Optimization of Traversal Queries on Distributed Graph Stores

A PROJECT REPORT

SUBMITTED IN PARTIAL FULFIMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

Master of Science (Engineering)

IN

Faculty of Engineering

BY

Abhilash Sharma

Department of Computational and Data Sciences

Indian Institute of Science

Bangalore – 560 012 (INDIA)

July, 2018

Declaration of Originality

I, **Abhilash Sharma**, with SR No. **06-18-02-10-21-15-1-12725** hereby declare that the material presented in the thesis titled

Optimization of Traversal Queries on Distributed Graph Stores

represents original work carried out by me in the **Department of Computational and Data Sciences** at **Indian Institute of Science** during the years **2015-18**.
With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discussions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Date: Student Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge, and I have carried out due diligence to ensure the originality of the report.

Advisor Name: Yogesh Simmhan Advisor Signature

DEDICATED TO

My Family and Friends

Acknowledgements

I would like to express a thankful note to my supervisor, Dr. Yogesh Simmhan, for his excellent guidance, meaningful insights and moral support. He has been supportive since the day I began working on this thesis and gave me the freedom I needed to explore this area of research on my own, while pointing me in the right direction in the times of need. I am also grateful to the Department of Computational and Data Sciences of IISc for providing valuable resources to aid my research.

I would like to thank all the labmates Aakash, Anshu, Diptanshu, Jayanth, Prateeksha, Pushkar, Rajrup, Ravikant, Sahil, Sarthak, Sayandip, Shayal, Shilpa, Shrey, Shreyas, Siddharth, Siva, Sumit, Swapnil, Venkatesh. A hearty thanks to my parents for encouraging and cheering me, in good and bad times. This thesis would not have been possible without them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

Large datasets are growing common, spanning web and social graphs, neural and protein interaction networks, and utility networks for transportation and power. It is increasingly common to see large datasets being modeled as graphs, be they search engines that serve webpage matches according to PageRank [37], source repositories modeling software dependencies and testing, or to match structural motifs in a database of chemical compounds [53]. There have been Big Data platforms for *batch graph processing* such as Apache Giraph [43], GraphX [52] and GraphLab [32] that perform graph algorithms that compute on the whole (large) graph, like finding the single source shortest path or PageRank. These weakly scale with the graph size but are designed for *high throughput processing*.

While Big Data platforms for batch graph processing exist [43, 32], there is much less emphasis on distributed graph databases that can support interactive queries and pattern mining over such large graphs [14, 41]. Such graphs are characterized by name-value property pairs assigned to the vertices and edges of the graphs, also called *property graphs* or *attributed graphs* [19, 35]. There is an increasing demand for graph mining frameworks [10] that can *interactively query* to find the existence of certain patterns, such as the reachability between two vertices or the k-nearest neighbors of a set of vertices, which helps domain experts or data analysts gain insights with *low latency*. Unlike graph algorithms that typically limit themselves to the graph structure or edge weights, these exploration platforms operate over *property graphs* [46]. Such platforms for distributed querying over property graphs have not been adequately studied. These are akin to "read only" *distributed graph databases* [41], which place an emphasis on scalable querying and analytics rather than ACID properties for updates. These queries also value low-latency concurrent execution rather than high-throughput batch processing as there are certain response-time constraints levied on the systems.

There is rich literature on *modeling graph datasets*. These include both relation and object-oriented models that graph are mapped to using tables or objects to represent nodes and edges, along with their properties [47, 18]. Other common data models to represent graph are semantic and property graph models [46, 19, 38]. These provide a formalism on which queries and analytics can be defined in a deterministic manner. The *graph query models* themselves often depend on the data model, and include the use of query graphs over simple graph models [6], path and reachability queries over property graphs [41], and subgraph pattern matching [19]. Formal languages like SPARQL and TinkerPop [9] exist that has query model over graphs.

Graph databases that implement these data and query models have been investigated as well. These databases may focus on data storage, cost model and query execution plans, updates to the data, and scalable execution. Relational databases have been used to store and query graphs by translating the queries to SQL that is efficiently executed [16]. Other support semantic graph queries over RDF triple representation [11]. Horton [4] provides an execution engine over property graphs with a query optimizer that uses predicate reordering to reduce size of intermediate results. There is a distributed version as well [41]. Our own work leverages a subgraph-centric distributed execution model to support queries over property graph [24]. Weaver, in addition, supports transactional updates for graph manipulation using a distributed graph store [14].

1.2 Challenges

Most of the graph pattern queries can be reduced to traversal queries. Traversal queries incorporates going through part of the structure of the graph by selecting starting vertices and going through the edges (outedges or inedges). At this point in time the frontier set of the traversal contains sink vertices of the edges, frontier set is filtered out based on predicate matching and same steps is repeated up to a certain depth. Some examples of traversal queries is BFS, Reachability and Path Queries. Path queries is a way for specifying graph pattern queries, hence we argue the hardness of the problem with respect to path queries.

Execution of path queries in a non-distributed setup requires the traversal graph structure as well as predicate matching. So the optimization entails finding of reduction or bottleneck points in the path query which causes the number of predicate matching to be vastly reduced when compared to other execution plans. Query cost models that can find such traversal behaviors are non-trivial due to the correlation between structure, predicate and between two predicates. Leveraging these query cost models for efficient distributed execution of path queries on batch graph processing systems is one of the challenges we address.

Further, the cost of a distributed query execution is affected by the time spent in compute as well as the time spent in communication and coordination. While distributed resources offer the opportunity for data parallel execution, they are still slower when compared to a shared-memory execution – if the graph will wholly fit in a single machine's memory. Communication and coordination costs can outstrip the compute advantages. So weak scaling is more feasible than Strong scaling. As a result, running on as few a number of machines as will fit the entire property graph in distributed memory is expected to offer the best performance. Java-based Big Data platforms suffer from object bloat, and are less memory efficient using a native object-oriented model for graph representation. Instead, using compressed data structures are likely to be more efficient. We explore the opportunities for these.

1.3 Contributions

Specifically, we propose to make the following contributions:

1. We leverage the query cost models proposed earlier in GoDB [24] to translate it to efficient distributed execution of path queries using the subgraph-centric GoFFish platform. We categorize different workloads of queries, and performing an in-depth empirical analysis using large real world datasets. These validate the scalability and performance of the execution and cost models.

2. We propose GoDBX, a compressed property graph model that builds on top of Succinct [5] and offers a relatively low in-memory footprint. However, this memory reduction comes at the cost of higher compute costs. We translate our distributed path query execution model to GoDBX for execution over this compressed data structure, and validate them for large real world graphs.

1.4 Thesis Outline

The remainder of the thesis is organised as follows.

In Chapter 2, we discuss the literature on different aspects to graph querying: the data model, query model and execution model. Graph Data Model and Query Model give a theoretical framework for performing querying over graphs. We discuss the different Graph Databases and elaborate on the data model and query model they use. Further we highlight different graph processing systems, the features that they support and their limitations when it comes to performing declarative querying. We also discuss frameworks for concurrent queries and discuss their limitations. We also give a brief description on Temporal Graph databases for completeness.

In Chapter 3, we offer background on the GoDB data model, query model, and cost model, and propose our platform architecture and execution model. We discuss different categories of workloads and how are they generated. Empirical evaluation on these workloads for the proposed distributed execution model are performed and compared against Titan, an open source graph database. We discuss benefits of cost model and explain in what scenarios it is advisable to use the cost model.

In Chapter 4, we discuss the advantages of using Succinct for compressing properties and graph structure using three data model variants. Further, we present the software stack of GoDBX and contrast it against GoDB. We then perform an empirical evaluation of GoDBX against Titan as well as GoDB. Using the heuristics based cost model for GoDBX, shows benefits as well as novelty of the cost model.

In Chapter 5, we highlight the key takeaways in this thesis as well as possibilities for future work.

Chapter 2

Background and Related Work

2.1 Graph Data Models

Graph data and query models offer a logical abstraction to the user to represent and access the graph datasets, and perform operations on them. These data and query models separate the specification from the implementation, and also determine the types of features and operations that can be supported. The query model is also tightly coupled to the graph data model. We review literature on graph data and query models to inform our model design.

Early work on graph databases did not make a clear distinction between the data and query abstraction, and the actual implementation. For e.g., *network and hierarchical models* allowed many to many relationships between entities, with the network (graph) model forming a superset of the hierarchical model [25]. Here, the vertices are stored as data structures and relations as pointers. These were used for recursive query processing. The application is fully aware of and tightly coupled with the data structure and its semantics. Since the structure is custom-designed for a particular graph, any structural changes to the graph requires changes to the data structure and pointer manipulation. While this is suitable for monolithic applications with read-only operations, it is inflexible for generic querying, data updates or schema changes.

Subsequently, the *relational model* [13] provided the separation between physical and logical levels of abstraction. Graph are modeled as two *tables* – nodes with node IDs and edges with source and sink node IDs. The properties of nodes and edges form additional columns in the tables [47, 48]. This model makes it costly to traverse the edges between vertices as it translates to multiple costly join operations. The relational model is suitable when the schema for vertices and edges is known, and if all vertices (and all edges) share the same schema. Otherwise, the addition of a property to one type of node/edge will create an extra column for all node types, which wastes space. This can be partially solved by introducing a table for each type of node/edge assuming known count of type of node/edge, but this adds additional join operations.

Object Oriented (OO) models [18, 20] have been used in computer aided design (CAD) and knowledge bases. Nodes are modeled as *objects* whose types are given by *classes* that can have complex values and methods. An edge is modeled as a references from one object to another, and provides either a containment (complex property) or an inheritance (parent-child) relationship between nodes. A *schema* defines class types and relationships between different classes. The OO model is more flexible than a relational model by supporting complex property values, and allowing the schema to be extended using inheritance or containment. It is also better for applications requiring a large number of traversals. However, the graph structure is limited to hierarchical or a DAG. GraphDB [17] proposes modeling graphs using object oriented paradigm, it defines link as first class entity which is used to model connected between two objects. It defines operators for functional querying over graphs, where a query is set of steps in which perform a chain of operations. It defines path as first class entity which is used to represent matched patterns and perform further querying.

An early survey [8] discusses different graph data and query models from the perspective of databases, and evaluates them on the basis of three data modeling concepts:

- Data Structures available at the schema and instance level
- Languages for querying and manipulating data
- Approaches to enforce data consistency

It posits that all **graph data models** operate upon a mathematical foundation of a graph, such as a directed or undirected graph, nodes or hypernodes, and hypergraphs [8], and impose additional features based on semantic or object oriented approaches. A *simple graph model* [18, 20, 6] has a schema and instances represented as digraphs. Information is modeled as labeled nodes and edges. *Hypernode* [30] extends this to support nested graph structures such that each node in itself can contain a graph at the schema and instance levels. Attributes are modeled as *(key,value)* edges. *Hypergraphs* [29] further generalize this to allow edges to connect more than two nodes.

With XML growing popular as a data representation and exchange format, processing and querying such XML documents has become important. XML documents can be viewed as a tree of nodes representing XML elements, with their children being either other XML elements nested within the parent node or literal values associated with a leaf element. XQuery is a query model over such XML documents [1]. [45] describes an *XML Database* with queries executed over such XML documents, internally stored in a relational database and mapped to SQL queries.

More generally, *Semantic models* [38] offer a wider range of well-defined meanings to relationships between data entities, such as aggregation, classification, instantiation, sub-classing, super-classing, attribute inheritance, and hierarchies. This goes beyond the parent-child relationship of XML and a generic source-sink relationship of a simple graph model, and can be used for richer and meaningful querying.

RDF [2] is a contemporary W3C standard for semantic models that uses a graph structure to capture linked data in an extensible way. RDF uses $\langle subject, predicate, object\rangle$ triples to refer to relationships (edges) between a subject (nodes) and an object (node or literal property), a set of which can be used to construct a labeled graph. SPARQL [3] [15] is a SQL-style query language to specify pattern queries over RDF which is translated to graph pattern matching operations. While RDF is very flexible and of particular use for reasoning over knowledge graphs, its flexibility and semantic underpinnings have its downsides in added complexity for specifying and executing queries. For e.g., since every item is modeled as a triple, even properties (literals) of a node are modeled as a separate node [46].

Property graphs [46, 19, 35] derive from the simple graph model, and instead of storing attributes of a node as a labeled edge, they store $(key, value)$ pairs as properties of nodes and edges. While some property graph databases define a schema graph to constrain relationships between different node types [34, 24], others do not enforce a schema and allow different nodes/edges to potentially have different attributes names and types [46]. Property graphs offer a flexible yet accessible model that allow them to represent and operate on simple graphs with ease, and include limited support for complex features such as semantics or hypergraphs as well. For e.g., to model hypergraphs, we first convert property edges into new property vertices. So, for each property edge $\langle src, sink, p\rangle$, we add a node n having property p and add edges from src to n and n to $sink$. A hypernode can be represented by a meta-vertex that points to all vertices that are contained in the hypernode, both original vertices and those created from edges. Similarly, a hyperedge can be modeled by creating a meta-vertex which points to all hypernodes (or vertices) in that hyperedge. Both semantic and hypergraph models are specialized and powerful constructs, but most real world applications can be easily mapped to property graphs, making them a suitable model as a starting point for our proposed work.

[1]W3C XML Query (XQuery), https://www.w3.org/XML/Query/
[2]W3C Resource Description Framework (RDF), https://www.w3.org/RDF/
[3]W3C SPARQL 1.1 Overview, https://www.w3.org/TR/sparql11-overview/

2.2 Graph Query Models

The survey [8] also categorizes **graphical query languages** over these data models. Several papers discuss query models over a *simple graph model* [18, 20, 6], where the user provides a query graph as a pattern of interest, and the result is a union of all the subgraphs in the instance that match the query graph. After getting the results of query, users may specify additional queries on the matched subgraphs [22]. Libkin and Vrgoc [31] provide a formal query language that can query over labeled graph. It extends regular path queries to conditional path queries where result of one step affects the next condition, and also comments on the tractability of the query execution.

Hypernode models define a logic based language to perform query and update on hypernodes, and allow the inference of new hypernodes from existing ones [30] For e.g., the HyperLog language [30] lets users define template hypernodes which can be matched against hypernodes in database. Insertions and deletions are performed on hypernodes based on rules specified along with the templates. HNQL is a procedural language that defines a set of operators for declarative querying and updating of hypernodes [30]. GROOVY supports querying over hypergraphs using HML which allows both queries and updates over labeled hypergraphs [29].

Property Graphs allow different types of queries to be defined over their graph structure and properties [46]. *Vertex/Edge queries* [24, 4, 46] filter and return vertices/edges that match a given property predicate. *Path Queries* [24, 4, 46, 28] specify a sequence of vertex and edge predicates along with edge directions which match paths in the graph that have that order of occurrence. Some limit these to paths with non-repeating edges [36, 28]. *Reachability query* [24, 4, 46, 9] indicates the start vertices and end vertices based on vertex predicates, and returns all shortest paths between the start and end vertices, optionally bounded by a distance [24, 9, 28]. Others [46] propose two forms of the reachability query, one that returns a boolean output for the existence of path, and the other which returns the shortest paths themselves.

A *Breadth first search (BFS)* [24, 21, 41] going outwards from starting vertices to a particular distance is also possible. *Graph Pattern-Matching* [9, 19, 28] allows for a sub-graph pattern to be searched in graphs, which is internally converted into a subgraph isomorphism problem, in the absence of labeled vertices. *Graph construction* [46] extracts subgraph patterns from a graph, similar to relational views, and enhances them by adding new vertices, edges or properties. Graph construction requires two patterns to be specified, one for subgraph matching and another for graph construction. The union operation [46] allows a user to merge set of graphs into a single graph.

Besides the abstract query models, formal query languages or query APIs have also been proposed. XQuery and SPARQL are W3C standards for XML and RDF graph data models, as mentioned earlier. SPARQL supports subgraph pattern matching while XQuery which proposes FLWOR (*For, Let, Where, Order by, Return*) expressions which support pattern matching over XML documents. PQL [28] is a query language for biological networks with an SQL-like syntax which supports subgraph pattern matching, and reachability queries over property graph model. Gremlin [9] is a functional language over property graphs model which support path, reachability and subgraph pattern matching queries. GraphQL [19] supports subgraph pattern matching over property graph model, and adopts the FLWR expression syntax.

Not all query models allow updates or construction of graphs, as opposed to querying over them [36]. Support for both updates and queries exist for simple graph models [18], hypernodes [30] and property graphs [36]. SPARQL also allows graph construction queries to be specified.

Lastly, **integrity constraints** over the data model are also considered by [8]. Based on the type of model, integrity constraints can be enforced to ensure its validity. Two common integrity constraints are *Entity integrity* and *Referential integrity*. Entity integrity ensures that the node/edge/Hypernode/Hyperedge have unique identifiers, while referential integrity verifies that the endpoints of edge/Hyperedge are valid. All models ensure these basic integrity constraints are met. Several graph data models [18, 20, 6] also use a schema graph which defines the relations that are allowed between entity types. This acts as an additional constraint on the data model.

Data redundancy is storage of same piece of data in more than one place. Literals are modeled as nodes in simple graph based models[18, 20] which may lead to multiple nodes for the same literal. Some models [18] provide operators to remove Data redundancy. Property graph [46] and Hypernode [30] based model suffer from data redundancy as properties of a node/edge is encapsulated by the node itself.

2.3 Graph Databases

Graph databases implement the graph data models described in the above section, and support efficient query execution above them. They may support both query and update capability while enforcing the constraints [16], or have a query-only feature that operates on the data in “read-only” mode [50]. They consider storage, indexing and cost models [11, 50]. Storage models attempt to reduce the memory footprint or complement in-memory querying with disk based computation. They may leverage existing relational or other execution engines [51], or build custom ones [50]. Execution models can either operate on a single machine [28], which typically assume that the whole graph can be retained in the machine's memory, or distributed models [24, 41] which execute queries over the graph that is distributed over a set of machines. Cost models helps choose one among several query plans during execution to reduce latency. Next, we discuss implementations of graph databases for various data and query models. We start with shared-memory databases, and then review distributed databases.

ObjectStore [27] is an *object-oriented database* that supports insert, update and delete operations. Traversal are preformed using containment queries over complex objects. It allows versioning of objects, so a user can instantiate a particular version of object (E.g., Employee with particular name), make changes to it and store it back. However, it works only for a tree data model.

GOOD [16] is an implementation of [18] which uses a *simple graph model.* It uses a graphical interface to help users specify subgraph pattern queries which describe parts of the database where graph operations should be applied. These include addition or deletion of nodes/edges, and abstraction operation to remove redundant nodes. GOOD is implemented using a relational backend where nodes and edges are stored as tables, so all GOOD operations are eventually translated to SQL operations.

GraphMiner [50] is a disk-based mining framework for structural pattern mining on large graphs. It uses a *simple graph model* and supports subgraph pattern matching queries. These can be used to compose algorithms such as mining frequent subgraphs. It uses an ADI structure that allows it to operate on large graphs without loading the entire graph into memory [49]. ADI stores three layers of information – edge list, graph information on each edge, and adjacency list The edge list stores the triplet pattern *(source label, sink label, edge label)* for each edge. A list of subgraphs is stored for each of these edge patterns. The final layer stores an adjacency list of each of these edge patterns appearing in a subgraph. Each of these layers can be swapped in and out of memory, based on runtime memory requirements, and this reduces the disk I/O for subgraph pattern matching queries.

DOGMA [11] proposes an index for fast subgraph pattern matching on disk and uses an *RDF triple-store* data model. DOGMA's index is based on a hierarchical binary tree where each node contains a subgraph and occupies a disk page. This ensures disk locality during runtime when performing pattern matching queries, which leads to fewer number of disk reads.

Jena [51] is also an *RDF database* that uses a relational database as the backend. It uses RDQL [42] which is SQL-like conditional selection language that is translated to SQL join queries on the relational backend that performs graph pattern matching. While Jena v1 used a normalized entity-relationship model for the triples using three tables, Jena v2 trades-off increased space for less time by using denormalized tables that translates to fewer joins at the cost of storing redundant information.

DEX [34] is a research database that operates on a *property graph model* with support for memory-efficient graph operations. It is a disk based system which uses an efficient graph representation by splitting the graph into smaller structures and caching recent data. It also uses object identifiers for nodes/edges, bitmaps for storing graph structure, and maps to store mapping from object identifiers to attribute. These three structures give a logical view of the directed property graph. These compact structures help DEX to efficiently access different parts of the graph in memory. However, it does not aim to optimize queries using cost models during execution.

GraphQL [19] also operates on *property graphs* and stores a lexicographic ordering of k-neighborhood of each vertex to the prune search space while performing subgraph pattern matching. However it is not distributed and assumes the graph can be loaded in memory in a single machine. Likewise, *PQL* [28] also uses a relational backend to represent property graphs. Apart from using two tables to represent the graph, it also uses “helper” tables that store all cycle-free paths that exists in graph, which helps in efficiently answering reachability queries. However, it does not aim to optimize these traversal queries during execution.

Neo4j [36] is an open-source fully-transactional graph database that is ACID compliant. It uses a *Property Graph* model backed by a Java-based persistent store that allows it to offer fast write throughputs. It does not optimize traversal queries. There is also a distributed-memory version of Neo4j that has several short-coming with respect to performance and correctness.

There are a few instances of distributed graph databases as well. *Weaver* [14] is a distributed graph store that uses a *property graph* model. It enables users to perform transactional graph updates, and allows ordering of graph manipulation requests for horizontal scalability. It does not optimize traversal queries. *Horton* [4] uses a *property graph* model, and provides a execution engine with a query optimizer that uses predicate reordering to reduce the size of intermediate results. Horton+ [41] extends this further and uses recursive predicate reordering by splitting the query to find the optimal query plan. It uses a vertex-centric Bulk Synchronous Parallel (BSP) execution model.

Our earlier work on *GoDB* [24] resembles Horton+ [41] in supporting distributed queries over property graphs. It uses our own GoFFish [44] batch distributed graph processing platform whose storage model stores the graph structure and its attributes separately so that only attributes required for the query are loaded into memory, to reduce the memory footprint and I/O latency. GoDB provides a distributed cost model based on predicate and graph topology statistics to find a query plan. It also offers an approach to executing these queries using the subgraph-centric BSP execution used within GoFFish. However, GoDB uses a static schema for both nodes and edges present in the property graphs.

Among the open-source distributed graphs databases, *Titan* [3] is the most popular. It is a scalable graph database built for high availability and fault tolerance, and support for both ACID and eventual consistency. It uses a *property graph* data model. Titan uses a tuple-based backend store such as Cassandra, and does not optimize for traversal queries. It uses the Gremlin Graph traversal language. Titan perform worse for traversal queries that have large intermediate resultset sizes, as shown by our earlier work [24].

2.4 Distributed Graph Processing

While graph databases support low-latency execution of concurrent declarative queries, there has been substantial work on distributed batch execution of graph algorithms using distributed graph programming and execution models. Graph databases may themselves use such graph processing frameworks to translate their data and query models to, just as relational database engines are used to implement graph databases.

Giraph [43] is an implementation of Google's Pregel [33] vertex-centric programming model. It provides a "think like a vertex" paradigm where logic for individual vertices are provided, and execute in an iterative bulk synchronous parallel (BSP) model, with bulk messages passed between vertices at synchronized superstep boundaries. Graph databases like Horton+ use such a framework to map their query execution to. Subgraph-Centric frameworks like GoFFish [44] allows users to operate on a coarser subgraph as a unit of execution, and this allows the application to perform more processing per superstep. Our GoDB graph database uses the GoFFish platform for its execution. While Giraph and GoFFish have a synchronous execution models, GraphLab uses an asynchronous execution model which perform better for some scenarios, and worse in others due to locking/unlocking required for consistency. Asynchronous programs are also harder to design. GraphX uses Spark [54] as the backend to store graphs as RDDs, and implement batch graph analytics. The memory footprint of GraphX is comparatively higher than others mentioned. Due to this, some works [55] even consider it to be non-scalable framework. NSCALE [40] lets users specify a subgraph on which subgraph-centric processing is performed. It only instantiates the subgraphs of interests to reduce the memory footprint but assumes that every subgraph of interest can be retained in a single machine's memory. This might only be true for k-neighborhood analytics, where k is small, for which this framework is optimized for.

As such, these framework are inherently designed for batch execution with high throughput and require applications to be imperatively specified. These frameworks are generally read-only. Our focus is on declarative OLAP queries that can execute concurrently, and with low latency, over property graphs.

2.5 Concurrent Queries

There has been limited work on executing concurrent graph queries in a distributed setting. Quegel [55] proposes a query-centric framework for execution of concurrent queries. Quegel leverages Pregel's *vertex-centric* programming model for specifying a query as well as its *BSP* based execution model. Multiple queries execute concurrently using *Superstep Sharing* where each query executes its computation within a BSP superstep thereby reducing the number of synchronization barriers incurred as opposed to each query using separate isolated superstep to execute. Queries can be queued. There is a capacity parameter which denotes the number of queries that can be running at any instant in the system. This parameter is specified by the user. Though this is simple and effective when the queries in a particular workload have similar characteristics for its superstep of execution, it breaks for workloads with mixed characteristics. Further this does not put any constraint on the response time per query, thereby focusing solely on maximum resource utilization.

Chapter 3

Distributed Execution for GoDB

GoDB [24] offers a design for a distributed graph oriented database, using a property graph model, path and reachability query models, and a query cost model. We first introduce the abstract design of GoDB as background material. We subsequently offer our contribution of a distributed execution model for GoDB using the GoFFish subgraph-centric batch processing system. These are subsequently validated for scalability and performance.

3.1 Background

In this section [24], the overall architecture of GoDB is described, this includes the Data model, Query Model, and Cost Model. The data model is how the graph and its attributes are represented. The Query model how a declarative query is specified, and it is tightly coupled with the data model as it constraints the access methods of the inherent database. The Cost Model uses heuristics based on Graph statistics to reduce the amount of communication and supersteps required.

3.1.1 Data Model

GoDB uses a property Graph data model. A property Graph is an attributed graph where each vertex and edges is associated with arbitrary number of $< key, value >$ pairs. There are often scenarios like social network where people and posts are modeled as vertices of the graph, edges of the graph are relationships between these vertices like friendship or who posted/commented in a post. Attributes of the vertices could be profile information of a person or post, while attributes of edges are time at which a relation was formed, relationship status etc. Similarly road networks have locations modeled as vertices in the graph while direct roads are modeled as edges. Attributes of vertices can be landmarks, name, latitude/longitude of a place, and so on.

The commonality in all these examples is that a vertex or an edge can have multiple attributes associated with it, which can be easily handled by property graph data model. Access methods of such data also plays a major role as it decides how a property will be extracted from the database. This overhead should be minimal because some of the queries which will be defined below will have a large number of accesses. Property graph data model ensures that accessing a particular attribute has minimal overhead.

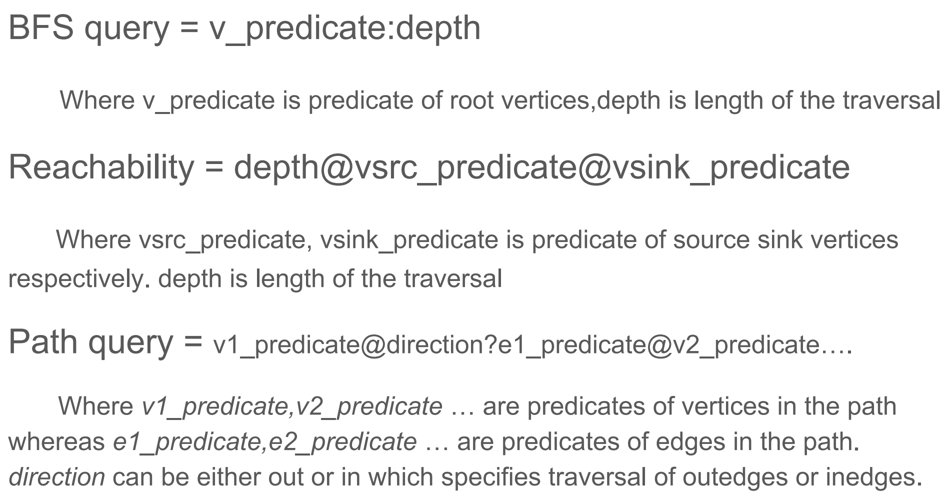
Before defining a property graph formally, we define some preliminaries:

- *Property* is defined as a triplet, denoted as $p =< key, type, value >$
- *Set of Properties* is defined as set of entity of type Attribute, denoted as $P = Set < p > ()$, where key should be unique in the set
- *Vertex* is defined as a node which has a unique id, denoted by $v =< id >$
- *Edge* is defined as having a unique edge id, along with a source vertex id and a sink vertex id, denoted by $e =< id, src, sink >$
- *Property Vertex* is a *Vertex* with *Set of Properties*, denoted as $pv =< id, P >$
- *Property Edge* is an *Edge* with *Set of Properties*, denoted as $pe =< id, src, sink, P >$
- *Set of Property Vertices* is defined as collection of entities of type Property Vertex, denoted as $PV = Set < pv > ()$
- *Set of Property Edges* is defined as collection of entities of type Property Edge, denoted as $PE = Set < pe > ()$
- *Graph*, denoted by $s =< id, \mathbb{V}, \mathbb{E} >$, is defined as set of Vertices and a set of Edges, with the constraint that the source vertex id and the sink vertex id of edge in $\mathbb{E}$ should also be present in $\mathbb{V}$.
- A *Property Graph* is also a graph, but whose vertices and edges can have properties. It is denoted as $sp =< id, \mathbb{PV}, \mathbb{PE} >$, where $\mathbb{PV}$ is a set of property vertices and $\mathbb{PE}$ is a set of Property Edges.

3.1.2 Query Model

The query model allows users to specify patterns to be searched over the property graph. GoDB supports three query types.

1. VE query: This query takes a predicate, i.e., a property name and value pair, defined on the properties of the vertex or the edge to return a set of vertices or edges that match the predicate.
2. BFS Query : This query runs a Breadth First Search (BFS) from root vertices and returns all the paths upto a certain depth. The root vertices are identified using a vertex predicate.
3. Reachability Query : This query returns the shortest path(s) from one or more source vertices defined by a vertex predicate, and one or more sink vertices, also defined by a predicate.
4. Path Query: This query specifies a pattern of alternating vertex and edge predicates that should match paths whose vertices and edges satisfy these requirements.



BFS query = v_predicate:depth

Where v_predicate is predicate of root vertices,depth is length of the traversal

Reachability = depth@vsrc_predicate@vsink_predicate

Where vsrc_predicate, vsink_predicate is predicate of source sink vertices respectively. depth is length of the traversal

Path query = v1_predicate@direction?e1_predicate@v2_predicate….

Where *v1_predicate,v2_predicate* … are predicates of vertices in the path whereas *e1_predicate,e2_predicate* … are predicates of edges in the path. *direction* can be either out or in which specifies traversal of outedges or inedges.

Figure 3.1: GoDB Query Specification

The specification of these queries can be found in Figure 3.1. Many other queries can be solved by reducing them to these basic queries as well. A VE query is specified by the passing a predicate as a Boolean expression. If it evaluates to true for an entity, it is said to have matched. While generally predicates may contain 1 or more variables which are combined using into an logical operators, GoDB's cost model only supports predicate having a single variable. BFS query can be specified by providing the starting predicate of the root vertices and the depth of traversal. As seen in Fig. 3.1, the vertex predicate and depth is specified separated by a colon.

E.g., $country = US : 3$ is a BFS query of depth 3 starting from all vertices having property name *country* with value as *US*. A reachability query is specified by giving the predicates of the source and sink vertices and a maximum depth of traversal. As shown in the query specification, the depth, source predicate and sink predicate are delimited by a @. E.g., $4@age = 30@age = 14$ is a reachability query of depth 4 from vertices having property value of *age* as 30 to vertices having property value of *age* as 14. Path query specification requires stating the predicates at each vertex and edge step of the path. E.g., $country = US@out?export = true@country = CA@in?import = true@country = CN$ is path query of three vertex predicates and two edge predicates. Edge predicates require specifying direction of edges, i.e., traverse along the out-edges or the in-edges. Wildcards are allowed for the predicates.

3.1.3 Cost Model

The cost model of GoDB [24] uses heuristics to calculate the cost of different execution plans for each query type. The different query types allow alternative forms of distributed execution, specifically the point at which the traversal can start and branch out, often using a BFS traversal constrained by the predicates matching on neighboring vertices or edges. These cost models allow the cheapest one to be picked for execution.

Since BFS query has predicates on the root vertices, there is no optimization possible as there is only one point to start a traversal. Reachability query has a source predicate and a sink predicate, and so there are two possible plans to start a traversal at either of them. The cost of traversal starting from source or sink predicate is calculated by optimizer and the plan with minimum cost is selected. A path query of length n has $n + 1$ vertex predicates and n edge predicates. So there are $n + 1$ plans possible, each corresponding to a vertex predicates in the query at which to start the traversal. The optimizer finds the cost of each of the plans and selects the plan with the minimum cost.

To emphasize the need for a cost model, consider an example path query of two vertex predicates and no edge predicates: `a@out?@b`. Here, $*$ is a wildcard predicate for the edges, indicating all edges are matched. Let us assume that the number of vertices that match predicate a and b are both 1. This is also called the vertex cardinality. Suppose the average out-degree of vertices that match a is 1000 and that of the vertices that match b is 50. Starting from a will require querying for starting vertices from the property index for the predicate a which will return one matching vertex. The out edges of the vertex matching a will be traversed and as there is no predicate on the edge, all 1000 neighboring sink vertices will be in the candidate pool for the next predicate b to be matched against. This match of the predicate for subsequent vertices is done using a vertex scan rather than an index since the index is global rather than limited to the prior matches. So 1000 vertex predicate matches need to be performed in the next step. These predicate matches using a scan are usually costlier than a lookup on the index.

Now lets consider when the traversal starts from b. Initially, the index is queried to find the starting vertices that match b, which is exactly one vertex. In the next step, all the out edges of this vertex is traversed leaving the candidate vertex to have 50 vertices. So we need to scan these 50 vertices to see if the next predicate a matches to find the path. It is clear that starting from b will thus yield faster execution time for this query since it is able to reduce the candidate pool to just 50 vertices rather than the 1000 vertices in the pool when starting with predicate a. This helps us establish the need for finding reduction points in the path queries.

Next, we reproduce the formal description of equations used in the GoDB cost model as formalized earlier in [24].

3.1.3.1 Preliminaries

Let $\mathbb{V}$ be the set of vertices and $\mathbb{E}$ be the set of edges in the Graph. Let $\pi(a)$ be a predicate defined on a property a of a vertex and $\widehat{\pi}(b)$ be a predicate defined on a property b of an edge.

Let $\mathbb{V}_{\pi(a)} \subset \mathbb{V}$ denote the set of vertices that match $\pi(a)$, with cardinality given by $n_{\pi(a)} = |\mathbb{V}_{\pi(a)}|$. Definitions used for edges is similar, $\mathbb{E}_{\widehat{\pi}(b)} \subset \mathbb{E}$ denotes the set of edges that match the predicate $\widehat{\pi}(b)$, with cardinality given by $n_{\widehat{\pi}(b)} = |\mathbb{E}_{\widehat{\pi}(b)}|$.

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

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

The *average local indegrees* and *output edge degrees* for predicates of vertices is defined as $\pi(a)$ be $\lambda^{in}_{\pi(a)}$ and $\lambda^{out}_{\pi(a)}$. 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 similar definitions for the average local out degree. Further, akin to local, definitions of remote are similar, the *average remote input* and *output edge degrees* of a predicate $\pi(a)$ be $\mu^{in}_{\pi(a)}$ and $\mu^{out}_{\pi(a)}$. *Remote edges* are edges that are incident on two subgraphs. 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 $\iota(a)$ be a function that returns benefit from starting traversal from a property a if it is indexed:

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

3.1.3.2 Reachability Query Cost Model

Let $q = d@\pi(v_{src})@\pi(v_{snk})$ be a reachability query of depth d from v_{src} to v_{snk} . Reachability query has two candidate execution plans: it can either start traversal from the source vertex predicate $\pi(v_{src})$ or start traversal from the sink vertex predicate $\pi(v_{snk})$. These two plans are denoted as $\Phi_{v_{src}}(q)$ and $\Phi_{v_{snk}}(q)$.

Reachability query execution needs to perform a BFS traversal from either of these ends. Hence, the estimated cost of reachability query from a predicate is essentially the cost of the BFS traversal. Estimation of the cost involves predicting the number of predicate scans, which check if the destination vertex is reached in the current pool's neighborhood, at each depth of traversal. Predicting the number of vertices in the candidate pool at the first depth is done by making use of the average out-degree of the source predicate and the average in-degree of the sink predicate. Subsequently, predicting for other depths requires the use of the average edge degree of the graph, which is common to all vertices. Therefore, the cost of both plans is dependent on predicting the vertices at the first step of BFS traversal. Thus, cost of both plans is given by:

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

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

The query optimizer selects plan which has the lower cost of the two plans, $\Phi_{v_{src}}(q)$ or $\Phi_{v_{snk}}(q)$.

3.1.3.3 Path Query Cost Model

Let $q = \pi(v_0)@out?\widehat{\pi}(e_0)@\pi(v_1)$ be a path query with two vertex predicates and one edge predicate. There are two candidate plans for executing this query: $\Phi_{v_0}(q)$ which starts traversal from vertices that satisfy predicate $\pi(v_0)$, and $\Phi_{v_1}(q)$ starting from vertices that satisfy $\pi(v_1)$.

There are two more terms that need to be defined. First is the *partial cost*, c, of doing predicate computations and the other is the *predicate cardinality*, n. These terms have direction associated with them: *forward* when it is along the same direction, according to specification of the query ($\overrightarrow{c}$ and $\overrightarrow{n}$), or *backward* if it is along the opposite direction from the query ($\overleftarrow{c}$ and $\overleftarrow{n}$).

There are three scenarios to consider in path query execution:

- Forward plan: The execution starts from the first predicate of the path query. This requires a forward BFS traversal in the same direction as the query specification.
- Reverse plan: The execution starts from the last predicate of the path query. This requires a reverse traversal, in a direction that goes opposite to the path specification.
- Mixed plan: The execution starts at a middle predicate, requiring both forward and reverse traversal to both ends of the path specification.

Forward Plan. This requires calculating the cost of the forward traversal. The cost for matching the starting predicate $\pi(v_0)$ and the cardinality of matching vertices is:

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

The cardinality of the starting vertices is calculated by multiplying the total number of vertices in the Graph by the probability of a vertex matching the starting predicate. The cost of finding the starting vertices can be found by using the indicator function $\iota(v_0)$.

Next, the cost of traversing the out edges from these starting vertices, and cardinality of edges matched is given by:

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

The cost of applying the predicate $\pi(v_1)$ to the candidate vertices remaining in the pool after applying the edge predicate, and the expected cardinality of vertices after applying the vertex predicate is:

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

Thus, the overall cost for the forward plan is:

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

The traversal shows recursive characteristics. Thus the generalization of the cost model for an arbitrary path query q_d of length d which has a starting vertex predicate $\pi(v_0)$ is,

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

The resulting forward query plan cost is:

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

Backward Plan. Similarly, this plan uses backward traversal using in-edges, and the cost model is similar to the *Forward Plan.* Working out the partial cost and cardinalities starting from the last predicate of the path query, the recursive equation is akin to *Forward Plan.* The overall reverse query plan cost is:

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

Mixed Plan. In this case, the starting predicate of the path query is in the middle. The execution of this type of plan requires performing forward as well as reverse traversals. This requires calculating the partial cost of both forward and reverse traversals. This can be thought of as executing two separate path queries, where one executes the *Forward Plan* starting at the middle, while other executes *Backward Plan* also staring at the middle. At last, it requires performing a join of the results of both the queries.

Given a path query of length d has $d+1$ the candidate query execution plans are $\Phi_{v_i}(q_d), i \in 0..d$, each starting at one of the vertex predicate in the path query. $i = 0$ and $= d$ correspond to the forward and backward plans, and the other intermediate values of i are the mixed plan. The cost model is used to find estimated cost for each of these plans, thereby choosing the plan with least cost.

The cost of joining results of forward and reverse traversals of the *Mixed Plan* is the *join cost.* The join cost depends the type of join used. In our proposed implementation, we use a *Hash Join.* So the join cost of a path query where a query q_d is split into two, $q_{0..i}$ and $q_{i..d}$ is:

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

Here, γ is an *join coefficient* which is ratio of cost of joining a pair of path and cost of performing a predicate computation. The overall query plan cost is the sum of estimated the cost of executing the forward and reverse traversals, and the join:

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

Network Communication Cost. Until now, the cost model assumes the graph to be on a single machine. However, in a *distributed execution*, there is cost involved in sending messages to neighboring subgraphs on a different machine, which hold the remote vertices. To model the cost of sending a message for the forward path segment, $\pi(v_i) \vdash \widehat{\pi}(e_i) \rightarrow$, we use the remote degrees of the heuristics. So the network cost is:

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

ω is a *network coefficient* that gives the ratio between the cost of transferring state to a remote subgraph and the query predicate evaluation time. Similarly, the backward traversal uses remote in-degrees for network cost i.e. $\overleftarrow{w_{\widehat{\pi}(e_i)}}$, $\mu^{in}_{\pi(v_{i+1})}$. This cost is evaluated for each of edge predicates in the path query. Summing up this cost for each edge predicate gives the total network cost for the query.

3.2 GoDB System Architecture

Here, we present an architecture for the distributed execution of the queries as planned by the earlier cost model proposed in GoDB [24]. This is part of this thesis' contribution and forms the GoDB system.

Figure 3.2 gives a pictorial representation of the GoDB platform stack. We use the GoFFish subgraph centric framework for execution of queries [44]. We leverage the heuristics-based query cost logic proposed earlier to find the execution plan for a query. The heuristics use statistics collected and maintained by *Statistics Collection and Maintenance* module to estimate the plan cost. This module provides a *Statistics API* to access the statistics. The *Query Optimizer* uses this to find the statistics required to estimate the cost of the candidate execution plans.

GoFFish v3 is implemented on Hama. The *GoFFish v3 API* has methods for accessing vertices, edges, and their properties, as well as methods for sending messages between subgraphs on different machines. HDFS is used as a distributed file system for storing and reading the initial property graphs into memory. A *JSON Graph Reader* facilitates reading a property Graph stored as JSON file on HDFS into GoFFish. The reader materializes the subgraphs into memory and creates the metadata required for sending messages. Upon completion, any subgraph centric program can be executed on the graph.

Indexing is used for executing the predicates corresponding to the starting points for a query. This is required for starting any traversal. GoDB uses Lucene for indexing vertices and edges. A Lucene index is created for each graph partition, and this is shared across all subgraphs in the partition. This avoids index fragmentation. The index stores subgraph information along

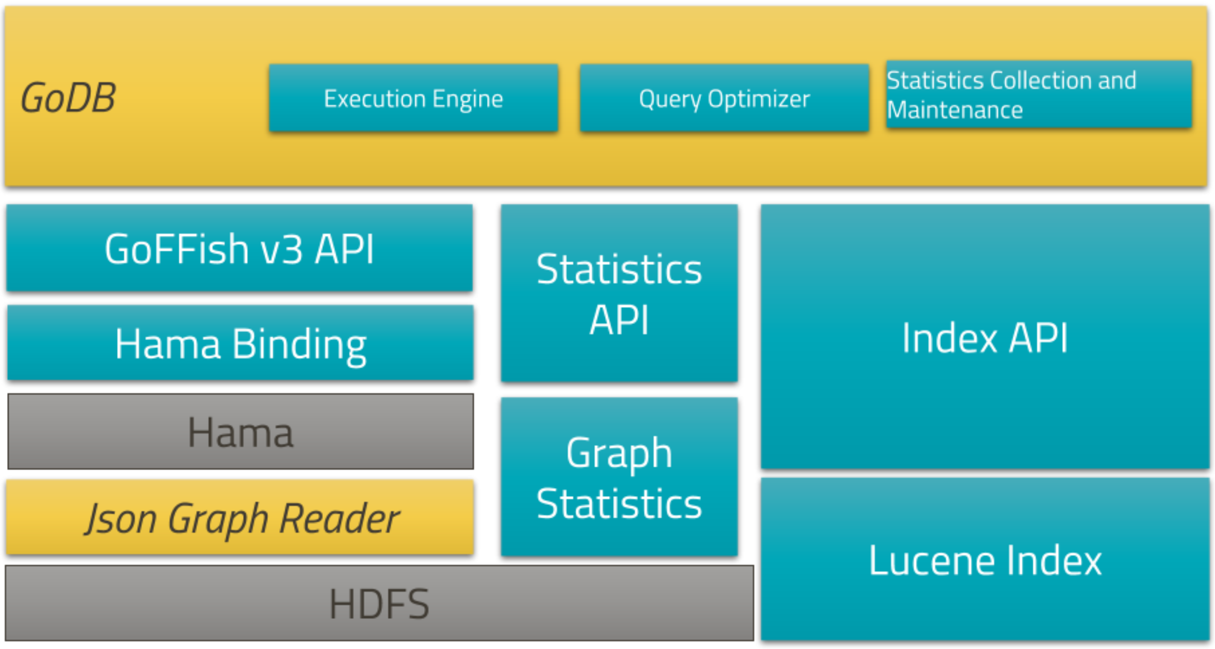


Figure 3.2: GoDB Stack

with the vertex id for the properties that are indexed. This allows a subgraph in a partition to query the index in a synchronized fashion so that all other subgraphs in the partition can use the results returned.

Figure 3.3 shows the distributed execution of GoDB, leveraging GoFFish. The graph is read from HDFS and materialized in memory on as many machines (workers) as the the number of partitions of the graph. A worker does computation on a partition. When a query arrives into the system, it is broadcast to all the workers. They locally parse and pass the query to the *Query Optimizer*. The Query Optimizer returns the start predicate for the query after consulting the statistics and evaluating the costs for the plans.

Each partition queries its Lucene index for the starting predicate to locate these starting vertices. Each worker then starts traversal and returns the set of paths as the result. Initially the traversals are local to a subgraph where the starting vertices reside. But when a remote vertex is encountered, a message is sent to the remote subgraph, which upon receiving the message continues the traversal. The message sent contains all relevant information about the current state of the traversal.

Query execution in GoDB can be demarcated into a *Traversal* phase and a *Result Collection* phase. Traversal involves exploring the graph and performing predicate scans on candidate

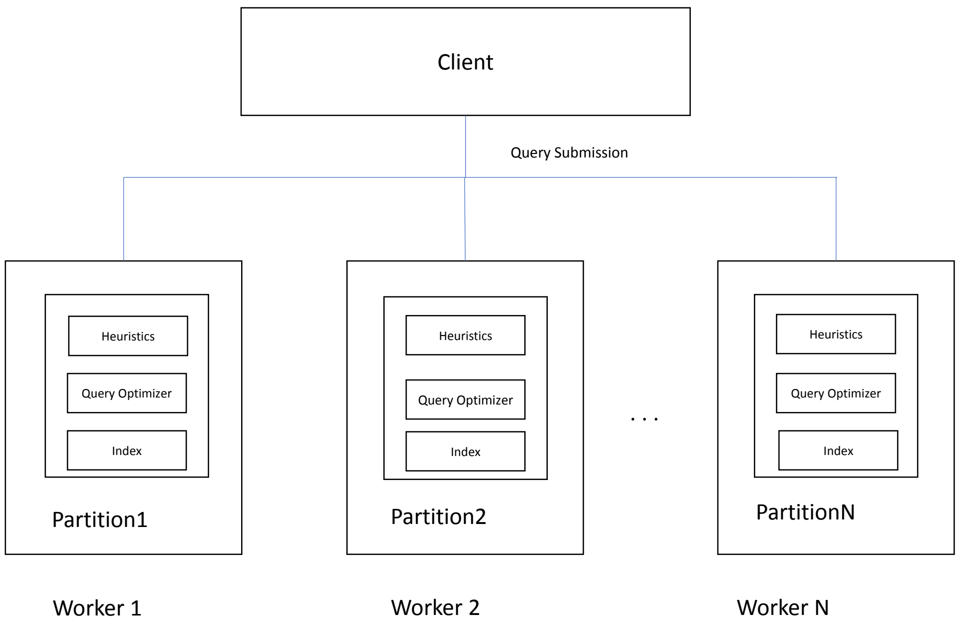


Figure 3.3: GoDB Execution

vertices to determine those that will be included in the next level of the candidate pool for further exploration. After the traversal ends, there are partial result set path(s) that needs to be aggregated and sent to the root vertex of the traversal. This is the Result Collection phase.

In the following subsections, we will explain the algorithms for the Traversal and the Result Collection phases for three query types.

3.2.1 BFS query Execution

GoDB supports two sub-classes of BFS queries:

- BFS with Revisits
- BFS without Revisits, also refereed to as a traditional BFS

One of the reasons to support BFS with revisits is to allow comparison against other graph query platforms that support the Gremlin query model which does not support a traditional BFS query without revisits.

3.2.1.1 BFS with Revisits

GoDB uses a subgraph-centric execution model for performing a multi-source BFS. In the first superstep, each partition queries the Lucene index the starting vertices based on the given

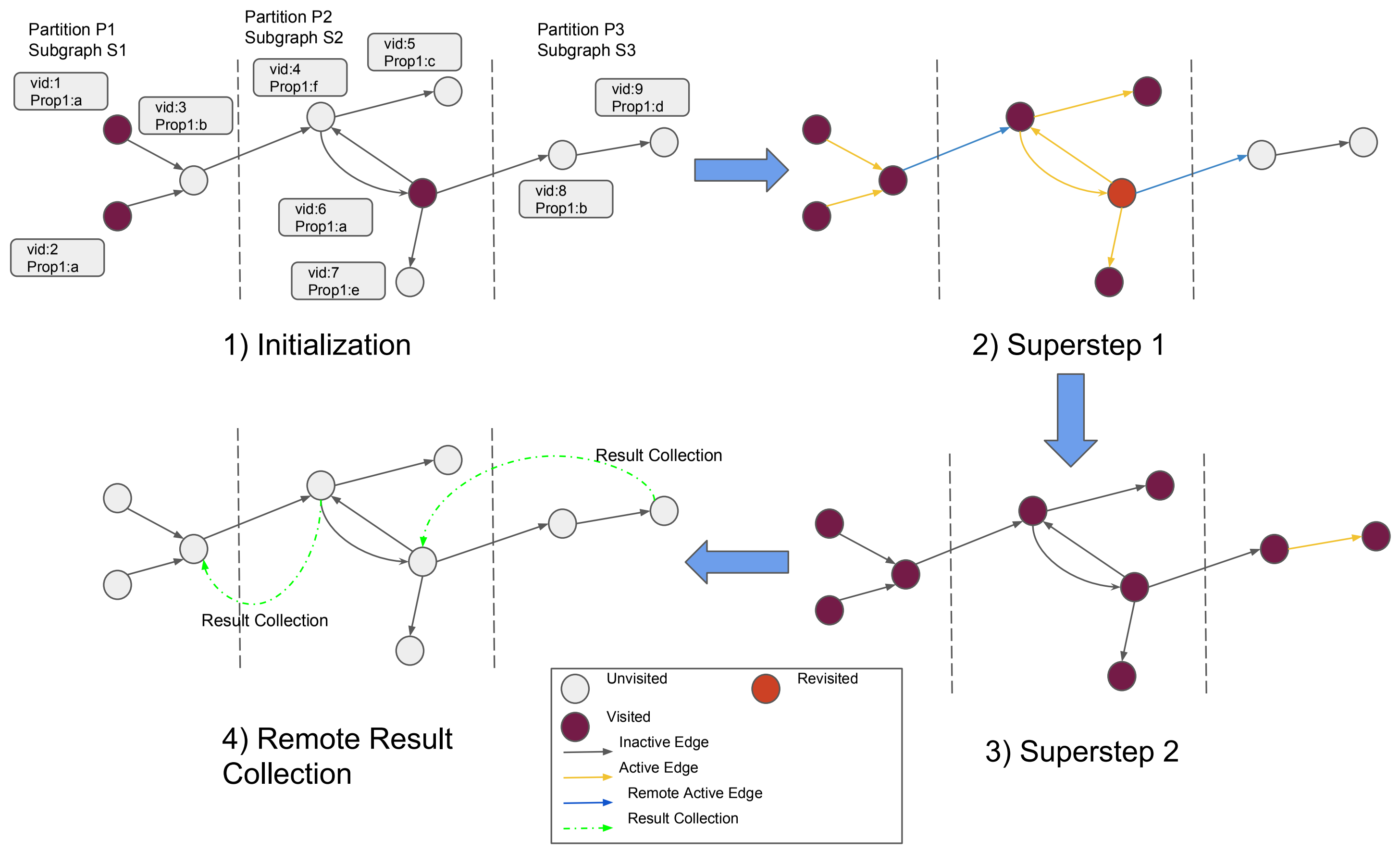


Figure 3.4: BFS Query Execution Model

25

predicate. The vertices are put in the local queues of their respective subgraphs in the partition. This is in the form of a *Traversal step* object which stores the state of the traversal: the current vertex id, path traversed, root subgraph id, and root vertex id. The path traversed contains the entire traversal of path from a root vertex.

Each subgraph in its first superstep polls for a vertex (traversal step) from the queue and starts traversing its out-edges. The sink vertex of an out edge can be either be local to this subgraph or remote, on another subgraph. If the sink vertex is local, then the vertex is add to the local queue so that it can be traversed later in the same superstep. The remote sink vertices are added to a remote queue. The local vertex traversals proceeds until the local is empty, and the BFS cannot proceed any further in this subgraph.Note that we can be performing a multi-source BFS as part of these traversals, and all the BFSs use the same local and remote queues. The step object has the root vertex which forms the context for the traversal.

Then the remote queue is iterated and messages are sent to the sink subgraphs before voting to halt this superstep. The message contains the traversal step object, the vertex id in the sink subgraph from which the traversal needs to continue, and the depth of current traversal. A subgraph upon receiving a list of messages in the next next superstep, extracts the traversal step object from each message and adds it to its local queue, thereby continuing the traversal. The same process as above proceeds. A traversal step terminates if it has reached the depth specified in the query or the vertex we reach has no out-edges.

Upon termination of a traversal step, the results need to be accumulated in the subgraph containing the root vertex of a traversal. This creates two separate conditions to be handled. First, if the current subgraph where a traversal step terminates is the root subgraph of the traversal, then it stores the results locally. Else, is current subgraph is not the root subgraph, the terminated traversal step needs to be sent to the root subgraph as a message. When there are no traversal steps in the queue as well as no output messages in flight, the query terminates.

Consider an sample execution of a BFS query $P_1 = a : 2$ on a hypothetical graph as shown in Figure 3.4. The Graph contains 9 vertices and 9 edges, and is partitioned into 3 as shown. So, each of the partition P1,P2,P3 contains one subgraph S1,S2,S3 respectively.

In the initial state, every partition queries its own index for the predicate $P_1 = a$. The index for partition P1 returns two vertices $[1, 2]$, P2 returns $[6]$ whereas P3 returns no vertices $[\ ]$. All Subgraphs iterate though their starting vertices and a Traversal step is created for each of the root vertices, with a step forming the tuple $(currentVID, depth, PathTraversed, rootVID, rootSGID)$. These steps are added to their local queue of their subgraph.

The traversal is in a subgraph initiated by polling a vertex from the local queue and traversing its out-edges, and adding the local sink vertices to the local queue and remote sink vertices to the remote queue of the subgraph. In partition P1, vid 3 is added twice into the local queue as this BFS is without revisits and we explicitly do not verify if the vertex has already been visited. When the local queue becomes empty, the remote queue is iterated and message is sent to remote vertex.

The state of the traversal in Superstep 1 is shown in the figure. All the grey vertices are visited exactly once, and black vertices are visited more than once. In P2, the traversal starts at 6 and reaches 5, and thus the Traversal step $(5, 2, 6 \rightarrow 4 \rightarrow 5, 6, S2)$ is the result. The location of this Traversal Step is local to the rootSGID and therefore result is stored in S2. Similarly, $(5, 1, 6 \rightarrow 7, 6, S2)$ is also a result.

In the next superstep, there are two message received by vid 4, but the traversal has already reached depth 2. Hence there are no further traversals, for this path. However, vid 8 starts traversal and visits vid 9. The state of the traversal is shown in Superstep 2 3.4. At this point, there are two remote results, one being at vid 4 and other at vid 9. So, the resulting path needs to be sent to rootSGID, as shown in *Remote Result Collection*.

3.2.1.2 BFS without Revisits

BFS without Revisits, uses a similar execution model as BFS with Revisits. The key difference is that a traversal from each of the root vertices will not visit a vertex more than once. To achieve this, we modify the execution model of BFS so that when polling the local queue for a Traversal Step, it checks if that vertex id has not already been visited through this root vertex. Logically, we maintain a list of visited flags for each vertex, one for each root vertex, using a bitmap data structure. The rest of the execution model is the same as BFS with Revisits.

The example in Figure 3.4 shows this query execution. The change here is that vid 6 will not get revisited since it was added to the visited list of itself.

3.2.2 Reachability Query

The Reachability Query uses the cost model described earlier to decide whether to start from the source predicate or the sink predicate. After selecting the predicate, a BFS is initiated from the starting vertices that match this predicate as queried from the index. The difference from BFS execution here is that the sink predicate (or the source predicate, based on the execution plan chosen) is evaluated at each step on the vertex being visited until a vertex satisfies it. When this happens, a message is broadcast to all partitions to set the depth of their query to the depth at which this vertex was found. This ensures that paths longer than this reachable path between the source and sink are not explored as this query finds all paths with the least distance between the vertices. When a match is finalized, the path is sent back to the root

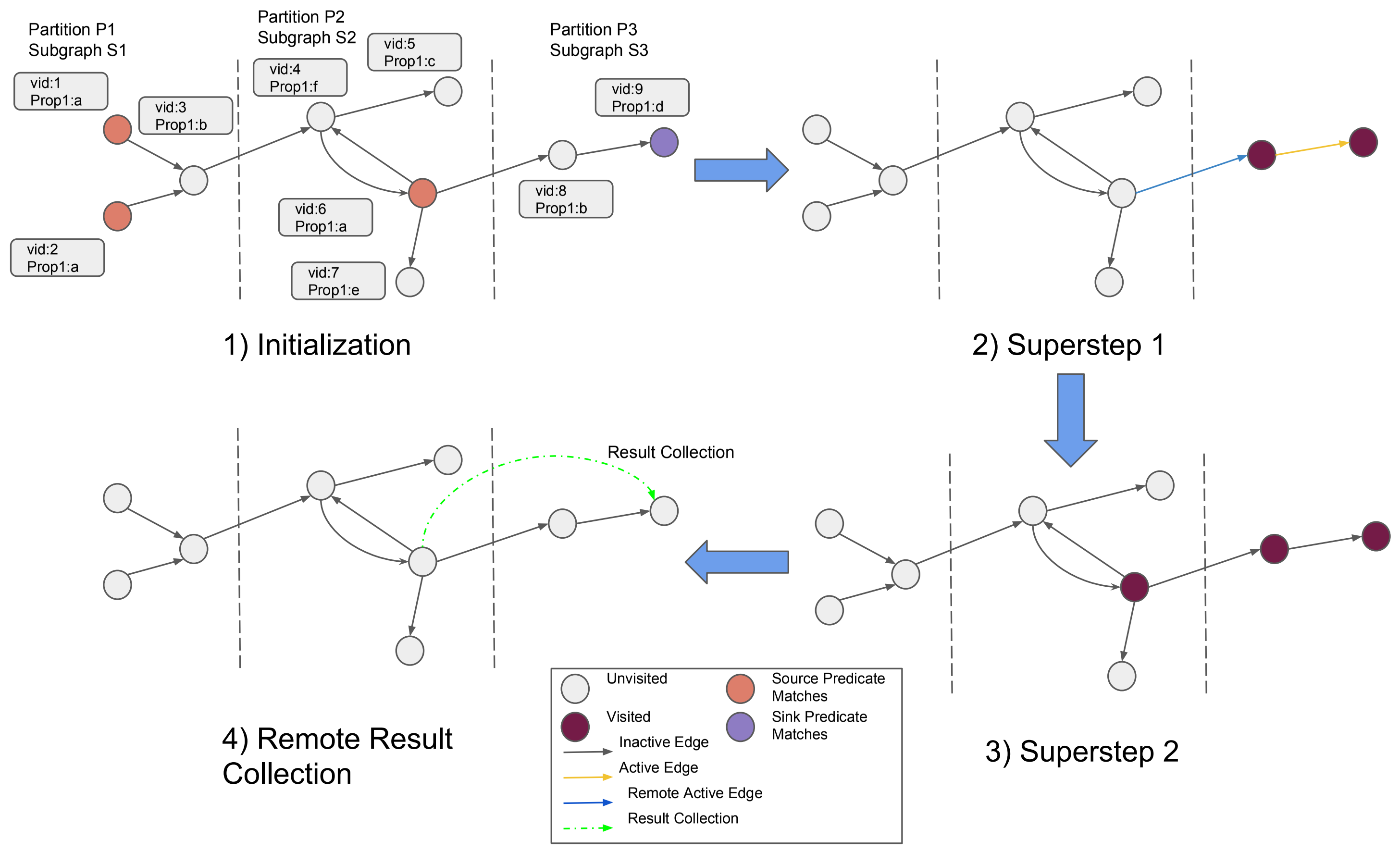


Figure 3.5: Reachability Query Execution Model

subgraph if the current subgraph does not have the root vertex of the traversal.

Consider an example Reachability query $4@P_1 = a@P_1 = d$ executing on a Graph having 9 vertices and 9 edges, as shown in Figure 3.5. The graph has 3 partitions P1,P2,P3 holding one subgraph per partition S1,S2,S3 respectively. There are two candidate plans: either starting forward traversal from predicate $P_1 = a$ or reverse traversal from $P_1 = d$. Here the cost from starting query execution from $P_1 = d$ is lower than that from $P_1 = a$. So the Cost Model selects $P_1 = d$ and the index is queried to find the initial set of vertices, which will be [9].

A Traversal Step similar to that in the BFS query is created and added to the local queue of Subgraph S3. Then the backward traversal is initiated from vid 9. There will be no other vertices in the local queue after vid 8 is processed and one remote message is sent to subgraph S2 so that traversal can continue from vid 6. Upon receiving the message, subgraph S2 starts traversal from vid 6, which matches predicate $P_1 = a$. As the rootVertex of the traversal is not in this Subgraph, the resultant path is sent to the root subgraph. A *STOP* message is also sent to other subgraphs so that they can set their new depth as length of current result. This reduces unnecessary traversals of depth higher than the minimum depth found so far.

3.2.3 Path Query

Path Query has the most complex execution model of the three query types. A query having n vertex predicates and (n-1) edge predicates has n possible execution plans. Each Vertex predicate serves as a possible start point of the traversal. Selecting the right starting point to start traversal is done by the *Query Optimizer* which runs the Cost Model and returns the starting point. Upon receiving this plan, the query execution is initiated by querying the index for the starting vertices using this starting predicate and adding them to the queues.

There are separate queues associated with forward and reverse traversal. Further, there are separate queues for local and remote vertices. If the starting point is the first predicate of the path query, then the starting vertices (traversal steps) are added to *forward local* queue. If the starting point is the last predicate of the path query, then it is added to *reverse local* queue. If it is neither, and is in some intermediate point in the path, then it is added to both *forward local* and *reverse local* queues. This allows the forward and reverse traversals to be executed concurrently from this middle.

Execution of the forward traversal entails polling a traversal step from the *forward local* queue and exploring its out-edges. During exploration, the next edge predicate is checked for every out-edge. The corresponding sink vertices (traversal step) for which the edge predicate is true is added to the *forward local* or *forward remote* queue, based on the whether the sink vertex is local or remote. When there is no traversal step left in the *forward local* queue, a similar

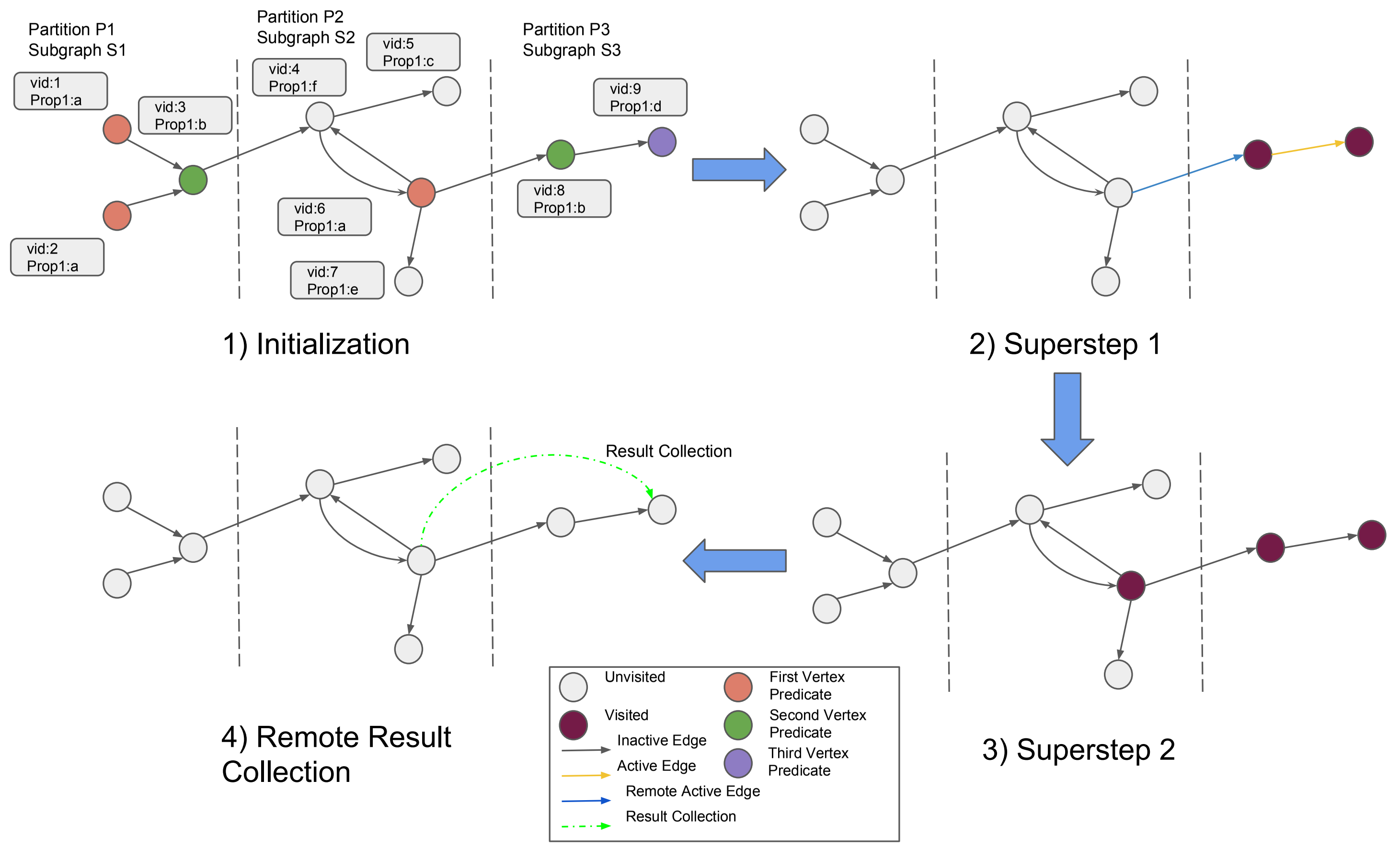


Figure 3.6: Path Query Execution Model

execution is initiated for the reverse traversal using the *reverse local* queue. When *reverse local* queue is empty, then no further local computation can be done in the current superstep.

The *forward remote* and *reverse remote* queues contain vertices that have to be activated remotely, so as to continue traversal. This is done by sending messages to the remote subgraphs which upon receiving the message, adds the traversal step to their appropriate local queue. The message contains the state of the traversal, which includes vertex id, next predicate to evaluate, direction (either forward or reverse), and the previous subgraph id. Based on the direction, the traversal step is added to either *forward local* or *reverse local* queue. The traversals resume as before.

When the forward traversal reaches the last predicate of the path query, then the results need to be collected. The path query execution model deploys a *Recursive Result Collection*. During the forward traversals, whenever a remote sink vertex is encountered, the traversed path is stored in the current subgraph state, while a message is sent to the remote subgraph. So whenever the traversal step matches the end predicate, the path traversed in this subgraph is sent back to previous subgraph which upon receiving this partial result joins it with traversed path in that partition thereby creating longer partial results. This is sent back to its previous subgraph and so on till the root subgraph of the traversal is reached. The reverse traversal also goes through the same algorithm for traversal and recursive aggregation of results. If the traversal had started in the middle, both these paths from the forward and reverse traversals will be present.

Recursive Result Collection helps reduce the network cost. When deploying this, the size of the message is reduced as the partial path is stored with the current partition, while only the state information is sent across the network with a minimal and fixed size. It provides further advantage by reducing the number of messages. It is observed that different traversals send messages to the same remote vertex. Some of these traversals have the same state apart from the path traversed. Thus only one message is sent on behalf of all these traversals after storing these traversal locally. The remote vertex continues the traversal and will return the partial results back at some point of the query execution, which will be shared by all traversals in that partition. Therefore this reduces the number of messages sent across the network thereby reducing the communication cost.

Figure 3.6 shows an example of a path query $P_1 = a@out?@P_1 = b@out?@P_1 = d$ executed on the graph with 9 vertices and 8 edges. There are 3 partitions P1,P2,P3 where each one holds one subgraph S1,S2,S3 respectively. The query is parsed and passed through the *Query Optimizer*. Let us assume that it returns the predicate $P_1 = d$ as the starting point of the query execution. Each partition queries its index, and P3 returns [9] for which a traversal step

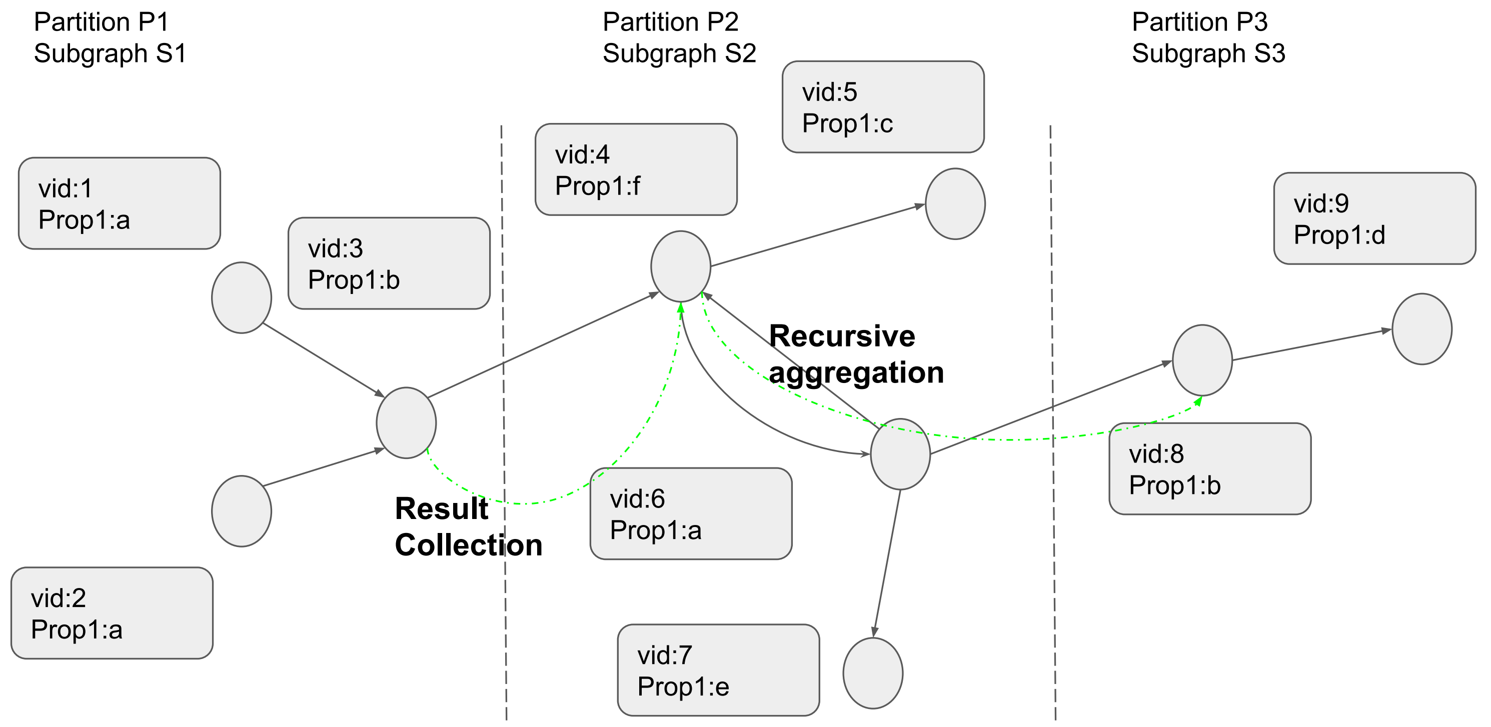


Figure 3.7: Path Query Recursive Result Aggregation

is created and added to the local queue of S3. The traversal step of the path query contains $(currentVID, currentSTEP, startVID, startSTEP,$
$prevVID, prevSGID, PartialPathTraversed)$.

The Backward Traversal starts from vid 9 and a partial path $9 \leftarrow 8$ is obtained in the first superstep. As a remote in-edge 6 is found, the partial path is stored in the current subgraph whereas $(remoteVID, STEP, currrentVID, currentSGID, Direction)$ is sent to S2, which creates traversal step from this and continues check for predicate $P_1 = a$. There is match in vid 6. Hence it uses recursive result collection to propagate $8 \leftarrow 6$ to S3 which joins this path with $9 \leftarrow 8$, thereby completing query execution.

3.3 Experimental Evaluation

In this section we provide a description of the workloads used for evaluating the GoDB cost model proposed earlier, and the distributed execution model described in this thesis. Select and Report queries are two sub-types of Path queries, with the distinction being the resultset size of the query. Select Queries have a resultset size between 1 and 10, whereas report queries have a resultset size more than 10.

We first introduce Titan, an alternative graph database for comparison against along with translating its query model to ours. Then we report comparative results for two graphs, Patent Citation Network (CITP) and Google Plus (GOOP), for these different query workloads.

3.3.1 Titan and Query Implementation in Gremlin

Titan is chosen as an alternate framework for empirical comparison it is a highly scalable Graph Database, and widely used. It is built on top of the Tinkerpop stack which uses Gremlin for specification of queries. Any traversal query can be mapped to the native Gremlin declarative query or by using the Gremlin API (as we do for BFS without revisits). It supports the property graph data model. It performs traversals during the query execution by performing predicate matches as well as traversing through the structure of the graph. Traversals are done by creating *Traversers*, which are similar to our *Traversal Steps.* The *Pull based* or OLTP execution model of Titan is highly optimized for light-weight declarative queries which traverse a small fraction of the graph, and we use this flavor. Here, a single Titan server coordinates the execution of the query but fetches necessary data from distributed Cassandra servers; the Titan server acts as a Casandra client. The Titan server is collocated on the same machine as one of the Cassandra servers. Hence, we have one machine hosting the Titan server and a Cassandra server, and 0 or more additional Cassandra servers involved in the execution.

In OLTP execution model, the traversal is done in in the single Titan server. After getting the starting vertices and creating *Traversers*, the traversals starts by pulling the relevant edges into the Titan server. These relevant edges may be *out-edges* or *in-edges* based on specification of the query. This could further initiate pulling of vertices. This pulling may go on for arbitrary number of steps based on the query. Cassandra is used as the backend to store the partitioned property graph, and it shows linear scale-out in Reads/Writes when we increase the number of machines. As a result, we try to find the most efficient configuration by varying the number of Cassandra nodes. For each dataset, we configure a certain number of Cassandra nodes, besides the single Titan server.

We next explain how the three classes of query supported by GoDB is mapped to Titan's Gremlin query model. Gremlin is a traversal based query language which allows us to specify traversals as a series of *Steps*. When a query starts, it creates *Traversers* from the starting vertices which store the state of the current traversal, and contains the current object (vertex/edge) of traversal, path, and the loop count. Further, traverser(s) can be created on the application of a step on each active traverser. Thus it takes a set of traversers as a stream and outputs a stream of traverser(s) for further exploration. So, a Gremlin query can be thought of as a pipeline of steps through which traversers flow through, and the traversers that remain at the end of this pipeline is the result of the query.

3.3.1.1 BFS query

Gremlin can naturally support BFS with revisits, while the traditional BFS without revisits requires us to write a wrapper code around the query Model to keep track of the visited vertices. We support both these models in GoDB, and this allows us to compare the native BFS model of Titan with revisits as well as the traditional BFS that we implement within Titan.

Below is a sample of defining a BFS with Revisits of Depth 5 using GoDB and using Gremlin:

GoDB: `country=LV:5`

Gremlin: `g.V("country","LV").out.loop(1) {it.object!=null && it.loops<6} {it.loops<7}.path`

BFS with revisits is mapped in Gremlin as declarative query as shown above. Initially, it starts by getting the starting vertices which is specified as *g.V("country","LV")* where g is an object of *StandardTitanGraph*. The second step is a traversal of the out-vertices by specifying *.out*, which is iteratively called by specifying *.loop*(1) where 1 specifies the backward jump in terms of steps for each iteration which is *.out* in this case. At every iteration, there is a *While Closure* that executes on the traversers, which is a boolean function that lets user specify the condition for continuing the next iteration. In this case, iteration for a traverser is stopped when it reaches a vertex with no out-vertices or the desired depth is reached. These two conditions is specified as $it.object! = null$ && $it.loops < depth + 1$, where it is the traverser object that gremlin provides.

The last part is the *Emit Closure* which is a boolean function that lets user specify if a traverser should be emitted. Here all traversers that terminated with a loop count less than depth+2 is to be returned, which is specified as $it.loops < depth + 2$. *.path*. This is used to specify that the traversers need to keep the path traversed in their state whereas not specifying this will return only last vertices of the path.

Below is a sample of defining a BFS without Revisits using Gremlin:

```
List bfsMultiPathList = new GremlinPipeline(titanGraph).V(key,val).as("x").
        out().loop("x", whileFunction,emitFunction ).path().toList()
```

BFS without revisits is not naturally supported. Though the *While Closure* is the same as BFS with revisits, the distinction comes in the specification of emit closure stated as *emitFunction*. In *emitFunction*, it is ensured that the traversers emitted to take part in next iteration of *While Closure* adhere to traditional BFS, i.e., no revisits. This is done by keeping track of vertices that have been visited. It is ensured that this tracking is separate for every root vertex.

3.3.1.2 Reachability Query

Below is a sample of defining a reachability query using GoDB and using Gremlin:

GoDB: `3@country="US"@country="CA"`

Gremlin:
```
g.V("country","US").out.loop(1) {it.object!=null &&
        it.object.country!="CA" && it.loops<4}.
path.filter{it.last().country=="CA"}.transform{it.country}.
        groupBy{it.size()}{it}.cap.next()
```

Reachability query execution in Gremlin is similar to the BFS query. First, it requires checking for the sink predicate at every step, which is incorporated in the *While Closure* and achieved by adding $it.object.country! = "CA"$ in the condition. The *While Closure* terminates when a sink predicate matches for a traverser. There is no *emit closure* specified. The traversers that remain have paths that may or may not match sink predicate. Thus it is required to filter out such traversers, which is specified as $.filter\{it.last().country == "CA"\}$. The rest of query specification essentially groups traversers by path length and returns the set of paths with the smallest length.

3.3.1.3 Path Query

Below is a sample of defining a path query specified using GoDB and using Gremlin:

GoDB: `country="AE"@out?@country="AU"@out?@country="US"@out?@country="0"`

Gremlin: `g.V("country","AE").out.has("country","AU").out.has("country","US").out.has("country","0") .path()`

The path query in Gremlin starts by creating traversers for the initial vertices. It then applies a series of vertex and edge predicate steps, thus performing a traversal of the graph and creating new traversers. Those that remain at the end of the last predicate step form the result of the path query. The specification of Path Query goes as *g.V(key1,val1).out(key2,val2).has(key3,val3).out(key4,val4).has(key5,val5).out(key6,val6).has(key7,val7)....*, where the vertex and edge predicates are specified with the traversal definition.

For instance the example path query shown above has 4 vertex predicates and no edge predicate. Initially it creates traversers for the starting vertices which is queried from the first predicate *V("country","AE")*. The next step is *out* that has no edge predicate specified, which means that there is no filtering on outedges. Thus it will create new traversers for all the out-vertices of starting vertices. Traversal goes one step further. Vertex predicate has *("country","AU")* is applied on the vertex object of the traverser, and those which remain will apply *out* again. This goes on until the last vertex predicate *has("country","0")*. The traversers that remain have the resultant paths in their state which is extracted by using the *path* step.

3.3.2 Results for CITP

3.3.2.1 Experimental Setup

The patent citation network (CITP) property graph is used in the first set of experiments [1]. It has 3.7M vertices and 16.5M Edges. GoDB is implemented on GoFFish V3 with Lucene 4.10.4 used for indexing. Azure D2 VMs with 2 Cores and 14GB RAM are used. The number of partitions is set to 4, with one partition per VM. The maximum limit of a JVM is set to 14GB. We use Titan 0.5.4 as a comparison, configured with Cassandra 2.0.15 and Elastic Search 1.4.2 running on 4 Azure D2 machines as well. DB cache is disabled for Titan, and the *row_cache_size_in_mb* in Cassandra is set to 0 (default). This avoids undue benefits due to hotspots in the workload. The maximum limit of its JVM is set to 14GB as well.

3.3.2.2 Workload Characteristics

Table 3.1 shows the workloads used for out evaluation. It shows the number of queries for each query type and category that are present.

There are 3 *categories* of workloads for each query type. These categories restrict the vertex predicates chosen in the query by adding constraints to their selectivity. *BFS query* has one vertex predicate, *Reachability* has two vertex predicates while Path Query can have arbitrary number of vertex predicates that can be restricted. The different workloads are:

Table 3.1: GoDB Experiments(Number of queries as entries)

Query Type	Length	T20	T100	T500	Total
BFS Revisits	3	54	31	14	99
	4	54	-	-	54
	5	54	-	-	54
BFS w/o Revisits	3	54	31	14	99
Reachability	3	100	100	100	300
Select	3	100	100	100	300
Report	3	100	100	100	300
VE	-	100	100	100	300

1. T20 : In this workload, vertex predicates in the query have a selectivity between 1 and 20 vertices.
2. T100: In this workload, vertex predicates in the query have a selectivity between 21 and 100 vertices.
3. T500: In this workload, vertex predicates in the query have selectivity between 101 and 500 vertices.

Varying the number of starting vertices impacts the number of vertices visited by a query during traversal. For example, consider two BFS queries of depth 2, query q_1 with number of starting vertices being 5 and q_2 with number of starting vertices being 25. If the average out degree of the graph is d, then the expected number of vertices visited is $5 \cdot d^2$ for q_1 and $25 \cdot d^2$ for q_2. So, this provides a good heuristics for finding workloads with diversity in the *expected* number of vertices visited during query traversal.

3.3.2.3 Synthetic Query Generation

T20,T100 and T500 workloads are meant to restrict the cardinality of the starting vertices. To find predicates that satisfy this, we use *statistics module* of GoDB to filter the predicates whose cardinality is restricted as per the workload definition. For instance, to find predicates of LT20, all the predicates are scanned from the *Statistics module* and those predicates are selected whose cardinality is between 1 and 20. Similarly, predicates for other workloads are found.

Subsequent generation of the query workload requires extracting paths from the graph. This is done by running a BFS from 1000 random vertices with a depth of 5, which is the longest path length in our workload. It also facilitates generating workloads of smaller lengths. The Paths from the BFS traversal tree are extracted, which contains properties for each of the vertices in

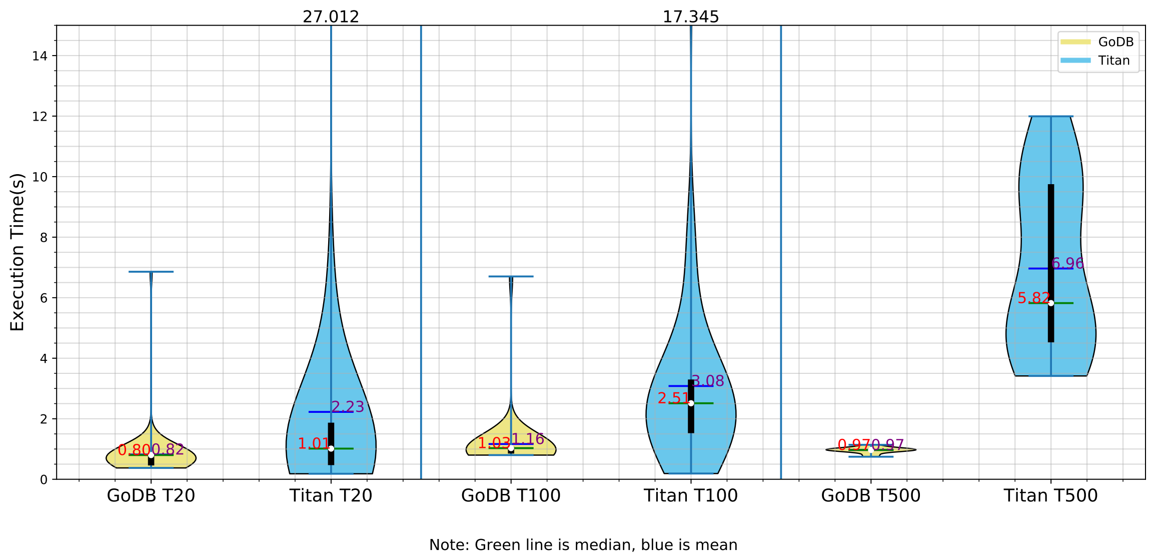


Figure 3.8: *BFS with Revisits* of *Length 3* for different *categories*, performed over *indexed* database.

the path. The paths are used to construct workloads for different query types. A BFS query is formed by finding paths where the initial vertex predicate satisfies the relevant category (i.e. T20,T100,T500). This ensures that there will be at least one result path of length 5, if the depth of the BFS query is 5. Reachability queries of length 5 are created by finding paths of length 5 that have the first and the last predicate satisfy the selectivity constraints. Reachability queries of lengths 3 and 4 are formed by finding paths of length 3 and 4 respectively.

Path queries of length 5 are formed by finding paths of length 5, and ensuring that each of vertex predicate satisfy the selectivity constraints as per the category. Similarly, Path queries of length 3 and 4 are formed. Apart from predicate selectivity, path queries are also categorized as select and report queries based on the result set size. Select queries have a result set size ≤ 10 while report queries have a result set size > 10. So the path queries generated need to be categorized by executing the queries (in GoDB) and checking the result set size. This generates a set of candidate Select and Report queries.

It is to be noted that BFS, Reachability, Select and Report Queries generated for different lengths, and satisfying selectivity constraints are candidates and not the actual workload. The actual workload consisting of n queries is generated by selecting n queries at random from the candidate queries.Table 3.1 shows the actual number of queries in the executed workload for each query type, which is used for evaluation.

For interpretation of the violin plots, a green horizontal line marks the median and a blue line marks the mean. The black bold vertical line indicate the first and third quartiles within

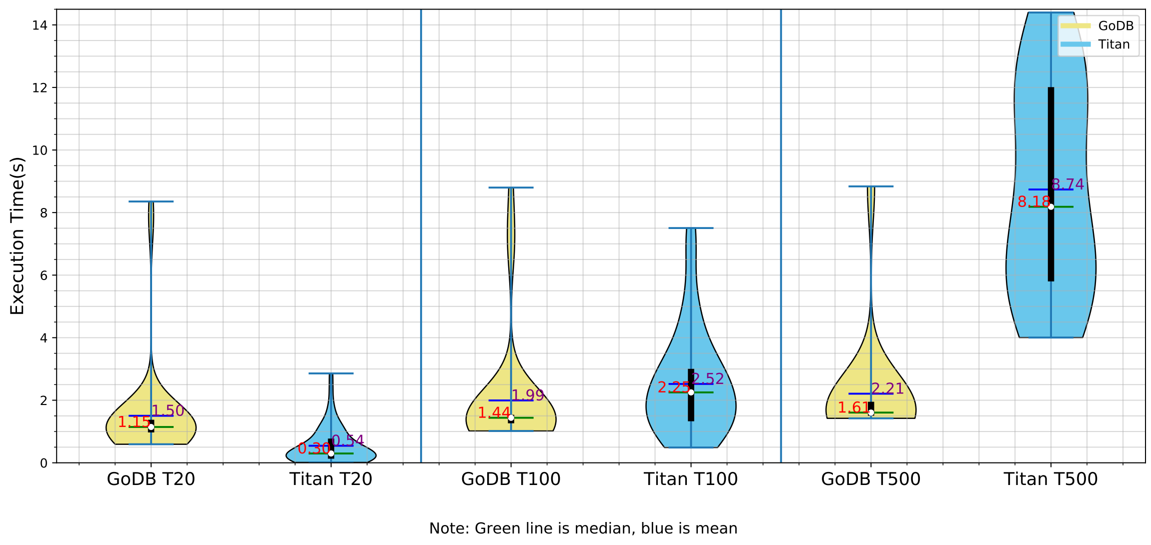


Figure 3.9: *BFS w/o revisits* of *Length 3* for different *categories*, performed over *indexed* database.

the violin.

3.3.2.4 BFS Query

Figure 3.8 depicts the execution time for GoDB and Titan for 100 BFS queries with revisits, of length 3 for T20,T100 and T500 categories. The BFS traversal is faster for GoDB by 2.7x to 6.1x than Titan. This is due to the subgraph centric execution model of GoDB as compared to OLTP model of Titan which pulls vertices to a single machine for traversal. As the number of starting vertices for the BFS traversal increases, it is expected that the number of vertices traversed will increase, thus increasing the computation and network overheads for Titan. This is observed for Titan for T20,T100,T500 workloads.

GoDB does not have an drastic change in the average execution time for the three categories. This is due to the synchronization time of BSP supersteps dominating the computation time as well as time for sending messages. On an average, $0.74s$, $0.81s$, $0.83s$ are spent in synchronization for T20,T100,T500 respectively. Thus the overall runtime becomes a function of number of superstep synchronizations which is approximately the same for the three workload categories.

Figure 3.9 shows the evaluation of the same three categories of workloads for a traditional BFS query, without revisits. Compared to BFS with revisits, there is an overhead of checking and updating currently visited vertices in a traversal. This is observed for both GoDB and Titan relative to the previous experiments. For Titan's T20 workload category, the number of vertices that are brought into the Titan server for traversal is reduced drastically as a result of

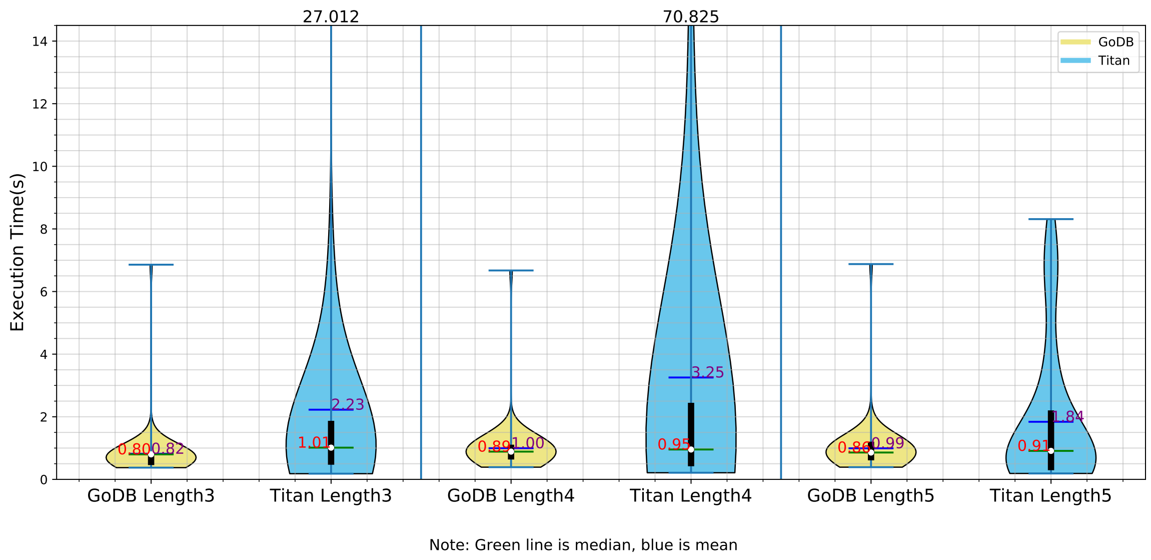


Figure 3.10: *BFS with Revisits* for *T20 category* for different *lengths*, performed over *indexed* database.

keeping track of the visits. Hence it performs better than GoDB. Otherwise, GoDB outperforms Titan, being approximately 4x faster than it for the T500 workload.

Figure 3.10 shows the execution time of BFS queries with revisits for the T20 category, for query lengths 3, 4 and 5. As the length of a query increases, it is expected that the number of vertices traversed will increase, thus increasing the computation and network overheads. This is seen from length 3 to 4. It is observed that these queries are dominated by synchronization time which are $0.74s, 0.83s, 0.81s$ for Length 3,4,5 respectively. The change in selectivity does not change the computation time for traversal by much when length changes from 4 to 5, thus execution times are approximately same in case of GoDB. In case of Titan, outliers in length 3 and 4 cause the average execution time to be higher than length 5. When outliers are removed, the average execution time are $1.73s, 1.78s$ for length 3,4 respectively. The advantages of the GoDB execution model and its robustness to varying lengths is apparent as it is 1.8x to 3.2x faster than Titan.

3.3.2.5 Reachability Query

Figure 3.11 shows the execution time for reachability queries for the 3 category of workloads. Reachability uses BFS to traverse the graph for finding shortest path between two predicates. Hence the increase in the number of starting points in the query increases the number of vertices needed to be traversed, thus increasing the average execution time. GoDB is 1.85x to 11x faster than Titan. For GoDB, the trend is same as that of BFS query, whereby the synchronization

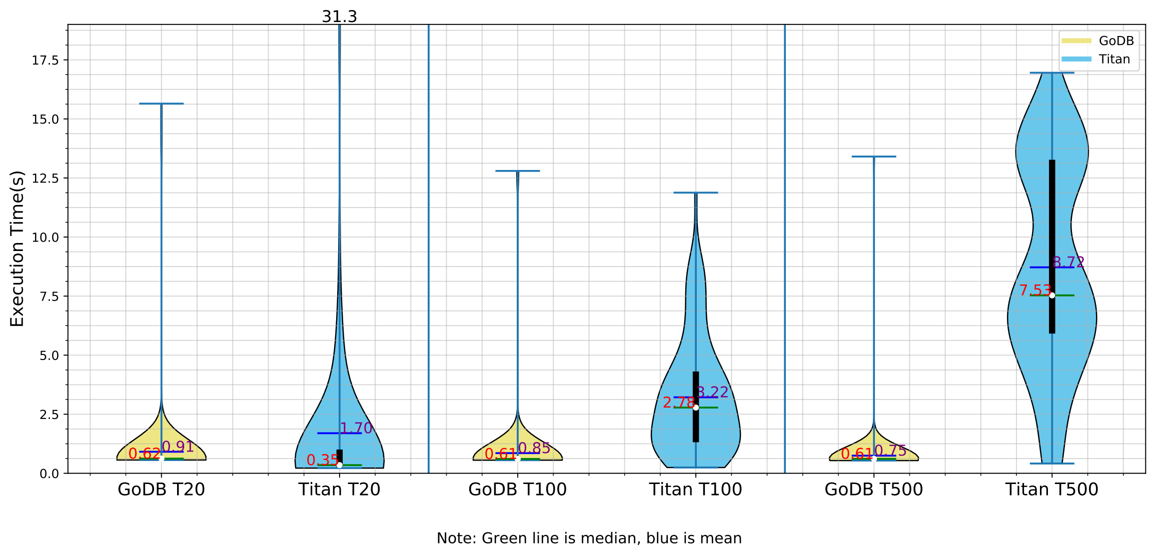


Figure 3.11: *Reachability* of *Length 3* for different categories, performed over *indexed* database, using *Cost Model*.

time dominates the overall computation time. So we see only minor changes in the execution time across the workloads.

Figure 3.12 shows the effect of using the cost model on reachability queries. The cost model is compared against fixed plans, i.e., starting at always the same end – either source or sink, and against an oracle model which always uses the best of the two plans based on actual runtime. The workload used here find the reachability between two unique vertex ids. It is seen that there is negligible advantage for using the cost model in this case. This can be attributed to the fact cost model as well as the fixed plan models are very close to the oracle model. Thus choosing either of the two plans will still be close to the oracle for this workload.

Figure 3.13 shows cumulative distribution plot of each plan against the optimal plan. X Axis shows the factor by which queries are slower than the optimal using a particular model, while Y axis shows the cumulative fraction of queries in the workload that are slow by this factor (or lesser). Our GoDB cost model chooses the optimal plan for 70% of the queries, and in the worst case chooses a plan that takes 0.5× more than optimal. On the other hand, the fixed plan with the source predicate has a worst case that is over 2.5× over optimal.

3.3.2.6 Select Query

Figure 3.14 shows the execution time for Select query for GoDB and Titan. GoDB outperforms Titan for all three workload categories, being 1.8x to 6x faster than Titan.

Figure 3.15 shows the advantages of our cost model against fixed plans, where the traversal

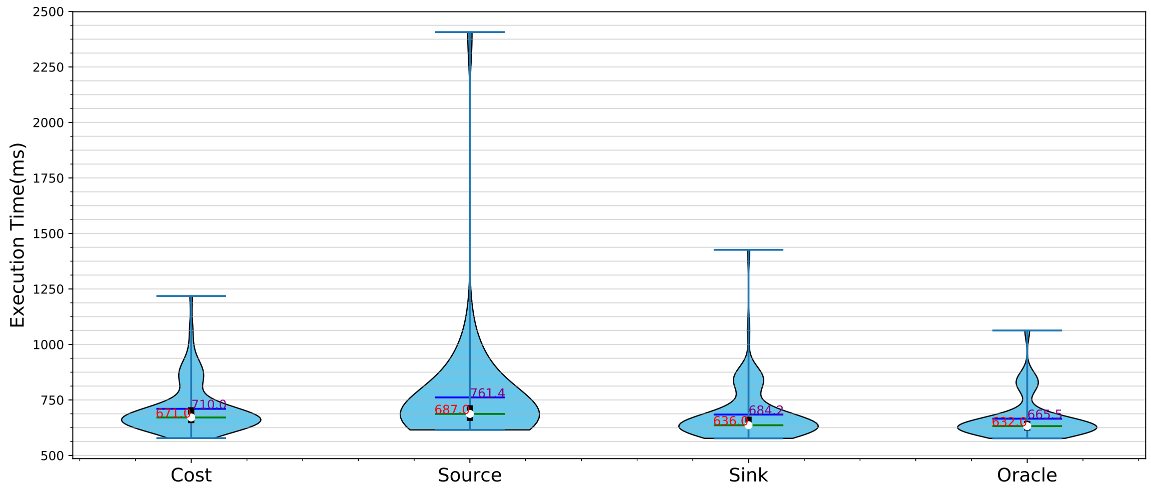


Figure 3.12: Cost Model benefits of *Reachability query* of *Length 3*, for *T20 category* over an *indexed* database.

starting point is fixed to one of the vertex predicates in the path query, and the oracle model. As the select query is light weight, we see that the effect of the cost model is not apparent as all the fixed plans have execution times close to each other. However, our cost model does help to avoid the worst plans, as seen by the long tail in V1. The same trend is observed for T100 and T500 workload categories in Figures 3.16 and 3.17.

Figures 3.18, 3.19, and 3.20 depict the cumulative distribution plots for T20, T100 and T500 respectively. It is seen that all the plans are close to each other.

3.3.2.7 Report Query

Figure 3.21 shows the execution times of report query for GoDB and Titan. GoDB outperforms Titan in all three categories of workloads, being 1.6x to 5.5x faster. There is a increase in execution time going through T20,T100 and T500, which is due to increase in number of predicate scans due to the increase in the number of starting vertices.

Figure 3.22 shows the benefits of using the cost model for selecting a plan as compared to fixed plan models as well as oracle model. It is observed that the cost model performs better than fixed plan model, attributed to the report query traversing a large number of vertices. Thus these workloads have some degree of improvement possible by using cost model to find the reduction point in the traversal.

Figure 3.23 shows the cumulative distribution plot of the cost model against the fixed plan models. It is observed that the cost model has a quicker convergence as compared to fixed plan model, thus showing the benefit of the cost model for such heavy workloads.

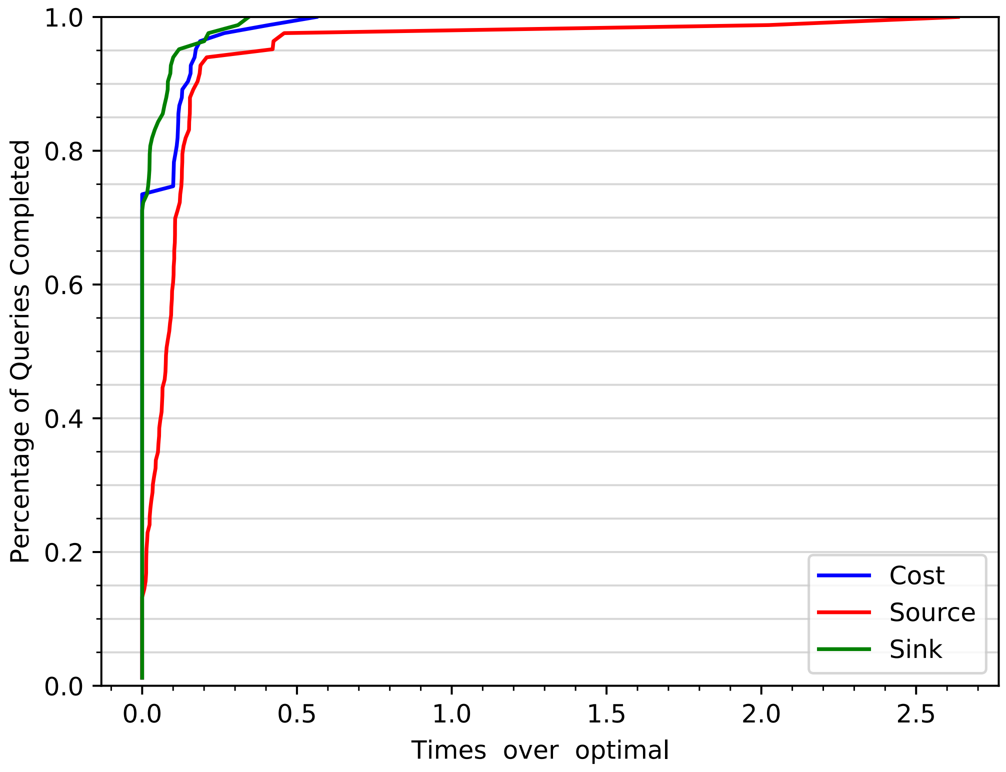


Figure 3.13: *Reachability query* Cost Model Cumulative for *indexed* database, Ratio against optimal.

3.3.2.8 VE Query

Figure 3.24 shows the comparison for VE query type, which are queries having either a vertex predicate or an edge predicate. In our workload we use vertex predicates, thus the queries essentially becomes a path query with one vertex predicate. This query returns the vertices that match the vertex predicate, which is either by using indexes or by traversing through all vertices and matching the predicate. It is observed that GoDB is slower than Titan. This is due to in-memory *Lucene* index of GoDB being slower than *Elastic Search* used by Titan. Comparing *non-indexed* GoDB and *indexed* Titan is also shown in Figure 3.25. Here, GoDB traverses through all vertices in each partition, which is costly, taking approximately 0.9s for all workloads. An improvement in index querying time will reduce the query execution time for all other types of queries (BFS,Reachability,Path) as well since the starting points for these queries are effectively a vertex query, thus making GoDB even faster than Titan.

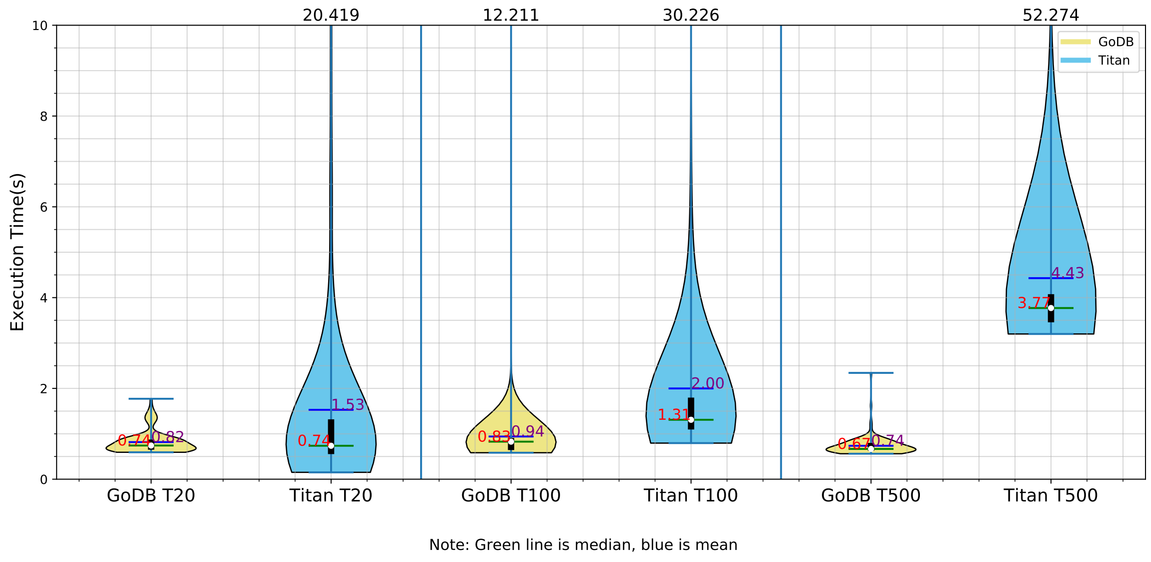


Figure 3.14: *Select Query* of Length 3 for different *categories* over Indexed database, using *Cost Model*.

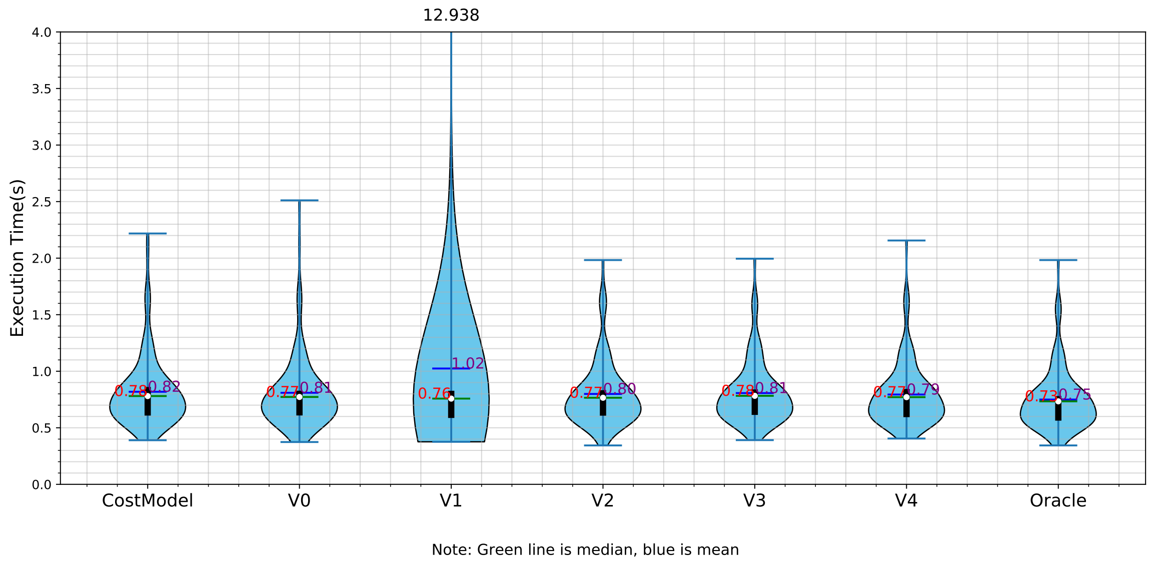


Figure 3.15: Select Query Cost Model benefit for *workload T20* for *indexed* database.

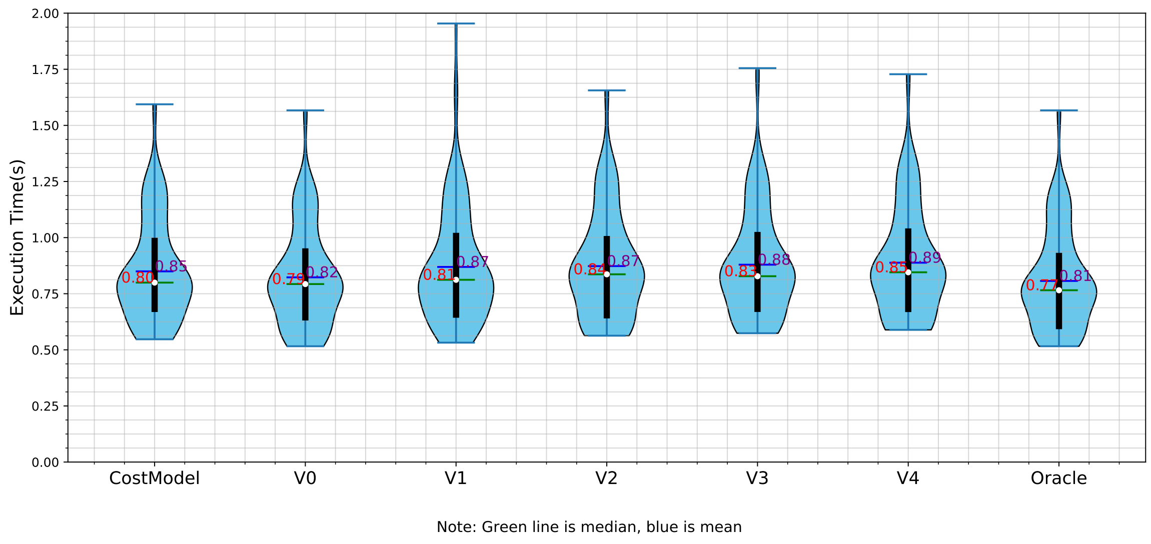


Figure 3.16: Select Query Cost Model benefit for *workload T100* for *indexed* database.

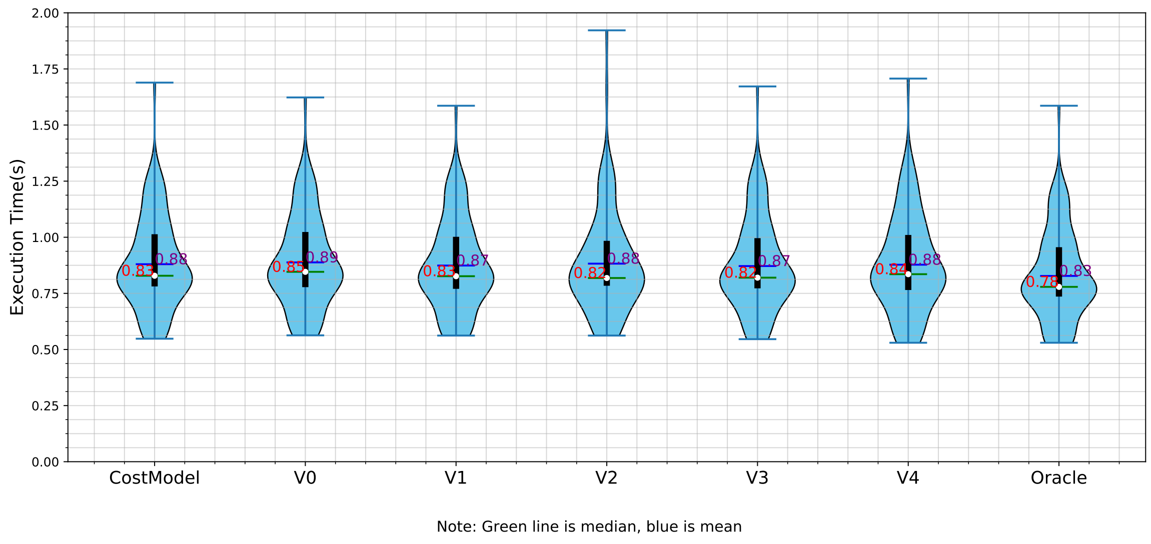


Figure 3.17: Select Query Cost Model benefit for *workload T500* for *indexed* database.

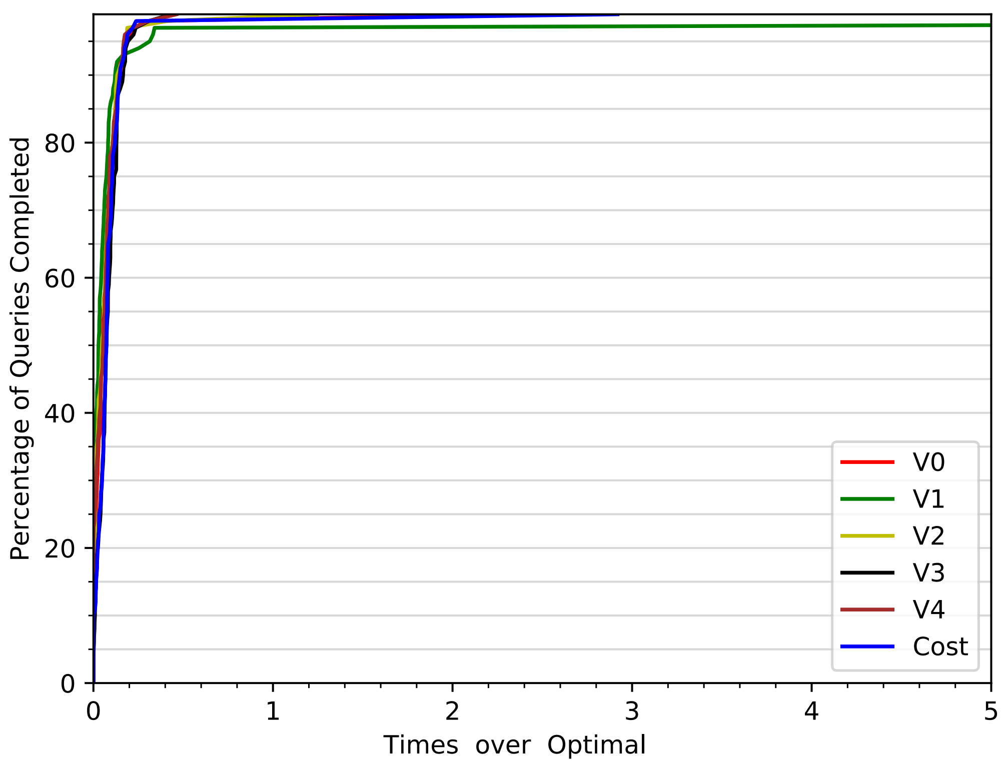


Figure 3.18: Select Query Cost Model Cumulative for *workload T20* for *indexed* database, Ratio against optimal.

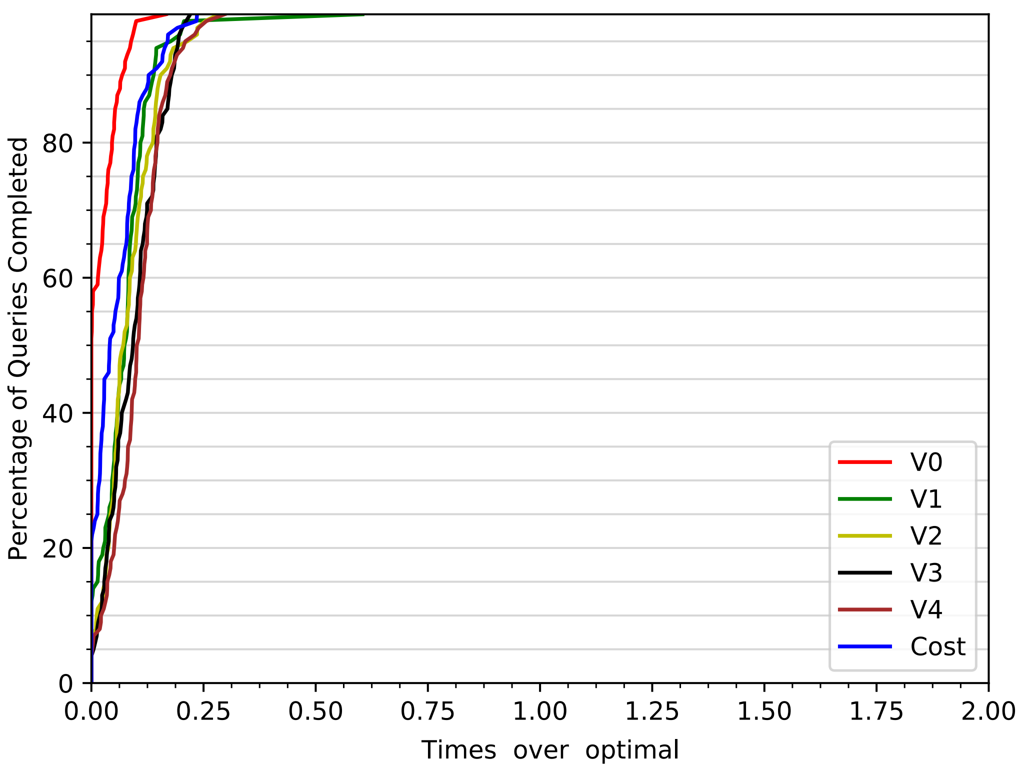


Figure 3.19: Select Query Cost Model Cumulative for *workload T100* for *indexed* database, Ratio against optimal.

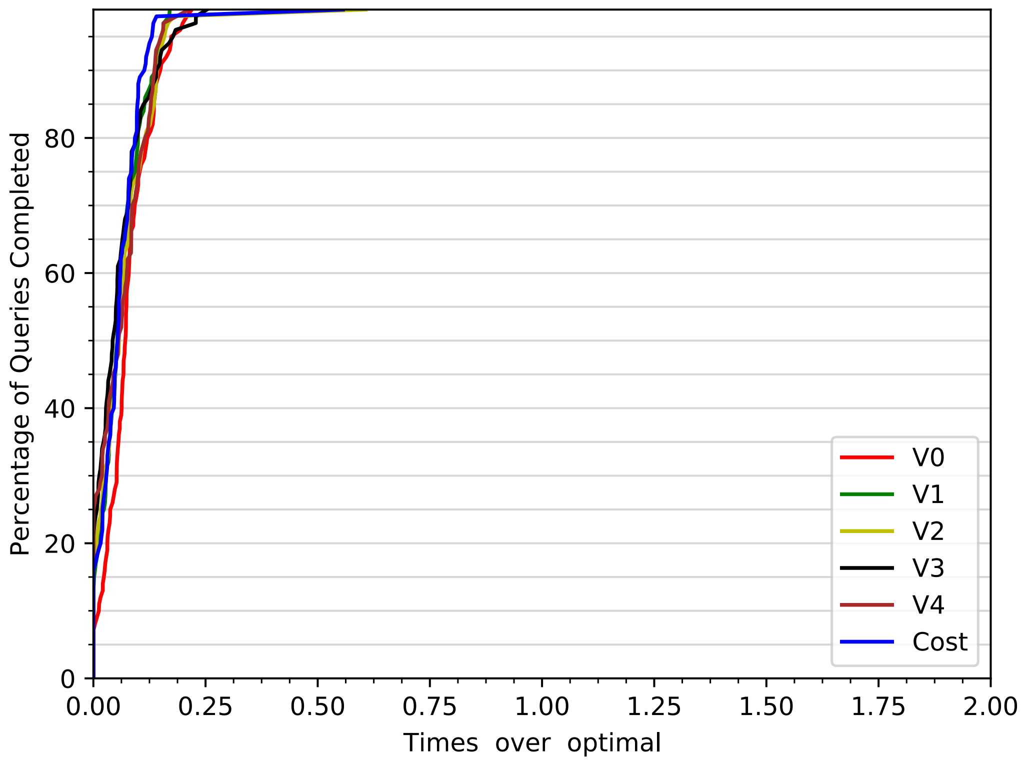


Figure 3.20: Select Query Cost Model Cumulative for *workload T500* for *indexed* database, Ratio against optimal.

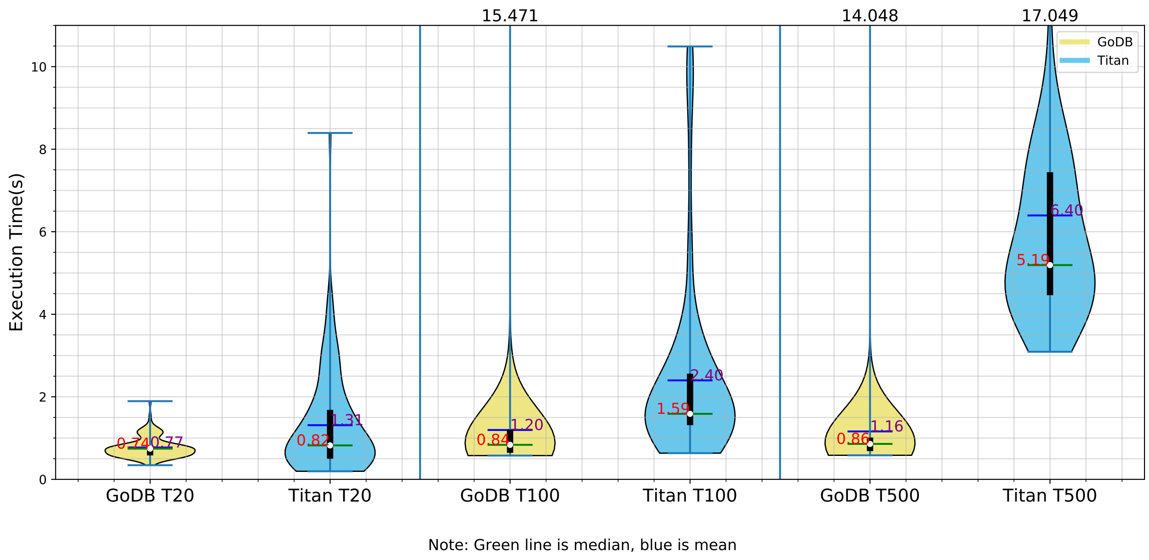


Figure 3.21: *Report Query* of *Length 3* for different *categories* over *indexed* database, using *Cost Model*.

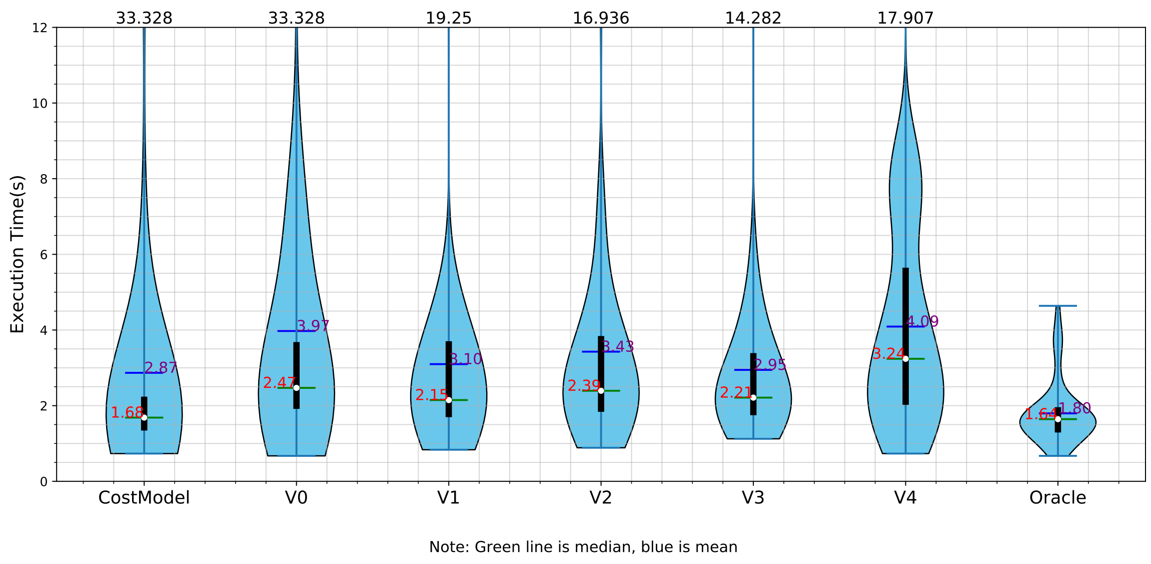


Figure 3.22: *Report Query* Cost Model benefit for *All Categories* together for *indexed* database.

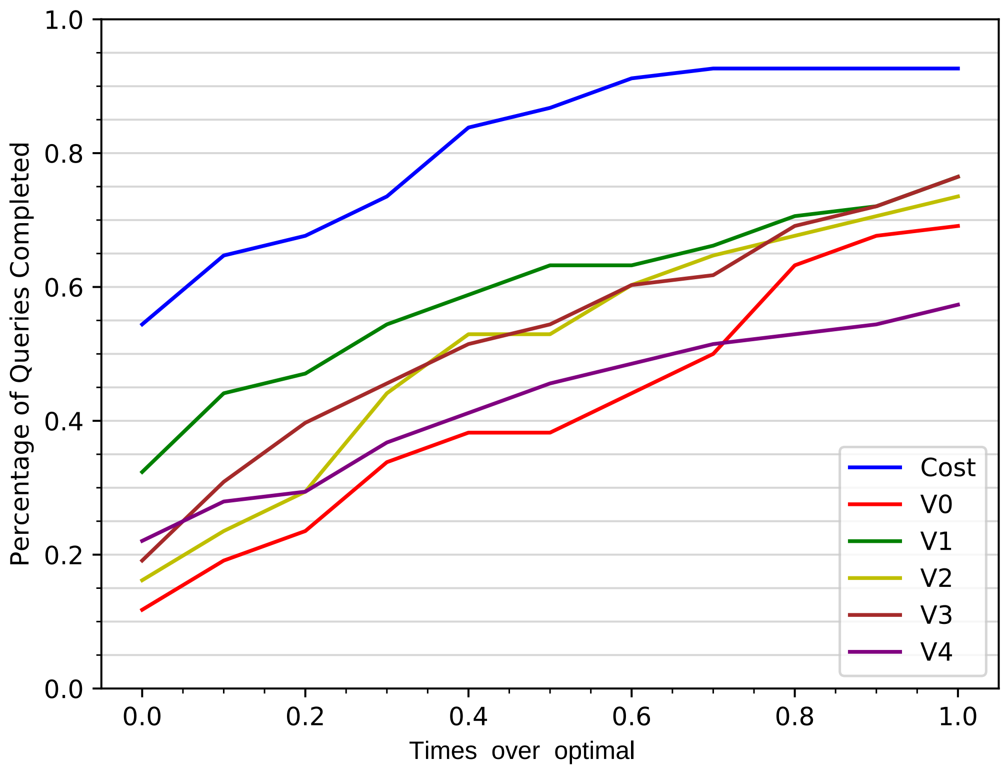


Figure 3.23: *Report Query* Cost Model Cumulative for *All Categories* together for *indexed* database, Ratio against optimal.

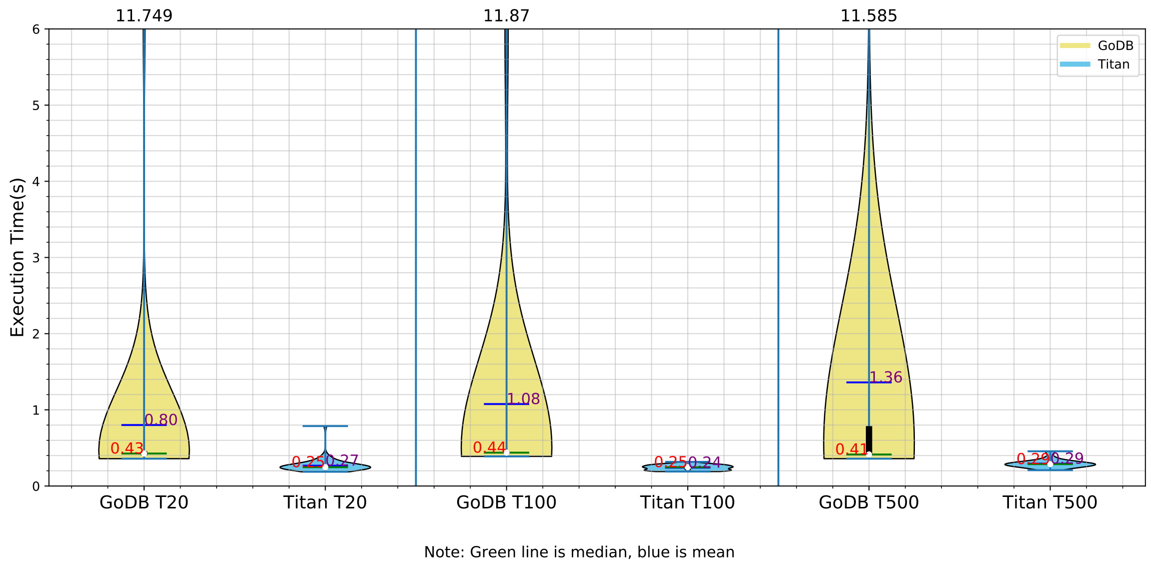


Figure 3.24: *VE Query* over *indexed* database.

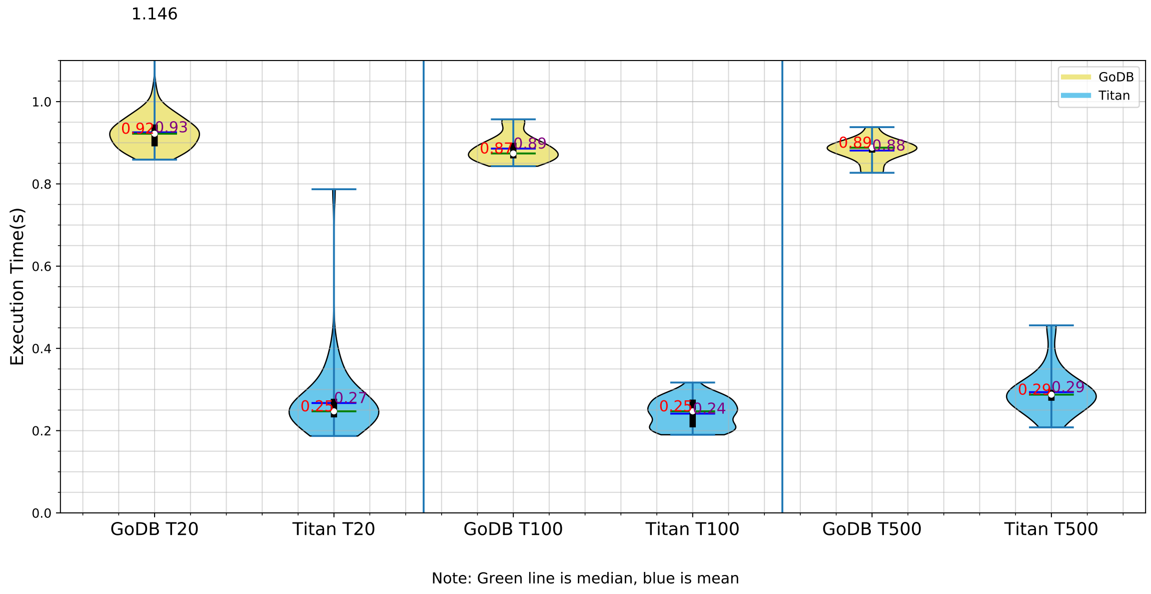


Figure 3.25: *VE Query* over *NonIndexed* GoDB and *indexed* Titan

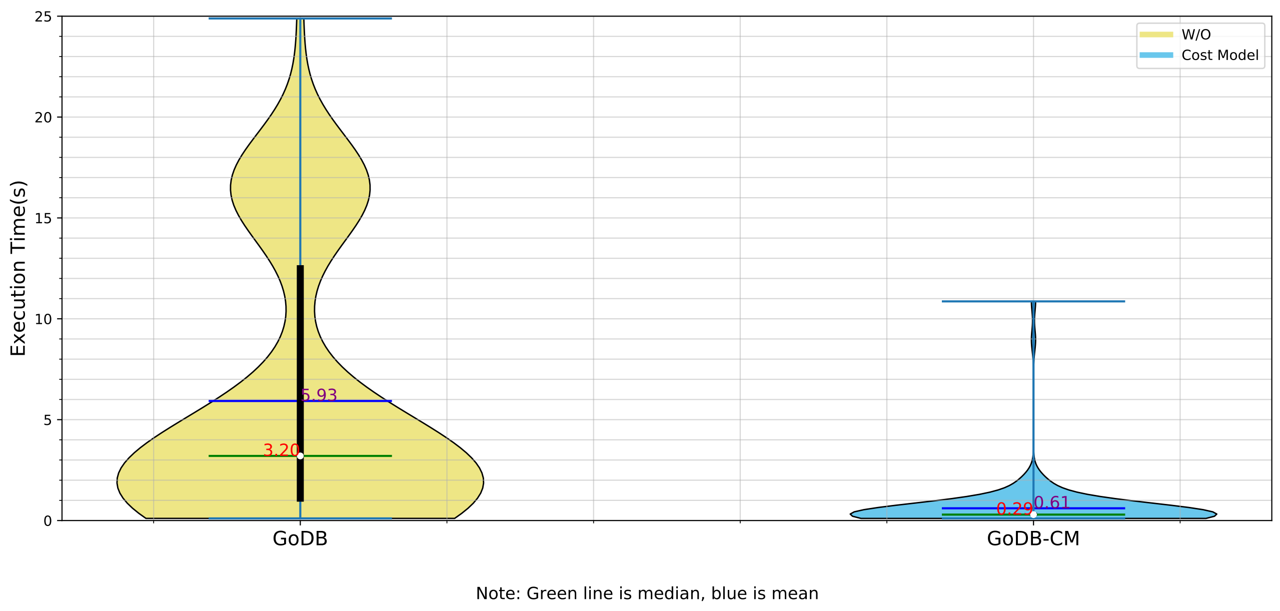


Figure 3.26: Cost Model Benefits for *Select Query* of *Length 5* having 16 Partition in 8 Machines

3.3.3 Results for GOGP

3.3.3.1 Experimental Setup

We also evaluate GoDB for a larger graph, Google Plus (GOOP), having 28 Million Vertices and 635 Million Edges (undirected). The experiments were run on a cluster having 24 nodes, each having AMD Opteron 6376 CPU with 8 cores and 32GB of RAM. GoFFish v3 on Hama v1.2 was deployed on the cluster with the number of partitions per machine varied according to the configuration specified in the plots. Here, we provide some of the experiment results and defer a more detailed examination to the next chapter where we compare it against both GoDBX and Titan.

3.3.3.2 Synthetic Query Generation

Generation of the workload requires sampling paths from the graph which is performed by running a BFS from 1000 random vertices. The depth of the BFS traversal is set to 5. Path queries are created from these sampled paths. A path query is formed from every path that satisfies the selectivity category constraints for every vertex predicate in the path. For Select Queries, at every step the vertex predicate, the selectivity was required to be less than 2000, while for Report queries, the selectivity was between 2000 and 10,000. This forms a set of candidate workload for the select and report queries. To create a workload of n queries, we choose n queries at random from candidate workload.

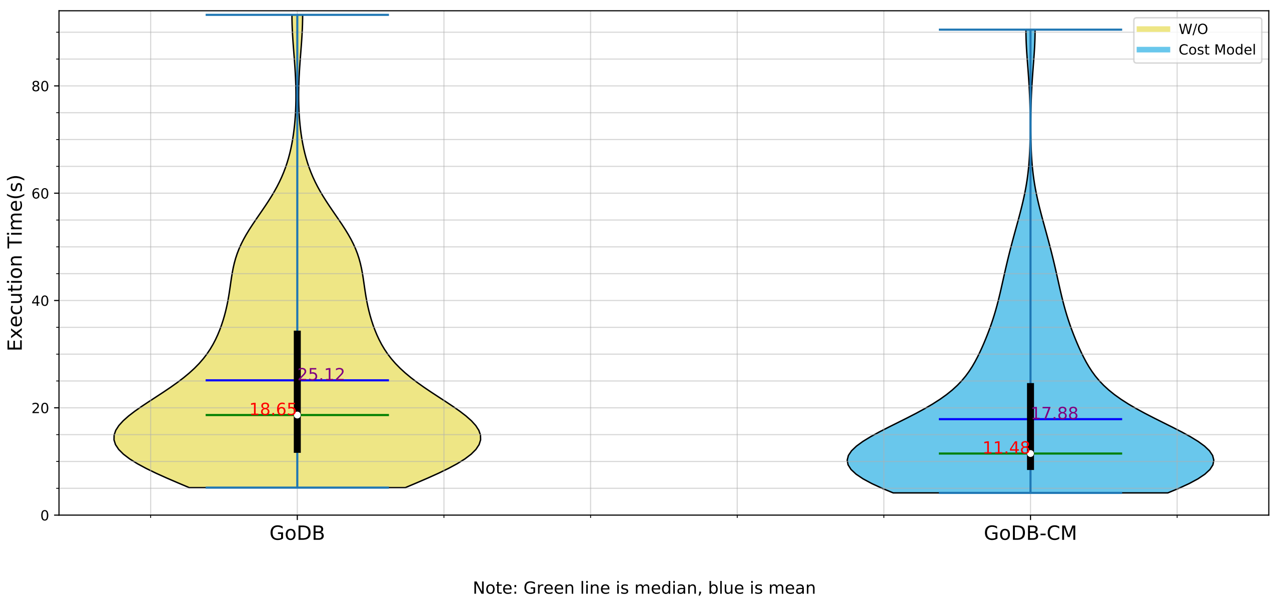


Figure 3.27: Cost Model Benefits for *Report Query* of *Length 5* having 16 Partition in 8 Machines

Figure 3.26 3.27 depicts the benefits of the GoDB cost model for Select and Report queries respectively run on GOOP. There is approximately a 10x reduction in the execution time for Select query while there is 1.4x reduction in execution time for the Report query.

Table 3.2 shows the reason why the cost model offers a faster performance. It provides the average number of vertex predicate scans for Select and Report query workloads. It is observed that average number of predicate scans drop sharply from *62,083 to 1,525* for Select queries while it drops marginally from *628,137 to 540,102* for Report queries.

Table 3.2: Average Predicate Computations

		Selectivity
Select	without Cost Model	62,083
	with Cost Model	1,525
Report	without Cost Model	628,137
	with Cost Model	540,102

Chapter 4

Querying over Compressed Property Graphs using GoDBX

GoDB uses Java-based property graph objects that are materialization in distributed memory for the entire graph before query execution starts. The graph entities present for each partition on a machine are the list of subgraphs in that partition, and for each subgraph, the list of vertices, and the properties and adjacency list for each vertex. While this gives fast access times to both the graph topology and the properties, it takes up a lot of memory. These Java objects are bloated due to class wrappers for simple types (`Long class` for an 8 byte long value), generics/template classes that are used, and the poor performance of string data types in Java – which property names and values use.

We address this memory limitation present in GoDB in this chapter. Specifically, we propose GoDBX that uses compressed data structures for memory-efficient storage of the graph entities, and with support for the same declarative path queries as GoDB over large property graphs. GoDBX retains the subgraph-centric query planning and execution model of GoDB [24], along with a mapping of the property subgraph data structure to the compressed data structures offered by Succinct [5].

This manages to sharply reduce the memory used by uncompressed Java objects in GoDB along with its Lucene index, while allowing more of a graph to be loaded in a single machine's memory. This offers an interesting trade-off between the number of machines used to keep the property graph in memory, the reduced communication and coordination costs for distributed execution, and the increased time used to query and extract data from the compressed model. We will discuss how succinct is used for compressing the graph structure and attributes in Data Model.

4.1 Background and Related Work

4.1.1 Succinct

Succinct [5] is a data store to efficiently store text data in a compressed form, in-memory. It supports character-based datasets which may be columnar in nature or just plain text files. It allows lookups that match substrings in the data that are executed efficiently using *suffix arrays* to return the offset(s) of the match. It further enables a subset of the data at given offsets to be extracted using *inverse suffix arrays*.

Succinct also uses sampling (a form of compression) to reduce the storage overhead below the $\mathcal{O}(n.log(n))$ space complexity used by these suffix arrays, for an input file with n characters. Yet another structure us used to compute and retrieve the unsampled values at given offset locations. Overall, for a sampling rate of α, Succinct's storage requirement are roughly $2n.log(n)/\alpha$ [5]. A (costly) pre-processing step is necessary to generate these data structures and is performed once, offline. In this sense, it offers the lookup efficiency of indexes while also allowing the actual source data to be retained for extraction in a compressed form, though the latter is costlier due to the "decompression" required. For more details on Succinct Store, we refer readers to [5].

Search and extract are two functions that are used over a Succinct data structure.

- Search: This allows for searching a string or wildcard within a compressed file and returns a list of offsets/records for those expressions in the file.

- Extracts: Upon receiving the list of offsets, each of these offset could be used to extract some part of text file and perform computation without decompressing the whole file.

More specifically, we use the indexed flat-file data object that is exposed by Succinct. Two queries from its API that we leverage for GoDB/X are *search* using the `getRecordID(val)` method that returns the index of a recordID which contains the string "val"; and *extract* using the `extractUntil(offset, delimiter)` method that returns the characters from an arbitrary offset until the first occurrence of delimiter.

4.1.2 Related Work

ZipG [26] is an existing graph store by the authors of Succinct that makes use of the capabilities of Succinct as well. Our graph data mapping that is represented using Succinct is inspired by mappings similar to those used by ZipG. However, the query model supported by ZipG is more restricted than GoDB, and their functionality is limited to the capabilities required by Facebook's TAO system [12]. We attempt to generalize such queries and combine the query capabilities of GoDB with the reduced memory footprint offered by Succinct.

FERRARI [7] which builds upon well-known node interval labeling scheme, is a scalable and efficient reachability index structure. This allows the user to trade-off the query processing and space by specifying restrictions on the index size. Two variants of FERRARI Index allow the specification of the maximum vertex labels size, or a global maximum size. FERRARI assigns a mix of exact and randomized identifier ranges to nodes to speed up both random and reachability queries. GoDB/X uses Succinct's sampling rate to allows users to make a similar trade-off.

Graph data structures like Compressed Sparse Row (CSR) are popular in graph stores. CSR has two arrays for a graph with the vertex/edge offset, and the edge list. The former holds the offset index in the edge array for a give vertex, and the edge array holds contiguous entries for the adjacent vertices. This structure provide memory locality among the edges of a vertex and can make traversals faster. Extensions to this CSR format to support properties and optimized for power-law graphs has also been proposed [23].

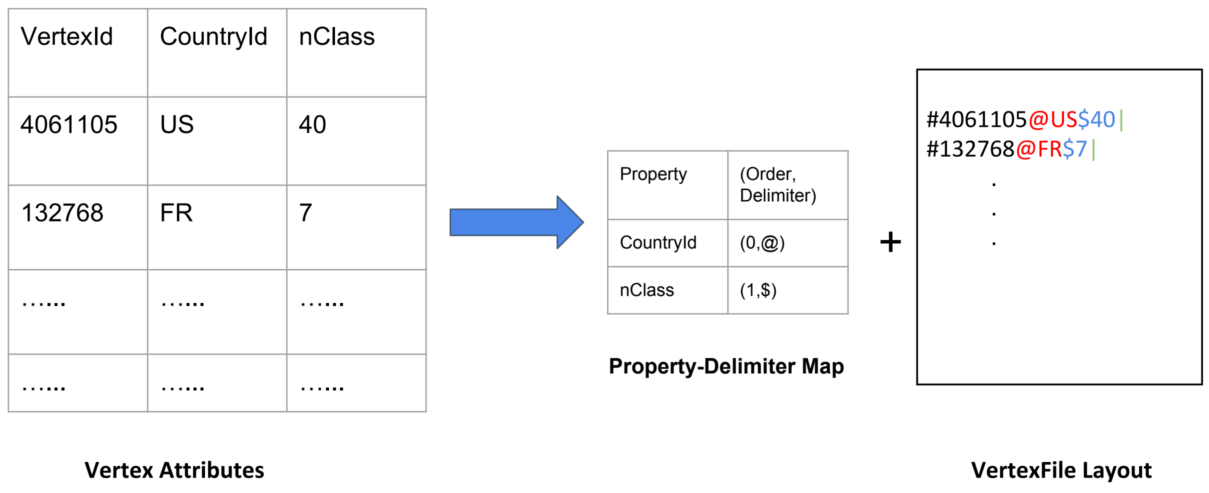
4.2 GoDBX Data Model

A transparent way of leveraging the existing GoDB platform is to implement the `Subgraph` Java object API that is used by the query engine into an implementation based on Succinct. As a result, there is minimal change required to the distributed query engine, and it allows for a direct comparison of GoDB with GoDB/X. We make use of Succinct's *indexed flat-file* representation that takes in a rows of text present in a file as input, and creates compressed in-memory data structures for each. next, we describe how we map the Subgraph API to such indexed flat-files.

4.2.1 VertexFile

Vertices in a property graph are identified uniquely by a `VertexID` and have associated name-value pairs as properties. The *VertexFile* data structure stores the VertexIDs and the set of property pairs in a single line of string that is mapped to a Succinct indexed flat-file. We use an approach similar to ZipG to model the structure [26]. Since there are just a few distinct property names in the graph, rather than store the entire property name for each vertex/edge, they map each property to a distinct non-printable character that is maintained in a hash table. The VertexFile itself only stores the vertexID, and a series of unique property name character and the property value pairs, with another special character used as line termination.

Figure 4.3 gives the pictorial representation of VertexFile. This depicts how properties are represented as an input text file that is translated to the our Succinct schema an stored as a VertexFile. Accessing of VertexFile requires first searching the vertex id on succinct files, then

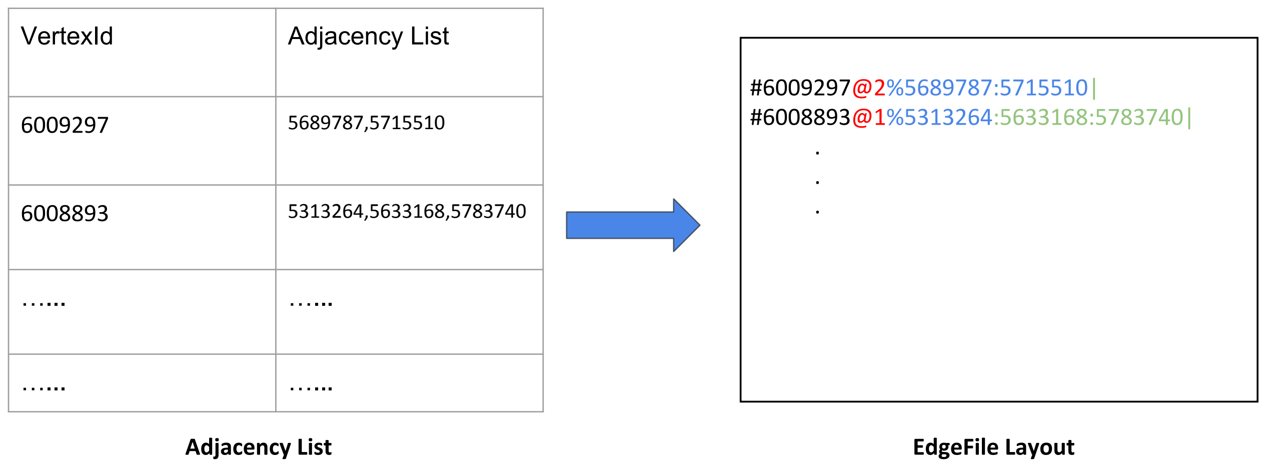


VertexId	CountryId	nClass
4061105	US	40
132768	FR	7
……	……	……
……	……	……

Property	(Order, Delimiter)
CountryId	(0,@)
nClass	(1,$)

```
#4061105@US$40|
#132768@FR$7|
    .
    .
    .
```

Figure 4.1: VertexFile Layout for vertex id and its properties for Patent Citation Graph[1]



VertexId	Adjacency List
6009297	5689787,5715510
6008893	5313264,5633168,5783740
……	……
……	……

```
#6009297@2%5689787:5715510|
#6008893@1%5313264:5633168:5783740|
    .
    .
    .
```

Figure 4.2: EdgeFile Layout for vertex id and its Adjacency List for Patent Citation Graph[1]

extracting the properties which returns delimiter separated properties, splitting the property string it based on the delimiters, and returning appropriate property value. So considering Figure 4.3, accessing a vertex property requires searching for #*vid*@ in the vertex data to locate that row record uniquely, and extracting the property needed from record obtained using the delimiters.

Specifically, we make use of two types of operations on top of this VertexFile to support the Subgraph API: (1) given a VertexID, extract the corresponding set of property names and values, and (2) given a property name and a value, return all VertexIDs which contain the given property. These can be mapped to a combination of search and extract operations performed on the VertexFile Succinct data structure, along with a mapping table maintained from the property name to the delimiter to help construct the search query.

E.g., if a vertex predicate is specified as *country=US*, then we first finds the delimiter associated with the property name *country*, namely @, and the delimiter for the next property name in the property name list, which is $. We then wrap the given property value with these two delimiters to get the string @*US*$, which we then *search* for as a substring in the VertexFile. The returned offset to that row can then be used to *extract* the VertexID from the VertexFile.

In addition to these, we also store an in-memory table that maps VertexIDs to SubgraphIDs.

4.2.2 EdgeFile

The EdgeFile is a Succinct data structure that stores the adjacency list for each vertex in each row. Layout of EdgeFile is shown in Figure 4.2. While there is some similarity to ZipG [26], here we also need to consider two types of out edges from a vertex – *local edges* that connect the vertex to other vertices within the subgraph, and *remote edges* that connect to vertices present in different subgraphs on a remote machine. Each edge record row contains a 3-tuple having the source VertexID, the number of local sink VertexIDs, and the list of local followed by remote sink VertexIDs as shown in figure 4.3 labeled as *EdgeFile*. The local vertex count (called as *lvcount* in Figure 4.3) allows us to distinguish the boundary between local and remote vertex IDs in the adjacency list. As before, these properties are delimited by unique non-printable characters.

The main operation supported by the EdgeFile is given the source vertexID, return all the local and the remote sink vertex IDs. Performing this for a vertex requires *searching* for #*vid*@ in the EdgeFile and *extracting* the local and remote neighbors. While we currently have not incorporated edge properties into this data structure, it can be extended similar to the VertexFile approach.

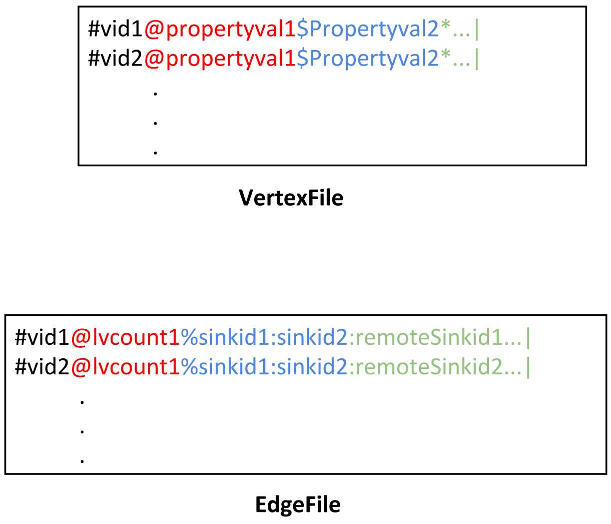

#vid1@propertyval1$Propertyval2*...|
#vid2@propertyval1$Propertyval2*...|
VertexFile
#vid1@lvcount1%sinkid1:sinkid2:remoteSinkid1...|
#vid2@lvcount1%sinkid1:sinkid2:remoteSinkid2...|
EdgeFile

Figure 4.3: GoDBX Succinct Schema V1, Unified Model

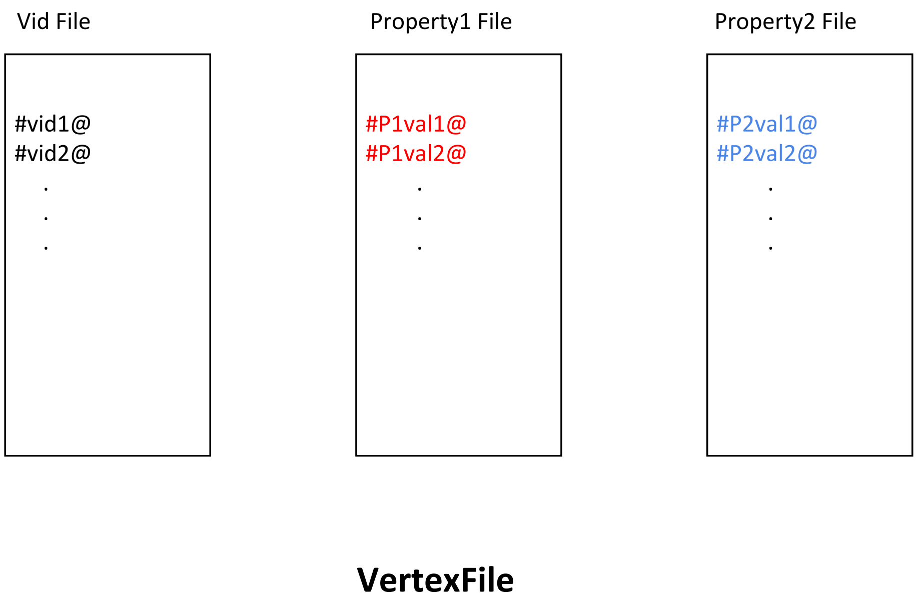


Figure 4.4: GoDBX Succinct Schema V2, Split Property Model

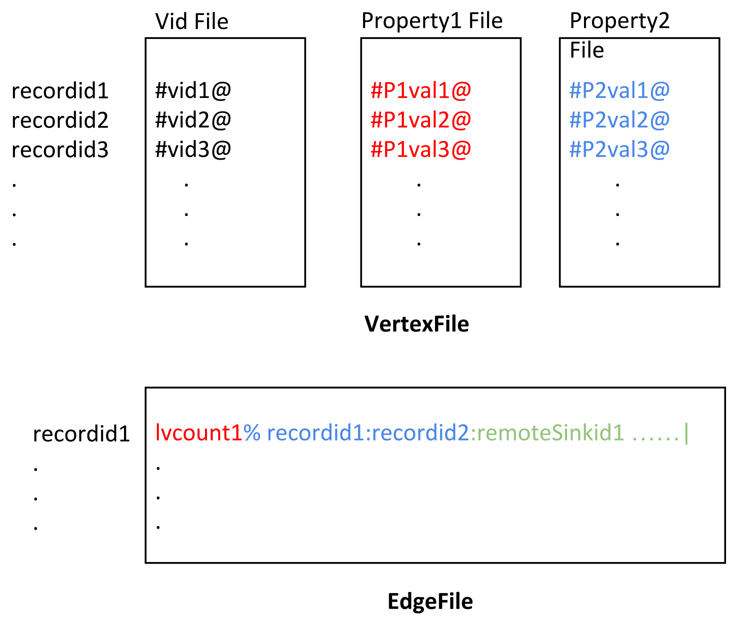


Figure 4.5: GoDBX Succinct Schema V3, Implicit Model

4.2.3 Alternative Models

Fig. 4.2 shows base Succinct layout that we use for the *GoDBX Data Model*, which we term the unified model or V1. There are two other data model variants that we propose, which help reduce the computation performed during traversals:

- Split Property Model (V2), Fig. 4.4
- Implicit Model (V3), Fig. 4.5

The operations supported by both these models are the same as described earlier and the key difference is in the representation. Figure 4.4 gives the pictorial representation for split property model (V2) where properties are stored as separate Succinct files from the vertex list. A separate vertex id Succinct file has the same vertex ordering as the corresponding properties in the property file. Thus, the record number for a particular vertex ID is the same as the record number for its property in the corresponding property succinct file. The key benefit of this is that it reduces the overhead of extracting all the properties from V1, and instead we limit ourselves to accessing only the property file of interest and extracting just a single value.

For instance, in Figure 4.4, finding Property2 for a particular vertex will require searching for *#vid@* in the *vid File* which returns a record id. Searching for this record id in the *Property*

2 file will the return value of Property 2 for this vertex. The Edge Data File remains the same as in the Unified model, V1. Thus, using this columnar based storage approach reduces the number of bytes extracted from the file, and reduces the extraction time.

Figure 4.5 gives pictorial representation of the Implicit model (V3). Here, we replace the local sink vertex IDs in the adjacency list with corresponding record IDs for those vertices in the succinct file. We also avoid storing the source vertex id since we align the adjacency list record ID with the vertex ID file's record ID for that vertex, as in V1. During traversals, extracting the adjacency list of a vertex returns the local sink record ids and the remote sink vertex ids. The local sink record ids is then directly used to extract the properties from the property files, and the adjacency list from the edge file, which saves us the cost of searching for the record id again in *vid file*. During traversals, this cost saved for all access of properties and adjacency list, thus reducing compute time. This V3 approach however requires preprocessing of VertexFile and EdgeFile to allow us to get the record id of a vertex for replacing the local sink vertex ID with.

4.3 Implementation

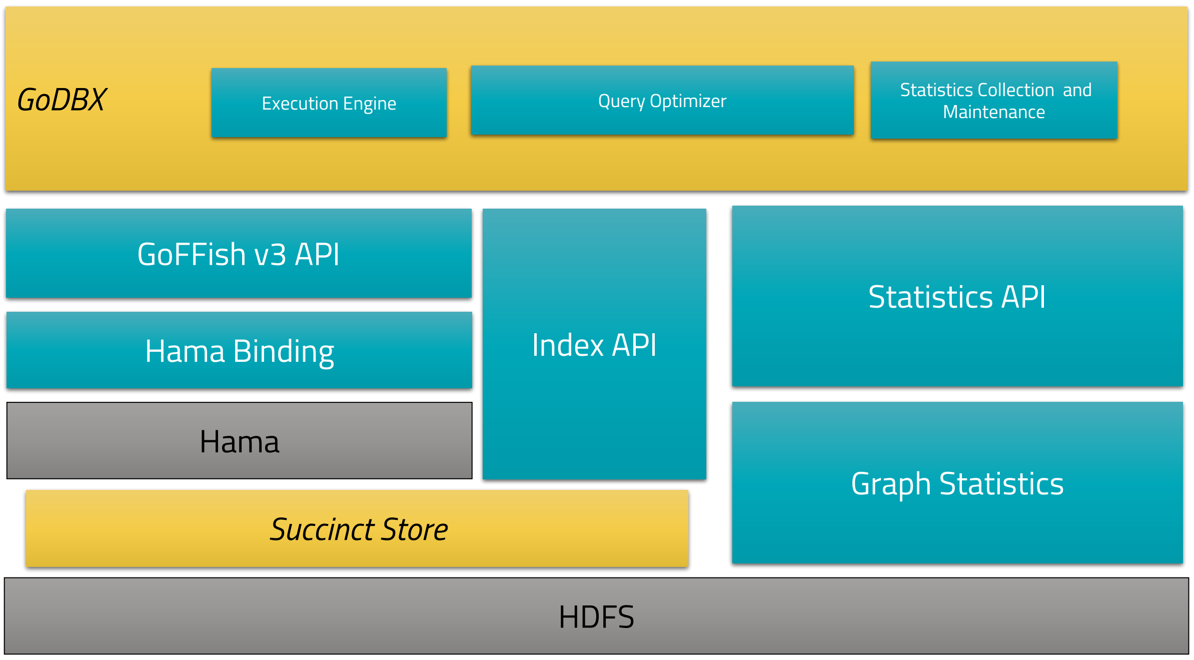


Figure 4.6: GoDBX Stack

Figure 4.6 gives a pictorial representation of the GoDBX stack. It is similar to GoDB's albeit with a few subtle differences. We primarily modify the implementation of the Subgraph API of GoDB that uses Java objects for storing the property graph, and replace them with the succinct files/objects that use the compressed data model. The Subgraph API along with the Lucene index expose three main methods that are used by the GoDB query execution engine: (1) return the vertex ID that matches a given vertex property predicate, (2) return the local adjacency list vertex IDs for a given source vertex ID, and (3) return the remote adjacency list vertex IDs for a given source vertex ID. These methods are implemented using operations on the VertexFile and EdgeFile data structures, as we have discussed above.

One additional challenge that we address is the limitation of Succinct in using an array internally to store its data structure contents. Arrays in Java are limited to having only 32 *bit* integer number of items, which limits us to storing only 2^{32} bytes of data in a single indexed file. This translates to about 2 GB of text input file, which is much smaller than the sizes of the VertexFile and EdgeFile for large graphs. We address this constraint by introducing an abstraction layer inside the Subgraph object that transparently wraps multiple Succinct indexed flat file objects as a single object and exposes search and extract operations that are automatically performed over these. We use this to store the VertexFile and EdgeFile as a collection of Succinct objects.

4.4 Experiments

4.4.1 Dataset

GITR Graph is a large synthetic graph which is created using *DataSynth Graph Generator* [39]. GITR graph represents a graph of Git repositories, and contains 170 Million vertices and 2.4 Billion Edges. The number of vertex properties is 4. The edge degree distribution parameter of Data Synth graph generator is set to follow the distribution of DBLP Collaboration Network [2] In addition, we also use the GooglePlus property graph having 28 Million Vertices and 635 Million Edges (undirected), as discussed in the previous chapter.

4.4.2 Query Workload Generation

Generation of the workload requires extracting paths from the graph, similar to the approach used for GoDB's workloads in the previous chapter. We perform a breadth first search from 1000 random vertices. The depth of the BFS traversal is set to 5. Path queries are created from the paths extracted from the BFS tree. As before, a path query is formed from every path that satisfy selectivity constraints for every vertex predicate in the path. For Select Queries, at every step the vertex predicate selectivity was threshold to be less than 2000, while for Report queries, the selectivity was between 2000 and 10000. This forms a set of candidate workload of Select and Report queries. To create workload of n queries, we choose n queries at random from candidate workload.

For our evaluations of GOGP, 100 Select and 61 Report queries of length 5 are used, while for GITR, 350 Select and 150 Report queries of length 4 are used.

4.4.3 Experimental Setup

The experiments were run on a cluster having 24 nodes each having 32GB of RAM, AMD Opteron(tm) Processor 6376 CPU. Goffish v3 on Hama v1.2 was deployed on cluster, with the number of partitions per machine varied according to the configuration specified in plots.

Succinct 0.1.8 was used for compressing the VertexData and EdgeData files. We use the indexed flat-file data object that is exposed by Succinct. Two methods from its API that we leverage for GoDBX are *search* using the `getRecordID(val)` method that returns the index of a recordID which contains the string "val"; and *extract* using the `extractUntil(offset, delimiter)` method that returns the characters from an arbitrary offset until the first occurrence of delimiter.

Titan v1.1 was used for comparison against GoDBX, deployed as per the setup in the GoDB chapter and with local configuration as specified in the plots. Cassandra 2.0.15 and Elastic Search 1.4.2 is used for backend storage and indexing, respectively. Titan is configured to use *pull based* or OLTP execution model. In OLTP execution model, the traversal is done in the single Titan server machine. After getting the starting vertices and creating *traversers*, the traversals starts by pulling the relevant edges from the Cassandra servers into this Titan server. These relevant edges may be *out-edges* or *in-edges* based on specification of the query. This could further initiate pulling of vertices. This pulling may go on for arbitrary number of steps based on the query.

4.4.4 Microbenchmarks

One of the configuration parameters for the Succinct data structures is the sampling rate α. Fig. 4.7 compares the space-time tradeoff of different α values. The file size on disk for the GITR VertexFile using V1 model on which this experiment was performed was 5.4 GB. As expected, the data size will increase as the sampling rate decreases, as seen along the right Y axis. E.g., it takes $\approx 8.5\ GB$ to store the data in-memory using $\alpha = 2$ and this drops to $\approx 2\ GB$ with $\alpha = 16$. At the same time, the Latency Time (μsec) to extract bytes from the compressed store is proportional to the sampling rate, and also linearly increases with the number of bytes extracted. E.g., it takes $\approx 70 \mu sec$ to extract 64 *bytes* of data with $\alpha = 128$ while this reduces

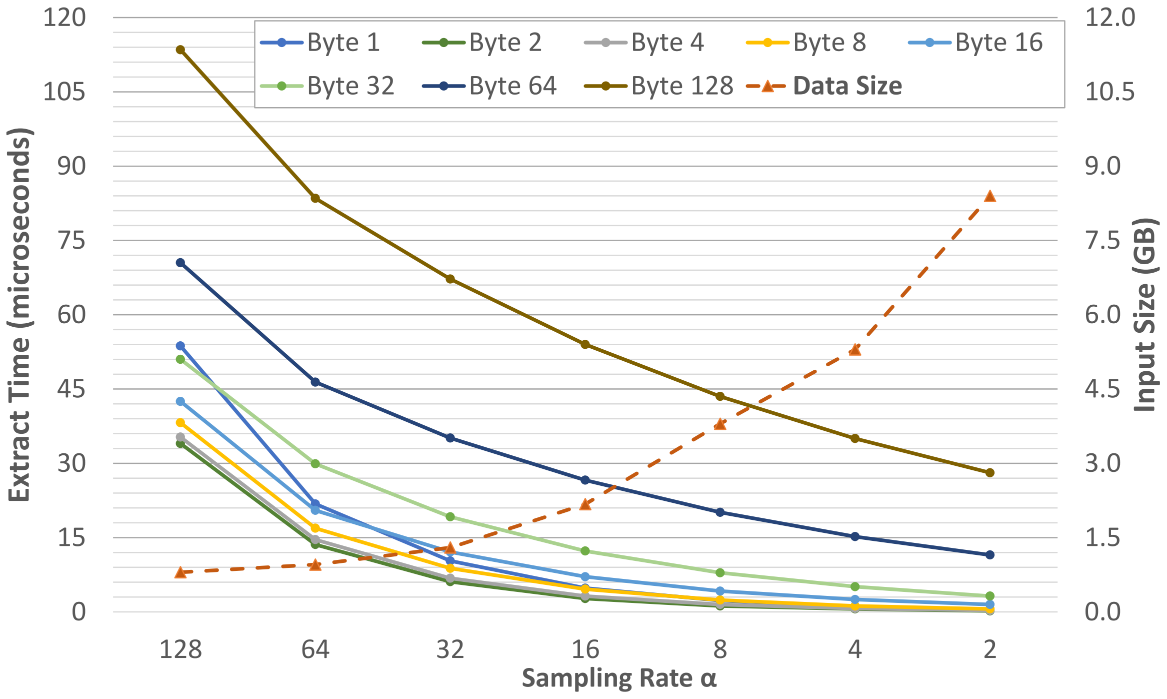


Figure 4.7: Space-Time trade-off of Succinct for different α values. The data size (right Y axis) and time to extract different numbers of bytes (left Y axis) for GITR VertexFile are shown.

to $\approx 1\mu sec$ with $\alpha = 2$. Based on these observations, we adopt $\alpha = 32$ in our experiments.

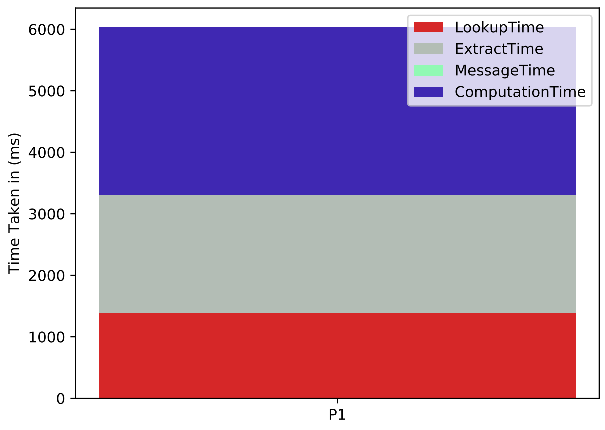
4.4.5 Results for V1 Model on GITR

This subsection discusses preliminary experiments for the *Path Queries* that help corroborate the three alternative models, V1–V3, that we adopt. These motivate the need for the V2 and V3 data models. We limit ourselves to a configuration where the entire GITR graph is on a single partition and machine, to scope this analysis to the impact of the data models.

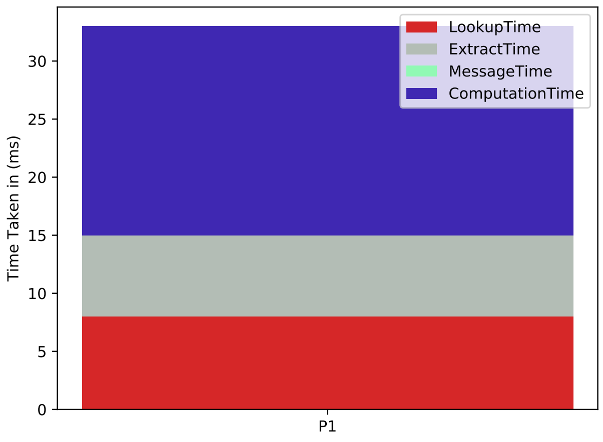
Figure 4.8 depicts stack plot of time taken by computation, succinct lookups and extracts during the execution of Select and Report queries. *It is observed that Extracts and Lookups take substantial time of the overall compute while Computation dominates query execution time.*

Figures 4.9 and 4.10 show a stack plot of Time and Counts for each type of lookups respectively. Lookups are performed during traversals are:

- Index querying (called as IndexLookup)
- Vertex property lookup (called as VertexLookup)
- Adjacency list (called as EdgeLookup)

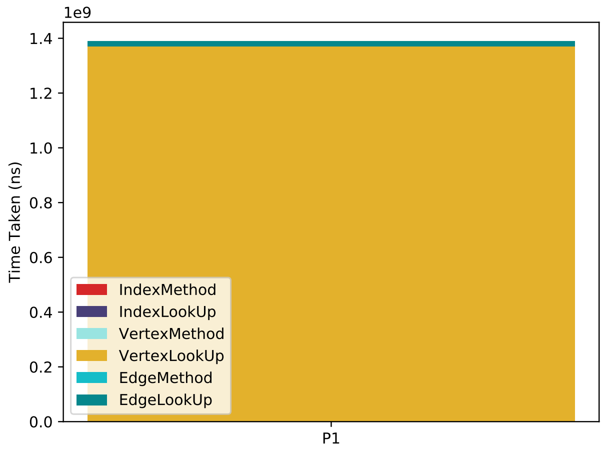


(a) Report, 1P no NW

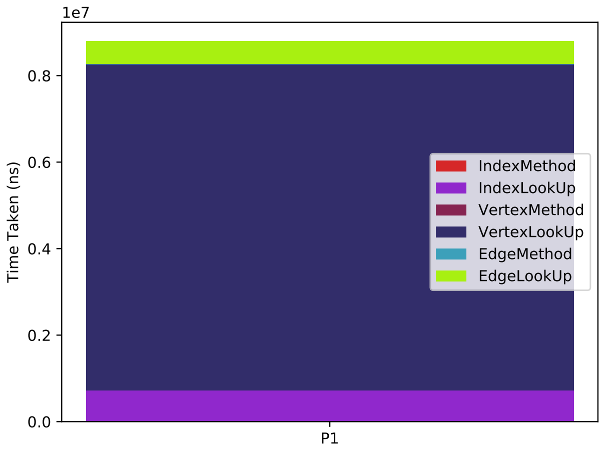


(b) Select, 1P no NW

Figure 4.8: Stacked plot for time taken by SELECT and REPORT Queries



(a) Report, 1P



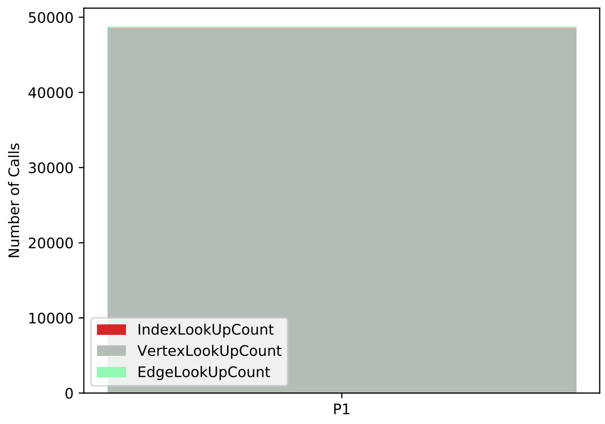
(b) Select, 1P

Figure 4.9: Stacked plot for time taken by Lookup

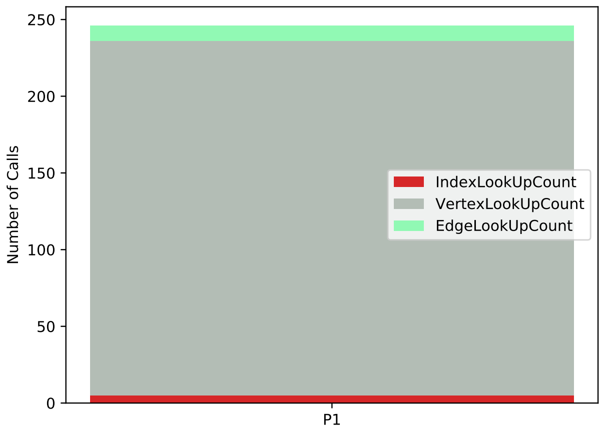
It is observed that Vertex lookup that searches properties of a vertex dominates all other types of lookups.

Figures 4.11 depicts the time taken against the number of bytes extracted for different type of extractions. There are three different types of extractions:

- Index extraction (called as IndexFile),
- Vertex property extractions (called as VertexFile),
- Adjacency list extractions (called as EdgeFile).



(a) Report, 1P



(b) Select, 1P

Figure 4.10: Stacked plot for count of of calls made to Lookup method

It is observed that the extraction time shows linearity with the amount of bytes extracted. Adjacency in EdgeFile has large variability in number of bytes as outdegree of vertex vary. Due to this, some edge extractions take larger time while index and vertex extractions have relatively smaller number of bytes extracted.

Figure 4.12 shows the correlation between Lookup Time(of all types) against File Size of the Indexed Flat-file of Succinct. *It is observed that there is no correlation between Lookup Time and File Size.*

Table 4.1 presents the Cumulative Time(Top table),Count(Middle Table) and Average Time per method(Bottom table) for different methods:

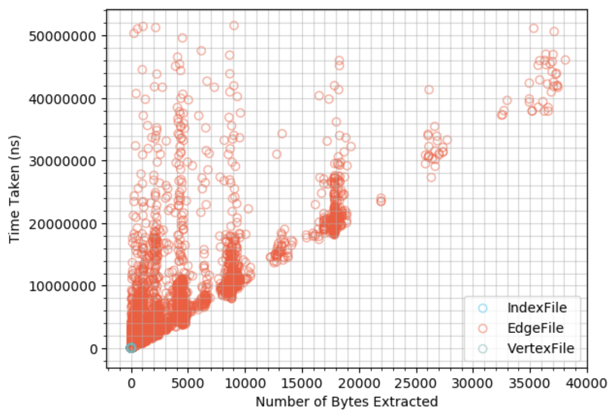
- GetVerticesByProp: This function performs index lookup of given predicate and extracts

Table 4.1: Cummulative Time per Method (ms) [Top], Count per Method [Middle] and Average time per Method call [Bottom]

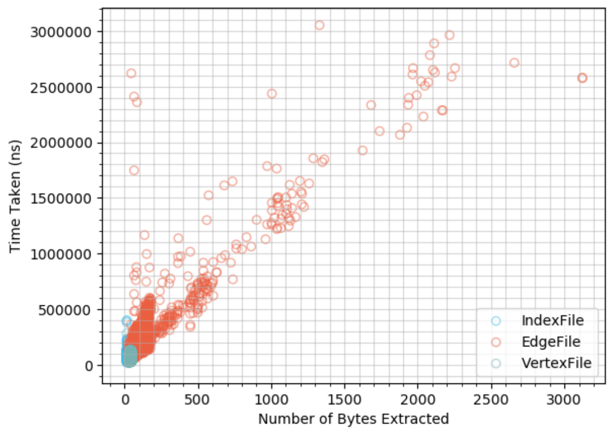
	GetVerticesByProp	GetPropForVertex	GetEdges
Select 1P	190	2,512	422
Report 1P	187	187,195	41,077

	GetVerticesByProp	GetPropForVertex	GetEdges
Select 1P	1,180	60,783	2,607
Report 1P	1,348	532,014	19,533

	GetVerticesByProp	GetPropForVertex	GetEdges
Select 1P	0.1610	0.04132	0.1618
Report 1P	0.13876	0.3518	2.1029



(a) Report, 1P w/o W



(b) Select, 1P w/o W

Figure 4.11: Scatter Plot for Extract Time vs Bytes extracted, colored by TYPE

vertex ids. This is added as part of Subgraph API in GoFFish. It takes vertex predicate as input and returns a list of vertex ids as output.

- GetPropForVertex: This function performs search and extraction of property for a vertex. This is added as part of Vertex API in GoFFish. This takes a property name as an input and returns property value for that vertex as output.

- GetEdges: This function performs search and extraction of adjacency List for a vertex. This is added as part of Vertex API in GoFFish. This returns a tuple of two list, one contains list of local sink ids while other contains list of remote sink ids.

It is observed that Property extraction takes most of the time because count of GetPropForVertex is high while count of GetEdges is low but edge extraction takes relatively higher time due to higher number of bytes being extracted.

To conclude, the optimization desired are:

- Reduce the number of bytes extracted.

- Reduce the number of lookups.

Based on this, we can conclude that the time taken is dominated by the extraction of the properties. Hence, our proposed *Split Property model (V2)* optimizes this by reducing the number of bytes extracted by creating separate vertexFile for each property while *Implicit model(V3)* further optimizes it by reducing the number of lookups by replacing local vertex IDs with record ID.

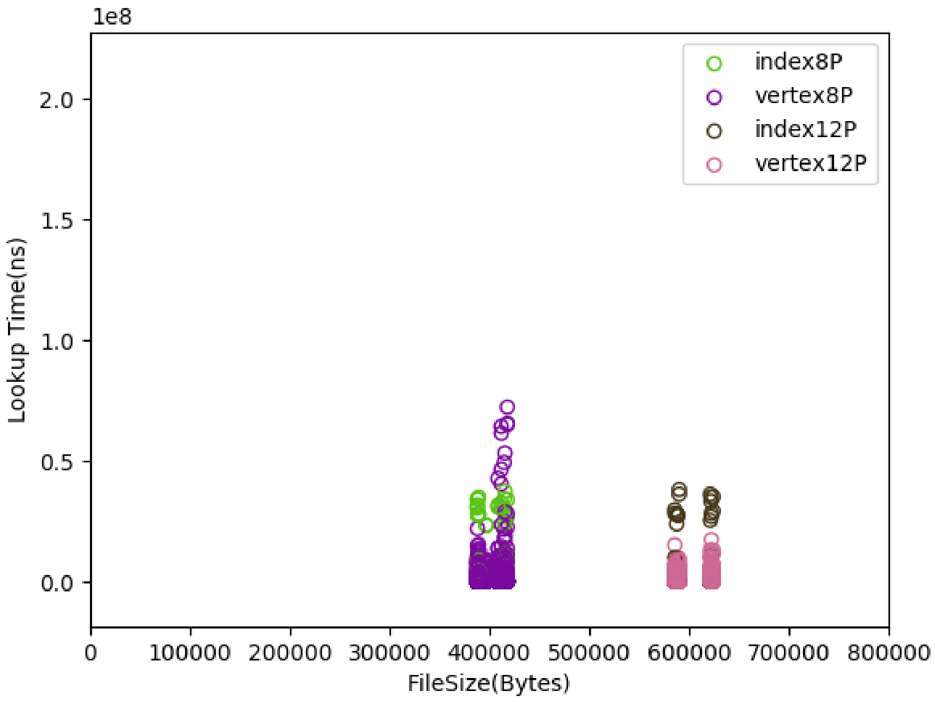


Figure 4.12: Lookup Time vs File Size

4.4.6 Results for all Models on GOGP

In the violin plots we present, the green line marks the median and the blue line marks the mean. There is black bold line between the first and third quartile within the violin. Each violin is labeled by combination of system and configuration. For example `GoDBX-V3-CM-2P1M` implies the `GoDBX` platform using the `V3` data model, with cost model `CM` enabled, and the graph partitioned into `2` partitions that are both placed on `1` machine.

Figure 4.13 shows the comparative evaluation of GoDB,GoDBX and Titan. It is seen that GoDBX is the fastest and Titan is the slowest of the three systems. GoDB and GoDBX leverages the subgraph centric execution model, whereas Titan pulls the vertex data into its single server machine for traversal. Note that in this case, we have just a single Cassandra node, and hence the Titan server is on the same machine as this Cassandra node. We see orders of magnitude difference in the execution time between Titan and our two GoDB and GoDBX systems.

GoDB is distributed across more machines due to memory constraints and thus requires more supersteps per query. GoDBX requires an average of 2.18 supersteps per query while GoDB requires 3.88 superstep per query. Select query being lightweight is dominated by syn-

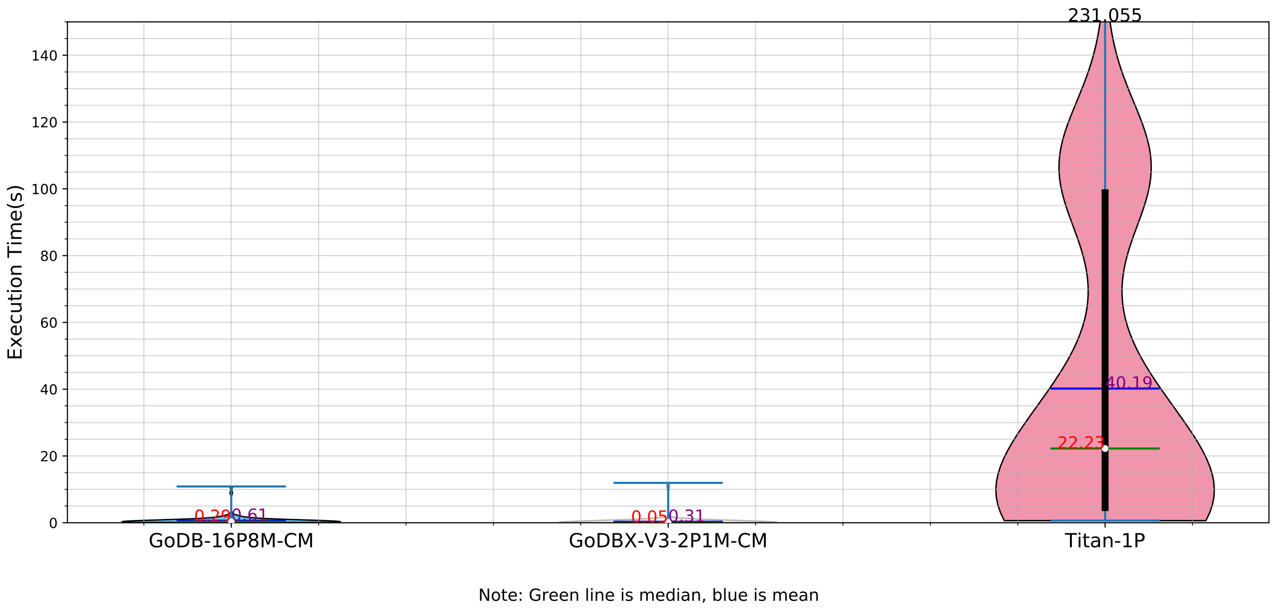


Figure 4.13: System Comparison of GoDB,GoDBX-V3,Titan for *Select Query* of *Length 5* - GoDB and GoDBX has 2 Partitions per machine with CostModel while Titan with OLTP is using 1 node

chronization time between supersteps, thus GoDB takes more time. Figure 4.14 shows a similar evaluation for report queries. Here it is observed that the GoDB is faster than GoDBX, while Titan gets worse. Report queries traverse a large number of vertices, with average of *497731* for the Report workload. It beneficial to use a higher number of machines which distributes the computation, thus achieving more parallelism for GoDB.

Figure 4.15 shows the benefits of different versions of the GoDBX data model as well as the additional benefits of the cost model. It is seen that there is benefits leveraged from data models V1 through to V3. When going from data model V1 to V2, the use of columnar storage of properties reduces the number of bytes extracted in vertex properties extraction, thereby reducing computation during traversals. There is further reduction in V3 as the lookup is skipped while querying succinct files for properties or edges. It is also seen that cost model reduces the number of vertex property extractions as well as edge traversals, thus reducing the execution time of the query.

Table 4.2 shows the average vertex property extractions and predicate computations in each partition for Select and Report query workload. It is observed that the cost model reduces the average number of vertex property extractions from *32744 to 782* and *26905 to 613* for two partitions respectively. There are few queries that traverse comparatively denser part of the graph, thus performing more extractions leading to higher latency which is apparent with

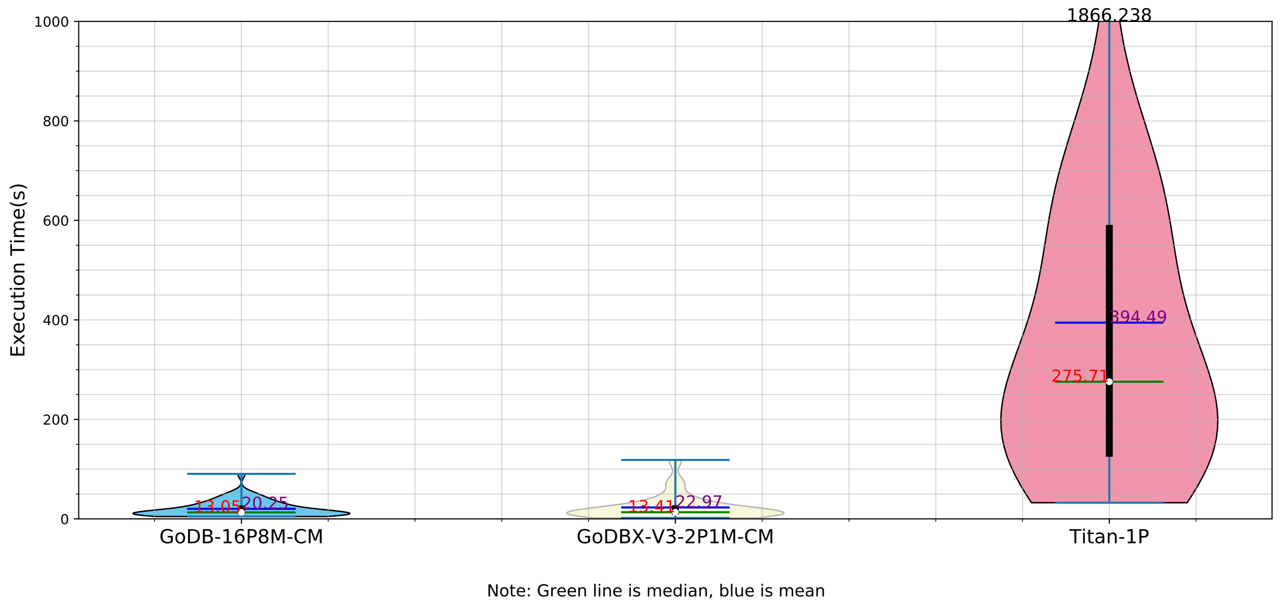


Figure 4.14: System Comparison of GoDB,GoDBX-V3,Titan for *Report Query* of *Length 5* - GoDB and GoDBX have 2 Partitions per machine with CostModel while Titan with OLTP is using 1 node

the long tails in the violin plots for $V1, V1 - CostModel, V2, V3$. For instance, long tail in V1,V2,V3 is due to a particular query traversing 134962 vertices.

Figure 4.16 shows similar benefits for Report query. There is no benefit from V1 to V2. Further there is no benefit by optimizing query using cost model for data model V1. For Report query, the long tails are due to higher number of predicate computations. For instance, the long tail in $V1, V2$ is due to a query traversing $2,477,170$ vertices. While it seen that the tail is shorter in $V2$ which is due to lower bytes extracted than in $V1$ Table 4.2 shows the benefits of cost model, where the average number of vertex property extractions is reduced from *526,456 to 497,731* and *183,863 to 133,632* for two partitions respectively.

Table 4.2: Vertex Selectivity of GoDBX2P

		Partition 0	Partition 1
Select	without Cost Model	32,744	26,905
	with Cost Model	782	613
Report	without Cost Model	524,656	183,863
	with Cost Model	497,731	133,632

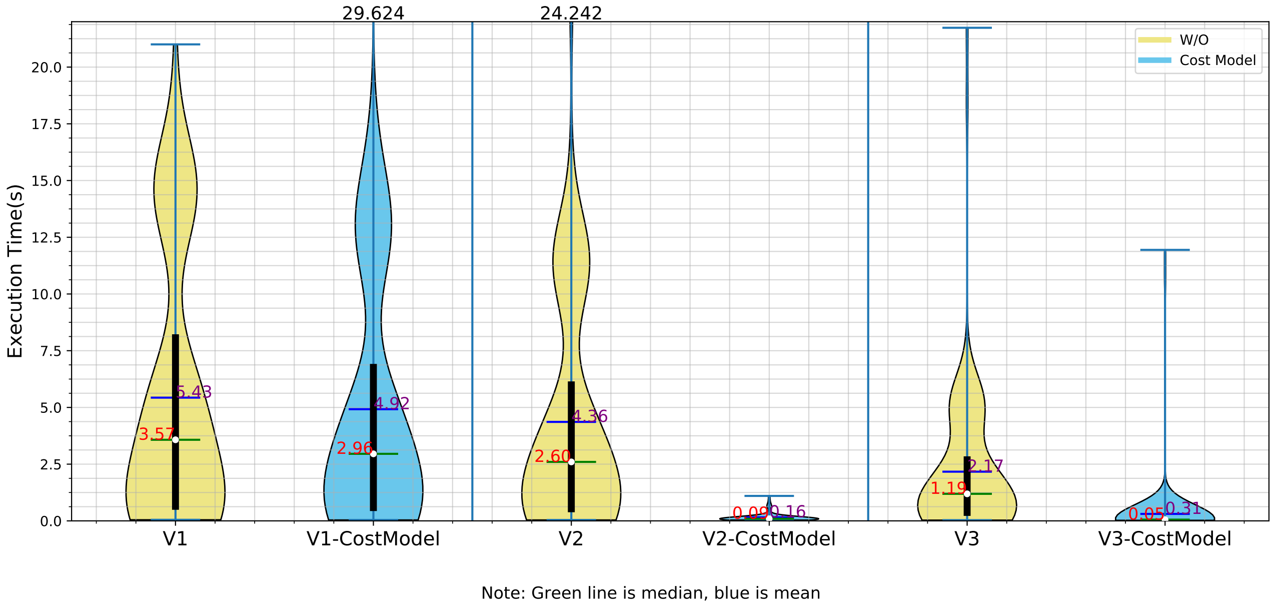


Figure 4.15: Data Model Benefits of GoDBX(V1,V2,V3) for *Select Query* of *Length 5* - without and with Cost Model, GoDBX with `2P1M` configuration

4.4.7 Scalability

Figure 4.17 and 4.18 show the scaling experiments of GoDBX for the GooglePlus graph for Select and Report query respectively. The same query workload is run across four different configurations which includes changing the number of partitions as well as the number of machines. For instance, `4P2M` signifies that the graph is partitioned into 4 parts, and executes on 2 machines (with 2 partitions each). Each partition uses 1 core, implying that increasing the number of partitions per machine will start using more cores, thus giving better performance. But there is limited memory on each machine and increasing the number of partition increases the number of remote edges. Messages can be sent to remote sink vertex, thus the mapping information of *vertex id* to *subgraph id* and *subgraph id* to *partition id* is kept. This mapping information increases the memory footprint of the framework and with our cluster configuration, memory gets exhausted before all cores can be used. In our experiments, we see that 2 partitions is the most that can be placed on a single machine for GoDB and GoDBX as beyond that, the runtime in-memory state when queries are executing overflows the physical memory.

It can be observed that initially there is an advantage for increasing the number of partitions from 1 to 2, but as we increase it further there is marginal decrease in execution time for both select and report queries. For select query the average execution time is 8.1 s for 1 partition, 5.4 s for 2 partitions, 3.8 s for 4 partitions, 3.6 ms for 8 partitions. This denotes that network

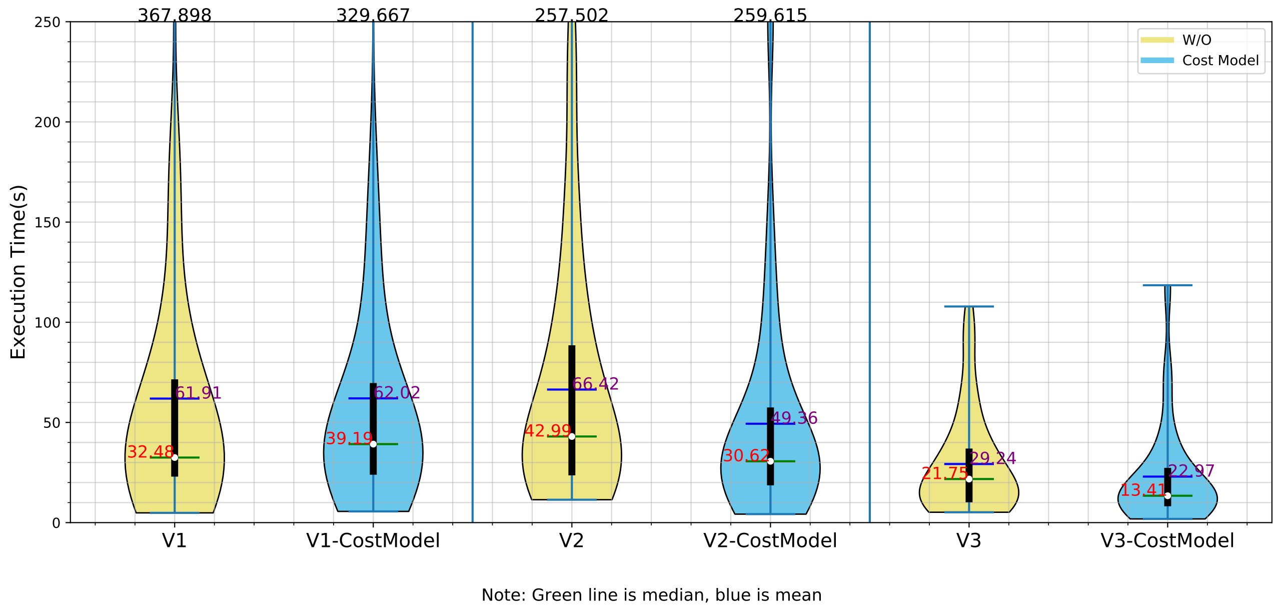


Figure 4.16: Data Model Benefits of GoDBX(V1,V2,V3) for *Report Query* of *Length 5* - without and with Cost Model, GoDBX with `2P1M` configuration

and synchronization overheads start to dominate the execution time, and thereby the execution time benefits start saturating. Similar observation can be made for Report query as it drops from 112.1 s to 61.9 s initial but starts saturating for 8P where it is 46.3 s. So we see a limited strong scaling for GoDBX.

Figures 4.19 and 4.20 show the scaling experiments for GoDB for Select and Report query respectively. For Select query it initially drops from 18.2 s to 5.9 s but there no benefit for 24 partitions as average execution time starts saturating at 5.7 s. Similar observations can be made for Report query as it drops from 74.6 s to 25.1 s and then starts saturating for 24 partitions as it is 23.8 s, thus providing no scaling benefits.

Figures 4.21 and 4.22 show scaling of Titan for Select and Report query respectively. Cassandra nodes is changed from 1,2,4. It is observed that for Select query, having 1 Cassandra node is the best configuration. While for Report query, there is marginal drop between 1,2 and 4 Cassandra nodes. Cassandra transfers data to the Titan server machine using Thrift which has frame size defaulted to 15 MB. Therefore using multiple nodes for lightweight *Select* queries increases the amount of unnecessary data brought in the client machine through Thrift packets, thereby increasing the query latency. The effect is the opposite for Report queries as it traverses larger part of the graph, and thus using multiple Cassandra nodes provides marginal improvements. It is expected to have negative benefits if the Cassandra nodes is increased further due to reasons specified for *Select*. So Titan has limited or no strong scaling either.

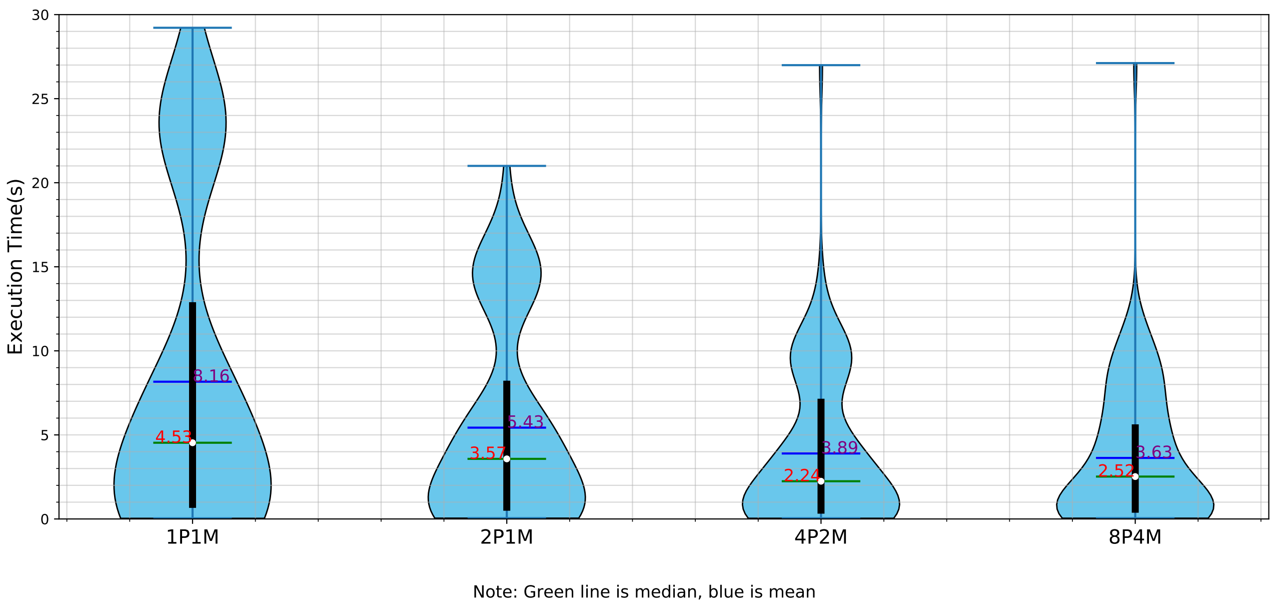


Figure 4.17: Scaling Benefits of GoDBX with Unified Model(V1) for *Select Query* of *Length 5*, without Cost Model

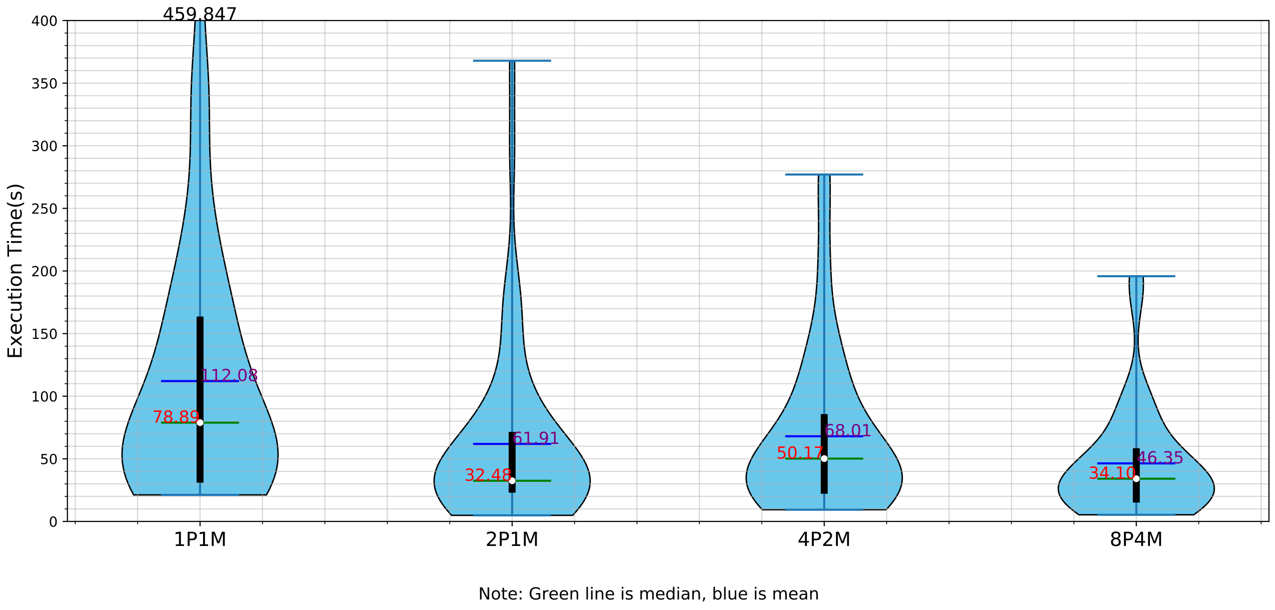
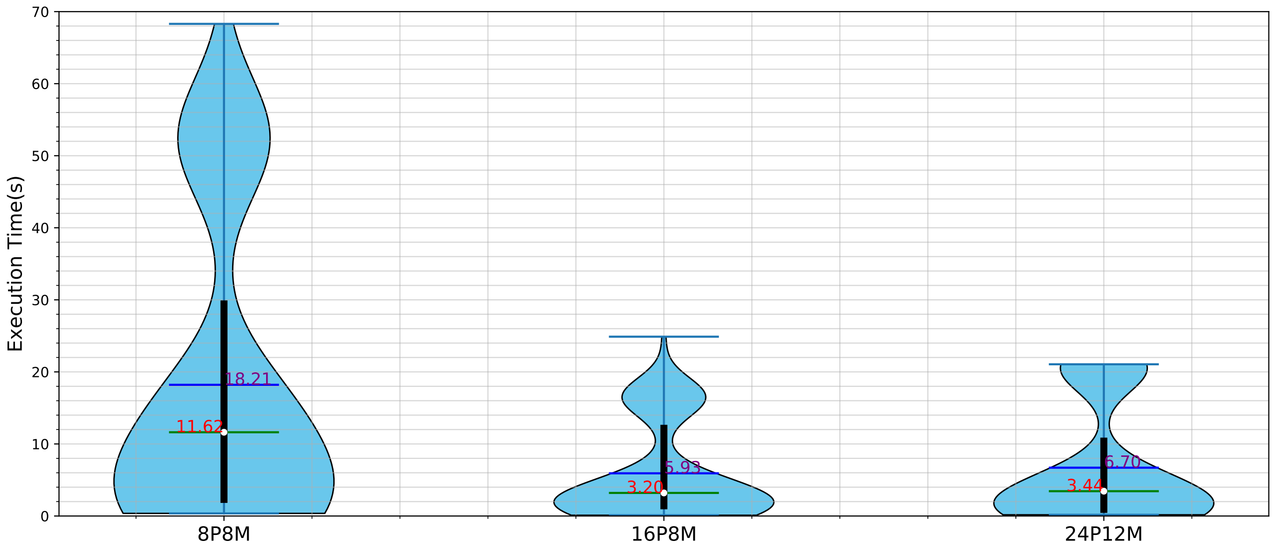
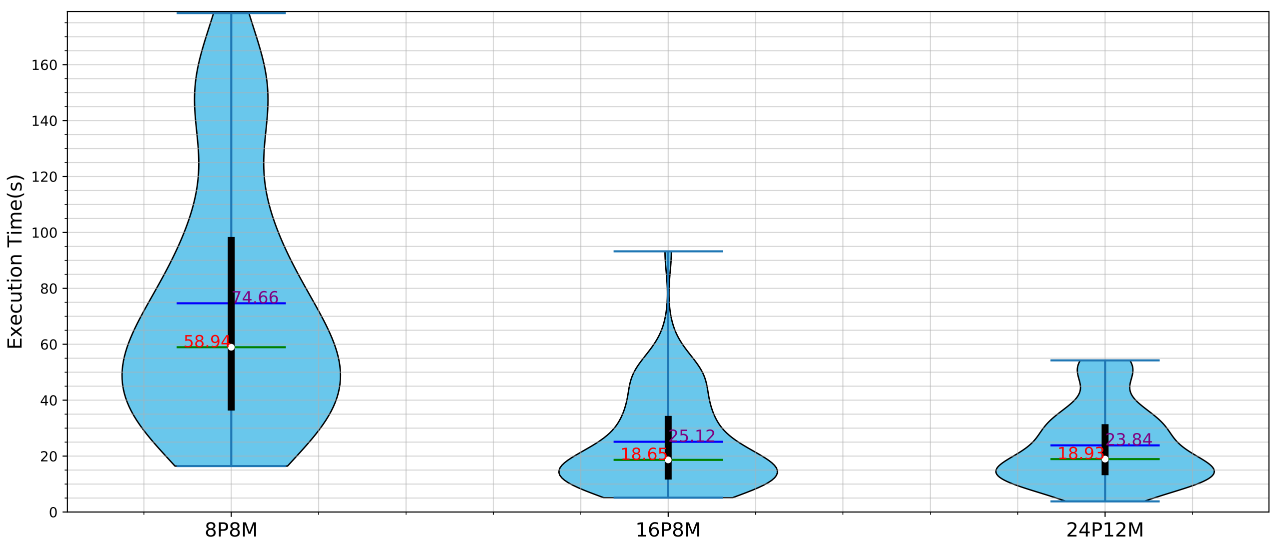


Figure 4.18: Scaling Benefits of GoDBX with Unified Model(V1) for *Report Query* of *Length 5*, without Cost Model



Note: Green line is median, blue is mean

Figure 4.19: Scaling Benefits of GoDB for *Select Query* of *Length 5*, without Cost Model



Note: Green line is median, blue is mean

Figure 4.20: Scaling Benefits of GoDB for *Report Query* of *Length 5*, without Cost Model

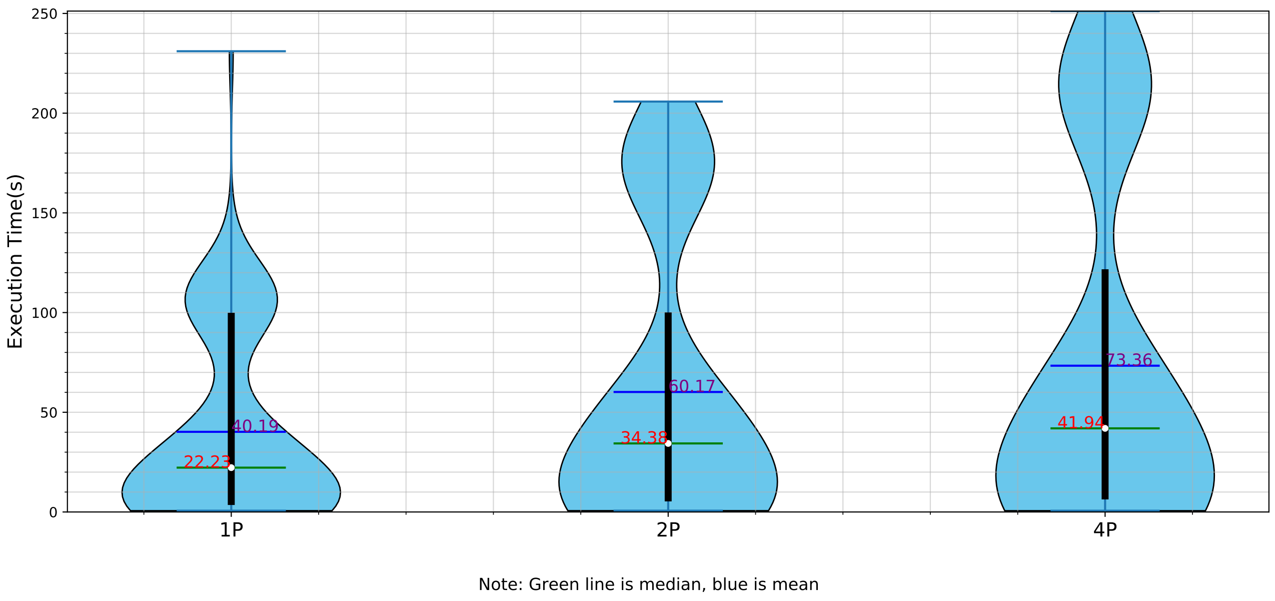


Figure 4.21: Scaling Benefits of Titan for *Select Query* of *Length 5*

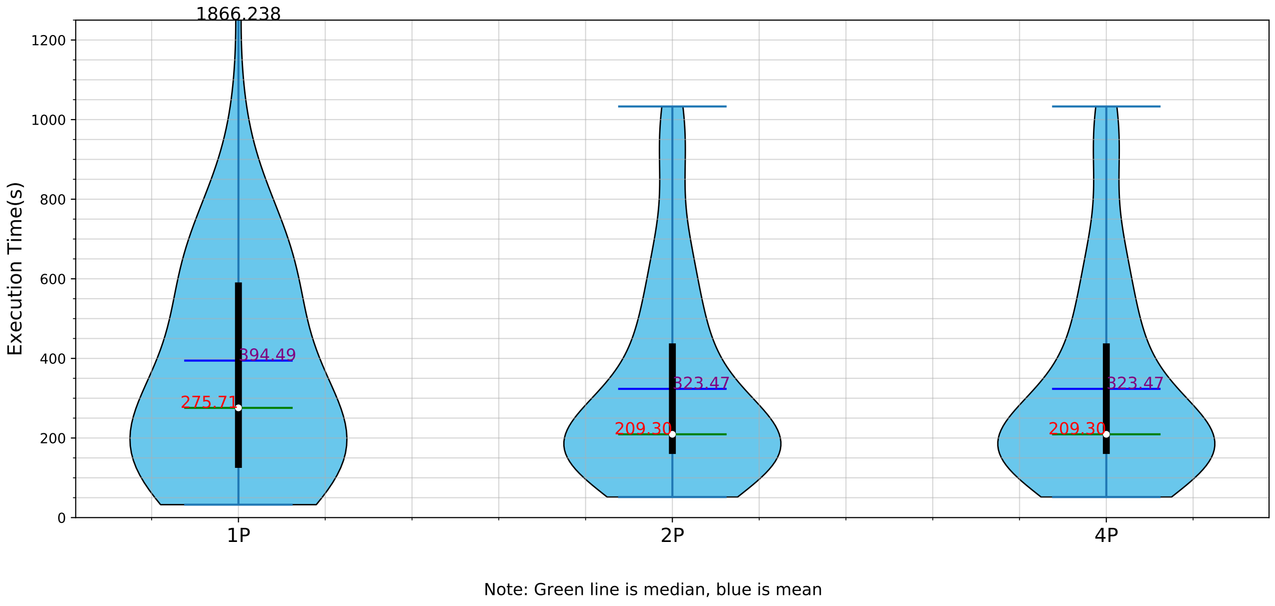


Figure 4.22: Scaling Benefits of Titan for *Report Query* of *Length 5*

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Graph databases are becoming increasing common as they emphasize on modeling data as graphs. Most of the Big-Data have inherent connections between representative entities which are modeled as edges of graph, making traversals of graph faster than relational counterparts. Most of these graphs follow power-law distribution, thus there are few vertices which have very large degree, most of vertices with relatively low degree. This makes it necessary to develop intelligent traversal techniques to avoid certain vertices that have extremely large degree, thus starting traversals from vertices which is expected to traverse very small portion of the graph.

In this context, GoDB [24] proposes a heuristic based cost model which tries to reduce the number of vertices traversed and thus reduces the number of predicate computation. In the distributed setup, it also tries to reduce the network costs by accounting about remote subgraphs in its heuristics. The work done in this thesis further cements the claims in [24] by proving the effectiveness of cost model for diverse set of workloads. For BFS workload, GoDB is seen to be $1.8x\ to\ 6.1x$ faster than Titan [3] for the set of workloads, thus showing the benefits of the execution model. Observations for Reachability and Path query workloads is akin to BFS query, Reachability being $1.8x\ to\ 11x$ and Path being $1.6x\ to\ 6x$ faster than Titan respectively for Citation Patent dataset. The cost Model facilitates selecting of plans that reduces the number of vertices traversed and avoid worse plans that has very high latency. Even For larger graphs like Google Plus, the benefits of Cost Model for Select and Report Query workload is observed.

GoDB uses property graph data model, which uses java based object representations of vertices,edges and properties. This has a high bloat in memory. We propose using compressed data formats such as Succinct [5] for representing these graphs. Succinct allows searching string in the compressed file and extracting a part of the file. We compare three data models called GoDBX-V1, GoDBX-V2 and GoDBX-V3. We also compare GoDBX-V3 against GoDB and Titan, which shows a trade-off between computation and network. It is seen GoDBX has a better performance for Select and comparable for Report Queries for Google Plus dataset while Titan performs worse.

5.2 Future Work

The GoDB *Query Optimizer* plans and optimizes each incoming query individually, with the assumption that each query will run in isolation and the queries and run sequentially. In real world scenarios, there will be more than one query in the system that wishes to execute concurrently with others. Further, the presence of distributed machines to host the data means that at a time, not all machines are used to full capacity. At the same time, there is a trade-off between response time and throughput when executing multiple queries simultaneously. Also, memory is a key constraint when executing queries over a graph present in distributed memory since the intermediate states of the queries can be large.

As ongoing work, we are working toward a *Prediction Model* which uses statistics based analytical model for estimating load on a distributed system for path queries. This model is able to predict number of supersteps and time taken per superstep with reasonable accuracy. Initial results shows that the prediction in the number of superstep is correct for 75 percent of the queries. In future we would use this model to propose a *Concurrency Model* which decides the concurrency of the system while adhering to response time and throughput constraints. This problem reduces to a optimization problem of finding the set of queries to be submitted to the system, given current set of queries running in the system and submission queue. There is a need for *on-line* method for solving this to reduce the latency induced by the *Concurrency Model*. This system will be built and integrated on top of GoDB as *Concurrency Control Manager* module. Further we would compare prediction model against basic models such as average model which uses average runtime of past query workloads. We would compare other models for concurrency control such as:

- Fixed batch model which submits fixed sized batches at a time
- Fixed concurrency model which tries to keep the number of queries running in system constant by submitting new query from submission queue when a query exits from the system.

The preliminary results are for Patent Citation network dataset. The *Prediction Model* and *Concurrency Model* will be validated for other larger graph datasets.

Besides this, there is extensive work on the different execution model for accessing and traversing a graph data store. There are further opportunities for using other heuristics in Cost Model. In current *Cost Model* of GoDB considers degree and count of vertices that match predicates to estimate the predicate computations at each step of the traversal. Other topological metrics(like centrality) can be incorporated in heuristics to estimate structure of the graph and detect bottleneck points in the graph for traversal. Orthogonal to using heuristics is to reduce the use of it and to progressively optimize queries during runtime. Concurrent queries should perform global query optimization which considers all queries currently submitted and detect an execution plan that increases throughput of the system. This will need to incorporate partial reuse of results across queries to reduce the access of inherent graph data store, thereby increasing query latency and throughput.

Bibliography

[1] US Patent Citation Network Graph. URL https://snap.stanford.edu/data/cit-Patents.html. vi, 36, 56

[2] DBLP Collaboration Network. URL https://snap.stanford.edu/data/com-DBLP.html. 61

[3] Titan. http://thinkaurelius.github.io/titan/. 12, 76

[4] *Horton: Online Query Execution Engine for Large Distributed Graphs (Demo Track)*, April 2012. URL https://www.microsoft.com/en-us/research/publication/horton-online-query-execution-engine-for-large-distributed-graphs-demo-track/. 2, 8, 11

[5] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. Succinct: Enabling queries on compressed data. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 337–350, Oakland, CA, 2015. USENIX Association. ISBN 978-1-931971-218. URL https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/agarwal. 3, 53, 54, 76

[6] Bernd Amann and Michel Scholl. Gram: A graph data model and query languages. In *Proceedings of the ACM Conference on Hypertext*, ECHT '92, pages 201–211, New York, NY, USA, 1992. ACM. ISBN 0-89791-547-X. doi: 10.1145/168466.168527. URL http://doi.acm.org/10.1145/168466.168527. 2, 6, 8, 9

[7] Avishek Anand, Stephan Seufert, Srikanta Bedathur, and Gerhard Weikum. Ferrari: Flexible and efficient reachability range assignment for graph indexing. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, pages 1009–1020, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-1-4673-4909-3. doi: 10.1109/ICDE.2013.6544893. URL http://dx.doi.org/10.1109/ICDE.2013.6544893. 55

[8] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1:1–1:39, February 2008. ISSN 0360-0300. doi: 10.1145/1322432.1322433. URL http://doi.acm.org/10.1145/1322432.1322433. 6, 8, 9

[9] Apache. The gremlin graph traversal machine and language for apache tinker pop. URL http://tinkerpop.apache.org/gremlin.html. 2, 8, 9

[10] Sabeur Aridhi and Engelbert Mephu Nguifo. Big graph mining: Frameworks and techniques. *CoRR*, abs/1602.03072, 2016. URL http://arxiv.org/abs/1602.03072. 1

[11] Matthias Bröcheler, Andrea Pugliese, and V. S. Subrahmanian. Dogma: A disk-oriented graph matching algorithm for rdf databases. In *Proceedings of the 8th International Semantic Web Conference*, ISWC '09, pages 97–113, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-04929-3. doi: 10.1007/978-3-642-04930-9_7. URL http://dx.doi.org/10.1007/978-3-642-04930-9_7. 2, 9, 10

[12] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. Tao: Facebook's distributed data store for the social graph. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 49–60, Berkeley, CA, USA, 2013. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2535461.2535468. 54

[13] E. F. Codd. *The Relational Model for Database Management: Version 2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990. ISBN 0-201-14192-2. 5

[14] Ayush Dubey, Greg D. Hill, Robert Escriva, and Emin Gün Sirer. Weaver: A high-performance, transactional graph store based on refinable timestamps. *CoRR*, abs/1509.08443, 2015. URL http://arxiv.org/abs/1509.08443. 1, 2, 11

[15] Bob DuCharme. *Learning SPARQL*. O'Reilly Media, Inc., 2011. ISBN 1449306594, 9781449306595. 7

[16] Marc Gemis, Jan Paredaens, Inge Thyssens, and Jan Van den Bussche. Good: A graph-oriented object database system. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 505–510, New York, NY, USA, 1993. ACM. ISBN 0-89791-592-5. doi: 10.1145/170035.171533. URL http://doi.acm.org/10.1145/170035.171533. 2, 9, 10

[17] Ralf Hartmut Güting. Graphdb: Modeling and querying graphs in databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 297–308, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc. ISBN 1-55860-153-8. URL `http://dl.acm.org/citation.cfm?id=645920.672980`. 6

[18] Marc Gyssens, Jan Paredaens, and Dirk van Gucht. A graph-oriented object database model. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '90, pages 417–424, New York, NY, USA, 1990. ACM. ISBN 0-89791-352-3. doi: 10.1145/298514.298593. URL `http://doi.acm.org/10.1145/298514.298593`. 2, 6, 8, 9, 10

[19] Huahai He and Ambuj K. Singh. Graphs-at-a-time: Query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 405–418, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6. doi: 10.1145/1376616.1376660. URL `http://doi.acm.org/10.1145/1376616.1376660`. 1, 2, 7, 8, 9, 11

[20] Jan Hidders. Typing graph-manipulation operations. In *Proceedings of the 9th International Conference on Database Theory*, ICDT '03, pages 394–409, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-00323-1. URL `http://dl.acm.org/citation.cfm?id=645505.656445`. 6, 8, 9

[21] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-marl: A dsl for easy and efficient graph analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 349–362, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0759-8. doi: 10.1145/2150976.2151013. URL `http://doi.acm.org/10.1145/2150976.2151013`. 8

[22] Peter T. Wood Isabel F. Cruz, Alberto O. Mendelzon. G+: Recursive queries without recursion. pages 645–666, 1988. 8

[23] Keita Iwabuchi, Scott Sallinen, Roger Pearce, Brian Van Essen, Maya Gokhale, and Satoshi Matsuoka. Towards a distributed large-scale dynamic graph data store. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, may 2016. doi: 10.1109/ipdpsw.2016.189. URL `https://doi.org/10.1109/ipdpsw.2016.189`. 55

[24] Nitin Jamadagni and Yogesh Simmhan. Godb: From batch processing to distributed querying over property graphs. In *IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid)*, 2016. [Core A]. 2, 3, 7, 8, 10, 11, 12, 14, 17, 18, 22, 53, 76

[25] Michael Kamfonas. Recursive hierarchies: the relational taboo. *The Relational Journal*, 27(10), 1992. 5

[26] Anurag Khandelwal, Zongheng Yang, Evan Ye, Rachit Agarwal, and Ion Stoica. Zipg: A memory-efficient graph store for interactive queries. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1149–1164, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4197-4. doi: 10.1145/3035918.3064012. URL `http://doi.acm.org/10.1145/3035918.3064012`. 54, 55, 57

[27] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The objectstore database system. *Commun. ACM*, 34(10):50–63, October 1991. ISSN 0001-0782. doi: 10.1145/125223.125244. URL `http://doi.acm.org/10.1145/125223.125244`. 10

[28] Ulf Leser. A query language for biological networks. *Bioinformatics*, 21(2):33–39, January 2005. ISSN 1367-4803. doi: 10.1093/bioinformatics/bti1105. URL `http://dx.doi.org/10.1093/bioinformatics/bti1105`. 8, 9, 11

[29] M. Levene and A. Poulovanssilis. An object-oriented data model formalised through hypergraphs. *Data Knowl. Eng.*, 6(3):205–224, May 1991. ISSN 0169-023X. doi: 10.1016/0169-023X(91)90005-I. URL `http://dx.doi.org/10.1016/0169-023X(91)90005-I`. 6, 8

[30] M. Levene and A. Poulovassilis. The hypernode model and its associated query language. In *Proceedings of the Fifth Jerusalem Conference on Information Technology*, JCIT, pages 520–530, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press. ISBN 0-8186-2078-1. URL `http://dl.acm.org/citation.cfm?id=100512.100601`. 6, 8, 9

[31] Leonid Libkin and Domagoj Vrgoč. Regular path queries on graphs with data. In *Proceedings of the 15th International Conference on Database Theory*, ICDT '12, pages 74–85, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0791-8. doi: 10.1145/2274576.2274585. URL `http://doi.acm.org/10.1145/2274576.2274585`. 8

[32] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012. ISSN 2150-8097. doi: 10.14778/2212351.2212354. URL `http://dx.doi.org/10.14778/2212351.2212354`. 1

[33] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0032-2. doi: 10.1145/1807167.1807184. URL http://doi.acm.org/10.1145/1807167.1807184. 12

[34] Norbert Martinez-Bazan, Sergio Gomez-Villamor, and Francesc Escale-Claveras. Dex: A high-performance graph database management system. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering Workshops*, ICDEW '11, pages 124–127, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-4244-9195-7. doi: 10.1109/ICDEW.2011.5767616. URL http://dx.doi.org/10.1109/ICDEW.2011.5767616. 7, 11

[35] Jayanta Mondal and Amol Deshpande. Managing large dynamic graphs efficiently. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 145–156, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1247-9. doi: 10.1145/2213836.2213854. URL http://doi.acm.org/10.1145/2213836.2213854. 1, 7

[36] Neo4j. Neo4j - the worlds leading graph database, 2012. URL http://neo4j.org/. 8, 9, 11

[37] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. URL http://ilpubs.stanford.edu:8090/422/. Previous number = SIDL-WP-1999-0120. 1

[38] Joan Peckham and Fred Maryanski. Semantic data models. *ACM Comput. Surv.*, 20 (3):153–189, September 1988. ISSN 0360-0300. doi: 10.1145/62061.62062. URL http://doi.acm.org/10.1145/62061.62062. 2, 7

[39] Arnau Prat-Pérez, Joan Guisado-Gámez, Xavier Fernández Salas, Petr Koupy, Siegfried Depner, and Davide Basilio Bartolini. Towards a property graph generator for benchmarking. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*, GRADES'17, pages 6:1–6:6, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5038-9. doi: 10.1145/3078447.3078453. URL http://doi.acm.org/10.1145/3078447.3078453. 61

[40] Abdul Quamar, Amol Deshpande, and Jimmy Lin. Nscale: Neighborhood-centric large-scale graph analytics in the cloud. *The VLDB Journal*, 25(2):125–150, April 2016. ISSN 1066-8888. doi: 10.1007/s00778-015-0405-2. URL http://dx.doi.org/10.1007/s00778-015-0405-2. 12

[41] Mohamed Sarwat, Sameh Elnikety, Yuxiong He, and Mohamed F. Mokbel. Horton+: A distributed system for processing declarative reachability queries over partitioned graphs. *Proc. VLDB Endow.*, 6(14):1918–1929, September 2013. ISSN 2150-8097. doi: 10.14778/2556549.2556573. URL http://dx.doi.org/10.14778/2556549.2556573. 1, 2, 8, 10, 11

[42] Andy Seaborne. An rdf netapi. In *Proceedings of the First International Semantic Web Conference on The Semantic Web*, ISWC '02, pages 399–403, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-43760-6. URL http://dl.acm.org/citation.cfm?id=646996.711294. 10

[43] Roman Shaposhnik, Claudio Martella, and Dionysios Logothetis. *Practical Graph Analytics with Apache Giraph*. Apress, Berkely, CA, USA, 1st edition, 2015. ISBN 1484212525, 9781484212523. 1, 12

[44] Yogesh Simmhan, Alok Kumbhare, Charith Wickramaarachchi, Soonil Nagarkar, Santosh Ravi, Cauligi Raghavendra, and Viktor Prasanna. Goffish: A sub-graph centric framework for large-scale graph analytics. In *International European Conference on Parallel Processing (EuroPar)*, 2014. 11, 12, 22

[45] Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and querying ordered xml using a relational database system. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 204–215, New York, NY, USA, 2002. ACM. ISBN 1-58113-497-5. doi: 10.1145/564691.564715. URL http://doi.acm.org/10.1145/564691.564715. 7

[46] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. Pgql: A property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, GRADES '16, pages 7:1–7:6, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4780-8. doi: 10.1145/2960414.2960421. URL http://doi.acm.org/10.1145/2960414.2960421. 1, 2, 7, 8, 9

[47] Gergely Varró, Katalin Friedl, and Dániel Varró. Graph transformation in relational databases. *Electron. Notes Theor. Comput. Sci.*, 127(1):167–180, March 2005. ISSN 1571-0661. doi: 10.1016/j.entcs.2004.12.034. URL http://dx.doi.org/10.1016/j.entcs.2004.12.034. 2, 5

[48] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. A comparison of a graph database and a relational database: A data provenance perspective. In *Proceedings of the 48th Annual Southeast Regional Conference*, ACM SE '10, pages 42:1–42:6, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0064-3. doi: 10.1145/1900008.1900067. URL http://doi.acm.org/10.1145/1900008.1900067. 5

[49] Chen Wang, Wei Wang, Jian Pei, Yongtai Zhu, and Baile Shi. Scalable mining of large disk-based graph databases. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, pages 316–325, New York, NY, USA, 2004. ACM. ISBN 1-58113-888-1. doi: 10.1145/1014052.1014088. URL http://doi.acm.org/10.1145/1014052.1014088. 10

[50] Wei Wang, Chen Wang, Yongtai Zhu, Baile Shi, Jian Pei, Xifeng Yan, and Jiawei Han. Graphminer: A structural pattern-mining system for large disk-based graph databases and its applications. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 879–881, New York, NY, USA, 2005. ACM. ISBN 1-59593-060-4. doi: 10.1145/1066157.1066273. URL http://doi.acm.org/10.1145/1066157.1066273. 9, 10

[51] Kevin Wilkinson, Craig Sayers, Harumi Kuno, and Dave Reynolds. Efficient rdf storage and retrieval in jena2. In *Proceedings of the First International Conference on Semantic Web and Databases*, SWDB'03, pages 120–139, Aachen, Germany, Germany, 2003. CEUR-WS.org. URL http://dl.acm.org/citation.cfm?id=2889905.2889914. 9, 10

[52] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 2:1–2:6, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2188-4. doi: 10.1145/2484425.2484427. URL http://doi.acm.org/10.1145/2484425.2484427. 1

[53] Xifeng Yan, Philip S. Yu, and Jiawei Han. Graph indexing: A frequent structure-based approach. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 335–346, New York, NY, USA, 2004. ACM. ISBN

1-58113-859-8. doi: 10.1145/1007568.1007607. URL http://doi.acm.org/10.1145/1007568.1007607. 1

[54] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1863103.1863113. 12

[55] Qizhen Zhang, Da Yan, and James Cheng. Quegel: A general-purpose system for querying big graphs. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 2189–2192, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3531-7. doi: 10.1145/2882903.2899398. URL http://doi.acm.org/10.1145/2882903.2899398. 12, 13