The HGDM Protocol
(Hypergraph Data Modeling)

How to Unify Hypergraph Database Architecture
for Multi-Institutional Biomedical Data-Sharing

Introduction

When controlled trials are not feasible, observational studies may be derived from multiple hospitals' case-series: in the absence of controlled trials, pooling data from more than one hospital can permit the comparative effectiveness of different clinical interventions to be quantified (with some degree of precision, even if not at the level of double-blind trials). These kinds of observational studies — as distinguished from double-blind clinical trials — require systematic data integration, so that data from each individual source is placed in its proper context. LTS has developed the **HGDM** (Hypergraph Data Modeling) protocol to facilitate multi-institutional data integration, particularly in the context of research-oriented data sharing.

The **HGDM** protocol is designed to facilitate certain data-sharing initiatives, particularly ones which emphasize software implementations — that is, where institutions participating in the initiative seek to have specialized software which encapsulates their specific contributions to the project, and where the data-sharing infrastructure should provide development tools to aid in the implementation of such software. Multi-site data sharing projects can take many different forms; the **HGDM** protocol would be appropriate for data-sharing architectures which have some common features. First, it is assumed that an organization seeks to provide restricted, selective access to their internal data.[1] Second, it is assumed that each organization's data-sharing protocols may be designed *ad hoc* to accommodate individual projects. That is to say, "data-sharing" in the sense that is relevant to **HGDM** does not involve a single, fixed technology (such as a web service) that (upon being deployed) provides access to some subset of organizational data according to a pre-determined, mostly unchanging protocol. Instead, **HGDM** is designed for environments where the protocol wraps/provides access to multiple distinct projects, each of which may potentially have their own *ad hoc*/customized software and their own data models. Here too, biomedical informatics offers a good example: suppose hospitals across a certain region agree to share case series for Covid-19 patients, the way that each hospital would expose clinical data for such a multi-site project should be determined by the foci of the observational analyses being performed. Thus, since different projects have different foci, each project might need its own data models and software implementations.

In short, preparing for the kinds of data-sharing scenarios targeted for **HGDM** is not primarily, or exclusively, a matter of setting up a large-scale service which outside parties utilize in an open-ended manner. Instead, organizations should have the capability to build *more focused* portals, which share data whose profile and structuration is targeted to the requirements/needs of individual data-sharing projects. This is so, because each project may require its own software components, serving as a layer and adapter intermediary between the organization's internal data sources and the outside parties' software applications that request data in the context of a given multi-site project.

[1] This outline will not address public **API**s or other technologies allowing wide-ranging, unmoderated general access to institutional data.

Overview of the HGDM Architecture

In order to analyze correct design patterns of database adapter-components — for the purpose of providing outside parties (such as researchers and Clinical Research Networks) with access to institutions' curated information — the sections below will focus on solutions that can be treated as extensions to database engines. The tenor of this exposition is somewhat abstracted from concrete implementations, because it is not necessarily the case that all data from a given organization would be housed in one single database. In practice, large institutions often have decentralized, heterogeneous information spaces spanning multiple forms of database design, as well as assets maintained outside any database altogether, such as via filesystems or network-connected software. However, to simplify the definition of **HGDM**, we can construct the essential common parts of the protocol in a hypothetical environment where there is one single database from which all information is obtained. Insofar as the goal of selective, curated outside-party access to some portion of an information space is manifest specifically in the context of a single database instance, the question then becomes how to architect/enhance the database engine so that programmers may implement selective views onto the database instance — permitting remote access to information whose extent and structure is determined by the parameters of a given data-sharing project.

As a framework to architect/enhance database engines, **HGDM** concentrates on one specific kind of database architecture, specifically that of "hypergraph" database engines. Hypergraph databases are well-suited to multi-faceted, heterogeneous data profiles that combine features of more restricted database designs. As a result, protocols developed in the hypergraph context may be adapted to other genres of databases as well (such as an **SQL**/relational database, ordinary graph database, or document-oriented database). In short, **HGDM** is first and foremost a data-sharing protocol in the specific context of hypergraph database engines, though it is not solely applied to hypergraph databases. On the contrary, it tries to promote a software design where servers and clients using non-hypergraph-based technologies can *adapt* those technologies to a hypergraph-oriented network protocol.

Data-Sharing Protocols in the Hypergraph Database Context

The central theme of **HGDM** is as follows: we have a hypergraph database instance, and intend to make some part of the data which it contains available within the context of targeted data-sharing projects. As mentioned above, hypergraph database engines are chosen because they support heterogeneous, flexible data profiles: information need not be constrained to a single format. Another reason for choosing hypergraph database engines is because application-integration is often easier via the hypergraph architecture than via alternative designs. That is, less development time or biolerplate code is needed to marshal data between application-runtime and database-persistent representations. Insofar as these are benefits of hypergraph database engines themselves, they also represent features which are reflected in protocols for outside database access. That is to say, the way in which external applications obtain data remotely from hypergraph database instances should reflect patterns in how hypergraph data is organized in the first place.

At the same time, there are several different hypergraph database engines used in production, each with distinct features and paradigms. As a result, we cannot point to one single, canonical hypergraph database model that could provide a common protocol applicable to all relevant technologies. This is why we recommend a multi-purpose protocol to integrate features specific to different individual engines. In so doing, the goal is therefore to formulate a system for describing remote hypergraph database access which could be applied as a mold for software components obtaining data from a remote portal backed by one of several different hypergraph

engines. The same accession framework should apply to multiple engines, integrating each one's distinct modeling features. In order to engineer a suitably integrated framework, it is useful to contrast different engines' designs and to develop an overarching terminology which can represent the common features and contrasts between them.

To give a concrete example, the term *projection* is used in the context of **HyperGraphDB** to refer to an individual component of an aggregate "node," or *hypernode* — "hyper" meaning that each node has, in general, mulitple smaller parts. Other engines employ different vocabulary for similar concepts — for instance, "roles" in **Grakn.ai**, or (albeit these are not technically equivalent) "properties" in **Neo4j**. In any case, hypergraphs are distinguished from ordinary graphs in that the essential components of hypergraphs — that is, individual hypernodes — are assumed to have internal structure; that is, to be composed of smaller units. Hypergraphs therefore have two different levels of organization: one scale of structure applying *between* hypernodes, which has graph-like properties (hypernodes are connected to other hypernodes via labeled edges), and a finer grain of detail through which individual hypernodes have their own internal parts. Any protocol for accessing hypergraph data thus needs to identify the difference between (information modeled through) these two distinct scales of organization.

In **HGDM**, the most rigorous application of hypegraph formations occurs at the level of "infosets," or deserializations of hypergraph data which result from parsing data sent by an **HGDM** *server* in response to a request by an **HGDM** *client*. The **HGDM** protocol, as such, stipulates how an infoset "layer" should parse and represent hypergraph data. **HGDM** also recommeds a protocol definition for servers and clients, to ensure that these components can productively interoperate with infoset layers. The overall **HGDM** protocol, as such, comprises procedures distributed over server, infoset, and client layers. This protocol-definition document will summarize each of these layers.

Protocol Outline for Server, Infoset, and Client Objects

As indicated in the White Paper introducing **HGDM**, there are three different contexts or stages where **HGDM** components would be implemented: *servers*, *infosets* (objects parsed directly from serializations), and *client* applications (values used directly by user-facing software components, which are initialized from objects of the prior two kinds). Server objects may be directly obtained from a hypergraph database, or alternatively from "adapters" (components that access non-hypergraph databases in a manner which emulates hypergraphs). The server, infoset, and client components will sometimes be referred to as "layers" or "endpoints" (considering that each is potentially the endpoint of a network communication governed by **HGDM**). Objects within each of these three layers can be conceptualized as *hypernodes*, meaning that they have both internal structure and labeled connections with other objects (although these hypernode structures are completely formalized only at the infoset layer).

The material below provides a summary of **HGDM** by enumerating certain procedures within the protocol specifications for each layer. The procedures are presented as grouped according to the layer where they are most directly relevant; however, it is consistent with **HGDM** for the server and client layers to implement procedures and data structures mimicking the hypergraph constructions of the infoset layer.

Server-Layer Procedures

This subsection will review procedures associated with server objects. Some of these procedures are associated with specific database architectures, so therefore they may not be implemented in every **HGDM** server-layer component — although providing capabilities concordant with multi-

ple database architectures allows components to serve as adapters integrating various databases into an HGDM network; as such, these procedures can be seen as a guideline for integration across heterogeneous information systems.

get_binary_encoding() Provides either a raw byte array or base-32 encoded byte array, which is a binary serialization of a server object. This encoding should be reversible, in that a procedure exists to recreate the serialized object through the byte array.

HGDM recommends a base-32 encoding scheme designed to be compatible with **QString** lexical casts (note that this encoding differs from other base-32 systems commonly used).[2] Base-32 streams in HGDM are case-insensitive, little-endian, and use digits **0**-**9** and letters **a**-**v**. The additional alphanumeric characters **w**, **x**, **y**, **z**, and underscore ("_") are employed as end-of-stream separators/markers, indicating 0-4 padding bytes.

get_decoder_class() Returns the name of a class whose objects are binary-encoded (and consequently decoded) by **get_binary_encoding**.

get_decoder_procedure() Returns/provides a data structure through which one obtains a procedure constructing decoder-class objects from byte arrays.

get_indexes() Returns a list of fields within a server object (construed as a hypernode) which are indexed for querying, meaning that one can identify individual objects based on the value for that field (and also potentially sort a collection of objects via that field).

Borrowing terminology from HyperGraphDB, HGDM refers to the smaller units within hypernodes as "projections."[3] At the infoset layer, all objects are fully-formed hypernodes wherein all data contained by each object is available within "hyponodes" — the nodes inside hypernodes; the hypernode is said to *project onto* each of its hyponodes. In the server-layer context, using a design analogous to HyperGraphDB, only some of the relevant information encoded via an object may be available via projections (the remainder is binary-encoded and therefore not accessible to query evaluators outside the object itself). In general, the data fields which have projections within server objects are those that have indices.

get_table_structure() Returns/provides a description of how tabular data can be mapped to a collection of hypernodes.

This procedure is associated with adapters for **SQL**-style databases. In general, data layouts which are technically distinct from the hypergraph perspective may all be embodied by relational tables in the **SQL** paradigm, obscuring the hypergraph-structural variations. This procedure compensates for that ambiguity by providing an outline of how a given table should be mapped to hypergraph formations. In the simplest mapping, each record becomes one hypernode, and each column corresponds to one projection. However, more complex mappings are also possible. For example, one column in the table might hold map keys, while the other columns are aggregated into hypernodes, so that the overall data structure is treated as a map (associative array) whose values are hypernodes. Or, columns may be aggregated into two sets of hypernodes, with the overall data structure treated as a graph asserting relations between pairs of hypernodes selected from those sets.

get_rdf_mapping() Returns/provides a data structure encapsulating how **RDF**-style graph data may be interpreted in a hypergraph context. This procedure is associated in particular with adapters working with Semantic Web services.

---

[2] **QString** refers to the principle character string/text class of the Qt C++ application development framework. The encoding is structured so that base-32 strings can be mapped to decimal or hexadecimal numbers via **QString** methods such as **.toInt(...)**.

[3] See http://www.hypergraphdb.org/docs/hypergraphdb.pdf.

In general, mapping Semantic Web or **RDF** ontologies to hypergraphs is non-trivial (certainly too complex for an automated process or a single algorithm), because there are numerous sorts of inter-node relations in a hypergraph context which may all be embodied as generic graph-edges in **RDF**. Therefore, adapter schemes for **RDF**-style data should model how **RDF** relations translate to hypergraph constructions — including hypernode-to-hypernode connections, hypernode-to-hyponode projections, hyponode-to-hypernode proxies, and various combinations of these links.

get_file_type_description(), get_file_type_class(), get_file_type_decoder() These procedures are principally intended for use when adapters encapsulate access to a local filesystem, or individual files therein, in accordance with a hypergraph protocol.

Infoset-Object Procedures

In **HGDM**, infoset objects are hypernodes containing sequences of individual parts (hyponodes), and also connected to other hypernodes via directed edges, or "connections." Each connection has a "connector" object which is itself a hypernode, but is not (directly) connected to other hypernodes, apart from the source-connector-target triple. All hypernodes and hyponodes have a type, and the hypernode type constrains the type of its hyponodes.

In **HGDM**, hypernodes may have a dynamically changing sequence of hyponodes — the length of this sequence may enlarge or contract. However, new hyponodes must be added according to a fixed type pattern. To be precise, all hypernodes have a (possibly empty) list of *structure fields*, which are hyponodes each of which is associated with a predetermined type and a string label. Each of these fields must have a value (perhaps a "null" value if such a value exists in the hyponode type); and hyponodes cannot be added or removed from the structure-field part of a hypernode. In addition to (and sequentially following) the structure fields, hypernodes may have *array fields*, which in the simplest case is zero or more hyponodes all with the same type. Within the span of the array fields, hyponodes may be added to or removed from the sequence. **HGDM** permits array fields to have multiple types, but in this case the types must conform to a predictable "cycle," meaning that every nth hyponode (for some n) has the same type. For instance, a map between strings and integers can be represented by a length-two cycle alternating between string-typed hyponodes and integer-typed hyponodes.

Many of the canonical **HGDM** procedures in the infoset context involve hypernodes, hyponodes, array fields, and structure fields. These include:

get_hypernode_type(...), get_hyponode_type(...) Returns a name or a data structure representing the corresponding node's type.

In general, each hypernode's type determines both the label used to access items in its sequence of hyponodes, and also the corresponding hyponode's type (note that only structure fields have string labels, however).

get_structure_field(...), get_structure_fields(...) Provides access to one or more of a hypernode's projections (i.e., its contained hyponodes). Each hyponode may be obtained via an index into the containing hypernode's hyponode-sequence, but using a descriptive label is more portable and self-explanatory than using raw indices. Variants on these procedures which take multiple labels will return or expose multiple projections accordingly.

HGDM recommends that access to projections be provided via a "functional" style as well as (or in place of) returning a handle to hyponodes directly. This means that the procedures have variants which accept a procedural value (e.g., a code block) which the called procedure will execute (assuming a proper hyponode can be provided), passing the relevant hyponode as a

value to the code passed as an argument. If multiple hyponodes are requested, this code should be called multiple times, one for each hyponode.

get_array_field(...), get_array_fields(...) Provides access to one or more hyponodes from the "cyclic" part of the hypernode.

Array fields are one-indexed (meaning that the field corresponding to the number "1" is the first field in the cycle, and so forth). Note that the array index is not (in general) the same as the index in the overall hyponode sequence.

Even if the primary role of a hypernode is to hold a collection of values — i.e., a resizable array whose hyponodes are indexed via array fields — hypernodes may pair these arrays with one or more structure fields carrying holistic info about the array. **HGDM** stipulates that hypernode implementations should indicate the length of a hypernode's total hyponode sequence; the boundary between the structure fields and array fields; and the "cycle length," which derives from the recurring type-pattern governing how hyponodes may be added as array fields. However, depending on how hypernodes encode their data, the array fields may embody a "logical" sequence of implicit values spread over multiple hyponodes. In some of these cases the number of distinct logical values encoded within a hypernode cannot be determined just from sequence- and cycle-length data alone. This is an example of the sort of holistic data which might reasonably be stored in structure fields preceding an array-field cycle.

As with structure fields, array-field access can be provided through functional variants, and through variants which return/expose multiple hyponodes. Variants should be provided which expose multiple array fields both via a list of indices and via a range of indices, indicating interest in each field indexed between the range start and end (inclusive).

get_field(...), get_fields(...) These are lower-level procedures which expose fields based on raw (zero-indexed) sequence numbers. These procedures do not distinguish whether the accessed hyponode is a structure field or an array field.

get_metadata_property(...), get_frame_relation_property(...), get_hyponode_property(...) Yields a property associated with a given hypernode. As with Property Graphs, **HGDM** permits properties to be defined on edges and vertices; in **HGDM**, however, properties can be encoded in several different ways, according to different hypergraph constructions. Properties can be hyponodes, or hypernodes connected to their target hypernodes via specially flagged edges, or they may be metadata ascribed to hypernodes in the context of a given frame.

get_role_projection(...), get_conceptual_role_structure(...) Exposes a hyponode via a description of the "role" played by that value in a hypernode, which is not (necessarily) the same as a structure-field label.

Following **Grakn.ai**, **HGDM** recognizes that some hypernodes are representations of events or situations which comprise multiple parts or "actors" playing different roles.[4] Labeling these roles provides an alternative interface for accessing the hyponodes onto which the roles project, in the context of a given hypernode. These projections may potentially be array fields as well as structure fields, particularly when the same role is played by multiple parties.

HGDM allows this concept of "roles" to be generalized to role-aggregates in a manner which borrows from Conceptual Space theory as well as conceptual-role semantics, which motivates the **get_conceptual_role_structure(...)** variant. Conceptual Space representations are also relevant to the next item.

[4] See http://klarman.me/academic-work/resources/MesEtAlCISIS17.pdf.

get_dimension_structure(...), get_domain_structure(...), get_conceptual_domain_structure(...), get_hybrid_conceptual_structure(...)

Returns/provides dimensional information about a projection, including numerical/statistical detail such as level of measurement (Nominal, Ordinal, Interval, Ratio), units of measurement, scale, ranges, or expected distribution within a range. The **HGDM** semantics of dimensions (and domains, which are dimension aggregates) is derived from the Conceptual Space Markup Language (**CSML**).

In accord with Conceptual Space theory,[5] dimensions may be grouped into *domains*, and domains grouped into "concepts." These aggregative structures can be defined or indicated via **get_domain_structure** and **get_conceptual_domain_structure**. **HGDM** also allows domains to be characterized in terms of roles as well as dimensions; as such, **get_hybrid_conceptual_structure(...)** yields a structural articulation which combines the features/details of **get_conceptual_domain_structure(...)** and **get_conceptual_role_structure(...)**.

get_connection(...) Given a hypernode (the source) and a description of a hypernode attached to a hyperedge (the connector), returns a hypernode which is the target of the edge whose source and connector matches the arguments.

If multiple edges match, and therefore multiple targets, the procedure can either return all matching hypernodes or else (assuming it is possible to order the targets) the first target which matches.

HGDM has a notion of *frames* and of *contexts* which might constrain connection matches. When contexts are in effect, certain connections (i.e., connections between hypernodes) may only be considered applicable when one or more contexts are "active." Similarly, "frames" are sets of hypernodes and/or hyperedges, and a certain connection may "exist" within one frame but not outside it. When frames and/or contexts could be in place which filter connections, any **get_connection(...)** procedure should take frames or contexts as parameters, and attempt to match edges within the confines of those structures.

get_proxy(...) Returns a hypernode which is the target of a hyponode "proxy." Proxying means that a hyponode has a *value* which designates or points to a hypernode. **HGDM** uses proxies as an alternative form of linkage between hypernodes — via one hyponode which holds the proxy value — in contrast to connection-based hyperedges.

consruct_hypernode_via_proxies(...) Given a sequence of hypernodes, constructs a new hypernode whose projections are proxies to each of those hypernodes, in turn. A variation of this procedure should be one which also takes a pre-existing hypernode as a parameter, and appends the proxies as array fields to that hypernode.

Note that for each hypernode type, proxies to that type represents a distinct type applicable to hyponodes. Hypernodes can only have fields which are proxies if those proxy types are part of the hypernode's type profile.

get_binary_proxy(...) The notion of *binary proxies* is available in an **HGDM** context if it is possible to encode sequences of hyponode values to a compact binary representation. In that case, it is possible to indirectly encode the data within a hyponode-set as a byte array. One form of **get_binary_proxy** should therefore construct a hypernode whose projections are translated from such a binary encoding. Note that the resulting hypernode is not considered to be part of

[5] See https://opus.lib.uts.edu.au/bitstream/10453/31756/1/2013007531OK2.pdf, https://www.researchgate.net/publication/221406067_Conceptual_Space_Markup_Language_CSML_Towards_the_cognitive_SemanticWeb, https://www.semanticscholar.org/paper/A-Metric-Conceptual-Space-Algebra-Adams-Raubal/521acbab9658df27acd9f40bba2b9445f75d681c, https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.302.8106&rep=rep1&type=pdf, http://ceur-ws.org/Vol-2090/paper4.pdf, or https://arxiv.org/pdf/1703.08314.pdf.

the graph — it is not connected to any hypernode via either hypernode connections or actual proxies.

A variation on this construction combines the properties of bonafide proxies and binary proxies. In this case, a hyponode contains both a proxy value and a binary value compactly encoding that proxied hypernode's content. When the proxied hypernode changes, the binary proxy is automatically updated.

Whether used alone or paired with a bonafide proxy, binary-proxy values may be part of a union type (i.e., a type which can be declared which *either* holds a binary proxy *or* some other value, such as an integer).

get_curried_proxy(...) "Curried" proxies involve hypernodes which model the semantics of multi-node hyperedges. This is particularly relevant for hyperedges which do not have an obvious semantic distinction between "source" and "target;" that is, an obvious directedness. In sentences such as "Abdul nominated Boris to be Chair of the Dance committee" — or "Abdul married Boris to Clara in Denmark" — the stated relationship has multiple participants (four, in these examples). Moreover, there is not an obvious strategy for clustering the participants into two sets which could each be hypernodes (that would form a "many-to-many" relationship, such as a list of co-authors who have collectively written multiple books). The Abdul/Boris examples are not many-to-many relationships, or even one-to-many or many-to-one relationships. Instead, they are multi-participant relationships. The option of using hyperedges to model multi-participant relationships is considered an important feature of hypergraph (as opposed to **RDF** or Property Graph) modeling.

However, multi-participant hyperedges add certain complexities to hypergraph architecture, particularly vis-à-vis graph traversal. With respect to many-to-many (or one-to-many/many-to-one) hyperedges, there is an obvious directionality to the hyperedges and therefore a clear traversal pattern: one moves from the source node-set (or one node within that set) to the target node-set. With a multi-participant hyperedge, by contrast, the traversal can follow different directions, depending on which piece of information is relevant at a given moment. For instance, from "Abdul married Boris to Clara in Denmark" we can conclude that Boris is married to Clara (which in graph-traversal terms means switching focus from the "Boris" node to the "Clara" node) if that is the detail we are interested in; but we can also ascertain that Boris was married in Denmark, or that he was "married by" Abdul. Because of how multi-participant relations have multiple "directions" (analogous to a traffic circle, which can be entered and exited in multiple ways), some systems model these relatons as hyper*nodes* rather than hyper*edges*, and have a hyperedge connecting each participant to the relevant hypernode.

HGDM recognizes a variation on this solution by treating the multi-participant relation hypernode as an assertion that can be considered from the "perspective" of different participants. The Abdul/Boris examples, for instance, present several directions from the *perspective* of Boris (or likewise for the other participants in these examples). Starting with Boris, one can conclude (by "visiting" the multi-participant relation hypernode) that he is married to Clara, or has been nominated to be Dance committee Chair. But likewise, starting from Clara, the second example permits the conclusion that she is married to Boris; starting from Abdul, the first example permits the conclusion that Abdul is credited with nominating Boris. In short, the hypernodes are — semantically speaking — a superposition of certain fact-clusters which are each centered on one participant.

HGDM adopts this theoretical perspective on multi-participant relation hypernodes and concretizes it through "Curried" proxies ("Currying" is a common term in Computer Science, borrowing the name of the logician Haskell Brooks Curry, referring to fixing one argument to an n-ary function to obtain an (n-1)-ary function). A Curried proxy designates its content to the

proxied hypernode but indicates one participant as the relevant "perspective" for that version of the hypernode. Curried proxies permit traversal to other participants but *not* to the perspectival participant, on the premise that these proxies will only be traversed when the perspectival participant is already in focus.

get_channel_structure(...) In **HGDM**, a *channel* is in effect a special form of frame which aggregates some connectors that share a target. Channels may have labels, and be regulated by a *channel system* which identifies what sort of channels may be applied and whether there are restrictions on how they are combined (e.g., whether a non-empty channel — of a given kind — leading toward one hypernode precludes, or alternatively requires, an additional non-empty channel leading to the same hypernode). The **get_channel_structure** procedure should, given a target node, identify the set of source nodes which connect to the target, grouped by channel.

Note that efficiently finding source nodes from target nodes is only possible in general if the **HGDM** software implements "reversible" connections, meaning that target-to-source relations are stored alongside source-to-target. Whether or not such reverse-connections are modeled in general, they should always be provided once connections are associated with channels (i.e., reverse edges should be represented as a data structure attached to channels' target hypernodes).

get_hyponode_set(...) Given a hyperedge, returns a sequence of hyponodes comprising all hyponodes in the source hypernode and then the target hypernode, effectively erasing the boundary between the two hypernodes in the edge context. This is an example of mapping hyperedges to hypernodes, which is a basic operation of hypergraph analysis.

Client-Object Procedures

In general, application-level objects are initialized from **HGDM** infoset objects, whose hypergraph-based structures are designed to encode many data structures in a flexible, expressive fashion. Client objects, in contrast to infosets, are not intended primarily for sharing between disparate applications; as such, these objects can be modeled directly according to the needs of individual applications, in terms of building compelling **GUI** front-ends and effective domain-specific analytic capabilities. Constructions which are specific to hypergraphs — e.g., role-projections, proxies, and inter-hypernode connections — are not necessarily appropriate in the application-level context (often they can be replaced by ordinary application-level type structures, such as pointers and object members with **get**/**set** accessors).

Nevertheless, the **HGDM** protocol recommends several design patterns or practices which can be enacted at the application level to help application code translate infosets to local type-instances, and in general to interoperate effectively with **HGDM** networks. These patterns are particularly applicable to "collections" types (the sorts of values encoded/decoded at the infoset layer via array fields) and to string/textual data (particularly in contexts where words or names from multiple languages may be involved).

With respect to data types considered in most contexts to be "atomic" values — e.g., strings, integers, and decimals — **HGDM** encourages developers to use types and encodings that transparently document scientific and implementational assumptions and requirements. For instance, magnitudes can be decorated with dimension units and range-restrictions. Floating-point values can be expressed with extra precision by employing ratios (integer quotient-pairs) or "posits" (a special number system invented by John Gustafson)[6] in lieu of traditionally-encoded floating-point representations.

With respect to non-atomic "struct" types, whose contents are accessed by named data fields,

[6] See http://www.johngustafson.net/pdfs/BeatingFloatingPoint.pdf.

HGDM recommends accessors in which the bare field-name serves as a get-accessor and the form "**set_**" provides setters. Procedures named "**get_...**" should not be used for accessors (which logically serve merely to return some member data) but for obtaining a value which requires some intermediate computation, or a value which may vary according to some context beyond the prior explicit setting of one single member value.

In terms of non-atomic collections types, HGDM recommends a collections protocol which overlaps with some patterns applicable to infoset array fields. In particular, HGDM recommends collections types whose underlying sequential data is one-indexed, although with a "zero" value allocated in memory to represent "default" or "null" values. Internally, that is, the sequences are zero-based — usually it is not necessary to modify access indices — but the actual value present at position zero is considered not a *logical* part of the sequence, but rather a placeholder for a value to use for out-bounds or other exceptional circumstances. If **arr** is an array and n is an integer, the expression **arr[n]** would then return **arr[0]** for any n greater than **arr**'s size.

This system addresses several suboptimal properties of both zero-indexed and one-indexed arrays. On the one hand, there is a correspondence between the length of an array and the index of it's last element: if n is the length of **arr**, then **arr[n]** is the last element in **arr**. Moreover, if one searches for an element in **arr** which is not found, the return value can be naturally **0**, numerically the same as boolean **false** — and not (as is common practice in zero-indexed environments) **-1**. Employing **-1** as a "sentinel" not-found value is problematic in several ways, one of which is that **-1** evaluates under boolean conversion to ***true*** rather than **false**. Indeed, **-1** does not even technically exist if the array index is an unsigned integer. Moreover, some systems allow negative indices to support counting from the *end* of the sequence — **arr[-1]** would be **arr**'s last element, **arr[-2]** its next-to-last, etc. In this case it is dubious to use **-1** both as a valid index value and as a flag for a index not found — i.e., for the *lack* of any index which returns a value that would satisfy some condition. With any logic, the proper flag value for a "not-found" condition would be zero. On the other hand, computationally offsetting every index by one is at least a little inefficient and potentially error-prone. A reasonable compromise is to leave an array internally zero-indexed, but treat the first (zero-index) value as a "dummy" value which is not logically part of the array. In that case, then, one can utilize this initial value as a default to be provided in out-of-bounds situations. With that practice, **arr[n]** becomes defined for all n — defaulting to **arr[0]** for n out of the proper index range.

The HGDM protocol recommends employing such a one-indexed system as a basis for multiple collections types that may be based internally on arrays, including lists, stacks, queues, and even matrices (two-dimensional arrays). HGDM also recommends providing "functional" accessors to collections types (that is, implementing variants of procedures which call a passed code-block with a value obtained from a collection, or potentially multiple calls with multiple values, in lieu of return values directly). Other guidelines, particularly those related to textual data, will be addressed in a later section.

at(...), get(...), value(...), get_at(...) These procedures provide indexed-based access to collections which support access at any index (not just the start or end of a list). The first element is at position **1**.

When implementing indexed-based access, the basic question to address is how to handle out-of-range indices. These distinct procedures use different strategies for such cases. Specifically, **value** (similar to QT's collections) requires a default value to be provided for out-of-bounds indices (a default-constructed value may be used for types which have one, so this default value would often not be explicitly passed as an argument). The **at** variant is recommended to return a reference to a value, defaulting to the "dummy" zeroth value if necessary. The **get** variant

returns a pointer to a value, or a **null** pointer for out-of-bounds indices. Finally, **get_at** can mimic the behavior of either **at** or **value**, but in either case **get_at** should provide a method syntax for the same operation achieved by overloading square brackets: i.e., **arr[n]** should be the same as **arr.get_at(n)**.

Note that programmers seem to differ on whether (in an ideal language) bracket access (viz., **arr[n]**) should return a (copied) value, a mutable reference, or a constant reference. **HGDM** allows for each of those options, to accommodate different computing environments. One note, however: in a **C++** context, it is possible to mix all three styles by an expanded implementation of **operator []**. The simple step here is using **const**/non-**const** overloading to provide two different forms of the procedure, one returning a constant reference and one a mutable reference. The more complex step (and one which some developers might deem too much a departure from idiomatic **C++**) is to associate **operator []** with an intermediate *functor* type which has a *cast* operator to a reference but a *call* operator to a value, via a provided default value. The details of this implementation are outside the scope of this summary, but the upshot is that with operator-overloading along these lines default values can be provided even with bracket notation via code such as **arr[n](1)** — observe the trailing "**(1)**" here which indicates that the number **1** should be used as a default if n is out of range.

push(...), pop(), top() These procedures apply to collections behaving like stacks — "Last In, First Out" data structures in which the element most recently added to the collection is removed by default, when no other criteria for an element to be removed is provided. All stack-like types should provide **push** (add an element to the stack), **top** (get the most recently added element), and **pop** (remove the top element). Stacks *may* also provide procedures to access and append/remove elements in other positions as well, if the stack can also be used as a different kind of collection; however, **push**, **pop**, and **top** are the characteristic functions associated with stacks.

enqueue(...), dequeue(), head() These procedures apply to collections behaving in the manner of queues — "First In, First Out" data structures in which items that have been present in the collection longest are removed first, by default. Every item added to the queue is placed logically "behind" the prior elements. This kind of collection is visualized like a line-up of objects, rather than as a vertical "stack," which explains the common terminology used for queue-oriented procedures rather than stack procedures. As with stacks, elements *may* be accessed out-of-turn, but **enqueue**, **dequeue**, and **head** are canonical functions associated with queues.

push(...), enlist(...), pop(), delist(), top(), rear() **HGDM**'s terminology for deques (double-ended queues) differs from conventional procedure-names more so than for stacks or queues. **HGDM** considers deques to be a combination of two stacks, one representing the "front" of a collection and one representing the "back." Deques in general are queues in which elements may be added to the front — where they are the first to be removed — as well as to the rear; in effect, an algorithm can select to allow some element to "skip the queue" when it is added. Logically, this has the effect of forming a "front" and a "back" stack, where the former is enlarged whenever an element is added which may "skip the queue" and where the latter is enlarged whenever an element is added in conventional "First In, First Out" fashion.[7]

HGDM accordingly uses non-standard procedure-names for the "back" stack, intended to convey the stack-like behavior: **enlist** adds an item to the rear, **delist** removes the item at the rear, and **rear** returns this item. Note that an item added via **enlist** may eventually be removed via **pop** (if all items ahead of it are removed first); however **delist** removes the rear-most item *first*

[7] It is possible to *literally* implement queues as double-stacks, but one must then track the position where the two stacks meet and accommodate the possibility that one stack may be empty.

(similarly, an item added via **push** might eventually be removed via **delist**).

is_row_major(), is_column_major(), is_diagonal(), is_symmetric(), is_skew_symmetric(), get_matrix_length(), get_matrix_size()

These procedures are applicable to matrices (or two-dimensional arrays), which in **HGDM** are assumed to be represented internally as a one-dimensional array. Most numerical or linear-algebra libraries adopt this convention, translating two-dimensional indices to a one-dimensional index into an internal array. Matrices in *row major* order store each row in contiguous memory, so the whole first role is at the beginning of the internal array, then the whole second row, etc. Conversely, matrices in *column major* order store columns in contiguous memory. Libraries differ in whether they employ row-major or column-major order, so **HGDM** recommends matrix classes support *both* options, to minimize memory-copying when importing data from other contexts. In general, row-major order is more efficient when it is desired to copy or reference individual rows, and column-major is better for copying or referencing individual columns.

Note that some matrices are neither row-major nor column-major. In particular, diagonal, symmetric, and skew-symmetric matrices have redundancies which make it unnecessary to store every item on its own. For instance, for diagonal matrices it is only necessary to have an array which is the length of the diagonal; all other elements (i.e., positions r, c for $r \neq c$) are zero. When the matrix in these cases is represented in a manner which minimizes the length of the internal array, neither rows nor columns are contiguous in memory.

Note also that these cases reveal a divergence between the logical number of elements in the matrix (the product of the number or rows and number of columns) with the length of the internal array. As such, **HGDM** recommends distinguishing matrix *size* (logical size) from matrix *length* (the actual array length). For instance, for a typical square diagonal matrix with n rows/columns, **get_matrix_length** would return n, while **get_matrix_size** would return n^2.

get_dimension_structure(), get_column_dimension_structure(), get_pseudo_constructor() Except where computing speed has to be strictly optimized, **HGDM** recommends using more detailed data types — even for numeric values — than conventional integers or floating-point decimals. For natural numbers, range and/or unit-restricted types ensure that values are used in scientifically reasonable ways. For a collections type, **get_dimension_structure** and related procedures would then return information about the range, scale, and units of measurement applicable to all elements in the collection. Along similar lines, **get_column_dimension_structure** would provide this information relative to an individual matrix column, in a situation where different columns could potentially have different dimensional details.

For each instance of a range-bound numeric type, **HGDM** also recommends proving a static factory function (a "pseudo" constructor) which confirms that the requested initial value is in the allowable range before calling an actual constructor. A pointer to that function — which may be obtained via a templated **get_pseudo_constructor** procedure — may then serve as a "passkey" value signaling that the provided constructor argument lies within the requisite range.

Conclusion

This outline has summarized the **HGDM** protocol by enumerating representative procedures which are recommended for software components implementing the server, infoset, and client roles within a network leveraging **HGDM**. This protocol definition is provisional: updated versions of this document will likely be provided in the future as the protocol evolves in tandem with emerging research developments.

For more information on the rationale and background for **HGDM**, please see the introductory paper at https://raw.githubusercontent.com/Mosaic-DigammaDB/CRCR/master/hgdm/


L
T
S


LTS