GraphLab 3 Design

January 22, 2013

Introduction

This document aims to lay the groundwork for a **minimal** specification of a GraphLab 3 computation system as well a Graph Database interface to support it. The specification is not meant to be complete and leaves open many possible extensions (for instance edge indexing, more datatypes, enforcing constaints, indexes on grouped fields, cursors, etc) that might be necessary for a true graph database.

Also not defined are the following:

1. How vertex insertions trigger computation.
2. Other database like operations[1]:
   (a) Atomic **REPLACE** operations (CAS) on edge data / vertex data
   (b) **UPDATE** operations on edge data / vertex.
   (c) **DELETE** operations on edges / vertices.
3. Re-partitioning (migration of vertices/edges)
4. Multiple simultaneously running computation engines.

The document is roughly split into 4 parts.

1. The 1st section (GraphLabDB) is a description of the database and of the interface.
2. The 2nd section (GraphLabCompute) outlines the GraphLab 3 computation model and describes a possible user-facing interface.
3. The 3rd section (Implementation) outlines one possible implementation (this is to ensure that the system is actually possible to implement in a moderately efficient manner.
4. The last section (Sharding) describes a new sharding model which generalizes the GraphChi design and has some interesting theoretical implications.

[1] To support these operations on arbitrary backends, it is likely that a set of "lock servers" will be necessary.

1 GraphLabDB

The graph is a directed multigraph of vertices and edges. Note that this is a relaxation from GraphLab 1/2: there can be multiple edges between 2 vertices. Self edges are not allowed.

Each vertex in the graph has a unique 128-bit ID which is user assigned. Vertex IDs need not be sequential and the entire 128-bit range must be useable. (The system may prescribe other internal IDs, but this is not strictly necessary)

All vertices share a fixed set of named fields where the fields are of the following defined types.

INT64: 64-bit signed integer

DOUBLE: 64-bit double

STRING: Bounded length string

BLOB: arbitrary binary object

This list of field types is not exhaustive and may be expanded in the future to include sparse/dense vector/-matrix types, or even JSON entries.

Every vertex share a fixed set of named fields. For instance, a web graph could have the following:

```
ID: VERTEX_ID (required 128-bit field)
PageURL: STRING
Body: BLOB
PageRank: DOUBLE
```

Similarly every edge also share a fixed set of named fields. For instance, a web graph could have the following:

```
LinkFontSize: INTEGER
LinkText: STRING
```

1.1 Null Values

If a field is not set, its value may be NULL, which denotes an missing value.

1.2 Indexing

INT64 and STRING fields on vertices may be marked as INDEXED, allowing these vertices to be identified by these fields. It is not necessary for the contents of the indexed fields to be unique. For instance, a web graph may choose to index the "PageURL" field, allowing direct searches for a vertex given a URL. NULL values are ignored (and cannot be searched).

1.3 Graph Layout

The database must define a sharding strategy based on vertex seperators where each edge in the graph is uniquely assigned to a shard. As in GraphLab 2, each vertex may appear in multiple shards and for which one instance of the vertex is designated the **master**, and all remaining instances are designated as **mirrors.**

A graph shard therefore contains the following information:

1. A list of all the edges stored in this shard as well as the edge data for these edges.
2. A list of all the master vertices stored within this shard.
3. For each master vertex,
   (a) A list of all the shards containing mirrors of this vertex
   (b) The vertex data

The API design is based somewhat on the MySQL C API. The goal is to minimize memory allocations within the interface itself thus simplifying its internal design.

1.4 Core Types

The following types are defined:

```
typedef uint128_t graph_vid_t;
typedef int64_t graph_int_t;
typedef double graph_double_t;
typedef std::string graph_string_t;
typedef std::string graph_blob_t;

enum graph_datatypes_enum {
        VID_TYPE,
        INT_TYPE,
        DOUBLE_TYPE,
        STRING_TYPE,
        BLOB_TYPE
};

typedef uint32_t graph_shard_id_t;
```

1.5 Graph Database Object

On the client side, the core interface object is the **graphdb** object.

It defines a single string constructor which describes how the graphdb object connects to the graph database. The details of the configuration string may be implementation specific.

```
graphdb gdb("configuration string")
```

The **graphdb** also allows query of the number of vertices and edges. This may be slow. (We may also define approximate versions of this)

```
uint64_t graphdb::num_vertices()
uint64_t graphdb::num_edges()
```

Field metadata may be obtained using:

```
const std::vector<graph_field>& graphdb::get_vertex_fields()
const std::vector<graph_field>& graphdb::get_edge_fields()
// find the column number of a particular field
int graphdb::find_vertex_field(const char* fieldname)
int graphdb::find_edge_field(const char* fieldname)
```

Where each **field** object provides the following interface.

```
std::string graph_field::name() const;
bool graph_field::is_indexed() const;
graph_datatypes_enum graph_field::type() const;

// If scalar type, returns the number of bytes needed to represent it.
// otherwise for string/blob types returns the maximum supported size.
size_t graph_field::max_data_length() const;
```

1.6 Common Data Types

Used in numerous places are also the **graph_row** object which describes the data on either the vertex or edge. The **graph_row** type is not copyable.

```
size_t graph_row::num_fields() const;
bool graph_row::is_vertex() const;
bool graph_row::is_edge() const;

// access individual entries in the row
graph_value* graph_row::get_field(int fieldpos);
graph_value* graph_row::get_field(const char* fieldname);
```

To access individual entries in the row, the **graph_value** type is used. The **graph_value** type is not copyable.

```
// returns true if this entry is NULL
bool graph_value::is_null();

// get the length of the data. Returns 0 if NULL.
// If scalar type, returns the number of
// bytes needed to represent it. Otherwise, for string/blob types,
// returns the length of the string/blob
size_t graph_value::data_length();

// Gets the pointer to the raw data.
const char* graph_value::get_raw_pointer();

// Gets the pointer to the raw data
char* graph_value::get_mutable_raw_pointer();

// returns false on failure or NULL
bool graph_value::get_vid(graph_vid_t* ret);
bool graph_value::get_integer(graph_int_t* ret);
bool graph_value::get_double(graph_double_t* ret);
bool graph_value::get_string(graph_string_t* ret);
bool graph_value::get_blob(graph_blob_t* ret);
bool graph_value::get_blob(size_t len, char* ret);

// exception versions of get. Throws exception on failure.
int graph_value::get_integer();
double graph_value::get_double();
string graph_value::get_string();

// returns false on failure
bool graph_value::set_integer(graph_int_t val);
bool graph_value::set_double(graph_double_t val);
bool graph_value::set_string(const graph_string_t& val);
bool graph_value::set_blob(const graph_blob_t& val);
bool graph_value::set_blob(const char*, size_t len));

bool graph_value::get_modified() const;
void graph_value::set_modified();

// delta commit
bool graph_value::get_use_delta_commit() const;
void graph_value::set_use_delta_commit();
```

1.7 Insertion API

To insert data into the graph, the user first creates a new vertex/edge data entry using **new_vertex_row** or **new_edge_row**. These create a row with all NULL values.

```
graph_row graphdb::new_vertex_row()
graph_row graphdb::new_edge_row()
```

Then once the data is filled,

```
bool graphdb::insert_vertex(const graph_row& data);
bool graphdb::insert_edge(graph_vid_t src, graph_vid_t target, const graph_row& data);
```

There is also a batch insertion interface:

```
bool graphdb::insert_vertices(const std::vector<graph_row>& v);

struct graph_edge_insert_descriptor{
  graph_vid_t src, target;
  graph_row data;
}
bool graphdb::insert_edges(const std::vector<graph_edge_insert_descriptor>& e);
```

It is possible to insert an edge **i->j** where either of the endpoints do not yet exist. In which case, the non-existant vertices are implicitly inserted with all fields carrying NULL values.

1.8 Read/Write API

There are two API groups which may be used to interact with the graph database. The first provides fine-grained access patterns while the other provides bulk access patterns. All access methods are by default **PRAM consistent** unless otherwise specified. In other words, writes issued from the same machine are seen in the same order, and the same machine immediately sees the effect of all writes. (This can be guaranteed by having all writes block until completion)

1.8.1 Fine-Grained API

The fine grained API allows for direct access to individual vertices and edges through a **graph_vertex** object and a **graph_edge** object. These objects are not copyable and are therefore manipulated as pointers. All returned graph_vertex and graph_edge objects must be freed using the free_vertex and free_edge functions.

Finding vertices

```
// returns a vertex for a queried vid. Returns NULL on failure
// The returned vertex pointer must be freed using free_vertex
graph_vertex* graphdb::get_vertex(graph_vid_t vid)

// returns the vertex ID in ret_vid for an index query. Returns false on failure
bool graphdb::find_vertex(size_t fieldpos,
                          graph_int_t value,
                          std::vector<graph_vid_t>* out_vid);
bool graphdb::find_vertex(size_t fieldpos,
                          graph_string_t value,
                          std::vector<graph_vid_t>* out_vid);

void graphdb::free_vertex(graph_vertex* vertex);
void graphdb::free_edge_list(graph_edge_list* edge);
```

Vertex Object The vertex object provides abstract to a single vertex. It is possible to query the vertex for the sharding information, as well as its adjacency information.

The vertex object caches the information stored on the vertex, which is accessible via the **data()** function. This data can be refreshed through the **refresh()** function. Modifications to the data can also be written through the **write_changes()** function. Only modified data is written since the **graph_row** object does modification tracking.

```
graphaph_vid_t graph_vertex::get_id()
// Accesses the vertex data. May return NULL if data was not read yet.
// in which case a refresh() call is necessary.
graph_row* graph_vertex::data();

// --- synchronization ---
// write changes to data. Wait for completion
void graph_vertex::write_changes();
// write changes to data. without waiting.
void graph_vertex::write_changes_async();
// synchronize local data with remote servers clearing local changes
void graph_vertex::refresh();
// writes local data to remote servers and reads the latest data
void graph_vertex::write_and_refresh();

// --- sharding ---
// get the ID of the shard owning this vertex
graph_shard_id_t graph_vertex::master_shard();
// returns the number of shards this vertex spans
size_t graph_vertex::get_num_shards();
// returns an array containing the shard IDs this vertex spans
std::vector<graph_shard_id_t> graph_vertex::get_shard_list();

// --- adjacency ---
// gets part of the adjacency list of this
// vertex belonging on shard 'shard_id'
// The returned edge pointers must be freed using free_edge
void graph_vertex::get_adj_list(graph_shard_id_t shard_id,
                                                    bool prefetch_data,
                                std::vector<graph_edge*>* out_inadj,
                                std::vector<graph_edge*>* out_outadj);
```

Edge Object The edge object provides abstract to a single edge. It is possible to query the edge for the shard it belongs to, as well as its adjacency information.

The edge object caches the information stored on the edge, which is accessible via the **data()** function. This data can be refreshed through the **refresh()** function. Modifications to the data can also be written through the **write_changes()** function. Only modified data is written since the **graph_row** object does modification tracking.

```
graph_vid_t graph_edge::get_src()
graph_vid_t graph_edge::get_dest()
// Accesses the edge data. May return NULL if data was not read yet.
// in which case a refresh() call is necessary.
graph_row* graph_edge::data();

// --- synchronization ---
// write changes to data. Wait for completion
void graph_edge::write_changes();
// write changes to data. without waiting.
void graph_edge::write_changes_async();
// synchronize local data with remote servers clearing local changes
void graph_edge::refresh();
// writes local data to remote servers and reads the latest data
void graph_edge::write_and_refresh();

// --- sharding ---
```

```
    // get the ID of the shard owning this edge
    graph_shard_id_t graph_edge::master_shard();
```

Example BFS We can write a single machine BFS in this way:

```
// initialize BFS queue
graph_vid_t root_vertex = 0;
std::queue<graph_vid_t> bfs_queue;
bfs_queue.push_back(root_vertex);
// assume every vertex has a integer visited field which is set to 0

while(!bfs_queue.empty()) {
  graph_vid_t cur_vid = bfs_queue.head(); bfs_queue.pop_front();
  graph_vertex* cur_vertex;
  gdb.get_vertex(cur_vid, &cur_vertex); // get the vertex object
  // check if we have visited this vertex before
  bool visited = cur_vertex->data()
                          ->get_field(visited)
                          ->get_integer() > 0;

  // the vertex was not visited
  if (!visited) {
    // set the field as visited
    cur_vertex->data()->get_field(visited)->set_integer(1);
    cur_vertex->write_changes();
    size_t numshards = cur_vertex->get_num_shards();
    // read the adjacency list
    for (size_t shard = 0; shard < numshards; ++shard) {
      std::vector<graph_edge*> out;
      cur_vertex->get_adj_list(shard, false, NULL, &out);
      // insert the out edges into the queue
      for(graph_edge* edge: out) {
        bfs_queue.push_back(out->get_dest());
      }
      gdb.free_edge_vector(out);
    }
  }
  // free the vertex
  gdb.free_vertex(cur_vertex);
}
```

1.8.2 Coarse-grained API

The coarse grained API provides access to entire shards.

```
    size_t graphdb::num_shards();
    // returns a shard
    graph_shard* graphdb::get_shard(shard_id_t shard_id);
    // gets the contents of the shard which are adjacent to some other shard
    graph_shard* graphdb::get_shard_contents_adj_to(shard_id_t shard_id,
                                                    shared_id_t adjacent_to);
    // gets a list of vertices in a shard
    void graphdb::get_shards_vertices(shard_id_t shard_id,
                                      std::vector<graph_vid_t>* ret);
    void graphdb::free_shard(graph_shard* shard);
    // returns a list of shards adjacent to a given shard id
    void graphdb::adjacent_shards(shard_id_t shard_id,
                          std::vector<shard_id_t>* adjacent_shard_ptr);
```

Shard object

```
    This is basically  a graph with graph_vertex and graph_edge.
    This object is going to be kind of big,
    so I will not describe it in detail here ...
```

2 GraphLabCompute

This is a draft of a compute interface and may change quickly as I figure out new and better ways to implement it. On a high level, it will provide the capability to perform map_reduce over the neighborhood (over just adjacent vertices, or the entire edge scope), as well as a matching transform operation, and a schedule operation.

The key implementation difficulty is that it will be highly desirable to pass additional information to the callback. For instance, if using C++11 Lambdas

```
double scale = 0.85;
double PR = 0.15 + context.mapred_in_vertices(
                        [=](graph_row& other)->graph_value {
                              other.get_field(pr).get_double() * scale;
                        },
                        [](graph_value& result, const graph_value& operand)->void {
                              result.set_double(result.get_double() + operand.get_double());
                        }
                    )
```

Observe that the value of "scale" is captured. This is the "optimal" way I would like to specify the update function. However, this is not possible to implement because it will be necessary to serialize the closure and send it across the network. However, C++ (or even GCC) does not provide enough access to do this.

The interface described below can pass an additional value through a "tag" parameter. This works, but is kind of annoying to use, especially if complex types are to be passed. An alternate non-functional interface based on classes might be possible however:

```
class pagerank_map_reduce: map_reduce {
  public:
    double scale;
    pagerank_map_reduce(double _scale):scale(_scale) {}

    // serializers
    void save(oarchive& oarc) const { oarc << scale; }
    void load(iarchive& iarc) { iarc >> scale; }

    graph_value map_function(graph_row& other) {
      return other.get_field(pr).get_double() * scale;
    }
    void red_function(graph_value& result,
                      const graph_value& operand) {
      result.set_double(result.get_double() + operand.get_double());
    }
};

double scale = 0.85;
double PR = 0.15 + context.mapred_in_vertices(pagerank_map_reduce(scale));
```

2.1 Compute Context

The user writes update function of the form:

```
void update_function(compute_context& v) {
   ...
}
```

Where **compute_context** has the following functions:

```
graph_vid_t& compute_context::vertex_id()
```

This needs to be updated to handle the new semantics as discussed in Dan Grossman's office Jan 15th.

For expert use, graphdb access is also provided.

```
graphdb& compute_context::db();
```

2.2 MapReduce

```
// map_function is a function of the form
//     graph_value map_function(const graph_row& g);
// red_function is a function of the form: (essentially a +=)
//     graph_value red_function(graph_value& result,
//                              const graph_value& operand);

graph_value compute_context::mapred_in_vertices(map_function, red_function);
graph_value compute_context::mapred_out_vertices(map_function, red_function);
graph_value compute_context::mapred_all_vertices(map_function, red_function);
```

This alternate map_reduce form allows an additional value to be passed on to the map and reduce functions

```
// map_function2 is a function of the form
//     graph_value map_function2(const graph_row& g,
//                               const graph_value& tag);
// red_function2 is a function of the form
//     graph_value red_function2(graph_value& result,
//                               const graph_value& operand,
//                               const graph_value& tag);
graph_value compute_context::mapred_in_vertices(map_function2,
                                                red_function2,
                                                graph_value& tag);
// ... also out and all versions ..
```

map reduce on the entire neighboring edge

```
// emap_function is a function of the form
//     graph_value emap_function(const graph_row& center,
//                               const graph_row& edge,
//                               const graph_row& other,
//                               bool is_in_edge);
graph_value compute_context::mapred_in_edges(emap_function, red_function);
// ... also out and all versions ..
```

As well as the corresponding tagged versions which we omit for conciseness.

2.3 Transform

Transform adjacent vertices

```
// transform_function is a function of the form
//     void transform_function(graph_row& g);

context.transform_in_vertices(transform_function);
// ... also out and all versions ..
```

Transform adjacent edges

```
// etransform_function is a function of the form
//     graph_value etransform_function(const graph_row& center,
//                                     graph_row& edge,
//                                     graph_row& other,
//                                     bool is_in_edge);
context.transform_in_edges(etransform_function);
// ... also out and all versions ..
```

As well as the corresponding tagged versions which we omit for conciseness.

2.4 Guarantees

Update functions which do not modify adjacent vertex data, do not use the ***_other_vertex** functions, and do not use direct graphdb access are guaranteed to have factorized consistency.

Writes to adjacent vertices or writes using ***_other_vertex** functions will result in eventual consistency

2.5 Scheduling

...

2.6 ...Details...

There are some details regarding the wrapping API (i.e. what is written in "main") which I am omitting since there are some implementation details which I have not thought about yet.

3 Implementation

This section describes one plausible implementation of the system.

3.1 Storage

Storage is split into 2 parts. A **vertex storage**, and an **edge storage**.

The **vertex storage** is held in a MySQL Cluster NDB table (or really, any fast key-value store) containing the following fields:

1. Vertex ID (Primary Key)
2. Master shard : Integer
3. Mirror shards : Integer List
4. ...vertex data fields...

All vertex data are therefore stored and read directly from the MySQL database. A key-value interface is required for the MySQL database (i.e. HandlerSocket or Memcached (http://dev.mysql.com/tech-resources/articles/nosql-to-mysql-with-memcached.html).

An additional set of **edge servers** provide a fault tolerant interface to shards, edge data, and vertex adjacency information. It does so by maintain a fault tolerant mapping of **graph shards** to **edge server**.

i.e. using the min-hash trick, shard i is replicated on servers with the 3 lowest values of $hash(i, serverIP)$. All edge data and structure read/writes are performed Dynamo style to enforce data consistency between all the edge servers. Unlike Dynamo we do not use vector clocks, nor versioning.

(An alternate design is to have structure read/writes be performed Dynamo style, since that must be maintained consistently. While data read/writes only go to 1 node: the machine with lowest $hash(i, serverIP)$. This maximizes performance, but at the cost that the failure of a single storage machine may result in some data writes being lost)

The edge servers must instantiate the shard in memory (to support the fine-grained access API) with periodic writes to disk.

3.1.1 Insertion

Parallel insertion is tricky since vertex shard information must be **atomic**. To properly support parallel insertion, we will probably require a transactional backend, or an additional set of **insertion servers.** (i.e. server "VID % K" manages the insertion of vertex VID thus sequentializing all operations on the same VID)

To insert a vertex with id ID with data: A vertex with ID is inserted into the MySQL table. The master shard is set to an arbitrary value and Mirror shards are set to NULL. An error should be triggered if the vertex already exists. This operation must be **atomic**.

To insert a edge from A–>B: Vertex A and B are read from the vertex storage. The streaming partitioning procedure then follows. If both vertex A and B exists, the edge is sent to edge servers managing the appropriate shard in the intersection of A.mirrors and B.mirrors. The mirror list of A and B must be updated appropriately. If A or B does not exist, the missing vertices must be inserted, using the streaming partitioning procedure to determine their shard locations.

3.1.2 Fine Grained API

graph_vertex Is basically an interface to the vertex K-V store. Adjacency information go to the edge servers appropriately.

graph_edge Again, nothing revolutionary. Simply an interface to the edge servers.

3.1.3 Coarse Grained API

Entire shards are communicated from the edge storage to the client. Vertex storage must support bulk read/write requests.

3.2 Computation

... still under design ... However there are 2 basic designs to be explored

If entire graph fits in memory

1. Replicate the entire graph on the computation engine.
2. Use **qthreads** to perform update function computations.

If entire graph does not fits in memory

1. Load subset of shards into memory such that some set of vertices have their compute adjacency structure loaded in distributed memory.
2. Use **qthreads** to perform update function computation on those set of vertices.
3. Repeat on another subset of shards.

See the next section for details on how this can be done by exploiting a generalization of GraphChi.

4 Sharding

We first consider the GraphChi sharding model.

Essentially, if we consider the graph as an adjacency matrix, and we consider what data is loaded at each round of computation, a GraphChi shard is essentially a contiguous block of entries in the adjacency matrix.

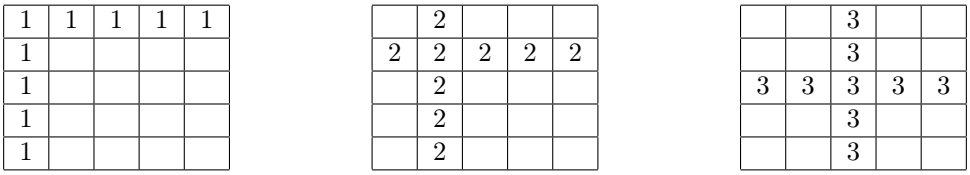

Figure 1: **GraphChi Sharding**: Shards loaded for first 3 rounds of GraphChi. Each square is a shard. i.e. In round 1, the entire first column and first row of shards is loaded. In round 2, the 2nd column and the 2nd row, etc.

This sharding can be thought of as an extremely constrained vertex seperator where only the shards on the main diagonal store the master vertices, and where edge placement is fixed. The advantage of the GraphChi model is:

1. Minimal random access.
2. The number of additional shards needed to contain the entire adjacency information of one shard is bounded. (i.e. if I need to run GraphLab on all master vertices within a particular shard, I only need to load a relatively small number of additional shards.)
3. New shards can be added easily by expanding the columns/rows.

The disadvantage is that

1. There is very little freedom for improving the partitioning. The only freedom available is in permuting the vertex IDs.

In the distributed setting, "random access" is not a serious issue. However, the "bounded" and "expandability" properties are highly desirable since this allows memory utilization to be better controlled. The issue is that in the distributed setting, partitioning quality is important, and so the lack of partitioning freedom is a critical issue.

4.1 Generalization

Lets consider the idea of a "constrained" partition. In other words, let there be k shards numbered from 0 to $k-1$. Let each shard i be associated with a set of numbers $A_i \subseteq \{0..k-1\}$.

Then we introduce the constraint that if a vertex v's master is in shard i, the mirrors of vertex v can only be in shards A_i.

This constraint can be introduced to the current streaming partitioning algorithm trivially. We consider the 3 cases considered by the streaming partitioning algorithm. On insertion of edge $u - v$:[2]

- If both u and v unobserved: insert edge arbitrarily.
- If u observed and v unobserved: We can insert $u - v$ into shard $master(u)$ or any shard in $A_{master(u)}$ which minimizes the objective.
- If both u and v observed: We can insert $u-v$ into any shard in $A_{master(u)} \cap A_{master(v)}$ which minimizes the objective.

Now then, we may consider what are appropriate choices for A_i ?

[2](Note that the algorithm described here requires the master for a vertex be designated early. However, this is not critical and it is possible to delay the choice with some additional care in the algorithm design)

4.1.1 Feasibility

We pay particular attention to the 3 cases of the streaming partition algorithm. The first 2 cases are always possible regardless of the choice of A_i. However, it is possible for the 3rd case to fail if $A_{master(u)} \cap A_{master(v)} = \emptyset$: i.e. there is no valid location to place the edge $u - v$.

We therefore introduce the following **feasibility constraint** for the sets A_i:

$$\forall 0 \leq i < j \leq k-1 \qquad A_i \cap A_j \neq \emptyset$$

In other words, every set A_i must intersect every other set A_j. This ensures that the streaming partitioning algorithm will always complete (though balance may no longer be guaranteed).

However, does there exist solutions for A which guarantee the **feasibility constraint**, other than the trivial solutions $\forall i, j\, A_i = A_j$?

In particular, we are interested in solutions which provide

1. **uniformity**: All $A_i's$ are the same size, or nearly the same size.
2. **balance**: $\sum_i \mathbf{1}(s \in A_i)$ are the same or nearly the same for every s. In other words, I don't get shards which are over or under-represented in the sets A_i.

The answer turns out to be indeed **yes**, with several interesting implications[3].

4.1.2 Solution 1: Perfect Difference Sets

Lets consider the solution form for A_i where, in Matlab notation, $A_i = P + i \,\%\, k$ where $P \subseteq \{1..k\}$ is some set of integers.

Then where P is the **perfect difference set** of order k, the feasibility constraints are satisfied.

The definition of a perfect difference set P of order k is that $\forall 1 \leq i \leq k\, \exists a, b \in P\, s.t.\, a - b = i \, mod \, k$. In words, every number from 1 to k can be expressed as a difference (modulo k) of numbers in P.

We omit the rather simple proof that this must satisfy the feasibility constraint.

Example: $k = 7$

Then a solution is $P = [1, 2, 4]$ and we have therefore have

$A_0 = [1, 2, 4]$ $A_1 = [2, 3, 5]$ $A_2 = [3, 4, 6]$ $A_3 = [4, 5, 0]$ $A_4 = [5, 6, 1]$ $A_5 = [6, 0, 2]$ $A_6 = [0, 1, 3]$

It is not difficult to verify by hand that this is valid.

The perfect difference set has size about $\sqrt{k}$.

4.1.3 Solution 2: Minimal degree diameter-2 graphs

Consider an auxialiary graph $H = (V, E)$ of k vertices numbered $0...k-1$, where vertex i is connected to vertices A_i.

Then the feasibility constraint is automatically enforced if H has diameter 2.

From the Moore graph bound: a graph of diameter 2 and degree d can have at most d^2 vertices (the true bound is slightly tighter than this).

Therefore, flipping the bound around, each A_i must have at least $\sqrt{k}$ entries.

4.1.4 Solution 3: Grid

Consider placing the shards in a 2 dimensional grid.

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

Then let A_i be all the shards along the row and column containing i.

[3](Note that having both uniformity and balance do not imply that the partitioning will be balanced)

For instance: $A_1 = \{2, 3, 4, 5, 6, 11, 16, 21\}$
and $A_{14} = \{4, 9, 11, 12, 13, 15, 19, 24\}$
It is not hard to see that every A_i must intersect every A_j.
This is essentially a generalization of the GraphChi sharding procedure where each A_i has size $2\sqrt{k} - 1$.

4.2 Implications

4.2.1 Communication

Assuming that balance is not an issue, the sharding constraints essentially place upper bounds on communication requirement. Essentially it limits the maximum number of mirrors each vertex can have. For instance, if solutions 1 or 2 are used, it implies that the replication factor for **any** graph on 64 shards is no higher than 8.

4.2.2 Asymptotic Optimality of Grid

Both solution 1 and solution 2 place lower bounds on the size of A_i to $\sqrt{k}$. The GraphChi-like grid solution is therefore no more than a factor of 2 from optimality.

4.2.3 Topology aware Partitioning

The constrained streaming partitioning algorithm described in 4.1 can be further extended to use "soft" constraints. Instead of placing hard constraints on the mirror set of each shard, we can permit violation at a high cost. This will allow the sets A_i to be used to describe network topology between shards/machines; penalizing communication which do not follow the network topology.

4.3 Related Work

Link, J. A. B., Wollgarten, C., Schupp, S., & Wehrle, K. (2011). Perfect Difference Sets for Neighbor Discovery: Energy Efficient and Fair. Proceedings of ACM ExtremeCom.

This paper explored the use of perfect difference sets for a sensor network like setting. Essentially, there are k sensors, and we would like every sensor to be able to talk to every other sensor. However, it is costly to keep the antennae on. Therefore, we would like a timing procedure where each sensor turns on its antennae at particular times while ensuring that for every pair of sensors, there are always some time for which both sensor's antennaes are on. The solution is essentially exactly the feasibility constraint. The paper describes the use of perfect difference sets, as well as the grid solution and a few other construction strategies. The diameter-2 construction was not considered.