Operational Amortization of Algorithmic Performance in a Graph Database

Frank W. Miller
Dept. of Computer Science
University of Colorado Boulder

Irene L. Beckman
Google Boulder

ABSTRACT

Scaling the performance of graph database engines executing graph algorithms on large datasets has been a challenge. This work presents a formal approach to the design of graph databases that yields amortized benefits for the execution of graph algorithms. The first result is the ability to support O(*n*) connectedness tests for large arbitrary graph datasets. This result is enabled by maintaining general graphs in a schema based decomposed representation. Many familiar relational database concepts map cleanly to this model including functional dependencies, a normal form, an algebra, and opportunities for optimizations based on both query semantics and performance and/or distribution requirements.

1. INTRODUCTION

This paper presents a model for database implementation based on a graph abstract data type. The approach has several goals.

The first goal is to address the performance of graph algorithm implementations as the graph datasets increase in size. This approach maintains graphs in a specific internal representation based on common schemas. The model allows for O(*n*) connectedness tests across the heterogeneous graphs by performing simple set intersections on a set of derived connected components. The ability to perform these tests at less than the O(*n*log*n*) limit is enabled by a relatively expensive (i.e. > O(*n*log*n*)) import operation that stores the graphs persistently as a set of connected components with common schemas. This import operation is thus amortized over many connectedness tests. We believe this technique can be generalized to other graph algorithms to improve their operational performance as well at the cost of the import.

A second goal is to try to put graph databases on a theoretical foundation in a manner analogous to that described for the relational model [1]. The specific internal representation we propose enables a set of well-defined operations that yield an algebra for manipulating graph components that is analogous to the relational algebra. It also takes into account semantic relationships in a manner analogous to functional dependencies in the relational model.

A third goal is to serve as design for a high-performance implementation approach. By dividing general graphs into sets of components with uniform schemas, low level implementations will be encouraged to store data in contiguous, homogeneous blocks. The ability to divide and group the data this way can catalyze the optimization of database features using already developed techniques from relational database experiences [11].

Consider the general description of a graph $G = (V, E)$ where V is the set of vertices and E is the set of edges in the graph. A graph database associates user data with the vertices and edges of graphs. The data associated with vertices is structured using vertex schemas (similar to the schemas defined for relations). User data is associated with the edges of the graph using edge schemas.

In the most general graph, no restrictions are placed on the vertex and edge schemas. Data can be hung arbitrarily from any vertex or edge. This means that different vertices can have different vertex schemas and different edges can have different edge schemas.

In this approach, such a general graph model is losslessly decomposed into a set of connected components where for each component, the vertex schemas are all identical and the edge schemas are all identical. This paper refers to this graph form as *normal form*. Conversions between the general graph model and normal form are described in Section 2.3.1.

2. DATABASE MODEL

The basic graph definition is extended for our graph database model as follows:

A *database* is a set of graphs, $D = \{G_1, G_2, \ldots\}$. Each *graph*, is a set of connected components, $Gi = C_i = \{c_1, _2, \ldots\}$. All the components in a graph share a common vertex id space, i.e. some vertex ids may exist in multiple components within the same graph.

The definition of connected component used in this here is specific, adhering to the definition of a *Common Schema Tree (CST)* which is given later. For the moment, the important property is that these components can be written as:

$$c_j = \{V, S_V, E, S_E\}$$

where $S_v = \{A_1, A_2, \ldots\}$ is the set of attributes associated with the vertex schema and $S_E = \{B_1, B_2, \ldots\}$ is the set of attributes associated with the edge schema. Each vertex has a unique id associated with it that is not part of its schema.

Figure 1 provides an example general graph. Each vertex has a vertex id. The vertices and edges also have arbitrary schemas associated with them. For example, vertex 2 contains a tuple that has attributes A and B and the directed edge from vertex 2 to vertex 3 has attributes C and D.

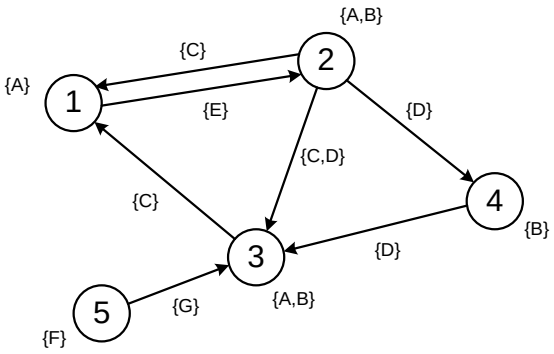

Figure 1: A General Form Graph

Using a notation where a different schema can be associated with each vertex and each edge, this graph can be represented as:

G = { {1{A},2{A,B},3,{A,B},4{B},5{F}},
{(1,2){E},(2,1){C},(2,3){C,D},(2,4){D},
(3,1){C},(4,3){D},(5,3){G} } }

Figure 2 illustrates the same graph after it has been decomposed into one instance of a normal form. Note that each component is connected and that some of the vertex ids occur in multiple components. In addition, each component has the same set of attributes for its vertices and the same set of attributes for its edges.

Using the notation introduced earlier, this normal form graph instance can be written as:

G = *C* = {{{1,2,3}, {*A*}, {(1,2),(2,3),(3,1)}, {*C*}},
{{2,3,4}, {*B*}, {(2,3),(2,4),(4,3)}, {*D*}},
{{1,2}, {}, {(1,2)}, {E}},
{{3,5}, {}, {(3,5)}, {G}},
{{5}, {F}, {}, { } } }

Since each of the components contains tuples that share the same schemas, they are kept on secondary storage together. The goal is to partition the graph data in a logical way to facilitate various features that perform better in the presence of locality. These range from basic functions like searches based on attribute conditions [5,6] to higher level functions like sharding for reliability and performance in a distributed implementations [12,13].

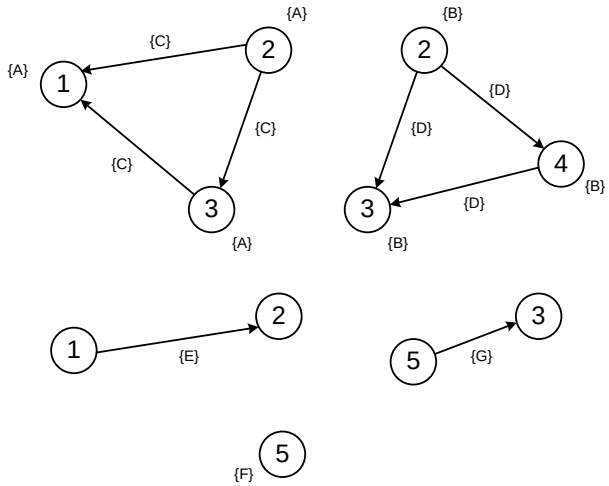

Figure 2: An Instance of a Normal Form Graph

2.1 The First Graph Algorithm: Connectedness

The first contribution associated with this paper beyond the model itself is that we can potentially exploit the structure of this decomposition to improve graph algorithm performance operationally on large datasets. Perhaps the most basic graph algorithm is the test for whether two nodes, v_1 and v_2, are connected. This test is typically done by executing the aforementioned Dijkstra's algorithm on the graph, or by a Breadth or Depth First Search [3]. The best known running time for these algorithms is O(*n*log*n*) where *n* is the number of vertices in the general graph.

The decomposition just described yields the ability to determine connectedness between vertices across components in O(*n*) time. This result is made possible by the relatively expensive graph import operation. The work associated with the expensive import is thus amortized across potentially many graph connectedness test operations.

There are a number of approaches to accomplishing set intersections in O(*n*) time, e.g. [14]. Hashing one of the input sets is a common technique. To take advantage of this performance result, we need to show that our set of connected components can be used to perform the general connectedness test using just a set intersection.

The proof is given for any arbitrary general graph, *G*. The overall proof is dependent on the property that the components in *C fully represent* the general graph *G*. This description is given in the Appendix but for our purposes here, *G* and the set of components in *C* share *exactly* the same vertices and edges.

There are two cases 1) if G is connected, then a set intersection between component vertex sets will prove connectedness and 2) if no intersection between vertex sets yields a single connected component that contains both vertices, then they are not connected in *G*.

In the first case, if *G* is fully connected, there exists a sequence of unions between the components in C that will yield a single connected component that contains both vertices being tested. A set of unions of the components in C will eventually testing for connectedness between elements of *C* means finding some set of intersections between vertex sets such that their union results in a single connected component. Since *C* fully represents *G*, testing connectedness can be accomplished using a set intersection between the vertex sets of the components in *C*.

In the second case, if *G* is not fully connected, it cannot be represented as a single component. As *G* and *C* are equivalent, there can be no union of connected components from *C* that form a single connected component that contains both vertices. Testing for not connected between elements of *C* means proving no intersections exists between sets that will allow them to be unioned into a single component. Since there is, by definition, no subset of C that can yield this connected single component after unions, testing for not connected between vertices in *C* means proving no series of intersections exists that allow them to be unioned into a single component.

As a *G* must be either connected or disconnected, testing for connectedness in *G* can be solved as a set intersection problem □.

Other graph algorithms may also be able to take advantage of this internal representation to yield amortized results. Note that this algorithm is based on a series of intersection operations between the components of *C*. Intersection is one of several logical operations that can be performed on the components based on they're have common schemata. Section 2.3 expands on this idea to define a set of operations and the basis for an algebra that is briefly outlined in Section 2.4.

2.2 Pattern Matching Affinity

The second observation regarding this graph database model is that the structures of these normal form graphs lend themselves to how queries are formulated and then executed against graphs. We give several examples.

Consider the two query languages SPARQL [6] and Cypher [5]. Both of these languages yield searches over heterogeneous graphs for existential paths. Path criteria are typically specified using conditions found in WHERE clauses, i.e. logical and data type expressions.

Consider the following example SPARQL query that matches the graph pattern on each graph in a dataset and forms solutions which have the **src** variable bound to the elements of the graph being matched.

```
SELECT ?src ?bobNick
WHERE
  {
    GRAPH ?src
    {?x foaf:mbox <mailto:bob@work.example>.
     ?x foaf:nick ?bobNick
    }
  }
```

The **WHERE** clause seeks the values of elements in the graph based on their schema names, e.g. **foaf:???**. Whether vertex or edge data, matches will search values associated with attributes in matching parts of the schemas to find solutions.

Consider the following example CYPER query:

```
MATCH
  (a:Person)-[:KNOWS]->(b)-[:KNOWS]->(c),
  (a)-[:KNOWS]->(c)
WHERE a.name = 'Jim'
RETURN b, c
```

This query is based on a graph with **KNOWS** as an edge label and vertices representing individual people. The pattern matching is based on the query variables, **a**, **b**, and **c**. Neo4j will search the graph for instances of paths that support the assertion that **a** and **b** who know each other and know other people in common. **c** represents the common person that **a** and **b** both know each other. Note the **WHERE** clause here is being used to look at a single attribute with its associated schema across the heterogeneous graph.

As a third example, consider again Dijkstra for computing Single Source Shortest Paths. This algorithm is representative of a variety of graph algorithms that are based on a comparison function that uses edge weights. This function will typically require that the two "weights" of the elements along the edge path be of the same type, or schema attribute. Since the normal form separates like data elements by schema attributes, comparisons relevant to the schemas of one normal form graph can be done independently from the rest of the general graph and efficiently since the data is stored together.

In all three examples, searches across a graph will tend to partition themselves into elements that have comparisons that are between two values of the same type. This basic behavior maps cleanly onto the separated components based on their schemas in this model representation. Each expression element will display locality during its iterative searches which bodes well for performance in algorithm execution.

This locality can also be exploited in distribution. For example, the current grdb code base uses a storage layer that is based on local storage devices. However, the database can be extended to make use of a distributed storage layer, e.g. distributed file systems or key-value databases [15]. Many of these implementations refer to their replicated, distributed chunks of data as *shards*. A natural mapping from this model would be to store individual components in shards.

2.3 Component Operations

Structuring data using graphs allows users to take advantage of various graph algorithms. This section provides a set of basic operations upon which graph algorithm implementations can be based. Figure 3

illustrates this operational flow in the **grdb** graph database implementation.

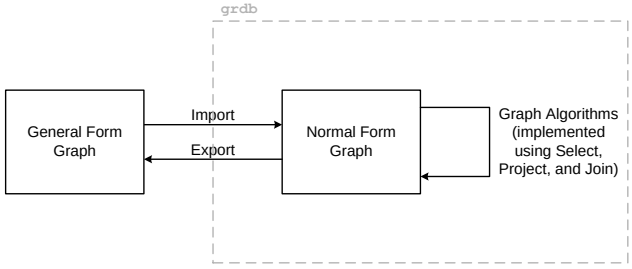

Figure 3: Graph Database Operations

Typically, a general form graph will be imported into the database implementation. This import executes an instance of the conversion from general to normal form algorithm. Likewise, the general form graph can be reconstituted by executing the export function.

Once imported, any graph level algorithms, e.g. Dijkstra, Minimum Spanning Tree (MST), Strongly Connected Components (SCC) [3], etc. are executed in the context of the normal form representation. A set of component level operations, i.e. select, project, and join, that are analogous to the those found for manipulating tables in the relational model are provided for manipulating the connected components within a graph to accomplish the higher level graph algorithm's functions.

These component operations also form the basis of an algebra similar to the relational algebra. This algebra can support query optimizations.

2.3.1 Decomposition to Normal Form

In the database implementation, conversion to normal form occurs during the import of a graph (perhaps from some standard file format, e.g. RDF [7], into the database.

This operation will be defined as follows for the inclusion in the component algebra discussed later.

$$C = \delta_f(G)$$

where f is a set of functional dependencies (see Section 2.3.1.3)

Convert to normal form is accomplished using an algorithm that bears resemblance to Kruskal's algorithm [3] for computing minimum spanning trees and Ullman's algorithm for subgraph isomorphism [10]. That is, a set of connected components is incrementally built that have the same vertex and edge schemas by decomposing the general graph.

While there are many ways to decompose a graph, in this work we focus on the schemas associated with the vertices and edges as the basis for decomposition. The algorithm builds a set of *constant schema trees (CST)* that fully represent the general graph. It does this by extracting CSTs from the general graph until the general graph is consumed.

Given our schema augmented representation of a general graph G, define a set of CSTs as a set of components, **C**, with fixed schemas initially as the empty set.

Select a vertex in $u \in V$ to begin a new CST. Search each neighbor of u, $v \in V$ from the general graph for the vertex with the largest common schema. A new graph c is then defined by the common schemas associated with u and v and the schema associated with the edge, $(u, v) \in E$. The result is $c = (\{u,v\},\{(u,v)\})$

```
C = {}
while (V != {}) do
        Select u ∈ V
        Select v ∈ neighbors(u) with the largest schema
        common to u and v
        if no such vertex v exists then
                Complete-Edges(G)
                Complete-Vertices(G)
                break
        endif
        new c = ({u,v},{(u,v)}
        CST-Grow(c, G)
        Project c out of G
        Insert c in C
endwhile
```

2.3.1.1 Growing the CST

The selection of the first and second vertices and the first edge are crucial to the resulting set of components that are yielded. This process can be steered by user semantics by introducing a concept analogous to functional dependencies. This idea is discussed in a later section.

Once the initial vertices and edge are chosen, the vertex and edge schemas for the new component are set. The algorithm then proceeds to successively search the neighbors of this growing component. During each iteration, another vertex and edge are chosen to add to the component. The choice of which neighbor is made by matching the vertex and edge schemas to those of the neighbors. If they are a subset of one or more vertex, edge pairs, then any desired criteria (probably driven by a query language condition of some sort) is used to decide which.

```
CST-Grow(c, G)
        N = neighbors(c)

        Search the edge set between c and N for an edge
        (u, v) with u ∈ V and v ∈ N that have the largest
        schemas in common with c

        If no such edge exists then return
        Remove v from G and insert (u, v) into c
        CST-Grow(c, G)
end
```

When a vertex and edge pair is selected for addition to the component (either initially or during growth), the vertex

and edge schemas may be subsets of the vertex and edge schemas from the general graph.

After the new component with its fixed schemas cannot be added to, it is used as a projection to remove attributes from the general graph. This procedure is then repeated to grow another constant schema component from the remaining general graph. When all the vertices and edges of the original general graph are consumed, i.e. *V* and *E* both become empty sets, the algorithm terminates.

2.3.1.2 Taking Care of Loose Ends

There may be a point in the execution of this algorithm when all the pairs of vertices have distinct attributes. After that, all the edges are made into components that contain an edge schema and the two vertices that have empty schemas. When all the edges have been removed, each remaining vertex becomes a single vertex component.

CST-Complete-Edges(*G*)
    If no edges remain in *G* return
    Remove an edge *e* = (*u*,*v*) from *G*
    new *c* = ({*u*,*v*},{(*u*,*v*)})
    Insert *c* in **C**
    CST-Complete-Edges(*G*)
end

Each *c* created in this construction has the entire edge schema from that edge. The two vertices have no schemas however, only their vertex ids.

A similar construction is then done with the remaining, edgeless vertices. Each *c* created by this algorithm takes on the entire vertex schema of that vertex.

CST-Complete-Vertices(*G*)
    if no vertices remain in *G* return
    Remove a vertex *v* from *G*
    new *c* = ({*u*},{})
    Insert *c* in **C**
    CST-Complete-Vertices(*G*)
end

2.3.1.3 Functional Dependencies

The selection of the first and second vertices and the first edge during the initial selections for a CST are crucial to the resulting set of components that are yielded. If the first vertex has a schema with more than one attribute and/or the first edge has a multi-attribute schema, it is possible to specify that subsets of these attributes be treated as *functionally dependent* for the purposes of decomposition. Such grouping might be appropriate based on data semantics for example.

The analogy here to functional dependencies in the relational model has to do with the observation that some relational normal forms, e.g. 3rd Normal Form, typically take functional dependencies into account when restructuring tables. This concept is analogous to grouping the vertex and edge schemas of the general graph to steer the decomposition algorithm yielding potentially different component sets.

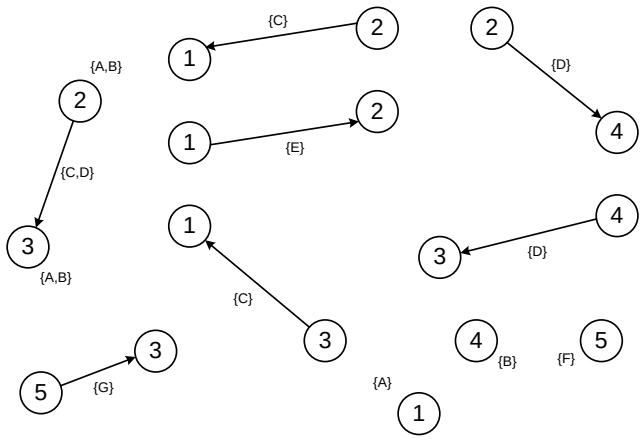

Figure 4: A Normal Form Graph Decomposition Based on a Functional Dependency

Figure 4 illustrates this idea. Assume that the schema associated with vertices 2 and 3 from Figure 1 are defined as A functionally determines B or A→B. This dependency is asserted when the decomposition algorithm is applied during the graph import. The schema {A,B} are not treated as separate attributes for purposed of deciding the greatest common schema during the initial seeding of a CST. This causes the component {{2,3},{(2,3)}} to be chosen first with the remaining components yielded as a consequence.

2.3.2 Intersection and Union

The remainder of Section 2.3 is a description of several operators that form the basis of a graph component algebra based on the normal form. For all of these descriptions, the general form graph in Figure 1 and the decomposition in Figure 2 will be used throughout.

The graph components are composed of sets of vertices and edges. The first operators are component intersection and union. As we have seen, the intersection operator is used during the execution of connectedness tests.

In the case of intersection, the vertex sets of each component and the edge sets for each component are intersected. For union, the vertex sets for each component and the edge sets for each component are unioned. In both cases, the schemas for the two components are unioned.

Given two components c_1 and c_2, the *intersection* creates a new component c' as $c' = c_1 \cap c_2$ where:

1) $V' = V_1 \cap V_2$
2) $E' = E_1 \cap E_2$
3) $Sv' = S_{V1} \cup S_{V2}$
4) $S_E' = S_{E1} \cup S_{E2}$

Figure 5 illustrates and example where the two large components in Figure 2 are intersected.

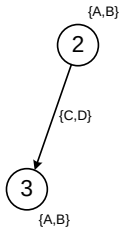

Figure 5: Intersection of the Two Large Components in the Decomposition from Figure 2

c' = {{2,3}, {*A,B*}, {(2,3)}, {*C,D*}}

Given two components c_1 and c_2, the *union* creates a new component c' as $c' = c_1 \cup c_2$ where:

1) $V' = V_1 \cup V_2$
2) $E' = E_1 \cup E_2$
3) $Sv' = S_{V1} \cup S_{V2}$
4) $S_E' = S_{E1} \cup S_{E2}$

Figure 6 illustrates and example where the two large components in Figure 2 are unioned.

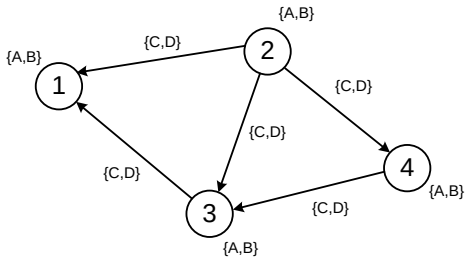

Figure 6: Union of the Two Large Components in the Decomposition from Figure 2

c' = {{1,2,3,4}, {*A,B*}, {(2,1),(2,3),(2,4),(3,1),(4,3)}, {*C,D*}}

The union is analogous to an *outer join* in the relational model. This is because the union joins the structures, collapsing on vertex ids, and *unions the schemas*. The resulting component has the same vertex schema for all of its vertices and the same edge schema for all of its edges. If, as in the case of these decomposed components, they were originally drawn from a general graph that had varying schemas across its vertices and edges they now potentially have a larger, more uniform schema than before decomposition. Thus, performing a union of the decomposed components does not yield the exact original general graph. The new schema attributes are given default values.

2.3.3 Projection

For a normal form component, a *projection* creates a new component such that:

1) the vertices and edges of the new component are equal to those in the given component
2) the vertex and edge schemas of the new component are subsets of the vertex and edge schemas of the given component

This operation is analogous to projection for the relational model. Instead of slicing a relation by attribute, we slice away some of the user data associated with the vertices and edges of the component while leaving the component structure intact.

The result is a component with the same vertices and edges but with fewer attributes associated with the vertex and edge schemas. Specifically, a new component, $c' = \{V', S_V', E', S_E'\}$, is generated by applying the projection operator, π, to c:

$$c' = \pi_{X,Y}(c)$$

where:

1) $V' = V$
2) $E' = E$
3) $S_V' = X \subseteq S_v$
4) $S_E' = Y \subseteq S_E$.

Figure 7 illustrates a projection on the component given in Figure 6. The structure of the component remains the same. Only the vertex and edge schemas change.

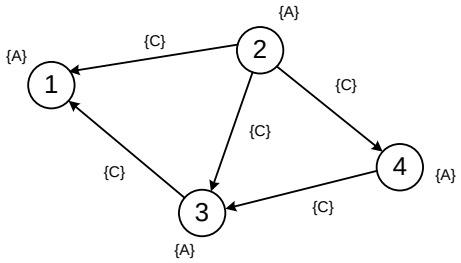

Figure 7: A Projection on the Component in Figure 5

$\pi_{A,C}(c)$ = {{1,2,3,4}, {*A*}, {(2,1),(2,3),(2,4),(3,1),(4,3)}, {*C*}}

If the attributes specified to the projection are empty sets, the data associated with the schemas is cleared. Projection can thus be used to clear all the data from a normal form graph. Simply iterate over the components of the graph and project out all their schemas. This operation clears all the data from the normal form graph but leaves its structure intact.

2.3.4 Selection

For a given component, a *selection* creates a new component where the vertex and edge schemas are equal to the vertex and edge schemas of the given component. Also, the vertices and edges contained in the new component have the following properties:

1) the vertices in the new component are a subset of the vertices in the given component
2) the edges in the new component are those from the given component such that each edge endpoint must be a vertex in the new component

This operation is also analogous to its counterpart in the relational model. Similar to slicing tuples out of a relation, this operation slices a component with fewer vertices and edges but with the same vertex and edge schemas.

Specifically, a new component, c*'*, is generated by applying the selection operator, $\sigma_{condition}$, to *c*:

$$c' = \sigma_{condition}(c)$$

where:

1) $V' \subseteq V$
2) $E' \subseteq E$ such that $\forall e \in E'$, such that if $e = (v_i, v_j)$, then both $v_i, v_j \in V'$
3) $S_v' = S_v$ and $S_E' = S_E$
4) all the vertices and edges in *V'* and *E'*, respectively, meet the condition specified by *condition*.

Figure 8 illustrates an example selection. Consider a condition where the value associated with the *C* attribute yields only the edges (1,2) and (2,4). The resulting component has a different structure but the same schemas as the input component.

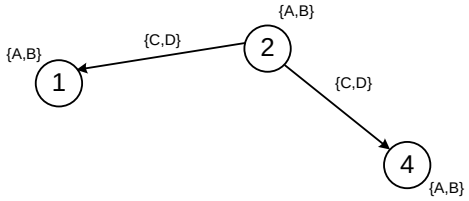

Figure 8: An Example Selection on the Component in Figure 6

$$\sigma_{condition(C)}(c) = \{\{1,2,4\}, \{A,B\}, \{(2,1),(2,4)\}, \{C,D\}\}$$

It is possible that this operation can result in disconnected components. The database implementation discussed later creates new components for each disconnected result.

2.3.5 Inner Vertex Join and Conversion to General Form

Consider the decomposition given if Figure 2. Since this set of components fully represents the general graph given in Figure 1, it is possible to reconstruct the general graph from the normal form components. This operation is referred to as an *Inner Vertex Join*.

Given a set of components $C = \{c_1, c_2, \ldots, c_n\}$, the inner join yields a general graph, *G*. The vertices of *G* are the union of the vertices across the components of *C*. The edges of *G* are the union of the edges across the components of *C*. The schema for each vertex in *G* is the union of the schemas in each component that the vertex appears in. Likewise, the schema for each edge in *G* is the union of schemas in each component that the edge appears in. This operation is denoted:

$$G = \bowtie(C)$$

Since C is a set of components, this expression can be written as:

$$G = \bowtie(\{c_1, c_2, \ldots, c_n\})$$

If this operation is applied to a set of components that were decomposed from a general graph, and these components fully represent the general graph, then this operation can be used to losslessly reconstruct the original general graph. For example, if C is the set of components in Figure 2, the Inner Vertex Join of these components yields the general graph originally given in Figure 1.

2.4 Component Algebra

An interesting observation yielded by the definition of these operators is the ability to treat them as algebraic. This means we should be able to affect query optimizations (not discussed here) based on transformations and cost models.

Table 1: Component Algebraic Rules

Definition	Commutative	Associative	Distributive
Decomposition $C = \delta_f(G)$	N/A	N/A	N/A
Intersection $c' = c_1 \cap c_2$	$c_1 \cap c_2 = c_2 \cap c_1$	$(c_1 \cap c_2) \cap c_3 = c_1 \cap (c_2 \cap c_3)$	N/A
Union $c' = c_1 \cup c_2$	$c_1 \cup c_2 = c_2 \cup c_1$	$(c_1 \cup c_2) \cup c_3 = c_1 \cup (c_2 \cup c_3)$	N/A
Projection $c' = \pi_{X,Y}(c)$	No	No	$c' = \pi_{X1 \vee X2, Y1 \vee Y2}(c) = \pi_{X1,Y1}(c) \cup \pi_{X2,Y2}(c)$
Selection $c' = \sigma_{condition}(c)$	No	No	$c' = \sigma_{cond1 \wedge cond2}(c) = \sigma_{cond1}(c) \cap \sigma_{cond2}(c)$
Inner Join $G = \bowtie(C)$	Yes, any component order	Yes, any sequence of pairings of components	N/A

3. IMPLEMENTATION

This section provides an overview of the implementation called grdb [16]. The implementation is targeted at a single engine running on a single system with local file storage. The database engine manages a set of graphs that are each a set of decomposed, connected components.

When new components are created, all user data associated with the vertices and edges are duplicated. This allows for query languages to use these operations as pass by value inputs and as intermediate values when building expressions based on the algebraic operations given in the last section. This approach facilitates the implementation of query transformations to address query performance.

Given the augmented graph description, the representation used to implement a component is based on an edge list representation and consists of:

1) Vertex schema
2) Edge schema
3) Set of vertices where each vertex is identified by a globally unique *vertex id* and each vertex has a tuple associated with it that is described by the vertex schema
4) Set of edges where each edge is defined by pair of vertex ids that identify a directed edge from id_1 to id_2. Each edge has a tuple associated with it that is described by the edge schema.

5) A list of enum types associated with this graph

The basic implementation provides base operations and a simple shell to support command line access to these operations. Work is in progress to port both OpenCypher and SPARQL query languages to grdb.

These basic data structures exist in two places. The graphs are maintained persistently on disk and elements of these structures are brought into memory as needed by various operations.

Each graph in the database has a numbered directory under the home **~/.grdb** Each graph directory contains a set of numbered component directories. Each component directory contains the files listed in Figure 9.

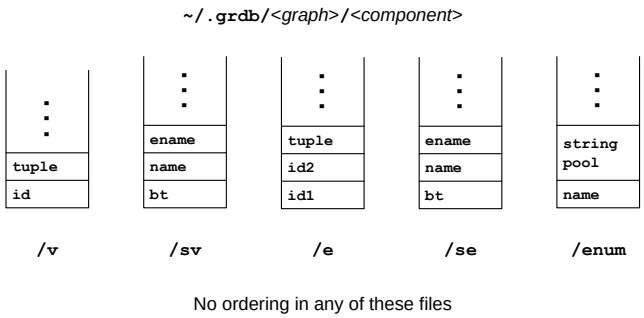

Figure 9: grdb On-Disk Graph Data Structure

The **/enum** file contains the enums defined for both vertex and edges schemas for this component. Each entry in this file contains the enum name and a list of strings that comprise the enum values. These enums are available for both vertex and edge types.

The **/sv** file contains the vertex schema. The **/se** file contains the edge schema. Each file is a list of attributes made up of a base type, attribute name and enum pointer if the base type is enum.

The **/v** file contains the vertices for the component and the **/e** file contains the edges. The vertex file entries are the vertex id followed by its tuple if it exists. The edge file entries are the pair of vertex ids followed by the edge tuple if it exists.

All of these files are maintained unordered. That means it is necessary to iterate over them to find specific elements. The grdb implementation reads and writes enums and schemas in their entirety. Enums and schemas should be relatively small compared to the graph vertex and edge data and should fit in main memory. Since the bulk of the graph data is kept in the vertex and edge files, operations that make use of these files typically shuffle portions of them in and out of main memory as needed. This is necessary since the assumption is that these data may be larger than main memory.

The division of component data into vertex and edge lists is intentional. Since the vertex file maintains a list of vertices that are all the same schema and the edge file maintains a list of edges that are all the same schema, each will be stored together and iterations take advantage of locality. The uniform schema also allows for indexes to be built in a manner analogous to indexes on tables in the relational model. This can be done if rapid searches across the vertex or edge data are required.

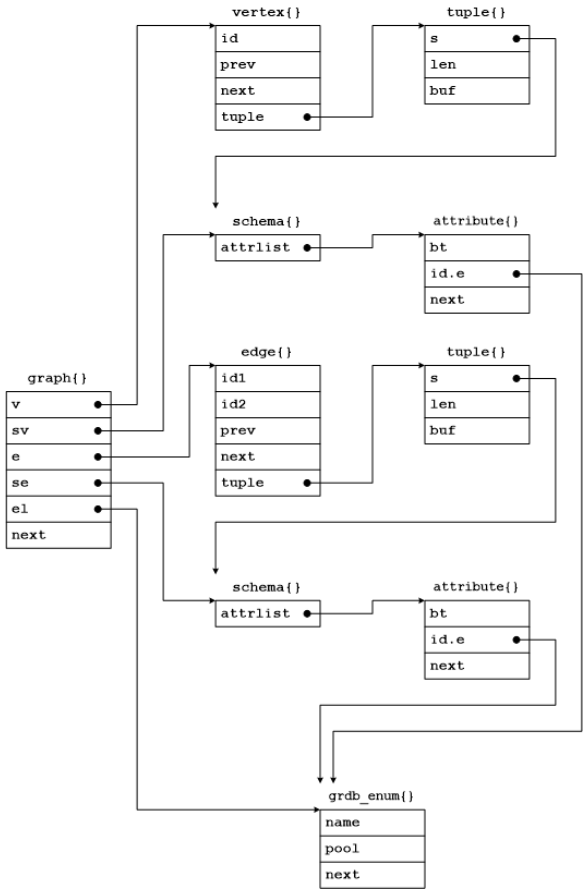

Figure 10: grdb In-Memory Graph Data Structure

Figure 10 illustrates this data structure when kept in memory. This diagram shows the relationships between various structures. This entire data structure is not expected to be instantiated at any given time. Rather its elements and structure serve as a cache template that allows portions of the on-disk data to be held in memory temporarily.

The implementation includes routines to bring portions of the on-disk structure into memory on demand. For enums and schemas, the entire structure is typically read in or written out to disk. These structures are small compared to the vertex and edge data and should fit in main memory. For vertices and edges, the assumption is that these files are larger than main memory so subsets of these files are cached into memory as needed.

3.1 Schema Base Types

The implementation uses base data types to represent graph data. The enum type is composed of two chars that

represent indices. The first is the index of the enum with respect to the containing graph and the second is the index of the item within the enum.

Base Type	Length (chars)
CHAR	1
VARCHAR	256
BOOL	1
ENUM	2
INT	8
FLOAT	4
DOUBLE	8
DATE	10
TIME	8

3.2 Enum Types

Enum types provide an efficient way to represent groups of string values as integer values. These types improve efficiency by allowing value comparisons to be done based on integer values rather than string comparisons. The importation of a graph evaluates strings in the graph data and converts them to enums internally.

Each enum type is represented internally by a string pool. A string pool is a list of strings and an array of indexes to their locations in a single allocated block of memory. This packed representation provides efficient storage for the strings on-disk.

4. RELATED WORK

Interest in graphs as a basis for database implementation has increased lately. Recent examples of large scale graphs are social networks, CRM, and content distribution. Internet scale graph datasets are proliferating, e.g. [18,19,20]. Recent database conferences contain many papers on algorithmic work on these large graph datasets. Less work has been done on the internal design and implementation of graph engines.

That said, the idea of decomposing graphs for efficient storage has also been explored recently. The example given in [8] represents a recent graph database design that focuses on graph decomposition as well. In this work however, the focus in on finding and storing subgraphs that have specified degrees, referred to as *k*-cores. The motivation is similar, to find a decomposition that supports efficient access to large graph dataset based on graph algorithms. The work focuses on identifying these commonly used graph cores in order to hold them in main memory. The work in [9] focuses on determining clusters within a large graph based on "structural similarity." This similarity can be described anecdotally as subgraphs that are well connected and that are each connected to each other by only a few paths. This overall approach bears similarity to the k-cores described in [8] in that both yield decompositions of a larger general graph based only on the structure of the graph. They do not take into account the schemata associated with the graph.

Neo4j [5] is a recent graph database product. This product provides extensive graph database modeling capabilities and an interesting graph database query language called Cypher. The Neo4j data model is based on general graphs in our terminology. They refer to this as the Labeled Property Graph.

In a sense, this work targets exactly what this reference calls, the "holy grail of graph database scale." (p.169) Rather than focusing on the modeling and language aspects of the database, this paper focuses on the internal implementation to directly address performance.

Regarding Neo4j's internal implementation:

> "Neo4j stores graph data in a number of different *store files*. Each store file contains the data for a specific part of the graph (e.g., there are separate stores for nodes, relationships, labels, and properties). The division of storage responsibilities −particularly the separation of graph structure from property data−facilitates performant graph traversals, even though it means the user's view of their graph and the actual records on disk are structurally dissimilar."

Neo4j also appears to use a separated internal representation of the graph but does so in a way differently than presented here. Graph structure (nodes and relationships) are stored separately from user data (labels and properties). While grdb will provide the same dissimilar internal representation from the user's view, the normal form graph representation keeps structure and user data together. This choice is driven by the desire to decompose based on schema to enable the formal treatment that the normal form enables.

Finally, we compare our approach to a survey of graph database models work up until 2008 [4] in order to place the recent work in context. Early approaches used a simple graph model as a basis but typically allowed for zero or one value to be associated with edges and vertices. The research direction moved towards allowing for more complexity, not necessarily uniformly typed, across the graph representation. This followed from the interest in object orientation and support for complex documents as data types in databases that emerged at the same time. The approach described here also extends the complexity of data associated the vertices and edges but provides an internal model that focuses on the similarities in the data across the graph to allow for efficient storage representations.

This reference discusses major features like integrity constraints, a query language, redundancy, and functional dependencies across the breadth of graph database implementations up to that time. This work focuses on the underlying implementation of the database engine and so

the feature set given in Table 2 that reflects grdb is based on that.

Table 2: grdb Implementation According to the Standard Characteristics defined in Appendix A of [4]

Characteristics/Database Model		grdb
		2016
Basic Foundation	Graph Model	x
Digraph	Node Labeled	x
	Edge Labeled	x
Support	Schema	x
	Tuples	x
Query Language	Algebraic – Procedural	x
	Logic – Declarative	x
	Query	SPARQL port in progress
	Path Queries	OpenCypher port in progress
Integrity Constraints	Schema Instance Consistency	Per Component
	Functional Dependencies	x
Implementation		x
Motivation		Efficiency for large graphs

5. DISCUSSION

The contributions of this paper provide a rich starting point for additional research. 1) The graph data model with its operators and support for functional dependencies, 2) the O(*n*) connectedness test across the general graph using component vertex set intersections as a template for other graph algorithm implementations, 3) the affinity for separating iterations yielded for queries based on WHERE clause expressions and 4) an implementation approach that takes advantage of file system locality

Schema spanning trees as the component structure should be explored in more detail. The simple algorithm outlined in this paper can be replaced with any number of algorithms. This work provides and analogy for functional dependencies that "drives" decomposition but other criteria for steering decompositions to address specific algorithms are possible.

The use of constant schemas for the normal form graphs makes indexing attributes across a general graph easier to implement. Vertex and edge files are maintained unordered but the the contiguous data storage enabled by this approach yields opportunities for efficient sparse indexes to be built on both vertex and edge sets.

Definition of additional normal forms that have their bases for decomposition other than uniform vertex and edge schemas should be explored.

The idea of functional dependencies and normal forms mean that graph database design theory, in a manner analogous to Enitity/Relationship diagrams [17] should be explored based on this model.

This graph database model can serve as a basis for undirected graphs. The typical conversion for each undirected edge to two directed edges is the basis here as well. The two edges will have identical edge tuples that will need to be kept consistent during graph operations. This consistency requires a pairing of edges to be added to the edge representation. This is considered to be an area for future work.

The authors will not patent this work. The implementation is and will remain available as open source at [16]. The authors would like to thank Craig Swank of SendGrid for motivational discussions.

6. REFERENCES

[1] Codd, E. F. 1970. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM 13*, 6 (Jun. 1970).

[2] Garcia-Molina, H., Ullman, J. D., and Widom, J. *Database Systems: The Complete Book (2nd Edition)*, Prentice-Hall, 2009.

[3] Cormen, T. H., *et. Al., Introduction to Algorithms Third Edition*, MIT Press, 2014.

[4] Angles, R. and Gutierrez, C., Survey of Graph Database Models, *ACM Computing Surveys, 40*, 1 (Feb. 2008).

[5] Robinson, I., Webber, J., and Eifrem, E., *Graph Databases: New Opportunities for Connected Data*, 2nd Ed., O'Reilly, 2015.

[6] W3C, *SPARQL Query Language for RDF*, https://www.w3.org/TR/rdf-sparql-query

[7] *W3C, Resource Description Framework (RDF) Current Status*, https://www.w3.org/standards/techs/rdf#w3c_all

[8] Wen, D., et. Al., "I/O Efficient Core Graph Decomposition at Web Scale", *Proc. of ICDE*, 2016.

[9] Chang, L., et. Al., "pSCAN: Fast and Exact Structural Graph Clustering", *Proc. of ICDE*, 2016.

[10] Ullman, J., "An Algorithm for Subgraph Isomorphism", *Journal of the ACM, 23*, 1, 1976.

[11] Chaudhuri, S., "An Overview of Query Optimization in Relational Systems", *PODS 98*.

[12] Chang, F, et. Al., "Bigtable: A Distributed Storage System for Structured Data", *Symposium on Operating Systems Design and Implementation (OSDI)*, USENIX, 2006.

[13] DeCandia, G., et. Al., "Dynamo: Amazon's Highly Available Key-value Store", *Symposium on Operating System Principles (SOSP)*, ACM, 2007.

[14] Ding, B. and Konig, A. C., "Fast Set Intersection in Memory", *Proc. of the VLDB Endowment, 4*, 4, 2011.

[15] Lakshman, A. and Malik P., "Cassandra – A Decentralized Structured Storage System", *Operating Systems Review, 44*, 2, 2010.

[16] *grdb Graph Database*, https://github.com/fwmiller/grdb

[17] Chen, P., "The Entity-Relationship Model – Toward a Unified Model of Data", *ACM Transactions on Database Systems, 1*, 1, 1976

[18] Van Rijswijk-Deij, R., et. Al., "The Internet of Names: ADNS Big Dataset", *Proc. of SIGCOMM '15*.

[19] Panchenko, A., et. Al., "Website Fingerprinting at Internet Scale", *NDSS '16*.

[20] *AWS Public Datasets*, **https://aws.amazon.com/public-datasets/**

APPENDIX

Theorem 1.

Given a graph G, fully represented (Def. 3) as a set of connected components (Def. 1), C, testing for connectedness G is equivalent to a set intersection problem between the components of C.

Proof. All paths in *C* exists in *G* (*Lemma 1*) and all paths in *G* exists in *C* (*Lemma 2*). *C* and *G* are *path equivalent*.

If there is an ordered list of components in *C* such that an intersection containing at least one common vertex exists between every component and the union of all its previous component's vertex sets, then *C* is connected. (*Lemma 4*). When C is disconnected, no such ordered list can be found (*Lemma 4*). Therefore, testing *C* for connectedness can be solved as a set intersection problem. Since connectedness determines connecting paths and *C* and *G* are path equivalent, testing for connectedness in G is equivalent to a set intersection problem. □

Definition 1. Let *c* be a subgraph of *G* (*Def. 2*). *c* is considered a connected component if there exists a $path_{ij}$ in *c* for all vertex pairs where *i, j* are elements of the vertex set *V*(*c*).

Definition 2.

Let *G* be a graph with a vertex set, *V*(*G*), and an edge set, *E*(*G*). Let *g* be another graph with a vertex set, *V*(*g*), and an edge set, *E*(*g*). *g* is a subgraph of *G* if *V*(g) is a subset of *V*(*G*) and *E*(*g*) is a subset of *E*(*G*).

Definition 3.

Given a graph *G* and a set of connected components $C = \{c_1, c_2, \ldots, c_n\}$ where each c_i is a subset of *G*, *C fully represents G* if and only if:

$$\bigcup_{c \in C} E(c) = E(G)$$

$$\bigcup_{c \in C} V(c) = V(G)$$

Lemma 1.

Given vertices i, j, for all $path_{ij}$ that transverse some component, c of C, these paths also exist in G.

Proof. By contradiction. Assume that $path_{ij}$ can exist in *c* but not in *G*. The path can be described as a ordered set of edges, $path_{ij}$ is a subset of *E*(*c*). Since *c* is an element of *C*, *E*(*c*) is an element of *E*(*G*) (*Def. 3*). By the transitive property, $path_{ij}$ is an element of *E*(*G*) and $path_{ij}$ can be found in *G*. This is a contradiction exists so if $path_{ij}$ exists in *c* it also exists in *G*. □

Lemma 2.

If a path exists in G then that same path is present in some subset of C.

Proof. By contradiction. Assume $path_{ij}$ exists in *G* but is not present in any subset of *C*. Since the path can be described as a ordered set of edges, $path_{ij}$ a subset of *E*(*G*). By definition of *fully represented* (*Def. 3*), $E(G) = \bigcup_{c \in C} E(c)$. Based on this equivalence, $path_i$ is a subset of $\bigcup_{c \in C} E(c)$. Since no new edges are created when components union together, $path_{ij}$ must exists in some subset of *C*. But, this is a contradiction and therefore, if a path exists in *G* then that same path is present in some subset of *C*. □

Lemma 3.

Two connected components c_i and c_j form a single connected component if and only if $V(c_i) \cap V(c_j) \neq \emptyset$.

Proof. By contradiction. Assume c_i and c_j are connected components where $V(c_i) \cap V(c_j) \neq \emptyset$, yet they do not form a single connected component. Since $V(c_i) \cap V(c_j) \neq \emptyset$, there is a vertex, *v*, such that $v \in V(c_i), V(c_j)$. A path exists between *v* and all elements of $V(c_i)$ and $V(c_j)$ (*Def. 1*). As such, a path exists between all elements of $V(c_i)$ and $V(c_j)$ forming a connected component. But this is a contradiction. Therefore, if c_i and c_j are connected components where $V(c_i) \cap V(c_j) \neq \emptyset$, they form a single connected component.

Now assume c_i and c_j could form a single connected component but that $V(c_i) \cap V(c_j) = \emptyset$. If c_i and c_j are a single component, then $path_{xy}$ can be found spanning both components (*Def. 1*). A path can be defined as an ordered set of vertices. For $path_{xy}$ to span c_i and c_j, there must be a subset of the path that is a member of both $V(c_i)$ and $V(c_j)$ which allows the transition from one component to another to occur. This means that $V(c_i) \cap V(c_j) \neq \emptyset$ which is a contradiction. Therefore, c_i and c_j can be considered a single connected component if $V(c_i) \cap V(c_j) \neq \emptyset$. □

Lemma 4.

C can form a single connected component if and only if there exists an ordering of the elements of C such that:

$$\forall_{c \in C} \left\{ \bigcup_{i=1}^{n-1} V(c_i) \right\} \cup v(c) \neq \emptyset$$

Proof. By Induction.

Base Case: $C_1 = \{c_1\}$, c_1 is by definition a single connected component so the base case holds.

Inductive Step: Assume $C_n = \{c_n\}$ for n=1 per the base case. If $C_{n+1} = \{c_1, c_2, \ldots, c_{n+1}\}$ then $V(c_{n+1})$ has a non empty intersection with $V(C_n)$, then C_n and c_{n+1} are a single component (*Lemma 3*). Therefore, all elements up to an including c_{n+1} can be considered a single component. If no such intersection exists, while C_n may be a single component, the elements of c_{n+1} are not included in that component (*Lemma 3*). □