Report
A Survey of Graph Processing Paradigms

Version 0.62

September 10, 2018

Contents

Chapter 1

Introduction

A **paradigm** is a "model" for doing something. This document is an attempt to survey in a relatively standardized way a variety of **computing paradigms** that provide significant support for expressing and executing algorithms that deal with graphs. The report is divided into several parts:

- Part I describes several explicit graph programming languages.
- Part II describes programming libraries that provide such support within conventional languages.
- Part III describes systems (combinations of languages, libraries, and run-times) , especially ones that run on parallel platforms, that support graph computation.

Part IV then compares these different paradigms in a systematic way.
The rest of this chapter is organized as follows:

- Section 1.1 provides some basic definitions.
- Section 1.2 provides some standard rules for syntax for describing statements in the programming systems discussed here.
- Section 1.3 discusses some common concepts found in many of the programming systems described here.
- Section 1.4 provides a change log for this report.

This report was developed as part of an NSF grant CCF-1642280, with drafts of many of the chapters written as part of CSE 60742, a Fall 2018 class in Scalable Graph Processing, at the University of Notre Dame. The names of the contributing students is included at the beginning of each chapter.

1.1 Basic Definitions

Many of the programming systems use different terms to mean the same thing. This report will standardize the notation used in the individual descriptions, with cross-references in each section to relate the terms used by the original sources.

Each of the following sections provides some of these standard definitions.

1.1.1 Relations

- An **object** is some data structure that may be distinguished from another object. All of the following definitions may be used to define objects.
- A **set** is an unordered collection of objects, each of which is unique, with no duplicates. A set with a finite number of elements is written as the list of elements surrounded by "{ }".
- A **bag** is an unordered collection of objects where there may be duplicates.
- A **k-tuple** (or **tuple** for short) is an ordered collection of k objects where the position of objects in the tuple matters, and where the same object may be used more than once (in different positions). Thus two tuples with the same objects but in different orders are different tuples. A k-tuple is written as a list of its components, in order, surrounded by "( )".
- A **pair** is another term for a 2-tuple.
- The **Cartesian product** of k sets $S_1, ..S_k$ is the set of tuples $\{(u_1, ...u_k)\}$ where u_i is an object from S_i. It is typically written as $S_1 x S_2 x ... x S_k$.
- A **k-relation** (or **relation** for short) R on sets S_1, ... S_k is a set of k-tuples where the i'th element of each k-tuple comes from set S_i. Some or all of the S_is may be the same set.

  This is equivalent to saying that R is a subset of the Cartesian product $S_1 x S_2 x ... x S_k$.

  In conventional terminology the term R$(u_1, u_2, ...u_k)$ means that the tuple $(u_1, u_2, ...u_k)$ is in the relation R.
- A **binary relation** R is a set of pairs { (u_i, v_i) } where all the u's come from a set U and all the v's come from a set V. U and V may be the same set, and if so, there is no requirement that if (u, v) is in R then so is (v, u).

  In conventional terminology the term R(u,v) means that the pair (u, v) is in the binary relation R.
- Very often we will say that there is a **relationship** or an **association** between two objects u and v if the pair (u, v) is in some relation.
- A **function** f is a binary relation where for each u in U there is exactly one v from V such that (u, v) is in f. If both (u, v) and (u, w) are in f then v=w.

  In conventional terminology, the object u is termed the argument for f and v is the result.
- A **partition** of a set S is the set of subsets of S that have no overlapping elements and which if unioned together form S.

In all of the above there is nothing preventing the sets from which tuples are created to be sets of tuples themselves.

Binary relations are equivalent to normal predicates in everyday usage. Thus if U = V = the set of integers, then > is equivalent to the binary relation {(u, v)} where numerically $u > v$.

Non-numeric binary relations are also common. For example, if U = V = set of all people, then father(Pete, Tim) is the binary relation where Pete is the father of Tim.

1.1.2 Graphs

Graphs as we use them in this study are closely tied to binary relations.

- A **graph** G is equivalent to a binary relation based on a set of **objects** where there may be some explicit relationships between pairs of objects. Equivalently $G = (V, E)$ where V is the set of objects termed "vertices," and E a set of "edges" representing the relationships.
- A **vertex** is the name of some unique object within a graph.

  The set of vertices for a graph may be partitioned into different classes of vertices.

  In many contexts the term "node" is used interchangeably with vertex, whereas here we will use "node" solely to define a part of a parallel system.
- An **edge** is the name of the expression of a relationship between two vertices in a graph, and is often written as a pair (u, v) where u and v are vertices in the graph.

  Strictly speaking, the set E of edges for a graph may be a bag, meaning that there may be multiple pairs between the same two vertices.

  Graphically, if a vertex is drawn as a point, then an edge (u, v) is an arrow drawn from u to v.
- If an edge (u, v) is **directed**, then the relationship implied by that edge only goes from u to v. If an edge (u, v) is **undirected**, then there are two relationships expressed by the edge: (u, v) and (v, u).
- Edges may also be **named**, where the subset of E that has the same "name" represents a set of edges that together express a particular binary relation. There is nothing preventing a graph from including many different types of edges.
- A **property** is a value that may be associated with either a vertex or an edge. An important example may be the name of an edge.
- The **out-degree** of a vertex u is the number of edges (u, v) that leave the vertex.

  There may be variations in this definition based on whether or not duplicate edges are permitted in a graph, and whether or not each such duplicate should be counted. Also, with there are multiple types of edges, the out-degree from a vertex may depend on the edge type.
- The **in-degree** of a vertex v is the number of edges (u, v) that enter the vertex.
- A **directed path** of **path length** n exists from a vertex u to a vertex v if there are edges (u, w_1), (w_1, w_2), ... (w_{n-2}, w_{n-1}), (w_{n-1}, v).
- The **transitive closure** of a graph G_1 is a graph G_2 where there is an edge from u to v in G_2 if there is a directed path (of any length) from u to v in G_1.

1.1.3 Computing Systems

- A **node** is a collection of cores and memory that represent a complete and minimal unit. Typically any thread executing in any core in a node can directly access any memory location in that node.

1.2 Syntax Definition

This report loosely follows the notation used in [1] to define rules of **syntax** for describing the set of valid character strings of a particular programming language, as follows:

- **Terminal symbols** are the basic characters from the language, and are expressed as the characters themselves, with the exception of when they are the same as meta-symbols discussed next, in which case they are written with a \ in front of them.
- **Meta-symbols** are characters in a syntax rule that are part of the rule and not of characters in the language being described. For this paper they include: |, ϵ, ?, [,], {, }, $^+$, *, \.
  - "{ }" surrounding a string means to treat that string as if it were a single unit in terms of other syntax rules, especially those using meta-symbols |, ?, $^+$, and *.
  - A "|" between two strings indicates that either one is an acceptable string.
  - "[ ]" around a string is shorthand for placing a "|" between each character in the string. Thus [abc] is equivalent to a|b|c.
  - ϵ represents a string of zero length.
  - ? used as a postfix to a string means "0 or 1" occurrences of that string. Thus "$-?number$" means the minus sign is optional in front of any string that represents a number. ? is often used after a possibly multi-character string surrounded by "{ }" to denote the whole string is optional, as in {+ |−}? *number*.
  - $^+$ as a postfix to a character string indicates that the character string may be repeated one or more times, with an unlimited number of repeats permitted. Again, the use of the meta symbols { } around a string allows a single $^+$ to refer to the whole string.
  - * as a postfix to a character that the character may be repeated zero or more times, with an unlimited number of repeats permitted.
  - A "\" in front of a letter, especially one of the excluded ones above, means the terminal character itself, not the meta symbol as described above.
- Strings of terminal symbols that represent **keywords** in the language are shown as the sequence of terminals, in **bold**.
- The names of **nonterminal symbols**, alternatively called **syntactic variables**, represents set of character strings that follow some rules, and will be shown as character strings in *italic*.
- A **production rule** is a statement in the syntax description that defines one way that strings of characters can be constructed. It will be shown in the form: "$head \rightarrow body$" where *head* is typically a nonterminal whose set of strings is defined in part by *body*.
- As a shortcut, if the name of a nonterminal ends in "*_list*", as in *nonterm_list*, then it is assumed that there is an implied production rule of the form: $nonterm\_list \rightarrow nonterm(,nonterm)^*$ that is a comma-delimited string of *nonterm*s.

An example of the definition of an arithmetic expression in the above notation may be:

$$
\begin{aligned}
digit &\rightarrow 0|1|2|3|4|5|6|7|8|9 \\
pos\_number &\rightarrow digit^{+} \\
number &\rightarrow \{+|-\}?\ pos\_number \\
factor &\rightarrow number \\
factor &\rightarrow (expression) \\
term &\rightarrow factor \\
term &\rightarrow term * factor \\
expression &\rightarrow term \\
expression &\rightarrow term\{+|-\}\ expression
\end{aligned}
$$

1.3 Common Concepts

This section discusses briefly some common programming concepts that show up in many of the graph systems described here.

1.3.1 Execution Model

An **execution model** describes how a sequence of computational steps is derived from a computer program, scheduled, and then executed.

A programming language with an **imperative execution model** is one where individual steps of computation are clearly identified in the program, along with their order of execution. An example is the C programming language where the unit of computation is a statement, statements are executed one at a time to completion before starting the next one, and the order of execution is in a text linear order unless explicitly changed by a program directive. The **program state** for such programs consists of a program counter pointing to the current program unit and the contents of all memory locations that are visible to the program.

A **single assignment model** is one where "variables" receive values "only once," and the units of computation combine the variables which must have received values with the expression to evaluate In such models, the "order" in which computation is executed is immaterial. The only ordering is in terms of which program units have all their initiating conditions satisfied.

The terms **data flow model** represents a computation as the flow and exchange of information between functions. When expressed graphically, it is a **DAG** (or Directed Acyclic Graph), where "computations" may be performed in any order as long as the required input data is available. The same DAG can represent both sequential and parallel executions, and even mixtures where partial data sets may be "streamed" through functions before all inputs are available.

A **declarative model** expresses ("declares") the conditions that must exist between different data values without expressing the control order for when, or even how, the computations to enforce those conditions are performed. Relational database languages like SQL/indexSQL are highly declarative. **Functional model** and **logic models** are similar.

1.3.1.1 Logical Variables

A **variable** in conventional "imperative" programming languages has as its execution model a mapping to a memory location which may be read and written multiple times during a program's

execution. The programmer is intimately concerned with how such a value may change in time.

In contrast, many graph programming languages are declarative in nature. In such paradigms, a **logical variable** is a "single assignment" variable, receiving a value "once." In addition, such languages sometimes are **pattern variables** where the don't represent the concept of "receiving" a value, but as a placeholder for a subset of all values in the variable's domain that make a surrounding expression true. Further, when such variables show up in multiple locations of a complex expression, the subset they stand for is the one that satisfies all the expressions.

1.3.2 Relational Operators

Many graph languages have characteristics that are derivatives of relational database systems. Such relational systems start with databases of tuples, where the components of each tuple is akin to the value of properties as previously defined, and where queries over these sets may be specified with the following general classes of operations:

- **Select**: create a subset of some set of tuples, where the components of each tuple in the subset pass some set of tests based on their values.
- **Project**: create a new data set from an existing one, where the properties of the tuples in the new set are taken from those of the originating set.
- **Join**: create a new data set from two or more data sets where the tuples of the new sets come from a subset of the Cartesian product of the input sets. Typically there is some predicate, such as equality, that must hold between properties of the input sets.

1.3.3 Grouping

Grouping is a capability in a programming language that allows a result that is a set of tuples to be grouped into a set of subsets of those tuples, where each subset has the same value in some property.

Such a capability has its origins in relational database systems in a function like **GROUP BY** in SQL. Such a function, when added to a query expression such as described in Section 1.3.2, partitions the results of the query.

Such functions are typically used in advance of aggregation functions as described in Section 1.3.4 that are applied over each subgroup to return a single value.

The use of groups in graphs is useful for expressing relationships such as the "neighborhood" of a vertex.

1.3.4 Aggregation

An **aggregation function** also has its origins in relational database systems, and, when given a set, returns some (typically numeric) property of that set. Typical aggregation functions include a count of the size of the set, or the sum, max, min, average, etc. of some property of every member of a set.

Aggregation functions are often applied to subsets identified by groups as discussed in Section 1.3.3

In graph computations such aggregations are often used to determine properties of objects, such as the out-degree of a vertex by counting the number of neighbors.

1.3.5 MapReduce

MapReduce, as popularized by Hadoop[15], is a parallel programming paradigm where sets of "key-value" pairs are distributed over a cluster of computing nodes. A single MapReduce step is a three-phase process:

1. MAP: A user-provided *map* function is applied to all key-value pairs to produce streams of modified key-value pairs.
2. SHUFFLE: All the key-value pairs are sorted by their key, and a new set of key-value pairs is created where there is one pair for each key where the "value" is a set of all the values from the output of the map phase that had the same key.
3. REDUCE: A user-provided *reduce* function is applied to each of the outputs of the second phase. This function typically is some sort of an aggregation on the value set. The output of this phase is arbitrary, and includes the possibility of generating new sets of key-value pairs for future MapReduce steps.

While MapReduce is designed for processing of large sets, and has trouble with expressing the kinds of recursive procedures found in many graph problems, there have been reports of the expression of some graph problems in the paradigm, as in the computation of Jaccard coefficients[2, 4]

1.4 Change Log

Version	Date	Changes
0.62	9/8/18	Change title to "Paradigms". Reformat to reflect use in CSE 40742 class. Addition of NetworkX as graph package, and removal of the benchmark section for inclusion in a separate document.
0.6	7/23/17	Completed GraphBLAS.
0.5	6/1/17	Added sections for GraphBLAS and Poplar. Initialized GraphBLAS. Edited KEL.
0.4	1/12/17	Partial update to GRAQL.
0.3	8/15/16	Initial version of GRAQL chapter.
0.2	2/10/16	Update to KEL chapter and adding in Jaccard coefficient kernel discussion.
0.1	2/1/16	Initial document constructions and initial version of KEL chapter.

Table 1.1: Change History.

Part I

SECTION I: Graph Languages

Chapter 2

Accumulo

Apache Accumulo [Apache 2015d], like Spark, is not necessarily graph-oriented, but has developed a significant graph-related user base. It is a distributed key-value store, has its origins in Googles BigTable [Chang 2008], and is written in Java on top of the Hadoop Distributed File System (HDFS). Keys in the key-value pairs have multiple attributes, including row, multi-attribute columns, and a timestamp.

Chapter 3

Cypher

Neo4j [Neo 2015] [Robinson 2013] is one of the most widely referenced graph database packages. It has both a graph database engine and a query language CYPHER. Datasets are described as property graphs where both vertices and edges can have attached arbitrary lists of (key, value) pairs representing properties. It is written in Java and Scala, with source code available on Github, and includes a separate API that allows direct access to the graph engine. The underlying query engine is multi-threaded and has been demonstrated on shared memory systems with 100s of cores. An effort is under way (http://www.opencypher.org/) to produce an open source version of CYPHER.

Chapter 4

GraphLab

GraphLab [Low 2010] is an open source framework to support scalable machine learning applications characterized by sparsity and asynchronous iterative computations. It was designed to scale from multi-core shared memory nodes up through distributed systems. Its data model has two parts: graphs where arbitrary data structures can be associated with both vertices and edges, and shared data tables that are represented as key-value pairs. A programmer can define update functions to be applied over neighborhoods of vertices, and sync functions which permit updates into the shared data tables.

Chapter 5

GraQL

GraQl[6] is a graph language designed at the Pacific Northwest National Labs to specify queries on graphs expressed either in terms either of sets of vertices and edges or as tables. In the table view, an individual row corresponds to a vertex or edge, with the columns of that row representing attributes or properties
indexGraQL!properties of that vertex or edge. The output of a GraQL query is a set of subgraphs of some graph that exhibit some property.

5.1 Data Structure Definitions

GraQL has its syntactic origins in SQL, and like SQL has as its basic data structure a table, with strong typing of the values in a table's columns. Graphs are then represented as "views" over such tables.

5.1.1 Tables

A table in GraQL is defined similarly to SQL:

$$\begin{aligned} \mathit{TableDeclaration} \rightarrow\ & \textbf{create table}\ \mathit{TableName}\ \backslash\{\mathit{AttributeDefinition\_List}\ \backslash\} \\ \mathit{AttributeDefinition} \rightarrow\ & \mathit{AttributeName}\ \mathit{Type} \end{aligned}$$

Each column is defined with a name for that attribute and an associated type. All values in the same column position in all rows in a table have the same type.

5.1.2 Vertices

Vertices are then defined as views off of such tables:

$$\begin{aligned} \mathit{VertexDeclaration} \rightarrow\ & \textbf{create vertex}\ \mathit{VertexSetName}(\mathit{VertexKey}) \\ & \textbf{from table}\ \mathit{TableName} \end{aligned}$$

The *VertexKey* should correspond to an attribute name of the corresponding table whose values represent a unique key into the table. In this case there is a one-to-one mapping between rows in the table and vertices of that class. For each row, the attribute in this *VertexKey* column is the

unique "name" of the corresponding vertex, and the other attributes represent properties of the vertex.

The overall graph defined by a GrapQL program has as its set of vertices the union of all vertex sets defined as above, and it is assumed that all such vertex sets are disjoint.

5.1.3 Edges

Edges are similarly defied as views on tables, but with some additional information given:

$$\begin{aligned} \mathit{EdgeDeclaration} \rightarrow\ & \textbf{create edge}\ \mathit{EdgeSetName}\ \textbf{with vertices}\ (\mathit{Vertex}, \mathit{Vertex}) \\ & \{\textbf{from table}\ \mathit{TableName\_List}\}? \\ & \{\textbf{where}\ \mathit{WhereClause}\ \{\textbf{and}\ \mathit{WhereClause}\}^*\}? \\ \mathit{Vertex} \rightarrow\ & \mathit{VertexSetName}\ \{\textbf{as}\ \mathit{Id}\}? \end{aligned}$$

Each edge declaration includes a list of two *Vertex* definitions, with the first defining the class of vertices from which a directed edge of this edge class may start, and the second defining the class of the target vertices.

An optional "**as** *Id*" after a vertex set name indicates that a vertex on that side of the edge should be given the alternative name *Id*. This is useful when both source and target are the same set.

If there is no optional **where** or **from table**. then edges are created as the cross product between all combinations of source and target vertices as defined by the **with vertices**.

A *WhereClause* is a boolean expression, typically involving the comparison of attributes in the source and target vertices. If there is an optional **where** clause in an edge definition, then each of the edges from the above cross product must pass the specified test before an edge is created.

If there is a conjunction of *WhereClause*s separated by **and**, then all of them must be true before an edge is added to the graph.

An *Id* defined by an **as** *Id* may be used in a separate *WhereClause* to refer to a specific end of the edge. This is useful when the classes of both the source and target of the edge are the same.

The optional **from table** defines one or more additional tables that may be used in a following *WhereClause*. Such tables are typically two-attribute tables, with attributes that are matched up to the vertex classes. Typically, an edge is then created for each row in this table that can satisfy the *WhereClause*s.

The overall graph defined by a GrapQL program has as its set of edges the union of all edge sets defined as above, and it is assumed that all such edge sets are disjoint.

5.1.4 Populating Tables

A series of **ingest** commands, as defined below, allow tables to be populated with data from external files.

$$\mathit{IngestCommand} \rightarrow\ \textbf{ingest table}\ \mathit{TableName}\ \mathit{FileName}$$

Ingesting a table also triggers generation of associated vertices and edges.

5.2 Table Queries

A query in GraQL is designed to find all subgraphs of a particular graph that match some *SubGraphExpression*. Its syntax is defined similarly to an SQL select statement as follows:

$$
\begin{aligned}
\mathit{Query} \rightarrow\ & \textbf{select}\ * \mid \mathit{AttributeName\_List} \\
& \textbf{from graph}\ \mathit{SubGraphExpression} \\
& \textbf{into table}\ \mathit{TableName} \\
\mathit{AttributeName} \rightarrow\ & \mathit{GraphLabel}\{.\mathit{AttributeName}\}? \\
\mathit{GraphLabel} \rightarrow\ & \mathit{VertexSetName} \mid \mathit{EdgeSetName} \mid \mathit{PathLabel}
\end{aligned}
$$

The output from a query is a table whose name is defined in the **into table** section, and whose attributes are defined by the *AttributeName_List* in the **select**. This latter list also specifies the graph sets where the values for the different attribute columns come from, with typing information inherited from the type of the source attribute.

An optional "*" in place of the attribute name list refers to all matching vertices and edges.

In general the graph sets used in the select are names of vertex or edge sets used in the *SubGraphExpression*. In cases where a graph set is used more than once in the same *SubGraphExpression*, an optional label defined by the programmer within the *SubGraphExpression* can define which set is to be used.

A GraQL program may include multiple such queries and/or may be followed by conventional SQL selects that do such things as take the top N items, where a value is derived from each surviving row in the referenced table, or where such rows are grouped, counted, ordered, summed, etc, again as done in SQL.

5.2.1 *SubGraphExpressions* and Basic Path Expressions

A *SubGraphExpression* is a description of a subgraph that is to be matched. It must start and end with a vertex set, and consists of a series of query steps defined as follows:

$$
\begin{aligned}
\mathit{SubGraphExpression} \rightarrow\ & \mathit{PathExpression}\ \{\{\textbf{and} \mid \textbf{or}\}\ \mathit{PathExpression}\}^* \\
\mathit{PathExpression} \rightarrow\ & \mathit{VertexSelection}\{\mathit{PathStep}\}^+ \\
\mathit{VertexSelection} \rightarrow\ & \mathit{StepLabel}?\ \mathit{VertexName}\{(\mathit{FilterExpression\_List})\}? \\
\mathit{StepLabel} \rightarrow\ & \{\textbf{def} \mid \textbf{foreach}\}\ \mathit{Label}: \\
\mathit{PathStep} \rightarrow\ & \{\mathit{ForwardEdge} \mid \mathit{BackwardEdge}\}\ \mathit{VertexSelection} \\
\mathit{ForwardEdge} \rightarrow\ & -\mathit{EdgeSelection}->\ \\
\mathit{BackwardEdge} \rightarrow\ & <-\mathit{EdgeSelection}-\ \\
\mathit{EdgeSelection} \rightarrow\ & \mathit{StepLabel}?\ \mathit{EdgeName}\{(\mathit{FilterExpression\_List})\}?
\end{aligned}
$$

The first *VertexSelection* on the left identifies the starting set of vertices for the subgraph to be defined by a *PathExpression*. This set is the name of a vertex set optionally followed by a list of boolean expressions in "()" that "filter" this set. Typically these boolean expressions test or compare attributes of each vertex in the vertex set. Only those that pass all the tests are allowed to start the subgraph description.

Following this *VertexSelection* is an arbitrary number of *PathStep*s, where each is the definition of an edge that must be in the subgraph. Each *PathStep* includes both an *EdgeSelection* and a *VertexSelection*. Each of these latter may include *FilterExpressions* as with the initial vertex set, and only entries that pass the filters are considered for the subgraph. We distinguish between an edge that starts with the vertex set on the left by "$-...->$" notation, and edges that start on the right by "$<-...-$" notation.

If we count each set (vertex or edge) that is referenced in a query as a **graph query step**, then if there are n path steps, there are 2n+1 path steps (n+1 involving vertices and n involving edge sets).

5.2.2 Result Sets

The result of a query computation is the union of the subset of vertex sets that satisfy the path. There is one such subset for both the initial leftmost vertex set and the sets used in each of the path steps. Each such subset is culled by both the step's associated filter and by the existence of appropriate edges that link the vertex into earlier and later vertices in the path.

We note that if there are multiple subgraphs which satisfy all the conditions, then the resultant set is simply the aggregate set of vertices in all such subgraph instances. There is no enumeration of the individual subgraphs, nor any indication of which combinations of vertices formed individual instances. A rationale for this is that the set of such vertices is of O(V), the total number of vertices, whereas the number of vertices when organized into n+1 tuples each representing a separate subgraph instance could grow exponentially, especially if there are cycles that match the pattern.

The result of a query in terms of the edges is similar - a subset of the total edge set representing all the edges that were traversed at least once.

The **select** part of the query identifies by name one of these sets (and its attributes). The only components of the named set that are used to form rows in the output table are those that survived the entire path process, that is they survived the initial filter and then appeared at least one in some satisfying subgraph.

5.2.3 Step Labels

A step label is a label placed on the specification of a vertex or edge set at an intermediate step of a path to capture results up to that point in the graph matching process. There are two kinds in GraQL: set and element-wise.

5.2.3.1 Set Labels

Prefixing either an edge or a vertex in a query step by a "**def** *Label*:" introduces a **set label**,. It indicates that the set of all the objects (vertices or edges) that are part of a matching subgraph in this position of the subgraph up to this point are to be given the specified label, and may be referred to again either in a later *PathStep* or in the query's *AttributeName_list* in the query's **select**.

It is important to recognize that a set label at path step i defines the set of all vertices (or edges) that were successfully part of the path expression so far from step 1 to the current step. They may include elements that will be discarded when steps later than i are considered. Thus if a set label defined at step i is referred to in step i+j, all vertices that were part of the set at step i will still be considered from i+j forward, even if some step between i and i+j had excluded them.

This is useful if, for example, the same vertex set is used multiple times in a path, and the **select** wishes to to separate out which vertices satisfied which path steps. In terms of the result, a

vertex that satisfies a path in a position with a **def** is considered as different from the same vertex that satisfies some other part of the path where the original vertex set name was not so overridden.

5.2.3.2 Element-wise Labels

Alternatively, a "**foreach** *Label*:" prefix defines an **element-wise label** that performs slightly differently than a set label. Assuming the label is on a vertex set at step i, at any later step i+j, a vertex can be in the set that satisfies that step only if it was also in the set labelled at step i.

5.2.4 Multi-Path Queries

A query may include more than one *PathExpression*, connected by either **and** or **or**. For **ors**, satisfying either is sufficient for inclusion in the result set.

For **ands**, both expressions must be satisfied by the same set of vertices and edges to make it to the result. This requires a step label in one of the paths to define an intermediate set, and the use of that defined set in the other.

5.2.5 Meta-variables and Variant Steps

The use of the syntax "[]" in place of either a *VertexSelection* or an *EdgeSelection* is equivalent to a wide card. If used for an *EdgeSelection* in a step it means that any edge out of the source of any type can match the pattern at this step. If used for a *VertexSelection* in a step it means that any vertex can match the pattern at this step. Together this allows for the description of type invariant paths.

A step that uses a "[]" is called a **variant step**.

No filters are allowed on "[]" as there is no guarantee that all vertex or edge types have a common subset of attributes.

Labels ar permitted on variant steps, and capture vertices or edges from any of the graph's defined sets.

5.2.6 Regular Expression Paths

The syntax of a *PathExpression* may be enhanced as follows:

$$PathExpression \rightarrow \ \backslash\{\ PathExpression\ \backslash\}\{+\mid *\mid number\}$$

The suffix specifies that the *PathExpression* inside of the "{}" may be repeated: 1 or more times ("+"), 0 or more times ("*"), or exactly some *number* of times.

5.3 Graph Queries

The prior section dealt with queries that generated tables. It is also possible to have the output of a query go into a graph format rather than a table' This replaces the

into table *TableName*

part of a query by

into subgraph *GraphName*

Notionally, this generates a new set of tables of vertices and edges with the same names as pulled from the *AttributeNameList* in the **select**, but prefixed in name by the "GraphName".

In this case the optional "*" in the select part of the query gathers up all the vertices and edges that matched the *GubGrapExpression*, creating all, and only, those parts of the original graph that matched.

The subgraph output of one graph can be accessed by another query by prefixing the names of the vertex and edge sets by the designated GraphName and a ".".

5.4 Notional Execution Model

Current GraQL documentation does not explicitly address the execution model, other than to say that it is targeted for scalable, multi-threaded, in-memory systems and is an upgrade from a first generation GEMS [13, 5] system that handled RDF triples and SPARQL query programs.

To Be completed later

5.5 Questions

Following is a list of questions that could not be answered by the available documentation.

1. It is unclear how to add properties to edges.
2. Equation 1 shows a projection is possible in the definition of a vertex set, but the syntax examples of Fig.2 show only identification of the key.
3. Equation 2 implies that there can be a select filter on the from table. Is such a filter to follow the table name, or is it part of the where clause.?
4. Equation 2 and associated text imply only natural joins for generating the cross product of possible edges. That corresponds to the "=" in the where clauses, but is "=" the only predicate allowed?
5. There doesn't seem to be any mechanism for modifying a vertex's properties.
6. In a select query can you have a list of attributes from one label, and/or a list of such labels?
7. Is "< −...− >" acceptable terminology for an undirected edge?
8. Equation 5 makes no sense since the "set" is defined as a logical conjunction of boolean tests. What it appears is meant is the union of the subsets of vertices from each vertex set in the path that together are interconnected in some way that satisfies the subgraph. If a vertex set appears multiple times in a path then a member of that set will be in the final set if it ends up in any of the path positions, and it is impossible for the select to know which position in the path it came from. That apparently is the reason for the **def** labels?
9. Can a **select** have more than one vertex or edge sets referenced as outputs to form the final table?

10. For element-wise labels, the text says that if step i has an element-wise label and if v is to be part of step i+j it must have been part of the set at step i. Given that the set of all vertices is partitioned into multiple vertex sets, shouldn't this constraint be modified to say that v must be part of step i+j if the set defined at step i+j was the same as that specified at step i?

11. What is the termination condition for the "+" or "*" regular expresions? When there is no change in the set of included vertices or when following one more iteration results in a null set? Also how is the output formed for either an **into table** or **into subgraph**?

12. The text associated with Fig. 14 (and a "*" in the **select**) indicates that the table generated has one row per subgraph, while earlier (Fig. 6) indicate that the table that is output is just a list of vertices from one step. There are cases where one might be preferred over the other. More clarity on how the output is formed would be seful.

Chapter 6

Gremlin

Gremlin [Rodriguez 2015] is a combination of a language for defining traversals within a graph and an execution engine for executing such traversals. Traversals may be defined as compositions of other traversals in a wide variety of functional patterns. The underlying engine was designed using a distributed BSP model. The Apache open source project TINKERPOP (http://tinkerpop.incubator.apache.org/) is developing code for this system.

Chapter 7

KEL

KEL (Knowledge Engineering Language)[12] was designed by LexisNexis Risk Solutions, Inc. to provide a declarative language for expressing logical relationships between entities on large commercial data sets. While these data sets are stored in files as table-like collections of individual records, the way the records are treated during processing is as if they represent potentially very large and complex graphs where vertices and edges all may have multiple and complex properties.

KEL is built upon **ECL** (Enterprise Control Language) [11], which is itself a declarative language for big data analytics. In KEL a user may define classes, data sets, and instances of both "entities" (equivalent to objects and graph vertices) and "relationships" or "associations" (equivalent to relations and graph edges), both with optional lists of properties. "Logic statements" then express in a Prolog-like way new associations, or additions to existing associations, that are drawn from finding consistent instances in chains of existing associations where entities have certain property values, or where entities referenced in different associations are the same. The code produced by such statements will, if fully executed, compute the transitive closure of all the rules. This allows graph queries that involve recursive definitions to be stated simply. To quote [12]:

> KEL presumes that the user wants control over the logical data model, the analytic logic and the mathematics - however, KEL also presumes that the user doesn't want to concern themselves with the details of algorithm selection, process construction, the physical data model or key building.

The KEL tool chain converts KEL source into ECL code, which is then compiled into large C++ programs. Since ECL is declarative, ECL (and thus KEL) programs can be converted into functional program flow graphs, with the output of particular function nodes in the program flow being data sets with certain characteristics. The ECL tool chain compiles the result into C++ code optimized to run on very large parallel clusters using LexisNexis' HPCC Systems[1] parallel software stack.

The resulting "execution model" assumes that the original source data may be partitioned over potentially very many parallel nodes, and the generated C++ code then takes the source data on each node through as many program steps as possible until some sort of inter-node redistribution is needed, after which the process is repeated. Significant optimization and function lifting is done in the generation of the program nodes to minimize intermediate data, especially ones that require inter processor node communication. Thus individual data records are "streamed" through as many function nodes as possible, without waiting for all data to be processed by one function before started the next.

[1] https://hpccsystems.com/

The structure and execution model of KEL has a lot of similarity to logic programming languages like Prolog. Essentially the concept of a "function" in ECL is replaced by that of a "predicate" in KEL.

7.1 Defining Graphs

In KEL both vertices (i.e. objects) and edges are termed **entities**. In addition, a relation (i.e. a set of edges of the same class) is termed an **association**.

Both kinds of entities may have arbitrary numbers of properties - values associated with them.

7.1.1 Defining Entity Classes

A KEL program can define a class of entities, with optional properties on each instance, via an **entity declaration statement**:

$$\begin{aligned} \mathit{entity\_declaration} \rightarrow \mathit{entity} := \mathbf{ENTITY}(\mathit{filetype}(\mathit{prop\_list})\{, \mathbf{MODEL}(\mathit{prop\_list})\}?); \\ \mathit{filetype} \rightarrow \mathbf{FLAT} \mid \mathbf{XML} \mid \mathbf{TEXT} \end{aligned}$$

The *filetype* defines the file format from which records defining entities of that class will be input.

A *prop_list* provides a list of properties (see Section 7.1.2) to be associated with each instance of an entity of the class as it is read in from a compatible file. Notionally, each property corresponds to a field in each such input record, or a column in the overall table of records.

7.1.2 Properties

A **property** is a "named" value to be associated directly with an entity. In the case of entitles that represent vertices, such properties do not need an "edge" to associate them with a particular vertex instance (as in RDF). The following syntax defines individual properties in KEL as they are used in *prop_lists* in the definition of entities:

$$\begin{aligned} \mathit{property} &\rightarrow \mathit{datatype}?\ \{\mathbf{UID} =\}?\mathit{prop\_name}\ \{= \mathbf{NULL}(\mathit{value}?)\}?\ \mathit{datafilefield}? \\ \mathit{prop\_name} &\rightarrow \mathit{name} \mid \mathit{submodel} \\ \mathit{submodel} &\rightarrow \mathit{name}?\ \backslash\{\mathit{prop\_list}\backslash\} \end{aligned}$$

If there is no data type mentioned, the property is assumed to be of type string.

Properties may be either **single-valued** or **multi-valued**. Single-valued properties are ones that for any instance of the entity, that instance has only one value with that property name. Multi-valued properties are ones for which a particular instance of the entity may either have multiple components or a set of different values. For an entity of class Person, a *birthday* is an example of a single-valued property; a *last_name* is an example of the latter.

Section 7.1.3.1 discusses the meaning of a **UID**; Section 7.1.3.2 does the same for **NULL**.

7.1.3 Models and Sub-models

A **MODEL** declaration in an *entity_declaration* is a list describing a subset of the main property list that together distinguish one instance of an entity from another. All the properties in the model list are to be considered single-valued, and thus may serve collectively as a unique identifier for the instance to which they are attached.

A **sub-model** is a method for defining a property whose value is a multi-component tuple. Its definition is a list of property names within a "{ }". If there is an optional name in front of the leading "{", the whole tuple goes by that name as its property name. If there was no leading names, the tuple takes on the same name as the first property in the list.

7.1.3.1 Unique IDentifiers

Each data set that is mapped into a particular entity must has one property that is a unique identifer for that entity instance. The name of a property in a property list that fulfills this role is preceded by **UID** =, where **UID** stands for **Unique IDentifier**. Any such property is single-valued.

7.1.3.2 Null and Initial Values

An KEL property may also be designated as possibly having an explicit **null** value, that is "no value." In such cases, when entity instances are being read in, if some property is empty or missing, the property's value is flagged as **NULL**. The syntax = **NULL**(*value*?) after a property name says that if a particular instance of the related class of entities is being read in, if the value for that particular property is missing or empty, then if there is a *value* as the argument for **NULL** then that value replaces the empty value, and if not then the particular property field is marked as null.

7.1.4 Defining Association Classes

As with entities, a KEL program can give a name to a class of edge associations, again with properties permitted on each instance, via an **association declaration**:

$$\begin{aligned} \mathit{association\_declaration} \rightarrow \mathit{association\_class} :=& \textbf{ASSOCIATION} \\ & (\mathit{filetype}(\mathit{entity\_list}, \{\mathit{prop\_list}\}?)); \\ \mathit{entity} \rightarrow \mathit{entity\_classname} & \end{aligned}$$

Such declarations are equivalent to defining the domains of relations defining edge sets. The *entity_list* typically is a list of normally two definitions corresponding to the source and destination vertices of an edge. Each definition has the class of an entity to which a vertex in an edge of that type must belong, and a name to be used when a particular edge is referenced. There may also be a *prop_list* associated with each edge for such things as edge weights.

KEL also includes options on each entity to override field names.

7.1.5 Inputting Graphs

An entity declaration only defines what entities look like, not what they are. A **USE** statement tells a KEL program from which file to read in a graph and what kind of entities are found in it, with syntax as follows:

use_declaration → **USE** *file_name*(*filetype*, *entity_list*)

The *file_name* is the name of a file that holds part of the graph. The type of the file is as in an *entity_declaration*. The *entity_list* is a list of the entity classes from which instances will be derived from the file's contents. While normally such a list is a single *entity*, it is permissible to have multiple listing, in which the same data in the file is re-entered into sets of different entities.

As an example, the following sample code defines two classes of vertices (Actors and Movies), some properties of those vertex classes, and a set of edges (Appearance) where each edge starts at a Who vertex of class Actor, and ends at a What vertex which is of class Movie. Both sets of vertices and the edge set are drawn from the same file ("IMDB").

```
    Actor := ENTITY( FLAT(UID=ActorID, Actor=ActorName) );
Movie := ENTITY( FLAT(UID=MovieName, Title=MovieName) );
Appearance := ASSOCIATION( FLAT(Actor Who, Movie What) );
USE IMDB.File_Actors(FLAT, Actor, Movie, Appearance);
```

7.2 Assertion Statements

Once a set of vertices and edges have been defined as discussed above, **logic statements** in a KEL program define **assertions**: rules that define the logical relationships that a particular graph should exhibit in terms of its vertices, edges, and properties. When such assertions are "enforced," the result may be the addition to the graph of additional entities (either vertices or edges), or properties to such entities. These statements don't specify *how* or *when* such computations should be done, only *what* conditions should exist after any enforcement. Thus the order of placement of such statements in a program are irrelevant.

The high level syntax for such statements is approximately:

assertion → (*scope***:**)? *predicate_list*? => *production_list*
scope → **GLOBAL** | *entity*
production → *entity*(*argument_list*)
production → *propertyname* := *expression*

An assertion says that "if" all the individual *predicates* in the *predicate_list* are true, "then" so must be all the *productions* in the *production_list*. The => is thus termed an **implication**.

If there are no predicates, the implication is always true.

The *scope* prefix, if present, refers to the class of entities that are affected by the productions in this assertion. **GLOBAL** means that new instances of entities or associations will be created. The use of an entity name, if present, is the name of a class of entities that should be used as a default in any of the predicates.

7.2.1 Predicates

A *predicate* is a boolean expression that must be true for the assertion's productions to be in play. If no *predicate*s are present, the assertion is deemed unconditional. Typical forms of predicates include:

- a comparison (such as $=, \neq, <, \leq, >, \geq$) between the properties of entities and other values. This comparison must be true for the predicate to be true. The values of properties are specified by using the appropriate *propertyname*s.
- an edge of the form $association(u, v)$ where u and v refers to vertex entities. Optional values for properties may be included as tests. The referenced edge must exist between the specified vertices for such a predicate to be true.
- Boolean expressions built from combinations of the above, such as **NOT**, **AND**, **OR**.

When more than two arguments are placed in the argument list of an edge description, the additional arguments are values that must match properties of the specified edge.

7.2.2 Pattern Variables

A **pattern variable** is a mechanism to indirectly specify either a particular instance of an entity (usually those representing vertices) or the value of a property of an entity (either a vertex or edge). The name of a pattern variable starts with a #, followed by either a positive number (as in #1) or a character string (as in $\#level$). There may be an arbitrary number of different pattern variables used in an assertion. If a string of numeric digits are used as the pattern name, its numeric value has no relation to anything and is simply there for reference.

Also, the scope of a pattern variable is limited to a single statement. Thus if two statements each use the "same" pattern variable, then all the instances in one statement are treated totally independently of the instances in the other. It is as if there is an implied extension on each pattern variable's name that is the "name" of the statement.

Pattern variables may typically be used in assertions to couple values across either predicates or from predicates to productions. When used across multiple predicates, they imply that the predicates are true only for all cases where all instances of each pattern variable take on the same value at the same time. When used across predicates into productions, they indicate that a separate production should be performed for each possible value for the pattern variable that makes the predicates true. As one example consider the following where there are two classes of edges (i.e. two different associations) possible between entities that represent vertices:

GLOBAL : $Parent(\#1, \#2) => Ancestor(\#1, \#2);$
GLOBAL : $Ancestor(\#1, \#someone), Ancestor(\#someone, \#3)$
$=> Ancestor(\#1, \#3);$

In the first statement, if there is an edge of class Parent between any two vertices, then there must also be an edge between the same two vertices of class Ancestor. The second statement has three pattern variables, and generates as a production a new edge of class Ancestor for each possible set of values to those variables that simultaneously satisfy both predicates (i.e. when *someone* is both an ancestor of #1 and has #2 as an ancestor).

Pattern variables can also be used in simple comparison predicates, as in $\#1 = \#2$.

7.2.3 Productions

A production represents some characteristic of a graph that must be present if all the predicates were true. If at some point in time all the predicates turn true and the productions are not true at that time, then changes are made to the graph to match the set of productions.

One class of productions add entities to the graph. These are typically edges, as in the following:

$$Parent(\#1, \#2), Ancestor(\#2, \#3) => Ancestor(\#1, \#3); \tag{7.1}$$

This assertion extends the edges of an association set Ancestor if one does not exist already.

Another class of productions adds or modifies properties of entities by assigning a new value to a specific property name.

7.3 Queries

A **QUERY** statement in KEL specifies what it is that the programmer wants to see as output when the name of the query is specified by the user. Its syntax is:

$$query \rightarrow \textbf{QUERY} : queryname\{(argument\_list)\}? <= output\_expression\_list;$$
$$output\_expression \rightarrow entity\ \{(filter\_list)\}?\ \{\backslash\{projection\_list\backslash\}\}?$$

Query statements do not by themselves generate any computation, only specify what outputs are needed if the user invokes that query.

The query definition may optionally include a list of arguments which, when invoked by the user, may be given values which will be passed to the expression list, and thus influence what is output.

Each *output_expression* in an *output_expression_list* starts with the name of an entity (i.e. either a vertex set or edge set) from which the output will be drawn. This may be optionally followed by a comma delimited list of $filter$s, surrounded by "( )". Each of these filters is a boolean expression similar to predicates. When executed, the filter expressions are applied to each member of the specified entity set, and only instances that pass all the filters are part of the output. If there are no filter expressions, the entire set is output.

The individual $filter$ expressions may be as simple as tests on entity properties, or, for entities that represent vertices, tests involving properties of other vertices for which there is an edge (an association) connecting them. Tests of the first kind are akin to "selects" in relational databases; latter kind are akin to relational "joins".

A projection list, if present, is surrounded by "{ }", and is a list of properties from the entity's data model that are to be output after successful filters. This is equivalent to a relational "project."

7.4 Aggregates

KEL allows aggregation functions to appear as values in expressions in several parts of the language. They can take two forms based on where the expressions are being used, termed $propertyform$ and $actionform$.

7.4.1 Property Form Aggregates

The *propertyform* is used with a $ as a postfix to a property name to signal that the value to be used in the rest of the expression should be an aggregation of the values with that property name over some set of instances of an entity class. The variations in syntax specify what is the scope of instances over which the aggregation is performed:

$$\begin{aligned} aggregation\_Propertyform &\rightarrow \{self\_scope|outer\_scope|explicit\_scope\} \\ self\_scope &\rightarrow propertyname\$aggregation\_function \\ outer\_scope &\rightarrow propertyname\$\hat{}aggregation\_function \\ explicit\_scope &\rightarrow propertyname\$entity\{(filter\_list)\}? : aggregation\_function \end{aligned}$$

A **self-scoping** aggregation is one where the aggregation function follows a "$" after a *propertyname*, and means that the aggregation is done over all instance from the current entity class that have the same UID, and individually for each distinct UID value in the set. For example, in the following:

$$\textbf{QUERY}: AboveAveIncome <= Person(income >= income\$Ave)$$

This will partition the entities of class Person into subsets based on common UID values, and compute the average of the income property over each subset. Then, in the above query there will be a list of person records where the record had an income not less than the average for all such records with the same UID.

The **outer scope** form of aggregation performs the aggregation over the specified property of all instances of the class of entities named just outside the "()", as in:

$$\textbf{QUERY}: HighestIncome <= Person(income = income\$\hat{}Ave)$$

This will return the instances of Person that have their income equalling the highest value found over all instances of Person.

The **explicit scope** form of aggregation performs the aggregation over the specified property of all instances of the entity class specified after the $. This class may be different from the one from the outer scope. In addition, there may be a *filter_list* following the entity name that is applied to eliminate instances before performing the aggregation.

7.4.2 Action Form Aggregates

The action form of aggregation can be used anywhere within a normal expression to provide a normal value. The typical syntax is something like:

$$aggregation\_actionform \rightarrow aggregation\_function(entity\{(filter\_list)\}?, propertyname)$$

With this notation, the specified entity class is filtered by the optional *filter_list* and the remaining instances have their specified *propertyname* values given as input to the aggregation function.

7.4.3 Grouping

In addition to the above, additional grouping is available by using the keyword **GROUP** in the scope of an aggregation:

$$implicit\_grouping \rightarrow\ propertyname\$\mathbf{GROUP} : aggregation\_function$$
$$explicit\_grouping \rightarrow\ propertyname\$\mathbf{GROUP}\{grouping\_list\} : aggregation\_function$$

The implicit form performs a grouping on the values in the specified *propertyname* from the same class of entities as identified in the outer scope form.

The explicit form performa a grouping on the specified list of properties in the *grouping_list* before grouping further on the specified property to perform the aggregation.

7.5 Execution Model

A KEL program is declarative in nature, as assert statements specify that when certain conditions are reached (the predicates of an assertion statement all evaluate to true) then other conditions must also be true (the productions of the same assertion), without specifying *when or in what order those conditions are made true*. In addition, query statements specify only what values are to be output - not when in the computation they are available.

There are several viable execution models for KEL, but KEL was designed to run best on a combination of two different classes of physical parallel platforms, called **Thor** and **Roxie**. Thor-class systems are very large parallel clusters of state of the art compute blades, with commodity interconnect such as Infiniband and support for large parallel distributed file systems. They are designed for massive batch processing, akin to what is found in large Hadoop clusters.

Roxie systems are typically smaller clusters designed for online multi-user query execution applications. Roxie systems use data sets and associated indices computed by Thor clusters to execute the online queries.

A simplified version of KEL's execution model on such platforms starts with the identification of the entities that are needed by the **QUERY** statements. Working backwards from those entity sets, the KEL tool chain will identify all the statements that affect those entities, either assertion or **USE** statements. For assertion statements, this process is recursive, adding onto the tree all entities that show up in the assertions' predicates. Once this forest of computation is identified, a parallel program is constructed to compute all, and only, the queried entities. In addition, there is no prescribed order in which the predicates of each assertion or each parameter-free QUERY in the tree is checked. Thus the set of predicates present in all relevant assertions may themselves be reorganized into forests of predicates, so that redundant tests can be avoided and tests can be re-ordered and re-composed so that later predicates need search less cases. This overall program runs on a Thor system.

In addition, if any of the **QUERY**s include parameters, the Thor computations may include preparing indices against the referenced properties of the referenced entities. These computed entities and associated index sets are then made available to a Roxie program.

The KEL program may have identified other entities that are not referenced by the current set of queries - these need not be computed.

The Roxie half of a KEL program then consists of code to compute and format the output expressions for the **QUERY** statements from the Thor-provided entities. Thus, when a user specifies a specific **QUERY**s, the Roxie program will perform the necessary projections and formatting to

deliver the results. If there were parameters in the **QUERY**s, the code for the Roxie system will use the prepared indices to accelerate return of just the desired data.

The parallelization of the Thor half of the code takes advantage of the declarative nature of a KEL program. The files referenced by the **USE** statements are brought into Thor's parallel file system and distributed so that different partitions are on different nodes. Computation can then proceed in parallel on these different partitions, much as in a Hadoop system. Further, since KEL is declarative, other than the compositional order dictated by the functional flow through the derived functional tree, there is no requirement that any one step be completed before the next one can be started, implying that in addition to the parallelization from the partitioned data, the elements of each partition can be "streamed" through the tree in a pipelined fashion.

Much of this reordering is performed by the ECL compiler, as in the LexisNexis tool chain the KEL source code is converted first into ECL source. Given that ECL is itself not only declarative but also a highly functional language, such reorganizations are highly effective and guaranteed correct.

When pattern variables are used in multiple predicates of the same assertion, then typically this converts into a join-like operation. Varieties of multi-level parallel hash joins as described in [10] can often result in excellent speedup at near linear computational complexity.

7.6 Kernel Code

7.6.1 Jaccard Coefficients

The following code reflects an outline of a basic implementation of the Jaccard Coefficients of Section **??**.

$$
\begin{aligned}
Persons &:= \textbf{ENTITY}(...IntegerMyNeighbors, ...);\\
Addresses &:= \textbf{ENTITY}(...);\\
ResidesAt &:= \textbf{ASSOCIATION}(\textbf{FLAT}, Persons, Addresses);\\
Gamma &:= \textbf{ASSOCIATION}(\textbf{FLAT}, Persons, Persons, Jaccard);\\
Persons &: => MyNeighbors = COUNT(ResidesAt);\\
\textbf{GLOBAL} &: ResidesAt(\#1, \#3), ResidesAt(\#2, \#3)\\
&\quad => Gamma(\#1, \#2, COUNT(\#3)\\
&\quad /(\#1.MyNeighbors + \#2.MyNeighbors - COUNT(\#3));
\end{aligned}
$$

The first three statements simple define the entities that make up two classes of vertices, and the edge set that connects them. Each Persons vertex has a property named "MyNeighbors" that will hold the out-degree from that vertex for edges of type Addresses.

The fourth statement defines an edge set "Gamma" between two Persons vertices, with each edge instance having a "Jaccard" property.

The fifth statement is an assertion over all Persons vertices that for each vertex sets the MyNeighbors property to a count of the number of edges of type ResidesAt that leaves that vertex. This is the out-degree from each vertex. There are no predicates here, so the single production is applied to all Persons vertices.

The last statement is also an assertion. There are two predicates that test for the existence of edges from two vertices #1 and #2 to some vertices #3. The single production adds a new edge of type Gamma between the two vertices #1 and #2. The property Jaccard receives the computation

of the coefficient from the out-degrees of the two vertices #1 and #2 and the size of the common neighbors, again using an aggregation function $COUNT(\#3)$.

We note that the first two predicates in this assertion are true only if there is at least one vertex of class Addresses that is a common neighbor. This ensures that the $COUNT(\#3)$ is non-zero, and thus both the computation of Γ is both valid (i.e. not a $0/0$) and non-zero and only non-zero edges are added.

Optimizations to this might include adding a predicate to ensure that we add a Gamma edge only when "$\#1.UID < \#2.UID$, so that we compute only one half of the coefficients (since the other half is symmetrical). A second production could add the reverse edge without the recomputation.

$$\begin{aligned}\textbf{GLOBAL}: & \#1.UID < \#2.UID,\ ResidesAt(\#1, \#3),\ ResidesAt(\#2, \#3) \\ & => Gamma(\#1, \#2, COUNT(\#3) \\ & \quad /(\#1.MyNeighbors + \#2.MyNeighbors - COUNT(\#3)), \\ & Gamma(\#2, \#1, COUNT(\#3) \\ & \quad /(\#1.MyNeighbors + \#2.MyNeighbors - COUNT(\#3));\end{aligned}$$

An interesting question that may depend on the specific execution model of a KEL implementation is that in the above assertion, the computation $COUNT(\#3)$ (i.e. the computation of the γ) is repeated four times. If a compiler looks for common terms, it is likely to find this repetition and compute the value only once, rather that a four-way repetition. Alternatives may split the assertion in two, with the production from the first setting a property for an edge of class "SmallGamma", and the second assertion testing for the existence of such edges before doing the computations. Such an alternative would also allow **QUERYs** that provide a threshold on γ to avoid recomputations over the whole graph.

Chapter 8

Poplar

https://www.graphcore.ai/blog/what-does-machine-learning-look-like

Chapter 9

SPARQL and RDF

RDF (Resource Description Framework) [W3C 2014] is an open and widely used notation for describing entity-relationship data models that is the basis for multiple graph programming systems. Each entry in an RDF-compliant store is a triple that nominally represents some named edge between two named vertices. An RDF query language is then a language that allows queries into an RDF-compliant store to be expressed. There are multiple implementations of RDF stores, including ones that scale over multiple nodes, clusters, and the cloud. An open http-based protocol has been designed [W3Cb 2013] to standardize scaling in distributed systems. An example open implementation that scales to at least 32 nodes in a shared nothing cluster is 4store [4store 2015]. The most popular RDF query language is SPARQL [W3C 2013], with a notation that is SQL-like but focused on triples and the kinds of attribute values defined in RDF.

Chapter 10

Trinity

Graph Engine [Microsoft-a 2015] is a Microsoft Research project to implement a distributed graph engine that runs in the cloud. It was previously called Trinity [Shao 2013], and has a downloadable SDK [Microsoft-b 2015] based on Microsofts VisualStudio. The underlying Trinity RDF store [Zeng 2013] was designed for distribution of an RDF data set across a cluster or cloud of distributed web servers. It was also designed to support both batch and online queries. Storage is as graphs, not tables of triples, and supports in-memory storage of the graph partitions on each server. The package supports a method for replacing joins as they might appear from an SQL-like execution of SPARQL queries with graph explorations that follow edges rather than perform indexed operations on columns. Scaling studies in [Zeng 2013] range from a few 10s of millions up to a billion vertices, and indicate a significant performance gain over several other RDF systems for a benchmark discussed in Section 7.2. . Queries are in a declarative language called Trinity Specification Language (TSL) [Microsoft-c 2015] where programs are compiled into Microsofts .Net environment for execution in a distributed environment.

Part II

SECTION II: Graph Libraries

Chapter 11

GraphBLAS

As discussed in [9], and summarized in Section **??**, many graph algorithms can be re-stated as linear algebra problems involving large, sparse matrices. GraphBLAS[1] is a package designed to perform linear algebra operations similar to those found in standard numerically-oriented linear algebra packages, but with a few options that make them especially useful for performing graph processing:

- The normal floating point addition and multiplication operations may be generalized and replaced by programmer-defined functions as long as they have certain properties.
- GraphBLAS objects, especially matrices, are designed to support sparsity, that is a high percentage of object values may be **structural zeros**.

In addition, the GraphBLAS specification is designed to be particularly amenable to support compliant implementations on a variety of non-traditional architectures. Such features include:

- The memory used during computations specified by a series of GraphBLAS functions is *opaque* to that of the original calling thread, meaning that GraphBLAS computations are done on data structures that are inaccessible to the original caller. This enhances the ability of GraphBLAS implementations to seamlessly and invisibly support a variety of platform architectures with a potential wide diversity of memory architectures such as GPUs or multi-cores with two-level memories.
- Different but still compliant GraphBLAS implementations may have internal execution models different from conventional sequential call-at-a-time semantics, without any changes to the application source code. This increases the portability of a GraphBLAS-based application to run on potentially parallel systems with widely different architectures without re-coding.

The first specification for GraphBLAS defines a set of C language bindings [3] to a library that may be called by a C application program.

11.1 GraphBLAS Functions and Methods

11.1.1 Predefined Operators

There are a set of predefined operators that are defined over all basic data types from Section 11.2.1, including basic arithmetic operations, comparisons, max, and min. For booleans, there are

[1] see the GraphBLAS Forum at http://graphblas.org/

also a set of logical operations. The comparisons and booleans are particularly useful for creating masks (Section 11.2.5).

11.1.2 Defining Semirings

GraphBLAS allows a program to create a linear algebra-based code that can work over any C-expressible **semiring** of operators and any set of compatible domains. Notionally a semiring definition identifies two functions to be used in an inner product-like operation between two vectors, with one of the functions (the "multiply") corresponding to the element-by-element operation between the matching elements of the two vectors, and the other (the "addition") reducing the results of the multiplies down to a single value.

Such definitions are accomplished by a set of GraphBLAS calls that allow definitions of new semirings as follows:

- **GrB_Type_new**: creates a type for values in the context from a type known to the application program. Such types can then be used to define domains for context operators.
- **GrB_UnaryOp_new**: defines a new **unary operator** to be used in GraphBLAS context computations. The arguments include the types of the input and output domains, and a function pointer to a C function that performs the associated operator.
- **GrB_BinaryOp_new**: defines a new binary operator to be used in GraphBLAS context computations. The arguments include three types (the two input domains and the output domain), and a function pointer to a C function that performs the associated operator.
- **GrB_Monoid_new**: specifies that a previously defined binary operator is to be considered a new monoid within a GraphBLAS context. The arguments include a single domain to be assumed for both inputs and results, and an identity element.
- **GrB_Semiring_new**: specifies a pair of operators to be used as a new GraphBLAS semiring. Both must have been defined earlier. The output domain of the binary multiply function must be the same as the domain used by the monoid addition function.

11.2 GraphBLAS Objects

11.2.1 Basic Built-in Types

GraphBLAS includes a variety of standard data types for individual data values: integers, floating point, and booleans.

11.2.2 Zeros

GraphBLAS takes special care in how "zeros" are identified, and what they mean in computations. When used in an inner-product-like computation with functions defined by a semiring, a zero in one of the vectors always corresponds to the value in the multiply's domain that is the function's **annihilator**, i.e. the value returned from the multiply is itself a "zero." In turn this "zero" must then be the **identity value** for the addition function, so that its combination with any other value in the addition's domain returns that value.

11.2.3 Vectors, Matrices, and Sparsity

The two major compound objects supported by GraphBLAS within a GraphBLAS context are one and two dimensional arrays termed respectively **vectors** and **matrices**. Their creation in a context occurs in two steps: first definition of their type and size via a **GrB_xxx_new** call (where "xxx" is either *Vector* or *Matrix*, and second, the population of values via either direct transfers from application space (Section **??**) or the results of GraphBLAS computations. Note that a created object has a size but is assumed filled with all zeros until it is populated.

The creating application can refer to a created object via a handle in application space, but cannot directly access any values in context space. Only through GraphBLAS calls can values be accessed.

While their dimensions are fixed when they are created, neither their values nor locations in memory are fixed. Unlike conventional languages, there is no assumed layout in memory for each structure that guarantees that element i (or (i,j)) is in some specific location relative to the origin of the structure. Instead, from the application program's view, a GraphBLAS vector or matrix is defined as a set of (index, value) pairs, with absolutely no guaranteed relationship to memory locations. If the current set for an object does not have an (index, value) pair for some element in an array, then that element is "undefined." Likewise a set for an object cannot have two values for the same index.

Notionally, undefined elements in an array correspond to some sort of structural zero as defined in Section 11.2.2. Arrays with large numbers of such elements thus correspond to sparse structures, and often can be stored in far more compact forms than conventional arrays that allocate memory to each element. Examples of such formats include **CSR** (**Compact Sparse Rows**), **CSC** (**Compact Sparse Columns**), or the structures defined by packages such as Stinger (Section 17). In GraphBLAS, however, the exact form is hidden from the application programmer, and is implementation-dependent.

While undefined elements are notionally structural zeros, unlike many conventional sparse representations, GraphBLAS makes no assumption about what their actual value is. Instead, the value of undefined elements is left up to the methods applied to the objects. In many cases, this assumed value is the value of the additive identity in the semiring of functions assumed by the executing method.

11.2.4 Index Arrays

GraphBLAS includes a concrete type (i.e. a type for an object stored in the caller's memory) called **GrB_Index** which is a pointer to a contiguous array of 64-bit uints. When passed to a method call, the associated array may be used as a set of indices into an opaque vector or matrix in the context.

11.2.5 Masks

Very often an irregular subset of a vector or matrix is desired. GraphBLAS has the ability to define **masks** as vectors or matrices of type *bool* of length equal to the dimensions of the original vector or matrix. Applying a mask to an object of equivalent dimension is equivalent to specifying that any element in the object whose index is matched by a *false* in the mask set is to be treated as a structural zero, regardless of its value.

The **structural complement** of a mask flips the boolean value of each element.

11.3 GraphBLAS Contexts

A GraphBLAS application has two memory spaces: the normal one associated with the process running the application, and a logically separate one, called the GraphBLAS **context**, that is accessed only by the GraphBLAS routines and is not visible directly to the original calling program. This separation of address spaces is done to optimize for systems whose architectures have multiple memory levels (such as GPUs or systems with HBM[2]) to keep GraphBLAS data in a memory that is both physically and logically separate from that of the application program.

11.3.1 Context Management

The current GraphBLAS spec permits initialization of exactly one GraphBLAS context by a single **GrB_init** call at the beginning of the application program. This call can have only one argument specifying the high level execution semantics allowed during the execution of later GraphBLAS method calls, as described in Section 11.5.

When a GraphBLAS implementation supports multi-level memories, accelerators, or other possibly parallel execution engines, this initialization of a context will most probably allocate some physical resources to the current process. Releasing these resources is done after the last GraphBLAS call is known to be completed, and is done via a **GrB_finalize** call.

Exactly one of each of these calls may be executed in a GraphBLAS-compliant application.

11.3.2 Object Management

GraphBLAS objects may have their internal representation saved in a memory space not accessible to the calling application program, but that program still needs to be able to refer to them in the specification of functions to be applied to them. This is done by executing calls to a variety of GraphBLAS routines that use handles returned by prior calls that created the object (Section 11.2.3):

- **GrB_xxx_new**: creates space for a new object (vector or matrix) of some specified uniform component type in the context, but does not assign values to it.
- **GrB_xxx_dup**: creates space for a new object (vector or matrix) in the context that is a duplicate of some other prior object, including that objects's current values.
- **GrB_free** with a valid handle as an argument frees all storage currently associated with the object in the context designated by the handle.
- Functions are included to return the number of values currently stored in a vector or matrix, and for matrices the number of rows or columns. Depending on the implementation, the number of values in an object may or may not equal the size. The difference may be the number of structural zeros as discussed above.

11.3.3 Caller to Context Memory Transfers

At some point, data must flow from the application space to the context, and computed values returned from the context into application space. Since the application program cannot directly

[2]HBM stands for "High Bandwidth Memory" and functions in many new systems as a high speed memory that is closer to a processor's core than main memory, and which may be configured as a region of memory that is separate from the conventional DRAM.

address into the context, GraphBLAS routines are available to perform such transfers. The following provide data transfers from application space into a previously defined context object (as before there are two variants of each of the following, where "xxx" is either **Vector** or **Matrix**);

- **GrB_xxx_clear**: removes all elements from an object in the context.
- **GrB_xxx_setElement**: takes an (index, value) pair (for a vector) or a double index and value (for a matrix) from application space, and changes the entry in the context vector corresponding to that index to the specified value.
- **GrB_xxx_build**: takes one or two vectors of indices and a vector of values from application space, and updates the corresponding entries in a context object.

11.3.4 Context to Caller Memory Transfers

The following then provide transfers from objects in context space back to application space memory. This is called **materializing** the computation:

- **GrB_xxx_extractElement**: returns the value associated with a specified index in an object in context space.
- **GrB_xxx_extractTuples**: stores in the caller's space two (or three for a matrix) equal-length vectors whose contexts are all the indices from the context object that are not structural zeros, and all the corresponding values.

11.4 GraphBLAS Operations

Operations in GraphBLAS are performed on objects identified by handles and stored in the current context, with results returned to other objects in the same context. Table 11.1 summarizes the major calls, their key operands, and the basic operation they perform.

For all these calls, there are typically a variety of options that can be specified by three additional operands in each call:

- An **accumulation function** that if specified, causes each value computed during the computation to be combined with the prior value in the target before being stored into the target location. This is equivalent to replacing $C =< expression >$ by $C+ =< expression >$, where "+" is the specified function.

  If the i'th element of both the target and the computed result are not structural zeros, then the final value is the result of applying the function to both values. If either one, but not both, are structural zeros, the result is the non-zero value. If both are structural zeros, so is the result.

- A **mask** which, as described in Section 11.2.4, is a boolean vector in the context whose dimensions match that of the target object. If a mask is specified, a false value in some position of the mask indicates that the corresponding element of the target shall not be changed by the call.

  If the accumulation option is included, the mask is applied after computing the accumulation.

- A **descriptor** is provided to each call as a separate argument to optionally modify the execution of the called operation, including:

Method	Target	Arg 1	Arg 2	Arg 3	Description	Page
mxm	C: matrix	A: matrix	B: matrix		$C[i,j] = A[i,*] \ \oplus . \otimes \ B[*,j]$	69
vxm	W: vector	U: vector	A: matrix		$W[j] = U \oplus . \otimes \ A[*,j]$	73
mxv	W: vector	A: matrix	U: vector		$W[i] = A[i,*] \oplus . \otimes \ U$	77
eWiseMult	W: vector	U: vector	V: vector		$W[i] = U[i] \ \otimes \ V[i]$	82
eWiseMult	C: matrix	A: matrix	B: matrix		$C[i,j] = A[i,j] \ \otimes \ B[i,j]$	86
eWiseAdd	W: vector	U: vector	V: vector		$W[i] = U[i] \ \oplus \ V[i]$	91
eWiseAdd	C: matrix	A: matrix	B: matrix		$C[i,j] = A[i,j] \ \oplus \ B[i,j]$95	
extract	W: vector	U: vector	I: Index		$W[i] = (U[I])[i]$	100
extract	C: matrix	A: matrix	I_R: index	I_C: index	$C[i,j] = (A[I_R, I_C])[i,j]$	104
extract	W: vector	A: matrix	I_R: index	J: uint	$W[i] = (A[I_R, J])[i]$	110
assign	W:vector	U: vector	I: index		$(W[I])[i] = \ U[i]$	114
assign	C: matrix	A: matrix	I_R: index	I_C: index	$(C[I_R, I_C])[i,j] = A[i,j]$	119
assign	C: matrix	U: vector	I_R: index	J: int	$(C[I_R, J])[i] = U[i]$	124
assign	C: matrix	U: vector	I: int	I_C: index	$(C[I, I_C])[i] = U[i]$	129
assign	W: vector	v: value	I: index		$W[I] = v$	133
assign	C: matrix	v: value	I_R: index	I_C: index	$C[I_R, I_C] = v$	138
apply	W: vector	f: function	U: vector		$W[i] = f(U[i])$	143
apply	C: matrix	f: function	A: matrix		$W[i,j] = f(A[i,j])$	147
reduce	W: vector	f: function	A: matrix		$W[i] = \ f/A[i,*]$	147
reduce	v: variable	f: function	U: vector		$v = \ f/U$	151
reduce	v: variable	f: function	A: matrix		$v = \ f/f/A[i,]$	155
transpose	C: matrix	A: matrix			$C = \ A^T$	160

Page numbers are relative to "The GraphBLAS C API Spec," Version 1.0.0, 05/29/2017
"Vectors" and "matrices" are in the GraphBLAS context.
An "index" is an array of 64b uints in caller's memory.
A "value" or "variable" is a scalar in the caller's memory.
"M" is an optional write index set used as a mask on the target.
"$\oplus$" is the additive operator from the specified semiring.
"$\otimes$" is the multiplicative operator from the specified semiring.
"$\oplus.\otimes$" is an inner product using the operators from the semiring.
"$\odot$" is an optional accumulating operator.
An operator by itself is applied element-by-element
The expression "f/" refers to the summation across all values in the operand using function f.

Table 11.1: GraphBLAS Methods.

- If a mask is specified, use the logical complement of the mask (a true in the mask becomes a false, and vice versa).
- For a particular input operand, if it is a matrix, use its transpose.
- For the target object, clear all its elements first.

11.5 GraphBLAS Execution Model

The execution of a GraphBLAS program starts with a GL_init (which builds a context), and ends with a GR_finalize (which frees up all resources). In between are a series of calls to other GraphBLAS routines which transfer data from the caller's memory into the context, do processing of that data, and extract results back out of that context. The GraphBLAS definition then allows for a variety of possible execution models, both at an **inter-call** and an **intra-call** basis. What models are permitted depends on the **execution mode** the GraphBLAS program is in (which in turn was established by the initial call to GR_init that created the context):

- In **blocking mode** all GraphBLAS method calls must run to completion before the calling program can resume. This is normal C semantics for C function calls.
- In **non-blocking mode** it is permitted for GraphBLAS method calls to all the calling program to resume even if the execution of the call has not yet completed.
- In **default mode** the program will use whatever mode is default for the given implementation.

11.5.1 Intra-Call Execution

From the intra-call basis, the internal execution of a single call to a GraphBLAS routine may be thought of as occurring in at least three possible steps:

- **Initialization**: Execution of the entry code for the call, including checking of the arguments and associated objects for validity and consistency, and accessing from the caller's space of any values that are needed during the rest of the call.
- **Computation**: In this phase, all computation required by the call on the opaque objects in the context are carried out, and whatever values need to be saved in opaque objects are saved. In most cases, completion of this step corresponds to completion of the call.
- **Materialization**: For some GraphBLAS calls, data must be transferred back from the context to the caller's space. Only when this is complete can the call be said to be fully complete, and the calling thread have any access to the computed values.

  The major calls that perform any such materialization are those discussed in Section 11.3.4. Also, only a few of the reduce methods from Table 11.1 materialize any data in this way.

In blocking mode, all three steps must be complete before the calling thread is given back control, and may continue. From the caller's perspective, the routine was executed sequentially, even though in real life at least the Computation phase may have been done in parallel.

11.5.2 Inter-Call Execution

In non-blocking mode there are many more possible execution models. For many calls, a call to a GraphBLAS routine may in fact return before the call has fully completed.

From the calling thread's perspective, at least the initialization phase of each call must be completed, so as to ensure that all information needed by the last two phases of the call is copied out of the caller's memory, so that no change by the caller to these objects can possibly affect the execution of the now-prior call. This is particularly true of the values in indexes and descriptors. However, with few exceptions, there is nothing that demands any particular order during the Computation phases of the sequence of called routines. This may range from simply enqueuing the work of each routine, and then executing each Computation step in order, but internally in parallel.

In contrast, a runtime analysis of opaque object usage may allow construction of a flow graph of what Computation step needs data from what other step, and allow multiple Computation phases that have no data dependencies to execute in parallel (perhaps each internally in parallel also).

Further, it may also be possible to stream computation across Computation steps, so that intermediate data sets are never computed in their entirety before starting a dependent execution. Instead, computation may start with streaming elements of an initial object piecemeal through the functions for one step to generate the elements of the output object that can continue be fed piecemeal through following Computation steps from other calls.

Finally, such execution may be done lazily, so that no computation is started until it is known that some later object that depends on that computation has been requested to be extracted from the context back to the caller's memory. In fact, if no future call ever references some output, no earlier computation on which that output depends need ever be done.

The semantics of GraphBLAS in fact explicitly allow this latter approach by specifying that only when a call is made to a routine that performs an extraction is there any guarantee that any internal computation is performed. In this case, there is a guarantee that at least the path of computation required for this extracted data is run to completion.

11.5.3 Return Codes

Each GraphBLAS call returns a code to the caller indicating the success or failure of the call. For blocking execution modes, this success or failure includes all possible error conditions that may have occurred during any of the three phases of execution. However, for non-blocking modes, the return code from a call may reflect only the status of execution through the Initial phase of the call, and it is only when a materialization is performed is an overall status of that computation path returned.

Chapter 12

GraphChi

GraphChi [Kyrola 2012] is a C++ open source library for building asynchronous vertex-centric algorithms that iterate over all graph vertices, using most recent property values rather than only some value from the previous iteration. It is also designed to handle streaming of graph updates. It assumes an out-of-core model where a subgraph is loaded from disk, the vertices and edges updated, and the results written back to disk. Vertices are partitioned into intervals, and edges partitioned into shards, all of which have their destinations in some interval (and which are sorted by their source vertices). Vertices are then processed one interval at a time, sliding a window down each of the other shards to pick up the out-edges from each vertex. To perform some application, the programmer provided a vertex update function. While scalability seems to be limited to a small number of cores, performance on large graphs seems to rival that of other system running on large clusters.

Chapter 13

GraphLab

GraphLab [Low 2010] is an open source framework to support scalable machine learning applications characterized by sparsity and asynchronous iterative computations. It was designed to scale from multi-core shared memory nodes up through distributed systems. Its data model has two parts: graphs where arbitrary data structures can be associated with both vertices and edges, and shared data tables that are represented as key-value pairs. A programmer can define update functions to be applied over neighborhoods of vertices, and sync functions which permit updates into the shared data tables.

Chapter 14

NetworkX

"NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks."[1]

[1]Taken from https://networkx.github.io/

Chapter 15

Parallel Boost Graph Library

Parallel Boost Graph Library (Parallel BGL) [Gregor 2005] is an open source C++ library that supports generic graph algorithms constructed to work on distributed graph structures to which a variety of vertex or edge data may be attached. Programs constructed using this library are typically SPMD/BSP in nature, with the library hiding the underlying communication needed for inter-node data exchange and synchronization.

Chapter 16

SNAP

SNAP[1] (Stanford Network Analysis Project) is a library developed to provide a general purpose network analysis and graph mining library. There are versions wrritten in C++ and Python. The SNAP website also includes over 50 large graphs available for test cases.

[1] http://snap.stanford.edu/

Chapter 17

Stringer

Stinger [Bader 2009] is an extensible data structure and associated API that was designed to store graphs that may receive frequent updates of new edges, vertices, or both. It was designed to be scalable, especially when created within a common address space. It allows properties to be added to both vertices and edges, and allows searches to be made through both vertices and edges of certain types. It was designed to support concurrent queries. Edges in particular can be time-tagged, allowing for both temporal searches and for simplified aging-out functionality. The current version supports additional functionality for streaming applications [Ediger 2012]. Stingers code base is available from Github under a 3-part BSD license, with documentation at [Bader 2015].

Chapter 18

System G

System G is an IBM Research project [Canim 2013] that combines a graph analytics libraries (largely in C++) developed on top of a graph database GBASE [Kang 2012] and a middleware API [Ekanadham 2014]. It was designed to support on scalable platforms a wide variety of graph applications. An open source download is available, along with source code on github.

Part III

SECTION III: Graph Systems

Chapter 19

DisNet

Chapter 20

FlockDB

FlockDB and Gizzard [Twitter 2015] are open source packages by Twitter to allow queries into twitter data. FlockDB is a distributed graph database designed for graphs where vertices have potentially very large out-degrees, but where updates come into the graph very frequently, where the typical queries do not require very deep hop traversals, and where there may be a very large number of independent queries at any one time. In particular, queries that might be handled by some sort of adjacency bit-vector like processing are particularly well-supported. Gizzard is a library and middleware layer that provides functions to manage the ingestion and partitioning of data that ends up in a FlockDB database.

Chapter 21

GEMS

GEMS [Castellana 2015] [Morari 2015] is a multi-threaded RDF database developed at Pacific Northwest National Lab (PNNL) that includes a SPARQL front end. It combines a highly specialized multi-threaded runtime with a PGAS (Partitioned Global Address Space) execution model that allows scaling across potentially many nodes. Experiments have been run on systems with up to 602 dual 16-core socket nodes, using the Berlin benchmarks, with graphs of 100M, 1B, and 10B edges. PNNL is one of the collaborators on this project, and intends on supporting scaling studies of benchmarks and mini-apps written using GEMS. A second-generation GEMS is planned during this project, and in fact scaling results from this project will be taken into account in its design.

Chapter 22

Graph Engine

Graph Engine [Microsoft-a 2015] is a Microsoft Research project to implement a distributed graph engine that runs in the cloud. It was previously called Trinity [Shao 2013], and has a downloadable SDK [Microsoft-b 2015] based on Microsofts VisualStudio. The underlying Trinity RDF store [Zeng 2013] was designed for distribution of an RDF data set across a cluster or cloud of distributed web servers. It was also designed to support both batch and online queries. Storage is as graphs, not tables of triples, and supports in-memory storage of the graph partitions on each server. The package supports a method for replacing joins as they might appear from an SQL-like execution of SPARQL queries with graph explorations that follow edges rather than perform indexed operations on columns. Scaling studies in [Zeng 2013] range from a few 10s of millions up to a billion vertices, and indicate a significant performance gain over several other RDF systems for a benchmark discussed in Section 7.2. . Queries are in a declarative language called Trinity Specification Language (TSL) [Microsoft-c 2015] where programs are compiled into Microsofts .Net environment for execution in a distributed environment.

Chapter 23

Graphulo

Chapter 24

HyperGraphDB

HyperGraphDB [Iordanov 2010] is a Java-based graph storage system for hypergraphs graphs where edges can have multiple destinations. The system also provides an API for querying such hypergraphs. It supports a distributed implementation using an agent-based peer-to-peer framework.

Chapter 25

JENA

JENA [Apache 2015a] is an open source JAVA-based project from the Apache Foundation that includes an implementation of SPARQL, called ARQ [Apache 2015b]. Its associated RDF store is called TDB [Apache 2015c], is also open source, but is currently limited to a single node, with query access not allowed from more than JVM at a time.

Chapter 26

Neo4j

Neo4j [Neo 2015] [Robinson 2013] is one of the most widely referenced graph database packages. It has both a graph database engine and a query language CYPHER. Datasets are described as property graphs where both vertices and edges can have attached arbitrary lists of (key, value) pairs representing properties. It is written in Java and Scala, with source code available on Github, and includes a separate API that allows direct access to the graph engine. The underlying query engine is multi-threaded and has been demonstrated on shared memory systems with 100s of cores. An effort is under way (http://www.opencypher.org/) to produce an open source version of CYPHER.

Chapter 27

Pregel

Pregel [Malewicz 2010] is a graph processing system built from a C++ library that was designed to scale well on distributed message-passing systems. It provides mechanisms for treating vertices and edges as first class objects, with attributes as needed, and distributes vertices over platform nodes. Its basic execution model proceeds in supersteps, where each superstep starts with a user-defined function being executed on each vertex. Such functions may read messages sent to it in the prior superstep, modify the attributes of a vertex, send messages to any other vertex, or may signal a deactivate for future processing (participate in no more supersteps if there are no more incoming messages). These supersteps are executed until all vertices have deactivated. Vertex information is kept in memory between supersteps, and messages may be combined on each node so that multiple vertex-to-vertex messages from one node to another can be sent as one physical message. Aggregators such as sum, max, and min are available to combine values from all these vertex function executions.

Chapter 28

Powergraph

[7]

Chapter 29

GraphX, Scala, Spark

GraphX is a graph processing library that combines two different views of graphs: as tables and as graph data structures with vertices and edges as classes. The former represents vertices and their properties as one set of tables, and edges as separate tables. Both kinds of tables are designed to be partitioned and distributed among different nodes of distributed computing systems.

GraphX is built on top of the programming language **Scala**, which in turn part of the **Spark** programming paradigm.

The following sections discuss first the Spark parallel framework, then Scala, and then GraphX.

29.1 Spark

Spark [19] is a distributed computing framework designed for high scalability in data-intensive applications, and addresses two issues present in MapReduce implementations: the need in many applications for iterations, and the desire to be able to do interactive analyses. While not explicitly graph-related, Spark has been drawing such a growing user base that it is worth understanding the characteristics that are of relevance to graph engines. Spark, like KEL, defines intermediate data sets declaratively. Also, in contrast to MapReduce, a Spark programmer can specify that these intermediate datasets be left in memory if possible, avoiding unnecessary flush to disk and then re-read. Further, datasets can be marked as RDD ("Resilient Distributed Datasets")[18] that are read-only, partitioned and distributed across nodes, and kept with enough redundant information so that if a node goes down, the partition that is lost can be recreated without affecting computations on other nodes. In addition, Spark supports two kinds of shared variables: Broadcast where copies can be made locally on each node, and Accumulator for global associative operations. In iterative Machine Learning applications these give Spark a reported 10X advantage over MapReduce. Spark, being written in Scala, also includes an interpreter mode, allowing dynamic queries to be written and run in an interactive mode. Finally, new "streaming" frameworks are being developed on top of Spark (c.f. [20]), which may be directly relevant to supporting some of the benchmarks that will come out of this project.

29.2 Scala

[14]

29.3 GraphX

[17, 16, 8]

Part IV

SECTION IV: Comparisons

Chapter 30

Comparative Summary

Table 30.1 summarizes a feature comparison for the different systems discussed in this report.

The features compared include the following:

- What characteristics can be specified for vertices:
  - **Vertex Classes**: vertices can be marked as being members of different classes.
  - **Vertex Properties**: vertices may have different properties that do not need to be specified by edges.
- What characteristics can be specified for edges:
  - **Edge Classes**: edges can be marked as being members of different classes.
  - **Edge Properties**: edges may have different properties that do not need to be specified by edges.
- **Multi-value Properties**: for those systems that have properties, a property for a particular instance may have multiple values at the same time.
- **Graph Persistence**: can graphs be left in memory independent of individual applications.
- Data Input:
  - **Bulk Input**: vertices and edges can be read in from files.
  - **Incremental Input**: vertices and edges may be input from outside in a dynamic fashion.
  - **Dynamic Create**: new instances of vertices and edges may be created dynamically during program execution.
  - **Dynamic Modify**: existing instances of vertices and edges may be modified dynamically during program execution. This is particularly true of their property values.
  - **Dynamic Delete**: existing instances of vertices and edges, and/or their properties, may be deleted from the graph dynamically during program execution.
- If an application can use a select-like expression to control what part of a computed graph can be used either within a computation or in describing what part of a graph should be output, what features of that graph can be tested to determine what goes forward:
  - **Select: class**: specify the specific class(es) that a vertex or edge must have.

Feature	Languages						
	Accumulo	Cypher	Graphlab	Gremlin	KEL	SPARQL	Trinity
Vertex Classes					Y		
Vertex Properties					Y		
Edge Classes					Y		
Edge Properties					Y		
Multi-value Properties					Y		
Graph Persistence					N		
Bulk Input:					Y		
Incremental Input							
Dynamic Create					Y		
Dynamic Modify					Y		
Dynamic Delete							
Select: class					Y		
Select: property					Y		
Select: property value					Y		
Support for Grouping					Y		
Support for Aggregates					Y		
Inherent Transitive Closure					Y		

Figure 30.1: Feature Comparisons.

  - **Select: property**: specify the specific property(ies) that a vertex or edge must have.
  - **Select: property value**: specify the value(s) of specific property(ies) that a vertex or edge must have.

- **Inherent Transistive Closure**: repeated application of edge following rules may be done without explicit programming.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] Jacob Bank and Benjamin Cole. Calculating the jaccard similarity coefficient with map reduce for entity pairs in wikipedia. Wikipedia Similarity Team, Dec. 16 2008.

[3] Aydin Buluc, Timothy Mattson, Scott McMillan, Jose Moreira, and Carl Yang. The GraphBLAS C API Specification: Review Draft Version 0.9.3. http://gauss.cs.ucsb.edu/ aydin/-GraphBLAS_API_C.pdf, April 2017.

[4] Paul Burkhardt. Asking hard graph questions: Beyond watson: Predictive analytics and big data. Beyond Watson Workshop, Feb. 2014.

[5] V. G. Castellana, A. Morari, J. Weaver, A. Tumeo, D. Haglin, O. Villa, and J. Feo. In-memory graph databases for web-scale data. *Computer*, 48(3):24–35, Mar 2015.

[6] Daniel Chavarria-Miranda, Vito Giovanni Castellana, Alessandro Morari, David Haglin, and John Feo. Graql: A query language for high-performance attributed graph databases. *Workshop on Parallel and Distributed Computing for Large Scale Machine Learning and Big Data Analytics (ParLearning'2016)*, May 27, 2016.

[7] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[8] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 599–613, Berkeley, CA, USA, 2014. USENIX Association.

[9] J. Kepner and J. Gilbert. *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial and Applied Mathematics, 2011.

[10] P.M. Kogge and D.A. Bayliss. Comparative performance analysis of a big data nora problem on a variety of architectures. In *Collaboration Technologies and Systems (CTS), 2013 International Conference on*, pages 22–34, 2013.

[11] LexisNexis. *ECL Programmers Guide*. HPCC Systems, Boca Raton, FL, 3.10.4.1 edition, 2013.

[12] LexisNexis Risk Solutions. Kel language reference version 5.4.2, 2015.

[13] Alessandro Morari, Vito Giovanni Castellana, Oreste Villa, Antonino Tumeo, Jesse Weaver, David Haglin, Sutanay Choudhury, and John Feo. Scaling semantic graph databases in size and performance. *IEEE Micro*, 34(4):16–26, 2014.

[14] Martin Odersky, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Matthias Zenger, and et al. An overview of the scala programming language. Technical report, 2004.

[15] Tom White. *Hadoop: the Definitive Guide*. O'Reilly Media, 2011.

[16] Reynold S. Xin, Daniel Crankshaw, Ankur Dave, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. Graphx: Unifying data-parallel and graph-parallel analytics. *CoRR*, abs/1402.2394, 2014.

[17] 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.

[18] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[19] 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.

[20] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.

Index