Object-Functional Typing for Queryable Data Sets

Nathaniel Christen

July 12, 2018

Abstract

This paper will explore techniques for querying data sets using ordinary application code structured to mimic the semantics and practical use-cases of query languages. The resulting query architecture is more strongly typed than an external query language; which, in the context of published data sets, helps establish a semantics for the data in question. The scientific and theoretical background from which a data set originates can be reflected in the type-representations, statistical/measurement properties, and "Dimensions Of Comparison" recognized in the relevant data types through their implemented methods and operators. Such a query environment uses a collection of types as a formal extension and representation of scientific (or, say, cognitive/linguistic) theories as well as a practical demonstration of theories' morphology. I advocate for a hybrid Object/Functional type paradigm, demonstrated via code which utilizes both Qt reflection and Haskell-like monads (or at least a partial C++ version thereof). Code discussed in the text is sampled from an accompanying self-contained demo repository that can be run as a Qt/C++ application. I will also discuss combining data from multiple sources, using types like QVariant which are less narrowly rigorous than classes uniquely implemented for each data set, but which still provide more type checking than SQL-like query languages. Combining data integration with inter-application messaging allows queries to be evaluated on mixed-source data while, via provenance techniques, allowing values to "route back" from looser-typed integrated data aggregates to their originating applications.

Open-Source data sets have become an increasingly valuable part of academic and scientific publishing. With several disciplines confronting recent "replication crises", greater transparency about authors' data and methodology inspires greater confidence in authors' research as something that will remain valid and can be built upon in the future [42]. However, providing access to data goes only so far; to understand and effectively use published data sets, researchers need the right tools and effective documentation. Ideally, many data sets are published along with computer code which, by accessing the downloaded files, can construct computationally manipulable and visualizable values, representing "deserialized" versions of published data. This code is then a starting-point for applications built around these deserialized values.

Size obviously determines whether an entire data set can be held in memory by one application at one time. If not, the "curator" has several options, include converting the data set to a database or loading only parts of the data set at a time (I'll say the *curator* is a developer who organizes and writes code for the data set, perhaps a different person than the researcher credited with making the data available in the first place). Even if size is an issue, it is important for semantic rigor to engineer data types tuned to the data offered, so that all data can at one time or another be held in memory as an instance of a specific data type, implemented in turn in a specific source code file.[1] Large data sets will be limited in that all of these values cannot be held in memory *at the same time*. In this situation the curator has to marry "in memory" representations of values with a structure conformant to whatever database is chosen to make the archive explorable as an integral whole — where "exploring" a data set means both forming queries to choose groups of values, and examining (visually and/or computationally) individual values.

But even if a whole data set is too large for full in-memory deserialization, individual values and sufficiently narrowed query result-sets *can* be held in memory and processed as regular application-typed values. Within an application, end users will be enacting interactive operations signaling their interests in certain data especially. To serve users effectively, applications need to map users' interests onto queries that, upon evaluation, obtain one or more relevant values.[2] Whether or not these queries are actually shaped as code in a query language like SQL, or resolved instead wholly via application function calls, the application should treat (if necessary, reconstruct) query *results* as live, in-memory values. These values then need to be associated with corresponding User Interface types so that users can view and examine them interactively [13]; [34], [15]. Properly preparing, evaluating, and following-up on queries calls for rigorous semantic models whether or not values are sometimes held in a database rather than live memory. For these reasons, it is a good strategy to build applications that

[1] Excluding, perhaps, resources which are naturally provided (not just *serialized*) as individual files, like photos and videos — but even here it is reasonable to design distinct data types for each file type, so a single value holds and provides access to a single file-resource that has been downloaded as part of the data set.

[2] In some cases it may also be desirable to allow users to compose their own queries.

work with all data as live values, at least at the prototype and testing stage. If the data set is sufficiently large, a separate step can then augment the application with abilities to query and interact with persistent storage rather than only live values; but this can be deferred until after queries, types, and GUI designs are well-developed without the added complication of external query processing.

Accordingly, this paper will focus on "querying" live, in-memory data using raw code rather than query languages *per se*. I will refer to examples implemented in Qt/C++, which has the benefit of the sophisticated Qt reflection mechanism as well as the relatively strong C++ type system, that allows for many functional-programming techniques (even though C++ is not a functional language as such [36], [7], [32], [23], [26]). My overarching claim is that semantically rigorous and systematically organized data sets should often rest on a "queryable data set" technology whose query environment is based, at least in part, on "queries" evaluated directly in application code. Moreover, "queryable" data sets, and their accompanying applications, will often benefit from a hybrid coding style using both Functional and Object-Oriented techniques.

They may also benefit from a hybrid style mixing narrowly-defined types with more general, dynamic types. In some contexts, "dynamic" and "static" type-concepts mix together: for example, the **QVariant** class allows many sorts of values, and so supports idioms more familiar in dynamically-typed scripting languages. However, user-defined types in **QVariant**s (those not part of the built-in Qt library) need to honor requirements relating to initialization, serialization, and compile-time declarations, so **QVariant**s despite their dynamic flexibility are subject to fairly strict type-checking.

One of my goals in this paper is to describe a "strongly typed" query framework by writing and evaluating queries in a reasonably strongly-typed language like C++ — actually, using C++ idioms chosen to maximize compile-time semantic checks. I contend that a query framework rooted in application code and types provides greater semantic rigor and modeling transparency than a strategy based on external databases and database-specific query languages. At the same time, I envision this underlying framework as a foundation for programming paradigms that stretch outside application code and even outside the built-in capabilities of application-development languages. A production environment employing similar code to the demo accompanying this paper would need an infrastructure extending beyond the underlying "queryable data set" code library. For instance, the demo requires the Qt Meta-Object System, which, technically speaking, augments the C++ language itself (in particular, it involves an extra compilation step). When used with published data sets, the techniques described here will also benefit from machine-readable descriptions of how code artifacts (like types and functions) map to data resources (like file types and serialized collections of values, insofar as they can be associated with application-level data types). Such associations can be asserted via annotations embedded both in code and in serialized data, which define structured code-to-data relationships — and therefore also call for tools to parse and act upon such annotations.

Furthermore, it may be impractical to write all queries directly in (say) C++, even if providing a library which *allows* direct C++ query-coding is a first step toward building a semantically rigorous, strongly-typed query environment. This raises the question of how the query-evaluators can be exposed to scripting engines and how query code can be dynamically generated. Taken together, a robust "queryable data set" therefore needs reflection, code-generation, and code-annotation technologies that provide an interesting case study for both language-level and tool-level support.

From the perspective of Software Language Engineering, scripting, reflection, and code-annotation are facets of implementation that tend to straddle the line between the capabilities of a language proper and those of its tools and IDEs. A technical question for SLE is whether and to what degree these facets are better implemented in the language (compilers and runtimes) itself, or in tools; and how the language-based and tool-based responsibilities can most effectively interact with and complement each other. The overall space of open data sets — including querying but also visualization, integration with academic texts, etc. — is a useful real-world domain for exploring these SLE questions.

A single data set, in isolation, may not be subject to change (though it also might be repeatedly updated); but how collections-types *may* change is an intrinsic aspect of their semantics, intrinsic to how the data evolves into its published form. How a data-set list of values accreted — as a stack, queue, ordered set, etc. — deserves to be modeled in its own right. So a data set may employ a range of collections types, even if some of these differ only in how they are updated. Collections have different internal organizations: they can be ordered or unordered; traversed in different directions; updated in different ways. A data set's semantics therefore includes how types map to various collections types. There are also other kinds of inter-type associations: the type representing numeric values measured by a scientific instrument can map to a timestamp type representing the moment when a measurement is taken, for example. Or, a type representing personal information maps to a type specifying access controls to protect privacy [17], [14], [31]; [10], [9], [30].

In short, most data sets reveal an assortment of data types with variegated inter-relationships, and how these types are defined and connected serves to semantically model the scientific/technical domain from where the data originates. The set's internal type-space is a logical extension or codification

of the scientific (or mathematical, engineering, etc.) principles structuring the research which a data set summarizes. This type-space plays both a practical and a theoretical/expository role. The significance of strongly-typed query frameworks is that they not only provide useful query evaluations; they also exhibit and operationalize a data sets' internal schema of types and their associations [43], [44], [16]; [12].

On the other hand, data sets may be most valuable when they are not used in isolation, but can also be synthesized and integrated with other data sets. So a fine-grained type-space hewn to data sets' internal, technical/scientific semantics should be augmented with broader types with less compile-time enforced semantics, for contexts where data is integrated from multiple sources. A comprehensive framework for querying data sets therefore requires a series of types which occupy different levels of semantic precision, and which reflect a variety of type-theoretic paradigms.[3]

When we lay out the requirements for a type system conducive to queryable data sets, the evidence consequently suggests that hybrid paradigms are better than monolithic ones. A scientifically thorough and practically effective type system needs to have both Object-Oriented and Functional characteristics; and to accept both more static and more dynamic programming styles. Each of these paradigms are appropriate type disciplines in their proper contexts.

Type theory in this point of view is not only present in its abstract mathematical guise and its practical, programming guise; it is also a technical link between scientific models and digital (computational, source code, HCI) models. The overarching organization of a data set is reflected in the overarching outline of its type relationships: does the data have a spatial, or a temporal dimension, or involve (maybe privacy-sensitive) personal information? These qualities correspond to inter-type maps: going from the type of an observed or measured value to the types of the spatial, temporal, or personalistic data providing observational context. Statistical properties of measurement dimensions (Nominal, Ordinal, Interval, and Ratio, for example) are expressed as operators defined (or not defined) on corresponding data types [37], [45]; [11], [5]. So a curator planning the large-scale organization of a data set is also deciding the central veins of the type system through which the data set will be digitally encoded.

[3] Similarly, evaluating complex queries calls for functional-programming techniques, because the code which handles successful queries is best treated as a function-typed value that can be passed between functions [27], [28]. Composing multiple queries then amounts to defining combiner operators on these functional values — for example, one such value may implement an anonymous function which performs another query, providing then another function-typed value, creating a chain of tests like an AND clause [18], [21], [41], [24]. On the other hand, Object Orientation allows queries to by dynamically mapped to application-level functions. Dynamic queries must refer to certain functions (like accessors to data attributes) via some kind of code or character string, rather than internally via function or member-function pointers [8], [4], [3], [6], [29]. In effect, these queries — filled in with script- or end-user-provided values — need to map names to function pointers and/or (as in the code autogenerated by QT's Meta-Object Compiler) **switch** on numeric or **enum** values standing in for class methods. The Object paradigm offers a rigorous foundation for reflection in this sense because dynamically invokable functions tend to form subsets of classes' public interface.

In this paper I will illustrate these principles by focusing on one specific kind of data set organization. Specifically, I will concentrate on a structuring design oriented toward "Events", which in turn are associated with measurements or assertions, which I'll call "Observations". Most content in such a data set is temporally organized; at a given time point there is some value that has been obtained from some sort of measurement or can be somehow asserted. These values may, but need not exclusively be, those "measured" in the manner of scientific instruments recording some quantity. For example, a student's score on a particular test is an Observation, tied to an Event, because it has both observed values and a point in time. But a teacher's decision to give a student remedial tutoring, and the details of this decision — the number of tutoring sessions and subject areas covered — can also be modeled as an "Observation". Observations may record human choices (an electoral vote, a commercial purchase, etc.) as well as actual measurements of physical magnitudes.

I will more specifically focus on organizations where Events and Observations are connected to *Grounds*, some value which ties series of event/observation pairs together. One student's test results, over time, collectively define a "worldline", a trajectory which can be treated as an integral whole because it has a "ground" in a person; Ontologically speaking, an enduring entity. Similarly, biomedical test results from one patient define the trajectory of their treatment over a course of time (say, a hospitalization, or a clinical trial) and, on a larger time scale, define a patient history. So Events and Observations are triangulated to "Ground" objects which provide a perduring context, one that stretches across time-points to unify Events into a history or worldline. Together I will refer the the Event-Ground-Observation organization of a data set as an "EGO" model.

Within an EGO data set, an additional layer of technology and organization can then produce a "Cohort" model, where sets of ground-objects are selected based on queries combining Event, Observation, and Ground data. The objects in a cohort share some common properties and history, although defining their shared background may demand multi-tiered queries: what they have in common may be identified not in one single Event or Observation, but instead in terms of the length of time between Events and/or quantitative differences between Observations.

The "Ground" objects I used as examples are persons (students, clinical patients), but EGO models can have other groundings: ground-objects might be nations, with Observations as macro-economic and sociopolitical data and Events as elections, laws enacted, as well as time points associated with economic measures (growth rate at the end of a quarter, say). If this is extended to a Cohort model, then Cohorts

might be nations whose economy expanded above a certain rate in the years after a policy decision like signing a trade deal. Or in other biological sciences: a ground-object may be a geospatial location where a bioacoustic sensor is deployed; Observations are quantitative models of acoustic signals and corresponding inferred species; and Events are the recording of a particular signal at a particular place and time. A Cohort may then be a map of geographic points where bioacoustic analysis suggests robust biodiversity, with multiple species recorded at well-distributed time intervals [35].

Once a Cohort is defined, it can be reused for further analyses. Cohorts offer different perspectives on a data set than "raw" event data, which can only be contextualized by basic temporal criteria or by "history" grounding vis-à-vis a single ground object. As a concrete example of Cohort models, consider "Clinical Looking Glass", a query and User Interface model developed at Albert Einstein College of Medicine [2], [1]. In this system, combining (what I am calling) Event, Observation, and Ground data produces a query architecture conducive to "patient-centered" care because it allows multiple events to be queried both within a single patient's timeline (like a diabetes diagnosis and results of subsequent tests) while also contextualized in Cohorts, such as all patients who receive a particular course of treatment.

Internally, Clinical Looking Glass generates queries combining each point on (what I call) the "EGO" triangle. This framework was built to improve on database layers that could evaluate temporal queries in isolation, or vis-à-vis single patients in isolation, but could not formulate more holistic queries that are both patient-centered (taking patient-specific events as criteria) *and* conducive to detailed comparisons between patient outcomes (via Cohorts). Conventional biomedical data stores support temporal queries on impersonal time-scales (how many patients are discharged from a hospital in a given month) but struggle with "patient-centered" queries whose time-axis must be keyed to patient's own histories (for instance, time "zero" being the moment of their hospital admittance). Via Cohorts, each patient (or more generally Ground Object) can carry its own "time zero", while still allowing comparisons across ground-objects based on time *intervals* evaluated respectively in each worldline.

As this example suggests, Cohorts extend EGO by varying how temporalized queries are interpreted: they can describe intervals within each ground-object worldline as well as time points and intervals in "collective" or "calendar" time. Clinical Cohort Models also point to practical considerations for integrating many data sources, because in principle biomedical data from many providers — not just one hospital system — can be integrated in similar ways.

While Biomedical use-cases like Clinical Looking Glass are instructive, I will focus here on simpler, hypothetical/expository education-related examples, illustrated via the accompanying code. I will use raw-coded queries to show how queries can be merged and integrated within a programming-language type system, making data sets "queryable" whether or not they actually have a database representation.

1 Heterogeneous Data Architecture

The Cohort Model used by Clinical Looking Glass typifies multi-paradigm data integration because it encompasses different kinds of data, which have different kinds of *roles*. One kind of data is the *patient*, which plays what I call a *Grounding* role. A unique patient identifier grounds and unifies multiple events, so that we can trace clinical histories and course of treatment. A second role is played by actual medical data, which I call *Observational*. These may be direct observations, like a blood pressure reading, or results from more complex clinical exams. Tests can generate a lot of information, but the data that gets directly integrated into the EGO queryable data set is usually in a summary form, like a tuple of numbers (and/or nominals). Finally, the building blocks of EGO data are *Events*, which have temporal form and unify "Grounding" and "Observational" data. So the basic data model unifies data playing the *Event*, *Grounding*, and *Observational* roles: each Event is the point or region in time wherein an Observation has been made in a Grounding context.

In educational software, the *Grounding* might be a student, the *Observation* might be assessments in one or several subjects, and the *Event* would be a time-specific assessment process (say, a test taken and graded). Observations can take different forms at different times. A practice exam can be graded to a simple tuple (percentages for reading, writing, and math, say, or "RWM"), or an observation can be one teacher's evaluation for one subject. Observations could also take the form of assertions about a course of study — that a particular student began to receive algebra tutoring in January, say, which continued until March. By combining temporal criteria with different sorts of Observation, we can formulate complex queries: How many students who received Algebra tutoring for two or three months in the Spring showed at least 25% practice test math improvement the following Fall?

Each Event "projects" in two directions. From one Event — say, a test result — we can get details about the student, or query against the Observation (say, an "RWM" tuple). Modeling this in an OO language, we have a **testResult** object (representing the Observation); for RWM this class could have methods like **getMathScore** and **getReadingScore**. We also have a **student** object which grounds and serves as a center for many kinds of observations and data (test results, demographics, personal information, written evaluations by teachers and other educators). Each *Event* unifies a **student** and Observation object, so given an **event** object, we can obtain an associated **student**, or an associated Observation like a **testResult**, and

also query for temporal relations (whether an event occurs before or after a time-point or another event, or within a time-interval). So an EGO model (in an OO implementation) requires at least three classes (one each for "E" and "G" and at least one "O"). Consider these in turn.

First, the *Observation*. The basic criterion of an "O" class is that it should enable comparative queries that can reach across temporal and grounding data. For example, a test result can be compared to other tests by the same student at other times, or by other students at the same time. Most Observations unify multiple attributes, which for simple cross-Observation comparison will take numeric form (allowing greater-than/less-than comparisons) or at least nominals (allowing equal/not-equal and classification/sorting). So a test result can be summarized as numeric tuples, each of which provides an axis of comparison to other results. Or, when identifying that a student received tutoring, the Observation may use an enumerated list to identify which subject was isolated for remediation (arithmetic, algebra, vocabulary, reading comprehension, spelling, clarity of writing, and so forth).

Consider a basic "RWM" date structure, with attributes for reading, writing, and math scores. As a Relational Database table, these scores would represent table columns. In an OO class, they are presumably accessor methods (**get_math**, etc.). To make the RWM class *queryable*, it needs more structure than just accessors to the three numeric attributes.[4]

Listing 1: A Simple "Reading, Writing, Math" Class.

```
class Test_Result : public QCohortObservation
{
 Q_OBJECT
 int reading_; int writing_; int math_; (1)
 ...
 Q_INVOKABLE int get_math() const { return math_; }
 Q_INVOKABLE int get_ordinal_comparison
   (const Test_Result& rhs, int index) const;
 Q_INVOKABLE int get_ordinal_comparison (2)
   (int score, int index) const; ...
```

For discussion, Figure 1 shows part of a C++ implementation of an RWM class. The underlying data provided by this class is an RWM tuple (1), but the class also provides an interface to mimic SQL queries. For different kinds of database query models, we can identify what are salient features usable for selecting and filtering data, and then how to expose similar features by ordinary function calls in OO or functional contexts. Other kinds of data models call for other kinds of query models.[5] These differences need to be mirrored in how "queryable" objects' methods are designed.

A basic RWM tuple is clearly ordinal, so it must support comparative-magnitude queries. This is modeled through the methods labeled 2 in the sample code, which take a value or a reference to another test, and a numeric code identifying one of the axes of comparison (the **reading**, **writing**, and **math** attributes), returning a signed integer.[6] The attributes themselves (private data members) are mapped to numeric codes via the QT "meta-object compiler" (MOC), a QT-specific code generator that preprocesses C++ header files and outputs implementations of "Meta-Object" classes. Specifically, the MOC assigns numeric indices to certain methods, and the code sampled in Figure 1 uses the indices of get-accessors as codes mapping to/from the RWM attributes (e.g., the index for the **get_math** method serves as a numeric code for the "Math Score" attribute). In Figure 2, **get_attribute_code** converts a text string (like "MathScore") to the associated attribute's code, allowing queries to be derived from text sources (such as scripts) but also ensuring that the actual **get_ordinal_attribute_comparison** method is only called with a valid code (because an unchecked string cannot be used for the query; it has to be mapped to the attribute code instead). QT's Meta-Object system uses a special **QInvokable** flag to identify those methods which are indexed, and thereby can be called via meta-objects and string representations of method names. This is the primary mechanism whereby QT applications can be "scripted", using languages like LISP or PYTHON to fine-tune applications which are already compiled and deployed. The same technology can be used to automate the connection between GUI front ends and Data Sources: for example, QT applications can implement front-ends similar to Clinical Looking Glass, where users interactively build queries by selecting queryable attributes from drop-down menus.[7]

Listing 2: Using Qt Reflection to Lookup Class Methods.

```
int get_attribute_code(QString attr)
{
 int index = attr.indexOf('/');
 if(index == -1) return -1;
 const QMetaObject* qmo = get_metaobject_by_type_name
   ( attr.left(index) );
 if(qmo)
 {
  QString mn = attr.mid(index + 1);
  mn.append("()"); mn.prepend("get_");
  return qmo->indexOfMethod(mn.toLatin1());
 }
 return -1;
}
```

[4] Analogous to a Relational query filtered via a where-clause like **WHERE math** > 80. In Object-Oriented Programming environments, of course, there is no direct equivalent to a "query engine" or a WHERE clause; so a "queryable" class has to provide this semantics — not just attribute **GET**ters — indirectly.

[5] For example, doing social network analysis involving a student's friends and classmates might call for graph connectivity metrics which depend on traversal rather than comparative-magnitude queries (as in, How much are students' math scores influenced by their peers?, using graph analysis to identify peer groups).

[6] Positive if the THIS's score is better than OTHER's in the specified dimension, negative if the opposite is true, and zero if the scores are equal.

[7] In the sample code, classes inheriting **QCohortObservable** are expected to conform to conventions governing how queryable attributes are accessed and named. Via these conventions, a GUI builder can identify these attributes within any class whose objects can be used by Cohort queries, and allow users to construct these queries via GUI actions. Note that in Clinical Looking Glass itself, GUI actions are ultimately translated to SQL; in the demo, similar query models are instead directly interpreted as method calls via QT reflection; there is no actual "query code" generated.

Table 1 Some Dimensions of Comparison, and their Domains of Comparative Values

Dimensions	Values (by name)
Nominal	Equality, Non-Equality
Ordinal	Greater-Than, Less-Than, Equal, Greater-or-Equal, Less-or-Equal
Uniqueness	Identity, Non-Identity
Taxonomic	Unrelated, Same-As, Subsumes, Subsumed-By
Temporal	Before, After, Simultaneous, Not-Before, Not-After

In the example, the key query method is called **get_ordinal_attribute_comparison** because the attributes are compared by magnitude. Other classes can have similar methods for different kinds of queries: for example, a **get_nominal_attribute_comparison** would return either "equal" or "not equal". Suppose an object models a particular student being helped by a particular tutor in a particular subject, with a simple subject list like ⟨Algebra, Arithmetic⟩. The "subject area" attribute is then purely nominal, having no comparison beyond basic equality. Alternatively, subjects may be classified taxonomically, so Math is recognized as a subject subsuming Algebra and Arithmetic. Then there are four possible comparisons between attributes: they can be the same, unrelated, or one can subsume the other. So a **get_taxonomical_attribute_comparison** method should return one of four values representing these cases. There are other statistical categories as well, like "Interval" and "Ratio": Dimensional Analysis reveals a diversity of measures for comparing attributes based on their statistical profiles (as summarized in the above Table).

Cumulatively, the kinds of dimensional comparisons proper to different attributes exposed by a data structure define a query model specific to that structure, and an overall Query Model represents the total range of dimensional comparisons that may be recognized vis-à-vis queryable objects. This dimensional analysis is reflected in methods' names and signatures, so a class interface provides (or encodes) a semantic overview of the implemented data model with an emphasis on how it can be integrated into queries, analogous to a database schema but wholly implicit in source code. Different classes in an EGO model play different roles, and these differences are reflected in their specific query models; in the dimensional analysis of their attributes.

With respect to *Grounding*, ground objects are structurally and dimensionally different from Observation objects. The role of ground-objects is to unify many data points, so the objects themselves do not necessarily hold many attributes that allow cross-object comparison (the way that the quantitative data in the **Test_Result** example allowed for obvious cross-comparison between tests). In a hypothetical school example, a **Student** class may be an aggregate of other cases,

Listing 3: An "Event" class for an EGO Model

```
class QCohortBasicEvent : public QCohortEvent
{ ...
public:
 ...
 signed int
 get_temporal_comparison(const QCohortBasicEvent& e) const;
 ...
}
...
template<> class Comparison<Dimensions_Of_Comparison::Temporal>
  : Comparison_Base
{ ...
 template<Dimensions_Of_Comparison dc>
 Comparison<dc> after_and(std::function<Comparison<dc>()> fun)
 { return _and(1, fun); } ...
```

(1) (2)

each exposing a different facet of student data: demographics, personal information, course history. Personal Data may be placed in a separate class because it requires extra permission to access; Demographic data may be kept together so it can be integrated with queries (analogous to demographics in Clinical Looking Glass). Course-related data is likely to be predominantly "many-to-many": each student has many teachers, and vice versa. These different kinds of data have different "shape" and need to be accessed in different ways, in part for reasons of privacy and security. In short, a good design for "ground-object" classes may be to avoid making these classes *themselves* queryable, but rather making them umbrellas for other classes, some of which are queryable and combined with Event and Observation queries.

Finally, *Event* objects intrinsically represent temporal and temporalizable queries. Figure 3 shows part of a QT class representing Events in a Cohort Model context. The methods labeled (1) and (2) reflect dimensions of comparison specific to events: two events can be before, after, or simultaneous. These queries are provided via **get_temporal_comparison** methods, intended for use similar to attribute-comparison methods discussed for Observations.[8]

As outlined here, the basic strategy for implementing an "EGO" model is to perform a kind of elementary dimensional analysis based on statistical properties of the kinds of data that need to be queried — what are their dimensions of comparison?[9] If the system is implemented using QT, the Meta-Object Compiler will extract metadata directly from source

[8] A more sophisticated event class may also allow comparison between an event and a time-interval (inside or out) or an event-pair (between or not). But a simpler Event-class as in the demo needs a coarser algorithm for comparisons. In particular, the underlying data for the demo class is not a single time-point, but rather divided into fields like day, hour, minute, and second. The rationale for this data model is that systems may record different events at different levels of temporal granularity, and comparisons should not confuse this difference with recorded before and after. For example, in many contexts a test taken on May 1st should be deemed simultaneous to a test taken on May 1st at 2 : 00PM (so May 1st is not the same as "May 1st at Midnight").

[9] A more robust production system might need a more rigorous dimensional analysis and directly model details like units of measurement and reasonable ranges — which can be expressed in a class interface, for example by declaring Attribute-Comparison methods which compare value-plus-measurement-unit pairs instead of just values.

Listing 4: Installing an Embeddable Common Lisp Callback

```
#define BASIC_DEFINE_CALLBACK(x, y) \
define_callback( [] (void* pass_on,
  cl_cxx_backend::cl_arglist arglst) -> cl_object \
{ \
 Runtime* r= reinterpret_cast<Runtime*>(((void**)pass_on)[0]);\
 Generator* g = r->generator(); \
 if(arglst->frame.size > 0) \
 { \
  QList<MS_Token> tokens; \
  cl_arglist_to_ms_tokens(size, 0, arglist, tokens); \
  if(g) { g->y(tokens); } \
 } \
 return ECL_NIL;
}, "KB", #x, pass_on);
...
void define_callback(cl_cxx_backend::callback_t cb,
  QString package_name, QString symbol_name, void* argument)
{
 cl_object obj = cl_cxx_backend::symbol(
   package_name.toUpper().toLatin1().data(),
   symbol_name.toUpper().toLatin1().data());
 cl_cxx_backend::define_function(obj, cb, argument);
}
```

code which can map text names to queryable attributes, so scripts can declare queries using SQL-like-code that can be dynamically mapped to method-calls (Figure 4 shows the kind of code that registers C++ callbacks as Embeddable Common Lisp functions). Moreover, the text-to-attribute mapping can be used in reverse, allowing all queryable classes to be examined iteratively and compiling a list of queryable attributes to populate a drop-down list of options at the GUI, so users can create queries visually on-the-fly.

These techniques are somewhat similar to "Object-Relation Mapping" frameworks that rely on code-reflection, but there are important differences: the queries translate to method-calls which can work by examining object data directly rather than producing SQL and querying a database; and the methods work for many kinds of queries, not just those endemic to Relational Databases. For sufficiently large data, it may be impractical to fully eliminate database queries in favor of "live memory".[10] So to scale up a system needs to route queries in ways which *do* defer to an external database *when necessary*. However, the semantics, architecture, and User Experience can be established using the *internal* query mechanism as the definitive data and query model used by the system. The engineering process should start by building and refining a working prototype where all queries are resolved internally, and only then build alternative versions of just those parts needed to interface with an external Data Source, minimizing the amount of code for which "internal" and "external" queries need to be distinguished.[11]

[10] A few thousand test results as "Observation" objects in an EGO can be processed as ordinary QOBJECTS in an application; a few million — in a government database, say — probably cannot.

[11] One benefit of developing an *internal* query system is that decisions on how to export data to a database can be delayed until the system has been explored and prototyped. To take the bioacoustical example, specialists in this field have acknowledged that conventional databases are a poor match for bioacoustical data. If biologists deem that an EGO model is a good fit for interactively exploring bioacoustical data sets, the application can be designed around internal queries until the design is judged successful (the User Experience serves researchers' productivity); then the question can be addressed as to which database technology best serves the application in its prototype form.

In any event, my discussion so far has considered the EGO components in isolation, whereas the strength of the Cohort Model lies in how the parts are combined. Having previewed how the individual query-models suitable for Events, Ground-objects, and Observations can be expressed in an OO class interface, I will now focus on the unification of these query models.

2 Query Models and Inter-Type Relations

The *Event*, *Grounding*, and *Observation* components of an EGO model are logically separated because they have different dimensions of comparison. The key to the Cohort Model is building multipart queries that require multiple forms of comparison, so an OO implementation of a Cohort Model needs to pass queries between the EGO end-points. For this discussion, assume our goal is to build QT/C++ classes that provide similar query mechanics to SQL, targeted to a data set wih an EGO — and specifically a Cohort — model; but works on object data rather than an external database. Assume that this implementation will be centered on a **QCohortModel** class that provides an overview of the overall system, along with **QCohortEvent**, **QCohortGround**, and **QCohortObservation** base classes that are subclassed to form specific EGOs. Finally, assume that there is a **QCohortNexus** class that unifies a single event, ground, an observation: through an instance of this class (a "nexus") the individual EGO objects can be obtained.

For illustration, assume that each Nexus class is joined to a unique trio of EGO classes (this can lead to many different Nexus classes being created, but it is a simplifying assumption to start with). Figure 5 shows a sample Nexus class defined to work with the **Student** and **Test_Result** classes sketched in the last section. The code on the line labeled ❶ shows that this Nexus class is derived from a three-part template identifying the relevant EGO classes (where **QCohortBasicEvent** provides basic time-comparisons that can be used with multiple Nexuses). Inheriting from **QCohortEventNexus<QCohortBasicEvent, Student, Test_Result>** gives this class **protected** access to pointers representing the EGO classes.

The previous section discussed method calls to EGO objects directly, but here I will describe code that routes queries through Nexus objects instead. Rather than expose EGO objects to outside code, **QCohortEventNexus<EVENT_Type, GROUND_Type, OBSERVATION_Type>** holds them (kind of) like a Haskell monad (❷), and supports

Listing 5: Querying via an Event Nexus

```
class Test_Result_Event_Nexus : public QCohortEventNexus
  <QCohortBasicEvent, Student, Test_Result>  (1)
{ ...
 const Test_Result& test_result() const
 { return *observation_object_; } ...
 QVariant_Event_Nexus* to_qvariant_nexus();
};
...
// QCohortEventNexus.h
class QCohortEventNexus
{
protected:
 EVENT_Type* event_object_; ...  (2)
public:
 typename EVENT_Type::Comparison_Result_Monad_type
 event_comparison(const QCohortEventNexus& this_type,
   std::function<typename EVENT_Type::
   Comparison_Result_Monad_type
   (const EVENT_Type&,const EVENT_Type&)> fval)
 { ...
  return fval(*event_object_,*this_type.event_object_); (4)
 }
```

```
nexus.event_comparison(nexus1, [](
  const QCohortBasicEvent& e1, const QCohortBasicEvent& e2)
{
 return Comparison<Temporal>
   ( e1.get_temporal_comparison(e2) );  (5)
}).before_and<Ordinal>([nexus, nexus1]()  (3)
{
 return nexus.observation_ordinal_comparison(nexus1,
   [] (const Test_Result& t1, const Test_Result& t2)
 {
  static int index = get_attribute_code(
    "Test_Result/math");
  return Comparison<Ordinal>  (6)
    ( t1.get_ordinal_comparison(t2, index) );
 });
}).lt_do([nexus, nexus1]() { ...
```

queries by taking a callable value as an argument, yielding to this value with the **private** object (3 shows this on the calling side, using a C++ lambda). The return marked 4 invokes another monadic construction for Nexus queries. This last monad-like value captures comparison results via types specific to the kind of comparison performed, and will be discussed further in the context of multi-dimensional queries.

Since queries involving ground-objects can be more complex, I will first review Event-Observation combinations, such as an improvement in math scores: a second test-result event is *later in time* than the first, and the second's math score is *greater in magnitude*. The code at 5 and 6 in Figure 5 shows shows individual queries for each condition, and note that each query returns a monad that contains a value representing the comparison result, but via a type specific to the dimension of comparison. Figure 3 shows parts of code defining **Comparison_Result** classes, where relations like *time* A *is later than* B are distinguished from, say, *magnitude* A *is greater than* B. For example, the C++ type specialized as **Comparison_Temporal** has methods with names such as **before_and**, whereas the analogous methods for **Comparison_Ordinal** is called **lt_and**.

The consequence of these naming conventions is that code declaring queries needs strong semantic clarity in order to compile. Code has to model statistical and dimensional qualities more rigorously than either normal query languages or "normal" (say) C++ code, which is more likely to sift through data collections with brute force (**if(age_ < rhs.age_)**, and so forth). This strongly-typed query semantics extends to how simpler queries are combined into complexes. In the demo, query results are not used directly; instead, the comparers which yield these results take functions to call when comparisons succeeds (and sometimes when they fail). These functions may then add new query criteria, producing a "chain" of comparisons expressing an aggregate query.[12] The reason for this indirection is flexibility: a similar code base can be used for different query models, for example a more nuanced model of events with a wider spectrum of comparative relationships (if events are considered to be enduring rather than instantaneous, then immediately we have a network of relations similar to an Interval or Region-Connection Calculus (RCC) [19], [33], [25], [39], [40], [22], [20], [38]).

These "chains" of method-and-lambda calls allow for a kind of Lazy Evaluation in C++. As shown in Figure 5, most of the code blocks are wrapped in "lambdas" (that is, anonymous inline functions) and will only be called if all preceding queries return positive. Moreover, the types declared for "links" in the chain structure how queries can be combined. For instance, the **Comparison_Temporal** class does not have "do" methods analogous to "and"s; there is no **Comparison_Temporal::before_do**. The reason is that in an EGO Model, events (that is, the basic time-data for an event) are always intended to be queried in combination with other data. So a mere comparison of two events, taken out of context, is incomplete; the design of the **Comparison_Temporal** class prevents incomplete queries from being accessed in isolation. Also, method names ensure that calling code understands which dimensions of comparison are in effect at each step in a "query chain". A compilation error will result if code tries to call **lt_and** rather than **before_and** on a **Comparison_Temporal** object.

The current example covered a "two-part" chain combining Event and Observation queries, but not the Ground. To see whether a student's math score improved between tests, it is obviously important to check that the same student took both tests. So we want to express a query assuring that the student associated with one **Test_Result** object is the same as the student associated with a second **Test_Result** object. Such a "Uniqueness Query" is similar to a "Nominal" query in that, for Nominal data, the only basis of comparison between two values is equal or not-equal. However, the conceptual role of Nominal data is typically used to categorize complex data values into groups: voters into party affiliations, people into genders, addresses into postal codes, etc. It is assumed that many voters will be assigned to one party, and postal codes encircle many addresses; the labels which define the domain

[12] In Figure 5, for instance, the **before_and** method executes its received function only if the **Comparison_Temporal** object holds a value consistent with the pair of event objects (from which the comparison object is created) reflecting a before-and-after relationship. Internally, numeric codes are used to mark how pairs of values compare, but these codes are never accessed directly by application code.

of a Nominal data do not usually uniquely identify any object which has that label as a attribute. By contrast, I will use the phrase "Uniqueness Data" to describe values which, when they are an attribute of some other object or data structure, *uniquely identify* that object. For example, students may have many attributes recognized in a database (age, gender, address), but only specially designed attributes — like a **student_id** — are guaranteed to be unique for each student.

Given these extra logistical requirements on Uniqueness, it is a good idea to distinguish Uniqueness queries and the Uniqueness "Dimension of Comparison" from other queries. Internally, a Uniqueness attribute (like a **student_id**) may be represented as a simple numeric code, but it may be good design to encapsulate access to this code in a separate class, to ensure that a unique identifier is never improperly reused — and also to ensure that there are carefully designed protocols for obtaining these identifiers given sufficient alternative information. In the case of a **student_id**, what combination of data (name, address, date of birth) should be deemed sufficient to uniquely designate a student, so that code can be implemented to retrieve an id on the basis of this data? And how should requests to learn an id be authorized? These are design choices which are orthogonal to most other data management vis-à-vis a single student, which is why they should be developed in a specialized class in isolation.

Listing 6: Students' Uniqueness Comparison

```
try
{
 nexus.ground_uniqueness_comparison(nexus1, []
   (const Student& s1, const Student& s2)
 {
  static int index = get_attribute_code
    ("Student_Uniqueness_Ground/id");
  return Comparison<Uniqueness>( s1.uniqueness().
    get_uniqueness_comparison(s2.uniqueness(),index));
 }).same_do([]()
 {
   qDebug() << "Yes, the student ids are the same.";
 }, /* or else */ []()
 {
  qDebug() << "The student ids are different.";
 });
...
catch(Unrecognized_Attribute_Index_Exception uaie)
{ qDebug() << "Wrong Attribute Index: " << uaie.index; }
catch(Dimension_Of_Comparison_Mismatch_Exception dcme)
{ ... }
```

Figure 6 shows code where a "uniqueness query" compares two students (a deliberate error for purposes of exposition causes the query to fail). Figure 7, by constrast, shows a compound or "chain" query intended to compare *different* students. Both Figure 6 and 7 show the **Student** objects passed as arguments to functions declared by the "Event Nexus" type, but the lambdas passed to these functions use narrower types included in **Student** objects (the "Uniqueness Ground" and "Demographics", respectively). The class relationships can be seen in terms of containment/composition: the Event Nexus branches

Listing 7: Combining Ordinal and Nominal Queries.

```
nexus.ground_ordinal_comparison(nexus1, []
  (const Student& s1, const Student& s2)
{
 static int index = get_attribute_code(
   "Student_Demographic_Data_QObject/age");
 return Comparison<Ordinal>(s1.demographic().
   get_ordinal_comparison(s2.demographic(), index) );
}).eq_and<Nominal>([nexus, nexus1]()
{
 return nexus.ground_nominal_comparison(nexus1,
   [] (const Student& s1, const Student& s2)
 {
  static int index = get_attribute_code(
    "Student_Demographic_Data_QObject/gender");
  return Comparison<Nominal>( s1.demographic().
   get_nominal_comparison(s2.demographic(),index));
 });
}).ne_and<Ordinal>([nexus, nexus1]()
{
 return nexus.observation_ordinal_comparison(nexus1,
   [](const Test_Result& t1, const Test_Result& t2)
 {
  static int index = get_attribute_code(
    "Test_Result/math"); return Comparison<Ordinal>
     (t1.get_ordinal_comparison(t2, index) );
 });
}).lt_do([nexus, nexus1]() {
 qDebug() << "The female student outscored the male student: "
});
```

out to Event, Observation (Test Result) and Ground (Student) objects, and **Student** branches out to Uniqueness and Demographics. Queries "follow" these branches to reach the levels in the containment hierarchy which fit the queryability profile that works for a particular conceptual investigation that needs to be expressed in code: comparing male and female students' test results given the same age, comparing earlier and later results for one student, etc.

When implementing queries against application data types, we are not searching against types in general — only against types which are "queryable" in a fashion workable for a given query. So "executing" the query involves traversing inter-type graphs to find the correct data types to participate in the comparison described by the query. In the EGO model, the root of this traversal is the Event Nexus, which connects to an Event object for temporal queries, to a Ground object for Uniqueness queries, and to an Observation object (plus classes associated with the Ground) for other kinds of queries. Each EGO combination provides a distinct Event Nexus type. Queries which are hard-coded in C++ code get rigorous type-checking and serve as demonstrations for each types' query interface. However, such hard-coding is not necessarily consistent with the dynamic framework offered by systems like Clinical Looking Glass, where queries can be constructed on the fly via a GUI, without the user writing code.

This limitation can be redressed by sacrificing some compile-time type checking for more runtime flexibility. The remaining code in the demo shows variations in the EGO framework to make some query details dynamic rater than hard-coded. Figure 2, for example, shows a function which takes attribute names as character strings and uses QT reflection to map names to index codes. More generally, Figures 8-9 show

Listing 8: An "Observation" Class Based on QVariant Values

```
class QVariant_Observation:public QCohortObservation
{
 Q_OBJECT
 QMap<QString, QPair<Dimensions_Of_Comparison,QVariant>> data_;
 ...
 Q_INVOKABLE int get_comparison(QString index,
   const QVariant_Observation& rhs, QString rhs_index) const;
 Q_INVOKABLE int get_comparison(QVariant val, QString index)...
 void absorb_data(QDataStream& qds);
 void supply_data(QDataStream& qds) const;
};
QDataStream& operator<<(QDataStream&,
  const QVariant_Observation&);
QDataStream& operator>>(QDataStream&, QVariant_Observation&);
```

Listing 9: Supporting Multiple Dimensions in a Single QVariant-Based Comparison Provider.

```
int QVariant_Observation::get_comparison(QString index,
  const QVariant_Observation& rhs, QString rhs_index) const
{
 auto it = data_.constFind(index);
 if(it == data_.cend())
   throw Unrecognized_Attribute_Index_Exception({index});
 QVariant qvar = it.value().second;
 auto it1 = rhs.get_data().constFind(rhs_index);
 if(it1 == data_.cend())
   throw Unrecognized_Attribute_Index_Exception ...
 if(it1.value().first != it.value().first)
    throw Dimension_Of_Comparison_Mismatch_Exception ...
 QVariant qvar1 = it1.value().second;
 switch (it.value().first)
 {
 case Dimensions_Of_Comparison::Ordinal:
   return compare_raw_qv<Ordinal>({ qvar, qvar1 });
   ...
 case Dimensions_Of_Comparison::Taxonomic:
  return compare_raw_qv<Taxonomic>({ qvar,qvar1 });
 }
}
```

an Event Nexus implemented to work with fairly generic data structures, based on **QVariant**. The **QVariant** datatype holds many different kinds of data, including many Qt types and user-defined types as well. The **QVariant_Event_Nexus** class uses a **QVariant_Observation** class to hold observation data. Internally, **QVariant_Observation** combines attribute strings, dimensional metadata, and **QVariant** values, so it can represent the same information as most narrower Event Nexus tyes. Figure 10 shows a query combining two different Event Nexus types, where both are converted to the generic **QVariant_Observation** format. So **QVariant_Observation** can be a common format for many different Observation classes, allowing for them to be synthesized at runtime, albeit with fewer compile-time type-checks.

Using **QVariant_Observation**, diverse data sources can all be merged into a single **QVariant_Event_Nexus** EGO class triple. This generic Event Nexus can then be wedded to a GUI implementations where queries can be dynamically assembled from user input. However, the more rigorous type models and graph of inter-type relationships are still necessary for translating type-specific models to generic **QVariant_Event_Nexus** (as I will discuss next section).

To conclude this current section, note that choosing a common **QVariant** data representation requires that all data sharing a Dimension Of Comparison needs compatible **QVariant** representations. This is straightforward for numeric data, but can be a more subtle problem for other kinds of comparison. Return to Figure 9, which uses "Taxonomic" criteria: the values form an inclusion or refinement lattice (in the example, different tutoring topics are included in more general subjects; Group Theory refines Algebra which refines Math). The demonstration code uses a simple taxonomic representation where the name of one topic is joined to a list of character strings identifying more general topics "above" each topic in the lattice. This representation requires a specific algorithm for comparing two topics: whether they are the same, or one is another's "ancestor", or they are unrelated.

So long as queries doing taxonomic comparisons are narrowed to individual types, each type can implement taxonomic representation and queries in its own way. However, when we transition to supporting multiple data sources sharing a common **QVariant_Event_Nexus** type, we also have to define a canonical format for Taxonomically-ordered data, so that queries needing Taxonomic comparisons can be generically executed. The same is true for other non-numeric or qualitative queries that can be required on a data set, like connectivity or Sermantic Web queries supported by graph databases. So a production EGO environment needs to include canonical models for graph data, taxonomies, and other kinds of "qualitative" data.

3 Integrating Multiple Data Sources

Hard-coding queries in a language like C++ provides rigorous type-checking — especially when the types are engineered to recognize details like Dimensions of Comparison. However,

Listing 10: Dynamic Query Fields (1), or Representations (2)

```
void check_tutoring(const Tutoring_Event_Nexus& tnexus,
  const Test_Result_Event_Nexus& nexus, int threshold,
  QString subject, QString topic, int number_of_weeks)
{ ...
 return Comparison<Taxonomic>( Tutoring::get_taxonomic ...
 }).below_and<Ordinal>([nexus, threshold, subject]()
 {
  static int index = get_attribute_code(
    QString("Test_Result/%1").arg(subject.toLower()));
  return Comparison<Ordi  1  nexus.test_result().
    get_ordinal_comparis  reshold, index) );
```

```
tvnexus1.observation_ordinal_comparison(*vnexus, [](const  2
  QVariant_Observation& o1, const QVariant_Observation& o2)
{
 return Comparison<Ordinal>( o1.get_comparison("Tutoring/"
   "number_of_weeks", o2, "Test_Result/length_of_unit"));
}).gt_do([]() { qDebug() <<
  "The tutoring laster longer than the unit tested."; }
```

these individual types are only readily usable if each application can include the implementation files for each type during compilation. Applications can certainly be updated with new types as new data sources become available; but for the current discussion I will assume that manually integrating each new data source and recompiling applications is impractical for a hypothetical project. I will also set aside possibilities like inter-application messaging or loading dynamic or shared-object libraries at runtime. While these solutions may be acceptable for performing queries entirely within data sources exposed via dynamic libraries or external applications, it would be more difficult to use these techniques for hybrid queries, mixing types from distinct sources. Instead, I will assume that an application is designed to represent queryable data via general-purpose models such as **QVariant_Event_Nexus** discussed last section, and that this application can integrate other data sources so long as their data can similarly be represented via the canonical model.

So, for sake of discussion, assume that multiple applications use a Qt-driven EGO model and include the same implementation of a class like **QVariant_Event_Nexus**. Suppose one application is a preexisting "primary" application, and then a new project generates a new kind of data which can be productively added to the primary's data sources. Assume the new project can be extended by or embedded into a Qt application and can internally resolve EGO queries using techniques I have described here and also export data to a canonical Event Nexus class, say **QVariant_Event_Nexus**. That is, the "secondary" application uses a strongly-typed query model to test and evaluate its implementation, and — after its internal semantics are well-understood — its narrower Event Nexus types are given functions to export data to **QVariant_Event_Nexus**. This secondary application can then compile a data set to share with the primary application, in the form of many **QVariant_Event_Nexus** objects. At this point the primary application has no direct access to the secondary's internal data types, at least for Observation data; all data exposed between the two applications relies on the common **QVariant_Observation** class (I'll assume the two applications use the same Event and Ground classes).

There are several steps required for the primary application to incorporate the secondary as a data source. First, the secondary application needs to encode all its data (at least all data bundled into an Event Nexus) into a transportable binary or text stream. Via **QVariant_Event_Nexus**, this can be accomplished using Qt's serialization mechanism, particularly the **QDataStream** class. Most Qt types can be serialized automatically, so serialization for user-defined types tends to be fairly simple. For **QVariant_Observation**, the **QMap** and **QPair** templates will automatically produce the overall serialization once the user-defined types, like **Dimensions_Of_Comparison**, are serializable (see Figure 11). The primary application then needs to receive and decode encoded data sent by the secondary.[13]

Listing 11: Serializing an Event Nexus via QDataStream.

```
void QVariant_Event_Nexus::supply_data(QDataStream& qds) const
{
 qds<< *event_object_<< *ground_object_<< *observation_object;
}
QDataStream& operator<<(QDataStream& qds,
  const QVariant_Event_Nexus& qven)
{
 qven.supply_data(qds); return qds;
} ...
```

```
void QVariant_Observation::supply_data(QDataStream& qds) const
{
 //  this is a one-liner because QMap and QPair will serialize
 //  automatically, if Dimensions_Of_Comparison <<, >>
 //  overloads are known (since it is the only non-Qt type) ...
 qds << data_;
}
...
QDataStream& operator<<(QDataStream& qds,
  Dimensions_Of_Comparison dc)
{
 qds << (quint8) dc; return qds;
}
```

The primary application then deserializes the received data to obtain **QVariant_Event_Nexus** instances which, barring an error somewhere, are identical to the ones sent by the secondary application. These new objects are now part of the primary application's available data and can be queried alone, or — more significantly — in combination with other Event Nexus objects (as was shown with Figure 10).[14] When positive query results return matches incorporating values from this new data, this leads to a question of how the primary application should process and display matches involving the secondary data source. The primary application understands this data via a canonical format like **QVariant_Event_Nexus**; but this format is a reduced and transformed version of the original data, and it may not be suitable on its own for end-user visualization and manipulation of the data. The rest of this section will discuss these issues and concerns.

The demo code has two non-canonical Event Nexus types: one for **Test_Results** and one for **Tutoring**. Imagine a scenario where these classes come from two different applications — a primary one used by teachers and a secondary used by tutors. Integrating these applications allows compound queries, such as various comparisons of test results for those students who received tutoring against those who did not. To participate in these queries the tutoring and test data needs

[13] Assuming the applications use binary streams and Qt serialization, the delivered payload is just a byte array that can be sent as the body of an HTTP POST request, as a binary file, via pipes or streams, etc. Alternatively, the binary streams can be converted to Base32 or Base64 character strings and delivered as part of a text package.

[14] Bear in mind that the two applications need to use the same types, at least within serialized data. Care must therefore be exercised when storing user types as QVariants via **QDECLAREMETATYPE**, **qRegisterMetaType**, and **qRegisterMetaTypeStreamOperators**. In the demo, the **Tutoring_Topic** class needs special serialization; as discussed at the end of the laset section, for multiple data sources such qualitative (e.g., Taxonomic) data needs to use a common format shared between applications. This way the data types representing qualitative data for the purpose of **QVariant** and Qt serialization can be implemented in code shared by each application.

to be simplified into data aggregates with straightforward dimensions of comparison. This can potentially screen out a lot of information that users may want to access.[15] Most of this application-specific data is neither quantitative nor qualitative in a readily comparable manner, so it would not necessarily be included/exported within a queryable-data framework.

The data which *is* shared between applications is an Event Nexus with an EGO organization. This means that each object incorporated from a data source has a single and specific event which in principle is anchored to a precise point in time.[16] The "Observation" part of the EGO corresponds to data associated with each event *insofar as* this data can be marshaled into a queryable format. The same event — i.e., the same temporal anchor — can be joined to more complex and nuanced data as well; the Event object can then serve as a tie joining summarial Observation data with richer application-specific information.

In the current example, suppose the Tutoring Event Nexus is created (as a digital artifact) shortly after a tutoring unit (a set of sessions) has completed. The tutors' application can generate two different kinds of retrospective data: a simplified "Observation" object with basic info such as the topic covered and length of time, and apart from that a more detailed overview that can access other resources, like session reports and student work. If each multi-session "unit" of tutoring is assigned a distinct id code, then this id-code can be embedded with the Observation data as it is translated to **QVariant_Observation**. The primary application would not use the id-code directly, but when queries return a match that includes this **QVariant_Observation** the id-code can be used to follow up for more detailed information.

In the demo, a hypothetical student's math tutoring matches a query because it coincides with the student's math scores improving. Thus, that tutoring "unit" would be identified as part of match results for queries related to improved test scores following tutoring in the relevant subject. Such queries reduce the overall data space to smaller collections — e.g. the set of all tutoring units that correlate with improvements evident in a pair (or more) of tests, or perhaps one single test (one teacher's midterm exams, say). If the queries are sufficiently fine-grained, a user may want to explore the results one by one. Id-codes understood by "secondary applications" then become a key to work backward from the primary application where the query is evaluated, to the secondary application where the data involved in the positive match was generated. When the primary application evaluates a query that singles out a "unit" — a group of tutoring sessions — the primary may lack the libraries and capabilities to represent most details about these units to its users. Via the units' id-codes, however, it can potentially ensure that users access these details anyhow, by launching and sending id-codes to the tutors' applications.

Note that for this "secondary exploration" the merger of disparate data sources into a common representation (like **QVariant_Event_Nexus**) is no longer needed. The queries have already been executed; we can assume the query results are seeded with "id codes" pointing back to data structures from which simpler objects suitable for multi-source queries are derived. Having used the simpler objects for their target query-evaluation phase, there is no longer any technical rationale to avoid switching back to the original larger structures — except insofar as the ("primary") application which evaluates queries does not have the requisite components to process and render the larger structures. But this problem is redressed by storing id-codes designating the original data — and giving the primary application inter-application messaging capabilities, so it defers to secondaries in compensation for whatever capabilities it lacks to process externally sourced data structures directly.

For this kind of collaboration between the two applications to work, assume the primary application can send messages to the secondary. One kind of message requests any updates from the secondary's overall data. A second kind of request would be to show a package of information identified by a code or resource locator — e.g., a tutoring unit code. This setup would apply if the two applications reside on the same computer and the primary can launch and/or send data to a running instance of the secondaries.

A robust system should then provide checks and features to ensure that inter-application protocols along these lines work properly. For example, it is easy to imagine scenarios where a primary application acquires data *generated by* a secondary application without actually having that application locally installed. These concerns need to be addressed via what we can call a *Multi-Application Networking Environment*, responsible not only for guiding inter-application messaging but also for installing and configuring new applications to join the network. Application Identifiers — embedded in data generated by each application as it is shared with other applications — should point to resources useful for acquiring applications, such as locations for source-code repositories. In the context of published data sets, this source-code may be part of an overall data archive, and preparing applications for use in a Multi-Application Network becomes one step toward preparing a data set for publication.

Notice that a single primary application may work with multiple secondary applications; as such, any data generated by each secondary must be paired with provenance information and identifiers uniquely pointing to the originating application. These "application-identifiers" are then keys that map to versioning, executable file paths, port numbers, and other information which the primary may need to launch and com-

[15] Imagine the tutors' application shows more detailed reports about students' work: notes about sessions and assignments, session breakdown with lists of topics covered, digital records of students' writings and workbooks, and so forth.

[16] The demo assumes that this anchor "attaches" to the point in time when the tutoring (i.e., a set of sessions extending over a period of weeks) has ended, with the length of the tutoring being a Dimension of Comparison.

municate with secondaries. With enough interface consistency, new applications can join in the network dynamically, adding new data sources on the fly.

4 Conclusion

The most comprehensive frameworks need to both integrate data from multiple sources *and* identify which applications are optimized for which kinds of data. Some user goals call for querying data which is mixed from many sources; others need applications implemented for narrowly delineated kinds of data, with special visualization and interaction protocols. Supporting both kinds of user goals requires query and type mechanisms that extend outside the boundary of a single application; perhaps as libraries or protocols that multiple applications can use.

Joining such networks requires more commitment from an application-development point of view than just exporting data in a standardized format, or even using data types designed under the influence of Ontologies or other controlled conceptualizations. Designing applications in the context of interoperating multi-application networks requires a deeper, more type-theoretic model of application-integration than that implicit in Ontologies or APIs. This is particularly apropos for published data sets, where type theory can add a level of semantic and meta-scientific rigor, not just practical coding, to scientific projects.

The techniques I have explored here lie at the intersection of three broader subjects — data set publication, implementing "queryable" data types, and Multi-Application Networking — each of which deserve extensive research in their own right. It's reasonable to claim that although multi-paradigm languages provide a rich set of tools for implementing advanced frameworks fulfilling each of these areas of need, the language- and/or tool- level support for such projects remains primitive. I have tried, for instance, to demonstrate by example that Qt has powerful features especially relevant for this paper's topics — we should treat Qt as a dialect of C++, not just a library — but Qt remains a specialized community within the C++ ecosystem. Whether the push for broader adoption of comparable techniques should come from frameworks like Qt, or from language standards and implementation directly — from tools, or languages (compilers and runtimes), or both — is a question Software Language Engineering may be best positioned to answer.

References

1 Eran Bellin, "Riddles in Accountable Healthcare: A Primer to develop analytic intuition for medical homes and population health". CreateSpace Independent Publishing Platform, North Charleston, SC, USA, 2015

2 Eran Bellin, *et. al.*, "Democratizing Information Creation From Health Care Data for Quality Improvement, Research, and Education -- The Montefiore Medical Center Experience" https://pdfs.semanticscholar.org/ad02/adebdfe8d51c6defb120aac9f6f102e16596.pdf

3 Robert L. Bocchino Jr., *et al*, "A Type and Effect System for Deterministic Parallelism in Object-Oriented Languages", 2009. http://jeff.over.bz/papers/2009/UIUCDCS-R-2009-3032.pdf

4 Joana Campos and Vasco T. Vasconcelos, "Channels as Objects in Concurrent Object-Oriented Programming" https://arxiv.org/pdf/1110.4157.pdf

5 Hong Chen, "Towards Design Patterns for Dynamic Analytical Data Visualization" https://pdfs.semanticscholar.org/ba7d/dcbf608e6da8a49239593c25207e136617b1.pdf

6 Tharaka Devadithya, *et. al.*, "C++ Reflection for High Performance Problem Solving Environments" http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.510.2930&rep=rep1&type=pdf

7 Ian Dees, "Misusing the Type System for Fun and Profit"Excerpt from PNSQC 2015 Proceedings. http://uploads.pnsqc.org/2015/papers/t-131_Dees_paper.pdf

8 François-Nicola Demers and Jacques Malenfant, "Reflection in logic, functional and object-oriented programming: a Short Comparative Study". *Proceedings of the IJCAI'95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, pp. 29-38, August 1995 http://www-master.ufr-info-p6.jussieu.fr/2006/Ajouts/Master_esj_2006_2007/IMG/pdf/malenfant-ijcai95.pdf

9 Erik Ernst, "Family Polymorphism", J. Lindskov Knudsen (Ed.): *ECOOP 2001, LNCS 2072*, pp. 303-326, 2001. http://www.cs.au.dk/~eernst/tool11/papers/ecoop01-ernst.pdf

10 Sara Irina Fabrikant, "Visualizing Region and Scale in Information Spaces". Proceedings, The 20th International Cartographic Conference, ICC 2001, Beijing, China, Aug. 6-10, 2001, pp. 2522-2529. https://pdfs.semanticscholar.org/526a/09e4767ff634c4cfbc51e6f7f4ebb700096a.pdf

11 Timothy Lindsay John Ferris, "Foundation for Medical Diagnosis and Measurement". Dissertation, University of South Australia, 1997. http://search.ror.unisa.edu.au/media/researcharchive/open/9915959949401831/53112353030001831

12 Andrew U. Frank, "Ontology for Spatio-temporal Databases". In T. Sellis et al. (Eds.): Spatio-temporal Databases, LNCS 2520, pp. 9-77, 2003 http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.113.9804&rep=rep1&type=pdf

13 Michael Fuchs, *et. al.*, "Semantics of Query Construction User Interfaces of Web-Based Search Engines for Digital Libraries". https://pdfs.semanticscholar.org/07ba/b109183c9dfe379f4eae07bae1e2e6d02277.pdf

14 Mary Geetha and Sriman Narayana Iyengar, "A Frame Work for Ontological Privacy Preserved Mining". International Journal of Network Security & Its Application (IJNSA), Vol.2, No.1, January 2010. http://airccse.org/journal/nsa/1010s2.pdf

15 Ghislain Hachey and Dragan Gašević, "Semantic Web User Interfaces: A Systematic Mapping Study and Review", 2012. http://www.semantic-web-journal.net/system/files/swj316_0.pdf

16 Jeremy Gibbons, "Datatype-Generic Programming". https://www.cs.ox.ac.uk/jeremy.gibbons/publications/dgp.pdf

17 Yolanda Gil, *et. al.*, "Privacy Enforcement in Data Analysis Workflows". https://www.isi.edu/~gil/papers/gil-etal-peas07.pdf

18 Sergey Goncharov, *et al.* "Kleene Monads: Handling Iteration in a Framework of Generic Effects" https://www8.cs.fau.de/staff/schroeder/papers/kleene.pdf

19 Azadeh Izadi, *et al.* "Multidimensional Region Connection Calculus", 2017 http://qrg.northwestern.edu/qr2017/papers/QR2017_paper_8.pdf

20 Robert Jeansoulin, and Christophe Mathieu. "Revisable Spatial Knowledge by means of a Spatial Modal Logic", 1995 https://www.researchgate.net/publication/228713541_Revisable_Spatial_Knowledge_by_means_of_a_Spatial_Modal_Logic

21 Morten Krogh-Jespersen, *et. al.*, "A Relational Model of Types-and-Effects in Higher-Order Concurrent Separation Logic" https://cs.au.dk/~birke/papers/iris-effects-conf.pdf

22 C. Maria Keet, *et. al.*, "Representing mereotopological relations in OWL ontologies with OntoPartS" https://pdfs.semanticscholar.org/eb97/dd87ef05b376ad1dc6f2214d19d964ba7259.pdf

23 Andrew Kennedy and Claudio V. Russo, "Generalized Algebraic Data Types and Object-Oriented Programming"OOPSLA'05, October 16-20, 2005, San Diego, California, USA. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/gadtoop.pdf

24 Oleg Kiselyov and Chung-chieh Shan, "A Substructural Type System for Delimited Continuations"

http://homes.sice.indiana.edu/ccshan/binding/context.pdf
25 Igor Kriz and Aleš Pultr, "Categorical Geometry and Integration without Points", 2012. https://arxiv.org/pdf/1101.3762.pdf
26 Sunil Kothari and Martin Sulzmann, "C++ templates/traits versus Haskell type classes", 2004. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.78.2151&rep=rep1&type=pdf
27 Martin Leinberger, *et. al.*, "The essence of functional programming on semantic data" https://eprints.soton.ac.uk/404009/1/ESOP_2017_paper_39.pdf
28 Daniel Lincke, "A Transformational Approach to Generic Software Development Based on Higher-Order, Typed Functional Signatures". Dissertation, Technischen Universität Hamburg-Harburg, 2012 https://pdfs.semanticscholar.org/5fa2/a5f97b148e1fc0ba6d2cbccee235f025b891.pdf
29 Fengyun Liu, "A Study of Capability-Based Effect Systems". Thesis, École Polytechnique Fédérale de Lausanne, 2016 https://infoscience.epfl.ch/record/219173/files/thesis-jan-15.pdf
30 Zhaohui Luo, "Coercions in a Polymorphic Type System", 2007 https://www.cs.rhul.ac.uk/home/zhaohui/WIT07.pdf
31 Toby Murray, "Analysing Object-Capability Security". https://www.cs.ox.ac.uk/files/2690/AOCS.pdf
32 Philippe Narbel, "Functional Programming at Work in Object-Oriented Programming" in Journal of Object Technology, vol. 8, no. 6, September-October 2009, pp. 181-209. http://www.jot.fm/issues/issue_2009_09/article5.pdf
33 Özgür L Özçep and Ralf Möller, "Spatial Semantics for Concepts". http://ceur-ws.org/Vol-1014/paper_38.pdf
34 Heiko Paulheim and Florian Probst, "Ontology-Enhanced User Interfaces: A Survey" International Journal on Semantic Web and Information Systems (IJSWIS), Vol. 6, No. 2, 2010. https://pdfs.semanticscholar.org/47d4/76f1203ef597683b9ed4ca3324f639bd2bcb.pdf
35 Ilyas Potamitis, *et. al.*, "Automatic Acoustic Identification of Insects Inspired by the Speaker Recognition Paradigm". Proc. of the InterSpeech-2006 -- ICSLP, Pittsburgh, PA, USA, Sept 17-21, 2006, pp. 2126-2129 https://pdfs.semanticscholar.org/bddc/666635973e12fa0a58f711530ea8db7f2dd4.pdf
36 Rose Hafsah Binti Ab. Rauf, "Integrating Functional Programming Into C++" Dissertation, University of Wales, 2007. http://www.cs.swan.ac.uk/~csetzer/articlesFromOthers/roseHafsahAbdulRauf/roseHafsahAbdulRaufPhDThesis.pdf
37 Ioannis Votsis and Gerhard Schurz, "A frame-theoretic analysis of two rival conceptions of heat" *Studies in History and Philosophy of Science*, vol. 43 (2012), pp. 105-114. http://www.votsis.org/PDF/Votsis_Schurz_A_Frame-Theoretic_Analysis_of_Two_Rival_Conceptions_of_Heat.pdf
38 Hedda R. Schmidtke and Michael Beigl, "Positions, Regions, and Clusters: Strata of Granularity in Location Modelling" http://www.teco.edu/~michael/publication/ki2010.pdf
39 John G. Stell, "Granulation for Graphs" https://pdfs.semanticscholar.org/9e0f/a93a899e36dc3df62feabc004a0ecef4365d.pdf
40 John G. Stell and Matthew West, "A Four-Dimensionalist Mereotopology" https://www.researchgate.net/publication/250891385_A_Four-Dimensionalist_Mereotopology
41 Jesse A. Tov and Riccardo Pucella, "A Theory of Substructural Types and Control". *Studies in History and Philosophy of Science*, OOPSLA'11, October 22-27, 2011, Portland, Oregon, USA. http://users.eecs.northwestern.edu/~jesse/pubs/substructural-control/CtlURAL.pdf
42 Wilkinson, M. D. *et al.* "The FAIR Guiding Principles for scientific data management and stewardship". Sci. Data 3:160018 doi: 10.1038/sdata.2016.18 (2016). https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4792175/
43 Conglun Yao, "Strongly Typed, Compile-Time Safe and Loosely Coupled Data Persistance". Dissertation, University of Birmingham, 2010 http://etheses.bham.ac.uk/1186/1/Yao10PhD.pdf
44 Martin Zalewski and Sibylle Schupp, "C++ concepts as institutions: a specification view on concepts"LCSD '07 Proceedings of the 2007 Symposium on Library-Centric Software Design, pp. 76-87. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.214.7815&rep=rep1&type=pdf
45 Frank Zenker, "From Features via Frames to Spaces: Modeling Scientific Conceptual Change Without Incommensurability or Aprioricity". In T. Gamerschlag, D. Gerland, R. Osswald & W. Petersen (eds.), *Frames and Concept Types: Applications in Language and Philosophy*, Dordrecht: Springer, 2014, pp. 69-89. https://www.researchgate.net/publication/228966901_From_Features_via_Frames_to_Spaces_Modeling_Scientific_Conceptual_Change_Without_Incommensurability_or_Aprioricity

Title
Object-Functional Typing for Queryable Data Sets

Author
Nathaniel Christen

Date
July 12, 2018

For More Information
Please contact Linguistic Technology Systems, Inc.