WHITE PAPER

A New Protocol to Improve Research-Oriented Biomedical and Clinical Outcomes Data Sharing and Across Multiple Institutions

Introduction

In recent years we have seen a proliferation of research in bioinformatic systems, or more broadly "biomedical knowledge engineering," yielding new insights into optimal database design and data-mining strategies.[1] What has become evident in light of this research — albeit as a phenomenon not newly discovered, but newly prioritized — is that clinically and scientifically important information is intrinsically *heterogeneous*, spanning a diversity of formats and subject areas (including bioimaging, genomics, epidemiology, biochemistry, sociodemographics, immunology, clinical outcomes, and patient "quality of life" or related patient-centered evaluations). Medical data sharing — in addition to the routine exchange of patient histories for patient care occurring "in real time" (which is governed by strict regulations, such as **HL7**, that inhibit novel, experimental technologies) — encompasses use-cases such as sharing observational studies; case series; clinical trial results; evaluating patient outcomes and cost/benefit analysis for different treatment options; and similar kinds of clinical, interventional, diagnostic, epidemiological, or translational research. In this research-oriented setting, there are still guidelines and requirements in effect (e.g., patient data must be suitably anonymized, and the reporting for clinical trials must accurately reflect trial design); but there is nonetheless considerable flexibility in how research-oriented medical data can be structured, modeled, and communicated.

In order to integrate all available information relevant for a given biomedical research/scientific project — both within single institutions and across multiple institutions — technology must be flexible enough to adapt to different data formats and profiles. This has led to two related (but philosophically distinct) paradigms: first, large-scale adoption of "Semantic Web" techniques oriented to knowledge-acquisition and Artificial Intelligence; and second, the emergence of "hypergraph" database engines, which aspire to unify many different database architectures into a multi-purpose totality.

Both of these approaches offer the promise of a more tightly integrated biomedical information ecosystem: data and files reflecting many disparate scientific paradigms accessible through one platform that is shared across multiple institutions. With robust multi-institutional data sharing in place, technology can start to redress shortcomings in existing biomedical research practices. To wit, more extensive documentation of patient outcomes can be used to assess the effectiveness of treatments within a broad spectrum of quality-of-life concerns (not just noting short-term clinical consequences of treatment modalities); and the merging of observational studies can be used to compare and contrast a statistically diversified array of interventions and demographic/diagnostic profiles, which can, in effect, compensate for a paucity of clinical trials in emergent subject areas (Covid-19 being a case in point).

Despite these potential benefits, the large majority of biomedical data continues to be stored via conventional database and/or filesystem technologies. Moreover, data-sharing initiatives tend to reinforce traditional data-modeling paradigms, working primarily or exclusively through **SQL**-style tables — as with, for example, the Patient Centered Outcomes Research Institute Common Data Model (**PCORnet CDM**) and the Observational Medical Outcomes Partnership

[1] See e.g. (among many works that could be cited) https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3555311/ or http://nemo.nic.uoregon.edu/wiki/images/8/88/978-3-938793-98-5_Munn_Ontology.pdf.

Common Data Model (**OMOP CDM**)[2] — or through **RDF** graphs which have no explicit contextualization or data-collections/one-to-many representations (consider the Open Biological and Biomedical Ontology Foundry). The relational/**SQL** and **RDF** paradigms embody two opposite extremes of information organization, rooted either in record-tables with well-defined intra-record structure but limited explicit inter-record relations (**SQL**) or in single-tier graphs with well-defined inter-node relations but limited intra-node structure (**RDF**).

Hypergraph database architecture, by contrast, fully embraces and accommodates both inter-node relations and intra-node structuration, absorbing the representational paradigms of both **SQL** and **RDF**. In theory, therefore, hypergraphs should be a preferred representational device for data-sharing initiatives whether they embrace a more **SQL**-style model (as with **PCORnet** and **OMOP**) or a more **RDF**-style model (as with the **OBO** Foundry). However, there are non-trivial differences among the designs of each hypergraph database engine, as well as between hypergraph database architecture (in general) and the Semantic Web. These differences present an obstacle to fully leveraging the data-modeling innovations intrinsic to the hypergraph and Semantic Web paradigms, particularly in the context of multi-institution data integration — even though such integration is a primary motivation for these paradigms.

Against this backdrop, Linguistic Technology Systems proposes a new "Hypergraph Data Modeling" (**HGDM**) protocol, whose goal is to unify the principal structures of multiple hypergraph and Semantic Web frameworks, yielding a truly general-purpose clinical/research data-sharing system. The **HGDM** protocol is focused on research-oriented communications between and among biomedical institutions.[3] **HGDM** aims to compensate for limited industry adoption of hypergraph-database and (to some extent) Semantic Web technology — not by deploying hypergraph database engines in place of legacy systems, but by facilitating the implementation of software layers which allow information spaces to mimic the behavior of hypergraph database engines with respect to multisite data sharing.

In short, **HGDM** articulates a data-sharing model which is flexible enough to accommodate structural variation in the kinds of information packages typically communicated in a biomedical or clinical-outcomes research-oriented context. **HGDM** achieves this flexibility by adopting hypergraph database architecture. This architecture has a similar purpose to **HGDM** — unifying disparate data-structuration models — except that hypergraph databases are engineered for the local storage and persistence of information, whereas **HGDM** is focused on a sharing data between different applications and institutions.

Limitations of Existing Clinical Outcomes Data-Sharing Initiatives

As suggested in the introduction, existing biomedical data-sharing models tend to gravitate toward either classical **SQL** structures or toward "single-tier" **RDF** graphs, which means graphs that do not structurally or syntactically indicate multiple levels of organization (even if different conceptual "tiers" are indirectly suggested by the semantics of an Ontology's types, classes, and properties). To be more rigorous, one could note that existing biomedical data-sharing frameworks can be roughly grouped into four classes:

Table-Oriented Frameworks This is the model evidenced by large-scale projects such as **OMOP** and **PCORnet**. In these contexts, "Common Data Models" are defined in terms of table schema, and any given data package fills in one or more tables conformant to the schema — each table having a fixed number of columns associated with every row. As with **SQL**

[2] See e.g. https://journals.plos.org/plosone/article/file?id=10.1371/journal.pone.0212463&type=printable.

[3] There are actually no technical details in **HGDM** which restrict it to a clinical, health-care, or bioinformatic context, so implementations could potentially be beneficial in other sectors.

databases, this format excludes one-to-many and many-to-many relationships, and in general is suboptimal for describing relations and connections among units of information.

RDF-Oriented Frameworks The **RDF** (Resource Description Framework) paradigm evolved as a corrective to the limitations of tabular data formats — specifically, the fact that relational data bases can only represent data connections and networks indirectly. In contrast to **SQL**, **RDF** puts network structures at the forefront: **RDF** data is intrinsically a graph, where each node represents a single unit of information and all edges are labeled with a string that identifies some sort of connection, asserting that such connection applies between the two nodes (e.g., that a particular patient has a particular doctor as her primary-care physician).

In **RDF**, individual nodes — unlike **SQL** records — have only a singe identifier; they do not internally contain multiple fields, in contrast to **SQL** records with multiple columns. This limitation (which is only partially addressed by linking each node to other nodes) has inspired the idea of *property graphs*, which essentially unify the **SQL** and **RDF** models: each node is a record with multiple fields (properties) and also multiple connections with other nodes (comparable to **RDF** edges). However, although property graphs are popular for database engineering (as in products such as **Neo4j**), they have not been systematically embraced by Semantic Web projects (e.g., the **OBO** Foundry).

Document or Template-Oriented Frameworks The concept of "Document-Oriented" database architectures involves a special technical understanding of the word "document" — specifically, a nested and hierarchical way of organizing information, concretely manifested in formats such as **XML** and **JSON** (the documents need not be, though sometimes they are, text artifacts in the usual sense). In a document-oriented database, hierarchical data structures — rather than nodes within graphs or records within tables — are the basic units of information. Some commercial database engines with this structure (e.g., MarkLogic) are widely used in the health-care industry. But, outside of actual database technology, the *design* of document databases is implicitly reflected in some data-sharing initiatives, such as the Clinical Data Interchange Standards Consortium (**CDISC**).

Without explicitly choosing this model, document-oriented architecture is also evident *de facto* in formats that are based on "templates," wherein a doctor or researcher provides information by using software or programming tools to complete information whose organization is suggested by a visual or structural template. An example of this model is provided by **DICOM** (Digital Imaging and Communications in Medicine), specifically **DICOM-SR** (**DICOM** Structured Reporting),[4] or other radiology/bioimaging platforms (e.g. "ViSion"[5]).

Compared to **SQL** or **RDF**, document-oriented and template-based presentations are more flexible, but they are also less predictably structured, which makes it difficult to parse specific information from the overall data package. This can be redressed by carefully designing the document structure, but in that instance the document or template needs to mimic the layout of **SQL**, **RDF**, Property Graphs, or some other rigorous format, which means that the technology consuming the provided information has to be engineered to work with those alternative kinds of data models — meaning that the surface-level document-oriented representational profile is actually backed by structures adhering to some different paradigm.

Software-Oriented Frameworks Lastly, we can identify data-sharing initiatives which are based around special-purpose data and file formats that do not fit into generic categories such as **SQL**, **RDF**, or Document-Oriented. Domain-specific file formats are associated with particular scientific disciplines: **PDB** (Protein Data Bank) with Organic Chemistry; **FCS** (Flow Cytom-

[4] See https://pdfs.semanticscholar.org/ddb5/f9c5d132ab68af7184140c83eef3a6525515.pdf.
[5] See https://epos.myesr.org/poster/esr/ecr2013/C-1784.

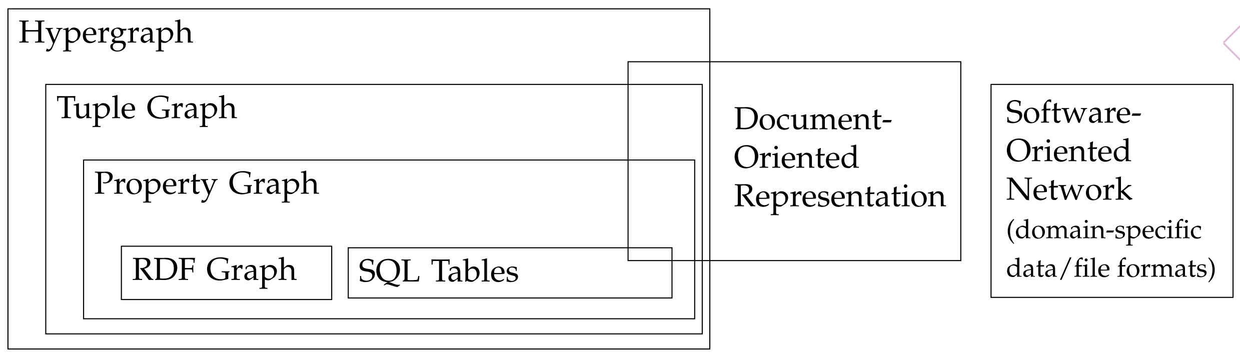
Hypergraph
Tuple Graph
Property Graph
RDF Graph
SQL Tables
Document-Oriented Representation
Software-Oriented Network (domain-specific data/file formats)

Figure 1: Relationships Between Different Kinds of Data Models

etry Standard) with immunology and serology; **OME-TIFF** (Open Microscopy Environment Tagged Image File Format) with microscopy; **DICOM** with radiology; and so forth. Research initiatives grounded in those disciplines — the "Flow Repository" (for **FCS** data), the Radiological Society of North America's Imaging Data Repositories, the International Nucleotide Sequence Database Collaboration, etc. — thus provide data principally through these special formats, assuming that anyone using their data has the requisite software on hand (so as to open the relevant files). In short, these initiatives rely on shared compatible software capabilities — rather than generic data formats — to ensure proper alignment between different communication end-points.

The interrelationships between these different general data-modeling paradigms — and hybrid models that seek to unify multiple paradigms — are sketched outs in Figure 1 (this is a simplified picture, which perhaps elides some technical details that preclude comparing formats as neatly as the figure implies, but it nevertheless gives a bird's-eye view of the technical landscape). Document-oriented architectures overlap with other models when the document structure is sufficiently regulated. Property graphs can be generalized to "tuple" models and then to hypergraphs proper, which may be seen as tuple-graphs augmented with extra details characterizing the fields encompassed by individual nodes. As one proceeds from more restricted to more flexible data models, we theoretically achieve capabilities to represent larger classes of data structures, so that general-purpose data-sharing initiatives should in principle embrace broader frameworks (such as property or tuple graphs or hypergraphs proper) in lieu of narrower ones (such as **SQL** or **RDF**).

The main point is not, however, that simpler data models cannot encode more complex or generic structures — with suitable packaging and encoding, hypergraph structures may be embedded in **SQL** or **RDF**, and vice-versa. The priority when choosing data models is not what can *theoretically* be encoded, but rather how effectively data models are integrated with software development practices. Existing biomedical data-sharing initiatives — such as the New York City Clinical Research Network (**NYCCRN**), **OMOP**, **PCORnet**, or **CDISC** — operationally focus on data formats through which information shared among institutions is encoded (a case in point is the **PCORnet** Common Data Model). In particular, these initiatives prioritize "neutral" formats that allow disparate institutions' software platforms to interoperate. While such neutrality is important, it is equally important to prioritize data representations which are conducive to flexible, dynamic coding practices — as well as to development tools and strategies that can respond to evolving research emphases.

Whenever data is shared between two endpoints, software applications have to be engineered to marshal the data into a common format (on the sender's end) and then parse the data out of that common format (on the receiver's end). The most important criterion for any data-sharing protocol is that application-development for these end points be as streamlined as possible. That is, when implementing a new software component which is to be plugged in to a data-sharing network — perhaps as a result of new scientific priorities (e.g. the sudden emergence of Covid-19) or a newly launched research project — one wants to minimize the development time prior to that component's deployment. As a consequence, one would want to minimize the time required to write code which sends, receives, and parses data according to the data-sharing protocol. **HGDM** posits that the hypergaph architecture is an *application-friendly* paradigm insofar as rapid software development is a high priority.

Creating a Truly Unified Data-Sharing Framework

HGDM *first and foremost* defines a protocol whose canonical use-case is one of remote applications accessing hypergraph database content via a semantically-constrained network.[6] However, this protocol merely stipulates a contract between server and client software *applications*, without the need for explicit requirements on the server's physical data storage. An **HGDM** server could, consequently, be a software layer adapting filesystems or database instances of different kinds, and not only hypergraphs. On a practical level, this need not entail programming encompassing the *entirety* of an institution's local data; instead, developers could selectively curate information fitting some constrained criteria, as part of a targeted data-sharing project.

In short, **HGDM** governs data-sharing activity coordinated between three different software entities (*see* Figure 2). That is, a "server" (which is not necessarily a **TCP** server in the conventional internet sense) responds to requests for information against some data space (potentially but not necessarily a hypergraph database instance). After pulling relevant content from this data space, the server serializes the response data and sends it to an intermediate software component, which parses the response into a hypergraph data structure. **HGDM** introduces a novel "Hypergraph Exchange Format" (**HGXF**) for encoding/serializing hypergraph data. The intermediate software, as such, then receives **HGXF**-encoded data and parses this content to create an "infoset," similar in purpose to an **XML** post-processing infoset. **HGXF** utilizes an extended version of the **TAGML** markup language[7] for a hypergraph-based document syntax, and derives a hypergraph-based semantics from the **HGDM** infoset protocol. The information contained in this infoset is then translated into "application-level" objects — viz., instances of data types which are natively recognized or implemented by the client application that initially formulated the **HGDM** request. Accordingly, the third end-point for **HGDM** components would be a library embedded in applications, one which implements functionality to initiate **HGDM** requests and can translate **HGXF** response-data infosets into application-specific data.

The **HGDM** protocol establishes guidelines or requirements for each of these three data-sharing endpoints/layers. The computational units within these components will be generally referred to as "objects," so that we have *server objects*, *infoset objects*, and *client objects*. **HGDM** regulates communications between server, infoset, and client components by stipulating or recommending that these objects (and the data types of which they are instances) support specific methods and procedures, many of which are oriented toward sustaining hypergraph-based data models across each phase in the data-sharing process.[8]

[6] According to semantics defined by the protocol: hypergraph structures are embraced as representations for content shared between applications, whether or not the communicating end-points adopt hypergraphs natively.

[7] See https://pdfs.semanticscholar.org/bbd9/9215f6bed393c9274f8e0642bebf42d8f633.pdf.

[8] According to the goal of representing, at a minimum, the important structural contributions of major hypergraph database engines, one question then arises as to how to identify such engines as a subset of overall database technology. Considering a range of commercial and academic projects to be examples of hypergraph-oriented models — to varying degrees of theoretical rigor — **HGDM** draws from (in particular) **HyperGraphDB**, **Grakn.ai**, AtomSpace (see https://aiatadams.files.wordpress.com/2016/01/cogprime-overview.pdf), **LMNtal** (see https://waseda.pure.elsevier.com/en/publications/lmntal-a-language-model-with-links-and-membranes), **Neo4j** (see http://www.we-yun.com/doc/books/Neo4j%20in%20Action.pdf or https://academic.oup.com/nar/article/48/D1/D344/5580911), and **WhiteDB** (see http://whitedb.org/ or https://arxiv.org/pdf/1910.09017.pdf), as well as several runtime (non-persistent) hypergraph libraries.

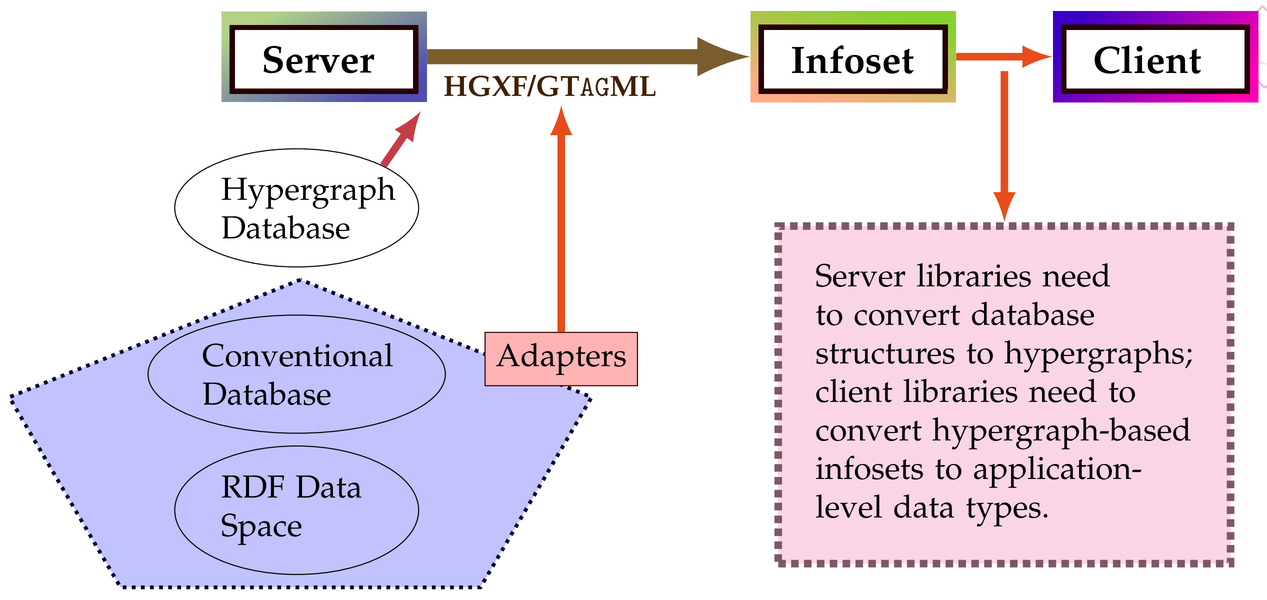
Server
HGXF/GTAGML
Infoset
Client
Hypergraph Database
Conventional Database
RDF Data Space
Adapters
Server libraries need to convert database structures to hypergraphs; client libraries need to convert hypergraph-based infosets to application-level data types.

Figure 2: Outline of HGDM Server, Infoset, and Client Layers

The following sections will outline some of the procedures standardized for the three protocol layers (server, infoset, and client) and will also address some details concerning how data within the hypergraphs should be encoded.

Protocol Outline for Server, Infoset, and Client Objects

As indicated in the introduction, 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).

Of the three endpoints, the intermediate "infoset" layer is the one most strictly regulated by **HGDM**. By contrast, the server and client components have more latitude, recognizing that these layers may be implemented in the context of disparate application and data-persistence environments. Nevertheless, the **HGDM** protocol recommends that certain procedures be implemented within the server and client layers (in addition to any capabilities tied to the server/client local environment) so as to streamline the coding required when these components interact with an infoset layer.

A representative list of **HGDM** procedures stipulated for the server, infoset, and client layers — along with technical discussions of the rationale for these prodedures — is available at https://raw.githubusercontent.com/Mosaic-DigammaDB/CRCR/master/hgdm/HGDM-protocol-definition.pdf.

The Hypergraph Text Encoding Protocol

The prior sections in this paper have considered hypergraph-based information encoding/serialization with an emphasis on numeric and collections types, recommending representational practices which document scientific and programming assumptions and requirements. The underlying principle — that data encoding should be rigorous and transparent so as to clarify the data's scientific background — also applies to textual data, implying the need for carefully designed text-representation protocols.

Properly encoding textual, Natural Language content is one of the more complex and challenging aspects of rigorous data sharing. The Unicode standard theoretically provides a consistent framework for representing all the world's languages, even under-resourced languages (plus many other symbols, e.g. those used in mathematics). However, there are several different Unicode encodings, with no guarantees that two separate communicating applications would employ compatible encodings. Unicode also has certain representational limitations (outlined below), so it is not a definitive textualization solution.

Most textual data needs fewer than 256 distinct characters — the most that can be represented with one byte — even if the text contains some special symbols, such as accented letters occurring in foreign names. Therefore, a lot of computer memory is wasted when allocating more than one byte per character, especially for long text documents. However, the specifics of *which* 256 characters are needed can easily vary from one text to another. Generic types holding textual data, such as **QString**s, often employ two bytes per character, instead, for greater flexibility. One limitation of this solution however — apart from extra memory-usage — is that characters cannot be paired one-to-one with two-byte units: in QT, for instance, some Unicode glyphs actually require *two* **QChar**s in sequence. The full Unicode system utilizes a combination of one-byte, two-byte, and four-byte glyphs; as such, there is no correlation between the index of a glyph in a character stream and the position of a byte in a corresponding Unicode array. To retrieve the nth character in an unrestricted Unicode text, for example, it may be necessary to examine every prior character so as to determine which byte(s) actually holds the character requested. A further consideration is that users often want to copy-and-paste across applications; given that distinct applications often employ incompatible character-encoding schemes, a thorough encoding system should support the notion of a "copyable" range which can convert the indexed characters to a neutral format.

In short, efficiently indicating individual characters or character-sequences by positional indices is oftentimes more important than strict adherence to Unicode (or any other) standards. For example, Natural Language text may need to be annotated wherein mentions of names or concepts are identified via index-ranges. In a biomedical context, strings representing proper names (e.g. patients or doctors), diagnostic codes, symptoms, clinical procedures, chemical formulae, and so forth, may all be detected via text mining (Named Entity Recognition, Lemmatization, Diarization, etc.). The representation of such annotation data could become increasingly complicated without a straightforward indexing scheme — that is, without there being at least one view onto a document such that the overall textual content is a list of characters, with text segments identified by providing a start and end index. For efficient indexing, every character should accordingly have *the same number* of bytes.

The technical question therefore emerges of how to flexibly encode a variety of characters (including certain rare glyphs only used once or twice in a document) while enforcing that all glyphs span the same length of bytes (preferably just one byte) and also preventing unnecessary wasted memory. There are two possible (but not exclusive) solutions to this problem. One option is to reserve a certain number of character codes which are not assigned, except for within individual documents — allowing them to locally map those codes to a more expansive character set, such as Unicode. Another option is to maintain a separate index — outside of the principal character array — for the handful of special glyphs which a document may need outside its normal character set. For instance, suppose three characters out of several thousand in a document require glyphs that do not fit within a one-byte character set. In such a scenario, this document could use a special code (e.g., **255**) for these characters in the main array; and then, separately, maintain a mapping from the three index points to (say) Unicode glyphs intended to be inserted into those three positions. Sometimes these two options can be combined.

For example, many non-English letters appearing in English-language texts are accented characters from other European languages, belonging to proper names or special phrases (cf. the "à" in *vis-à-vis*). These glyphs are typically described by providing a "base" letter plus a "diacritic" (accent) mark. In a one-byte-per-character encoding, one could place only the diacritic code in the main character array, and use a separate mapping to fill in those letters (e.g. a data structure which would assert that the acute accent at position 102, say, should be placed atop an "e"). Only those character codes which could potentially need supplemental bytes (e.g. diacritic codes) would trigger lookup in that external mapping. Such a system combines the ideas of having a designated subset of codes for "locally defined" nonstandard glyphs and of having an indexed-based lookup for nonstandard glyphs at specific index-positions in a character array.

Aside from encoding limitations of Unicode and other popular character formats (including **ASCII**), another problem with these existing encodings is that they tend to address the visual appearance of glyphs rather than their semantic meaning. Consider an ordinary period, which could have several different interpretations: the end of a sentence, part of an abbreviation, the decimal point in a printed number, part of an ellipses, or an operator in computer code. Failure to encode such differences can make text mining more complex than necessary — for instance, using separate character codes for end-of-sentence punctuation than for other uses of periods, exclamations, and question marks would eliminate the need for **AI**-driven "Sentence Boundary Detection." In some contexts, differences in glyphs' linguistic meaning may translate to visual typesetting as well: periods marking an end of sentence, say, should be followed by larger space-gaps than those inside abbreviations (e.g., "Dr."). Hyphens are often printed with different lengths when used as punctuation ("em" and "en" dashes) as opposed to within words or phrases (cf. *vis-à-vis* again) and within numeric literals (cf. "-1"). Quote marks are typically typeset as curved indicators when actually used to surround quotes, but as straight marks when used for feet and inches (e.g., 6'2" tall). Such presentational details may only be considered in depth at the camera-ready phase of a scholarly publication, but in fact there is an overlap between the proper visual cues expressing certain punctuation or other non-alphabetic marks and their semantic meaning, which is relevant for text mining. As such, these semantic differences should optimally be encoded in character sets themselves (rather than relegated to special markup, such as LaTeX commands).

For a further consideration, the basic Latin-1 character set is also inefficient in including certain obsolete **ASCII** codes (such as the "bell" character, which literally rings a bell via the computer's audio) that are almost never currently used. There are, in short, *de facto* unassigned code points, leaving room for duplicate glyphs which may have the same appearance but alternate interpretations (e.g. abbreviation-period vs. punctuation-period). Combining all of these ideas, then, **HGDM**'s recommended text-encoding protocol — dubbed **HTXN** ("Hypergraph Text Encoding") — uses diacritics, "interpretation codes" (disambiguating semantically distinct but visually similar glyphs) and the option to extend character sets arbitrarily for individual documents, to yield a flexible and expressive encoding. This encoding, in its raw form, requires more than one byte per character, but **HTXN** employs certain tricks to compress the character set so that it can typically fit into a single-byte context.

Character Encoding and TAGML

Having discussed the encoding of individual characters, it is appropriate to address **HGDM** recommendations for encoding markup and presentation details, at a higher scale than individual glyphs. As a general-purpose serialization format, **HGDM** suggests **TAGML**, which is similar to **XML** but allows certain constructions that are more expressive than **XML** (e.g., concurrent markup). However, **HGDM** describes a modified version of **TAGML**, one engineered to work with **HTXN** at the character-encoding level.

Intrinsically, **TAGML** does not specifically address character encoding; instead, this is assumed to be a property of each "node" which holds character data. In **TAGML**, certain nodes (called "prenodes" in **HGDM**) represent character-sequences, while other hypernodes embody markup, applied to groups of prenodes. The overall character sequence of a document is assumed to be retrieved by collating characters from every prenode, in order. This can be inefficient when it is necessary to define annotations, or other index-based ranges, which may span multiple prenodes (one cannot readily map single index numbers — assuming these quantify over a whole document — to character indices within individual prenodes). For this reason, **HGDM** recommends that a **TAGML** runtime hold character data outside of prenodes proper — specifically, that characters be stored in "text blocks" and prenodes use indices into these blocks to establish their character data. Each text block, in turn, uses **HTXN** to encode characters via single bytes (different text blocks may use variations on the basic **HTXN** encoding).

Note that this setup does not operationally alter the **TAGML** structure: while a text block *may* encode an entire document via a character array that can be accessed outside any prenode, this is only one possible use-case. Absent specific designs to the contrary, it should be assumed that the definitive character data for any document is accessible only via prenodes, and the use of text blocks is merely a memory optimization to streamline the initialization of prenode contents. In particular, prenodes which have overlapping indices to the same block are not overlapping markup — instead, each prenode holds multiple copies of a given character string, which have no logical relation to one another. For example, a distinct text block could be set up to hold proper names that have foreign characters. Individual names could then be carried within a prenode by giving the start and end indices of the name in its containing block. If a name appears multiple times in a document, several prenodes might then hold duplicate data — the same indices against the same block. But these prenodes do not overlap; each is understood to have a logically isolated copy of that name, as if the characters in the name were stored internally in the prenode's character data without the use of an external text block as the in-memory storage.

Separate and apart from the character-encoding issues analyzed above, **HGDM** also proposes extending **TAGML** in ways consistent with the intended use of **TAGML** as a general hypergraph notation format — in short, as the markup language for **HGXF** (when hypergraph serialization is done via text documents). Specifically, **HGDM** proposes a variant dubbed "Grounded" **TAGML** (**GTAGML**) in which **TAGML** hypergraph structures are refined to notate hypergraph relations within serialized data, not only (as in the current **TAGML**) among markup elements. That is, **GTAGML** structures are "grounded" in hypergraph relations intrinsic to the data being conveyed by **GTAGML** documents (further information about such "grounding" is outside the scope of this outline, but may be provided by Linguistic Technology Systems on request).

Conclusion

It is unrealistic to expect a robust data-sharing infrastructure to be maintained without a frequent fine-tuning of the relevant software, and without deliberate and collaborative effort among programming teams at multiple institutions. Data sharing in the "real world" (as compared to hypothetical scenarios in academic literature) does not happen because computer programs negotiate semantic protocols on their own; instead, it happens because one or more programmers on one site curate their data so that one or more programmers on an external site can productively access it, and vice-versa. This means that in the typical scenario "raw" institutional data is never exposed directly to outside entities; instead, there is always a software layer which adapts the raw data (selectively filtered and curated for open access; de-identification is requisite, for instance, to protect patient privacy) for third-party consumption. Moreover, the shared data must be integrated with applications in two environments: server-side adapters where the data originates, and client-side applications which request and then receive the data.

The **HGDM** protocol is designed on the premise that hypergraph data modeling is the ideal information architecture for organizing data spaces *when application integration is a priority*. Hypergraph database architecture embodies the most equanimous balance between data-format neutrality and compatibility with application-level data types: unlike relational (and non-hypergraph **NoSQL**) databases, hypergraphs do not force developers to work with a database schema which structurally pushes against the logical form of runtime data types; but, unlike object databases (which tend to bind persistent values closely to their runtime expression in a particular computing environment) hypergraph databases encode data in portable, context-neutral ways. **HGDM** lifts these benefits outside the explicit domain of hypergraph database engines proper, allowing the hypergraph data model (synthesized from multiple engines which each have their own design ideas and innovations) to be leveraged for general-purpose biomedical and clinical-outcomes research data sharing.

Appendix I: Developing Data-Sharing Client Libraries

To support the concrete implementation of server, client, and infoset components adhering to the **HGDM** protocol, Linguistic Technology Systems (LTS) aims to provide a development infrastructure and code libraries that would assist programmers writing code in the context of an **HGDM** network. One project serving that goal is **ConceptsDB**, LTS's new hypergraph database engine discussed in Appendix II. In addition to **ConceptsDB**, LTS is working on a common programming infrastructure which facilitates the implementation of client libraries and **GUI** components oriented toward biomedical research data. This common infrastructure is based primarily on **Qt** — a **C++** application-development framework — and secondarily on established scientific/medical platforms or projects, such as **CBICA** (Center for Biomedical Image Computing & Analytics), **SciData**, and **BioCoder** (a tool for documenting research/laboratory methodology). In addition to utilities and build tools, the new client libraries would include **C++** procedures to read and manipulate data in formats commonly used for clinical, diagnostic, and medical-outcome reporting, such as **OMOP**, **PCORnet**, **CDISC**, **DICOM-SR**, **FHIR** (Fast Healthcare Interoperability Resources), **RadLex** (Radiology Lexicon), and **LOINC** (Logical Observation Identifiers Names and Codes). The new client libraries will be focused on supporting custom application implementation, with an emphasis on **GUI** engineering via the **Qt** framework.

As has been emphasized in this paper, **HGDM** seeks to unify disparate representational paradigms into a single hypergraph format. In that sense, **HGDM** client libraries would first and foremost parse **HGDM** data (rather than alternatives such as the various "Common Data" or "Structured Reporting" Models). However, since a lot of the data which may be requested via **HGDM** already exists in non-hypergraph formats, code libraries are needed to manipulate this data so that it may be exposed on an **HGDM** network. In short, client libraries for commonly used extant data standards can act as bridge code, inputting these different standards and outputting **HGDM**-conformant data structures. As a concrete example, **C++** clients for the **OMOP** or **PCORnet** Common Data Models can serve as adapters, providing access to the respective **CDM** records so that they mimic **HGDM** hypernodes. Such capabilities, in turn, may be integrated within a larger array of client-library features, such as **Qt**-based networking and cloud services and custom **GUI** engineering.

Appendix II: The LTS ConceptsDB Database Engine

ConceptsDB is a new hypergraph database engine which LTS is developing as a native implementation of the **HGDM** protocol. In **ConceptsDB**, strucutural features of the **HGDM** infoset layer are expressly modeled within the database architecture. In the context of **HGDM**, then, the primary use-case for **ConceptsDB** would be as an auxiliary database serving as a primary back-end to an **HGDM** server. Rather than extracting **HGDM**-compliant data from an overall institutional information space in response to individual **HGDM** requests, this **HGDM**-ready data can be stored ahead of time in a **ConceptsDB** database instance. Such an arrangment simplifies the implementation of an **HGDM** server. **ConceptsDB** may also be used to prepare data sets to deposit in open-access research data repositories, in conjunction with academic publications. In this case, **ConceptsDB** would hold data that is evolving (and may be periodically updated), while a data set would be obtained by taking a snapshot of that data at a specific moment in time.

Internally, **ConceptsDB** employs the **WhiteDB** engine as its storage back-end (developers may directly access the **WhiteDB** instance if desired). The other main foundation for **ConceptsDB** is **Qt**; **ConceptsDB** can natively store all significant **Qt** objects. Each node may be completely (or almost completely) encoded via **Qt**'s serialization capabilities, which automatically binarize almost all **Qt GUI** and collections types (along with primitive **C++** data types). As such, in essence, **ConceptsDB** leverages **Qt** to provide binary encoding/decoding capabilities analogous to **HyperGraphDB**'s object storage. Programmers may then selectively isolate certain data projections (i.e., member fields within object data) to be stored as their own **WhiteDB** record fields, allowing them to be subsequently retrieved via queries.

ConceptsDB serves as an intermediary between programmers whose code adopts a **ConceptsDB** back-end and the underlying **WhiteDB** instance, providing a flexible interface for encoding hypernode data. Specifically, **ConceptsDB** constructs *record groups*, which are allocated **WhiteDB** records with certain preconfigured fields and node-links. In the simplest case, a record group is just one record where programmers can store both binary object-serializations and indexed field values. In other cases, records groups can have a more intricate structure, particularly when working with collections types that may append or delete items. **HGDM** uses complex record-group structures to encode collections types in situations where it is necessary to expose individual values within collections to a query engine.

Another noteworthy feature of **ConceptsDB** is direct support for serializing **Qt GUI** data, which makes **ConceptsDB** a good option for **Qt** applications' back-end logic. For instance, a **ConceptsDB** instance can conveniently store application state so that the application's layout, recent activity, and user preferences may be restored after a **Qt** application is exited and then later launched again. Similar capabilities support the exchange of relevant application state between distinct software products, which can help scientists recreate research environments in different computing contexts.

ConceptsDB also natively supports LTS's **HTXN** ("Hypergraph Text Encoding") textual data. Thus, **ConceptsDB** by design is unique because it is the only database engine which internally represents string/textual data with the *degree of granularity/specificity* made possible via **HTXN** (for example, recognizing alternative "interpretation codes" for visually indistinguishable but semantically distinct glyphs). In sum, in conjunction with hypergraph data-modeling and **Qt** serialization, this textual granularity makes **ConceptsDB** especially oriented to scientific research and publishing environments.

For more information please contact:
Amy Neustein, Ph.D., Founder and CEO
Linguistic Technology Systems
amy.neustein@verizon.net • (917) 817-2184