GigaScience, 2022, **11**, 1–12
DOI: 10.1093/gigascience/giac089
TECH NOTE

CORAL: A framework for rigorous self-validated data modeling and integrative, reproducible data analysis

Pavel S. Novichkov [1,†], John-Marc Chandonia [1,*,†] and Adam P. Arkin [1,2,*]

[1]Environmental Genomics and Systems Biology Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA
[2]Department of Bioengineering, University of California, Berkeley, CA 94720, USA
*Correspondence address. John-Marc Chandonia, Environmental Genomics and Systems Biology Division, Lawrence Berkeley National Lab, Mailstop Donner, Berkeley, CA 94720-3102, USA. E-mail: JMChandonia@lbl.gov; Adam P. Arkin, Environmental Genomics and Systems Biology Division, Lawrence Berkeley National Laboratory, 1 Cyclotron Road, Mailstop 955-512L, Berkeley, CA 94720, USA. E-mail: APArkin@lbl.gov
†Joint first authors.

Abstract

Background: Many organizations face challenges in managing and analyzing data, especially when relevant datasets arise from multiple sources and methods. Analyzing heterogeneous datasets and additional derived data requires rigorous tracking of their interrelationships and provenance. This task has long been a Grand Challenge of data science and has more recently been formalized in the FAIR principles: that all data objects be Findable, Accessible, Interoperable, and Reusable, both for machines and for people. Adherence to these principles is necessary for proper stewardship of information, for testing regulatory compliance, for measuring the efficiency of processes, and for facilitating reuse of data-analytical frameworks.

Findings: We present the Contextual Ontology-based Repository Analysis Library (CORAL), a platform that greatly facilitates adherence to all 4 of the FAIR principles, including the especially difficult challenge of making heterogeneous datasets Interoperable and Reusable across all parts of a large, long-lasting organization. To achieve this, CORAL's data model requires that data generators extensively document the context for all data, and our tools maintain that context throughout the entire analysis pipeline. CORAL also features a web interface for data generators to upload and explore data, as well as a Jupyter notebook interface for data analysts, both backed by a common API.

Conclusions: CORAL enables organizations to build FAIR data types on the fly as they are needed, avoiding the expense of bespoke data modeling. CORAL provides a uniquely powerful platform to enable integrative cross-dataset analyses, generating deeper insights than are possible using traditional analysis tools.

Keywords: FAIR data, contexton, microtype, data management, provenance, data analysis, Jupyter

One of the Grand Challenges of data science is to facilitate knowledge discovery by enabling datasets to be readily analyzable both by humans and by machine learning algorithms. In 2016, a diverse group of stakeholders formalized a concise and measurable set of principles, called FAIR, to increase the utility of datasets for the purpose of knowledge discovery [1]. The 4 principles of FAIR are Findability, Accessibility, Interoperability, and Reusability. *Findability* means that data are assigned stable identifiers and properly indexed. *Accessibility* means the data are easily retrievable by people authorized to have access. *Interoperability* means the data are clearly documented using a formal language, in order to facilitate integrated analyses that span multiple datasets. *Reusability* means the data are documented sufficiently well that datasets may be used by people other than the original data generators and that the provenance of all data is clear.

The problems of making an organization's data *Findable* and *Accessible* to its members are largely solved by modern databases, including relational databases such as SQL [2] and nonrelational "NoSQL" databases such as document stores [3]. The challenges of assigning a unique, permanent ID to each dataset generated within an organization, and then ensuring that the dataset is deposited into a database where it may be retrieved by appropriate people, are largely managerial problems rather than technical ones. On the other hand, making data both *Interoperable* (enabling powerful integrated analyses that span many datasets generated by different teams within an organization) and *Reusable* (enabling later use of datasets by somebody other than the person or team that originally generated the data) is extremely difficult, as we detail below.

The FAIR stakeholders recognized that a number of public databases for specialized data types have formatting standards that mandate rigorous formal documentation of each deposited dataset and its provenance and that this requirement is sufficient to ensure both *Interoperability* and *Reusability* [1, 4]. However, this is not the case for general-purpose datasets (i.e., data types for which rigorously defined formatting and formal documentation standards have not been developed). *Reusability* is challenging for general-purpose data because nonspecialized data storage formats often do not allow or require specification of key details, even basic ones such as units of measurement. As a result, undocumented assumptions and conventions can make it very difficult to reproduce or reuse data. This has produced a reproducibility crisis within the scientific research community [5, 6]. Nonreusable data are also very costly: life sciences researchers in the United States alone are estimated to spend over $28 billion each year on irreproducible preclinical work [7]. Ensuring *Interoperability*

Received: October 25, 2021. **Revised:** April 12, 2022. **Accepted:** August 31, 2022

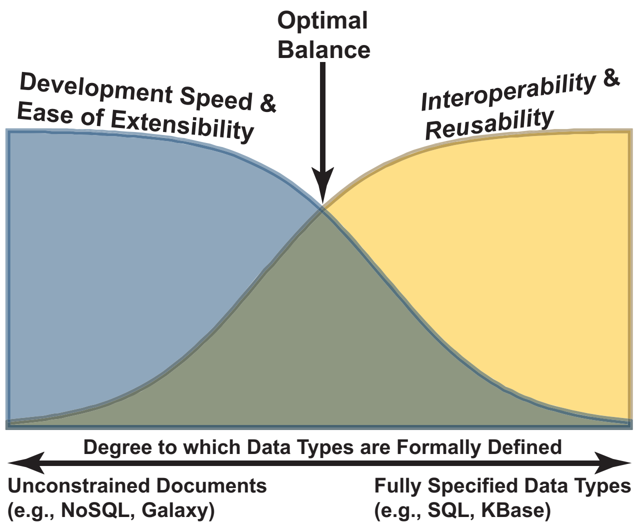
Optimal Balance
Development Speed & Ease of Extensibility
Interoperability & Reusability
Degree to which Data Types are Formally Defined
Unconstrained Documents (e.g., NoSQL, Galaxy)
Fully Specified Data Types (e.g., SQL, KBase)

Figure 1: Trade-offs between adherence to FAIR principles and the ease with which new data types can be added to a platform. More rigorous formal definitions of data types facilitate higher degrees of *Interoperability* and *Reusability*, but at the cost of a higher degree of difficulty in adding new data types to the system.

between datasets is challenging for many of the same reasons: when different groups within an organization produce data, impedance matching must be done in order to perform an integrative analysis. Some sources of impedance are differing units, incompatible scaling or normalization of different datasets, and different identifiers used by different groups to refer to the same objects.

The DOE Systems Biology Knowledgebase (KBase) is an open-source software and data platform that facilitates data sharing and integrated analyses of microbial and plant data [8]. KBase defines a large number of data types relevant to such datasets, and each of these data types is formally defined in sufficient detail to ensure compliance with FAIR principles. KBase also provides a Software Development Kit (SDK) that allows users to rapidly import new tools and algorithms into the system. However, the SDK does not currently allow custom data types, and development of new data types is a relatively labor-intensive process that must be done by KBase developers rather than end users [8]. This process of defining new data types is analogous to developing or extending SQL schema: each type must be carefully designed to handle multiple use cases and must integrate with other data already in the system.

In contrast to KBase, the Galaxy platform [9], also popular among microbial and plant researchers, uses a data model in which files uploaded by users are not validated or parsed in any way. Thus, both Galaxy developers and users can easily introduce new data types into the system to support new tools and algorithms. However, because files in Galaxy are stored without formal type validation—analogous to documents in a NoSQL database—they are not required to comply with any standards for documentation beyond what is required to successfully run a single tool or algorithm. Thus, the benefit of rapid development in Galaxy comes at the cost of *Interoperability* and *Reusability*. These trade-offs are illustrated in Fig. 1.

In practice, most data types can be represented using a small number of data models, such as arrays, graphs, trees, and hash tables. We surveyed hundreds of data types used by our colleagues in the ENIGMA project, a large consortium of researchers who study how communities of microbes interact with their environment (https://enigma.lbl.gov/). We discovered that the vast majority of data, from raw assays to processed results, can be represented as multidimensional arrays of scalars. We believe that this result is generalizable across many fields of research and business and not just true for ENIGMA. For example, climate modelers widely use the xarray library for storing data in multidimensional arrays, in which key-value pairs are used to label each dimension [10]. Similar libraries for handling labeled multidimensional arrays exist in most computer languages, and file formats such as HDF5 [11] and NetCDF-4 [12] are well-supported, mature technologies. However, a common file format alone is not sufficient to ensure adherence to the FAIR principles of *Interoperability* and *Reusability*: in addition to a standard file format, all data, dimensions, and units in these multidimensional arrays must also be formally and rigorously documented. Current state-of-the-art tools such as the CEDAR Workbench [13] greatly facilitate formal documentation of metadata describing entire datasets but do not readily allow documentation of components such as dimensions and variables. Likewise, Common Data Element (CDE) standards as specified in ISO 111179 [14] have been in use at the National Institutes of Health for over a decade [15] but are not applicable to internal components of data structures.

In this article, we describe the Contextual Ontology-based Repository Analysis Library (CORAL), a novel framework for data modeling and analysis, which aims to achieve an optimal balance between the ease of adding new data types and adherence to FAIR principles.

Findings

An overview of the CORAL data model is shown in Fig. 2, and further details are provided in subsequent sections. Our approach to modeling data enables new complex data types to be defined on the fly by users, thus avoiding high maintenance costs, but it also ensures that such data types are documented in the formal and rigorous manner that is necessary for *Interoperability* and *Reusability* of all data.

Microtypes and self-validated contextons

Formal documentation of data means that all data types are described using a limited vocabulary, in which all terms are clearly defined [1]. To ensure that users document all data in a formal way, CORAL data types must be built using a predefined set of building blocks, which we call *microtypes*.

Microtypes are atomic data types representing a simple concept relevant to a domain of interest (e.g., a gene name, an experimental parameter such as carbon source, or measurements such as optical density or pH). In contrast to complex data types, each microtype should correspond to a conceptual element that can be represented in a single scalar variable. Microtype definitions comprise a unique name, a formal definition of the data type, custom validators, allowed scalar types, and (if applicable) links to other data types. A well-chosen set of microtypes gives users the freedom to model all data types relevant to a domain of interest and limits ambiguity by restricting the ways in which context can be described to this defined set.

While each microtype defines a concept, additional information is needed every time the microtype is instantiated. In addition to the value of the scalar type, the *units* in which the value is defined provide another piece of critical context. We call this trio (the microtype, its value, and the unit of measurement) a

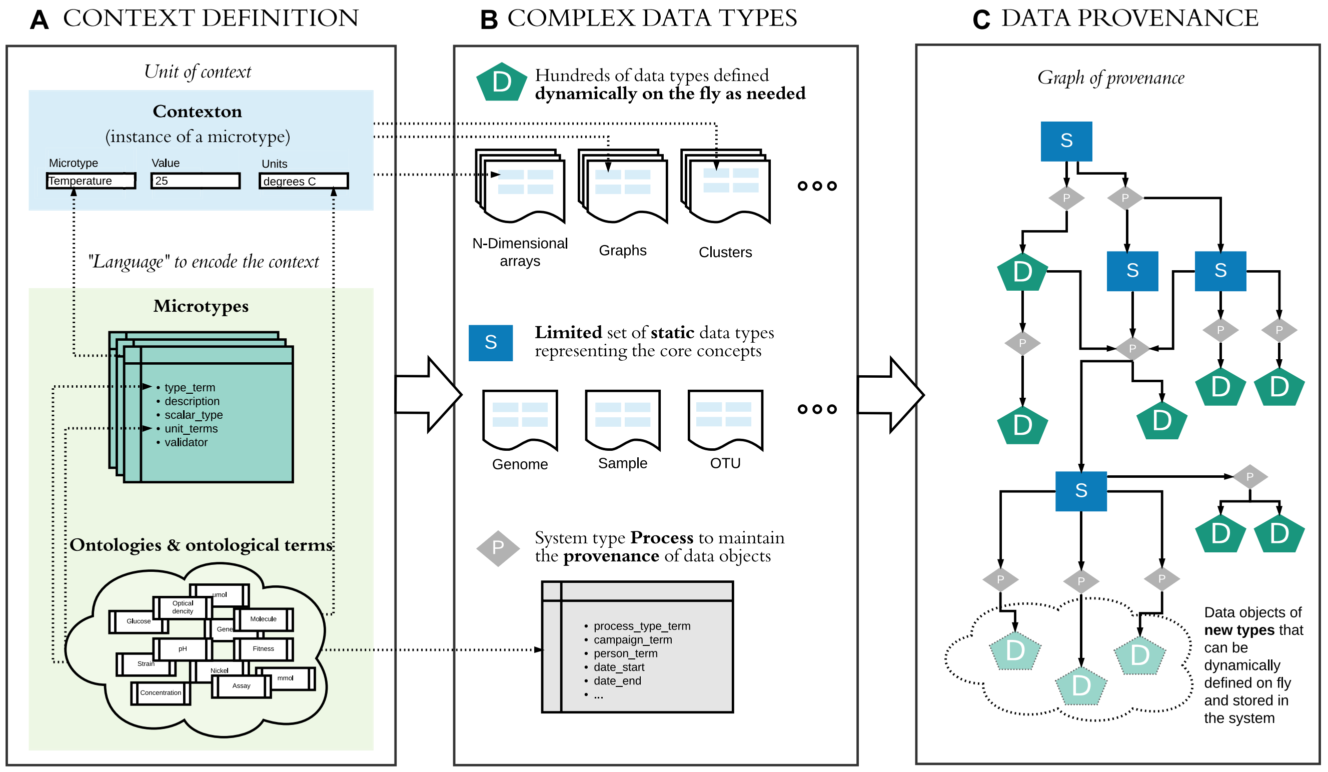

Figure 2: (A) To rigorously document context for all data, we introduce the concept of a "contexton," or unit of context. Contextons are built using "microtypes," which we define as atomic data types representing a simple concept relevant to a domain of interest. Both rely on ontologies, which define a controlled vocabulary for describing this domain. Together, the microtypes and ontologies defined for a particular instance of CORAL represent a language that allows users to formally describe all data in that instance in a way that is both *Interoperable* and *Reusable*. (B) Dynamic data types, which make up the vast majority of data in a CORAL instance, are defined by the users of the system as needed. These types are built by combining commonly used mathematical data structures with contextons. A limited number of static core types, which are fully specified traditional data structures, are also built using contextons in order to ensure *Interoperability* with the dynamic data. These static core types include the system type *Process*, which is a special core type needed to document the provenance of each data object in a CORAL instance. (C) All static and dynamic data in a CORAL instance are referenced in an object graph, where nodes are static or dynamic datasets, and edges are processes. This graph formally annotates the provenance of all data.

contexton, or an atomic unit of context. One or more contextons are required to provide formal context to all data in CORAL. Because the microtype on which a contexton is based includes a validator and a limited set of possible units of measurement, contextons are *self-validating* (i.e., the validity of all data in contextons may be confirmed independently of the contexton's role within larger data structures).

To provide a formal definition of all parts of a contexton, we make extensive use of controlled vocabularies of *ontological terms* [16, 17]. Requiring a controlled vocabulary has several advantages: synonymous concepts can be collapsed into a single ontological term, and errors due to misspellings are eliminated. Both the microtype names and the units of measurement are required to be ontological terms, rather than free text. We recommend the use of ontological terms rather than text strings as contexton values, where possible. Designers of microtypes should endeavor to use community standard ontologies wherever possible.

The relationship between contextons, microtypes, and ontologies is shown in Fig. 3.

Static and dynamic data types

The most FAIR data models require data scientists to completely specify detailed schema for all types of data within an organization, and the links between data types, before new data types can be deployed. This process is often slow and expensive to implement, especially as data types are added or modified over time, and is therefore impractical for dynamically evolving systems. In CORAL, one key design principle was to provide similar *Interoperability* and *Reusability* as the fully specified schema approach, but without these barriers to adding new data types.

We achieve this goal by dividing data types into 2 classes: *static core* types that are fundamental to an organization's business or science domain, and thus will remain largely unchanged over time, and *dynamic* types that may be rapidly defined as needed. In practice, only a small fraction (~10%) of data types need to be statically defined. These static core types must be defined in advance, with the associated development costs, but as fundamental types they should not need to be redefined significantly as an organization evolves. Dynamic types, which make up the vast majority of data types, are defined by the users of the system on the fly, with much lower development costs.

Static core types

Static core types are complex objects that are fundamental to the domain of an organization. Each static type is defined as a set of contextons, as shown in Fig. 4.

In this example, we define a static core type, *Well* (i.e., a groundwater well that is a location in which ENIGMA collects field samples) using a number of microtypes. Although a single Well may be represented by a set of scalar contextons, the set of all Wells

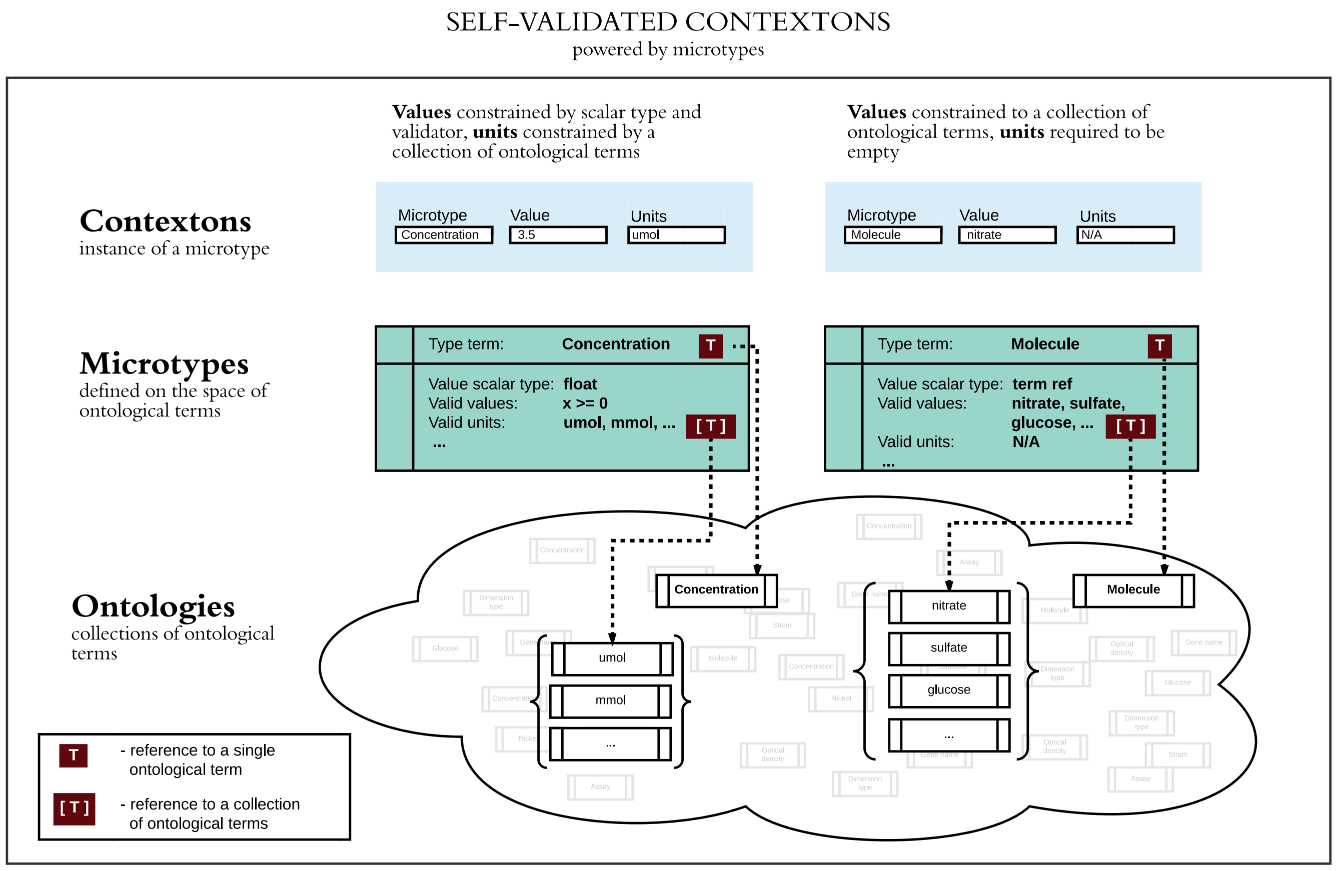
SELF-VALIDATED CONTEXTONS
powered by microtypes
Values constrained by scalar type and validator, units constrained by a collection of ontological terms
Values constrained to a collection of ontological terms, units required to be empty
Contextons
instance of a microtype
Microtype Concentration
Value 3.5
Units umol
Microtype Molecule
Value nitrate
Units N/A
Microtypes
defined on the space of ontological terms
Type term: Concentration T
Value scalar type: float
Valid values: x >= 0
Valid units: umol, mmol, ... [T]
...
Type term: Molecule T
Value scalar type: term ref
Valid values: nitrate, sulfate, glucose, ... [T]
Valid units: N/A
...
Ontologies
collections of ontological terms
Concentration
Molecule
umol
mmol
...
nitrate
sulfate
glucose
...
T - reference to a single ontological term
[T] - reference to a collection of ontological terms

Figure 3: A microtype is an atomic variable representing a single concept relevant to the field of interest, typically one that can be described using a single scalar value and (where applicable) a unit of measurement. Both the microtype names and units of measurement are required to be ontological terms. As part of the microtype definition, values must be constrained to be a simple scalar type, an ontological term, or a reference to a data object. A contexton is an instantiated microtype, in which the value and units are specified.

in a CORAL instance is represented by a set of 1-dimensional contextons. This is somewhat analogous to a single table in SQL, except that in CORAL, the use of contextons allows all data to be self-validated and relevant units to be specified.

Dynamic types

Dynamic data types are most often used for data related to the core types, such as measurements taken on core objects. Dynamic types are defined by combining a limited number of simple, mathematical data structures (e.g., matrices, trees, graphs) with contextons that provide critical context to the data, in order to allow users to build a large number of rich, complex data types on the fly. Contextons are used to provide formal context to all parts of the mathematical data structure: each axis, the values, and the object as a whole, resulting in a new dynamic data type. Individual instances of dynamic types (i.e., individual datasets) are called *Data Bricks*. An example is shown in Fig. 5.

As discussed above, we found that the mathematical data structure most relevant to ENIGMA data is the multidimensional array. Our first implementation of dynamic types is therefore based on mathematical N-dimensional arrays (NDArrays), but the same approach can be applied to support other mathematical types such as trees or graph structures.

Our NDArray data type is designed to model a homogeneous N-dimensional matrix of data, meaning that all the data in the matrix have the same scalar type and units of measurement. Contextons are required to provide context in 3 places:

1. Overall context for the data matrix. The mandatory microtypes *Data Category* and *Brick Name* are used to specify the overall type of data (chosen from an ontology of data categories defined by the instance administrators), as well as a unique name for each dataset. Additionally, each data brick is assigned a permanent *Brick ID* when it is stored in the system. Additional contextons that provide context for the data object as a whole (e.g., the instrument used or the experiment date) are optional. Note that contextons that provide context for the entire matrix are similar to CDEs; therefore, microtypes that are compatible with relevant community standards should be included in the set of microtypes defined by the administrators of a CORAL deployment.
2. Context for each dimension in the matrix. Each dimension has a mandatory microtype, *Dimension Type*, to specify what varies along that dimension. Like *Data Category*, dimension types must be chosen from an ontology of allowed types defined by the instance administrators. Each dimension must contain 1 or more contextons, each of which describes a single variable that varies along the corresponding dimension. These contextons usually correspond to the independent variables in an experiment. Each contexton contains a 1-dimensional array of values, with 1 value for every point along the axis of the dimension. All values within a contexton must have the same units; the contexton also includes a single ontological term documenting the units of measurement.

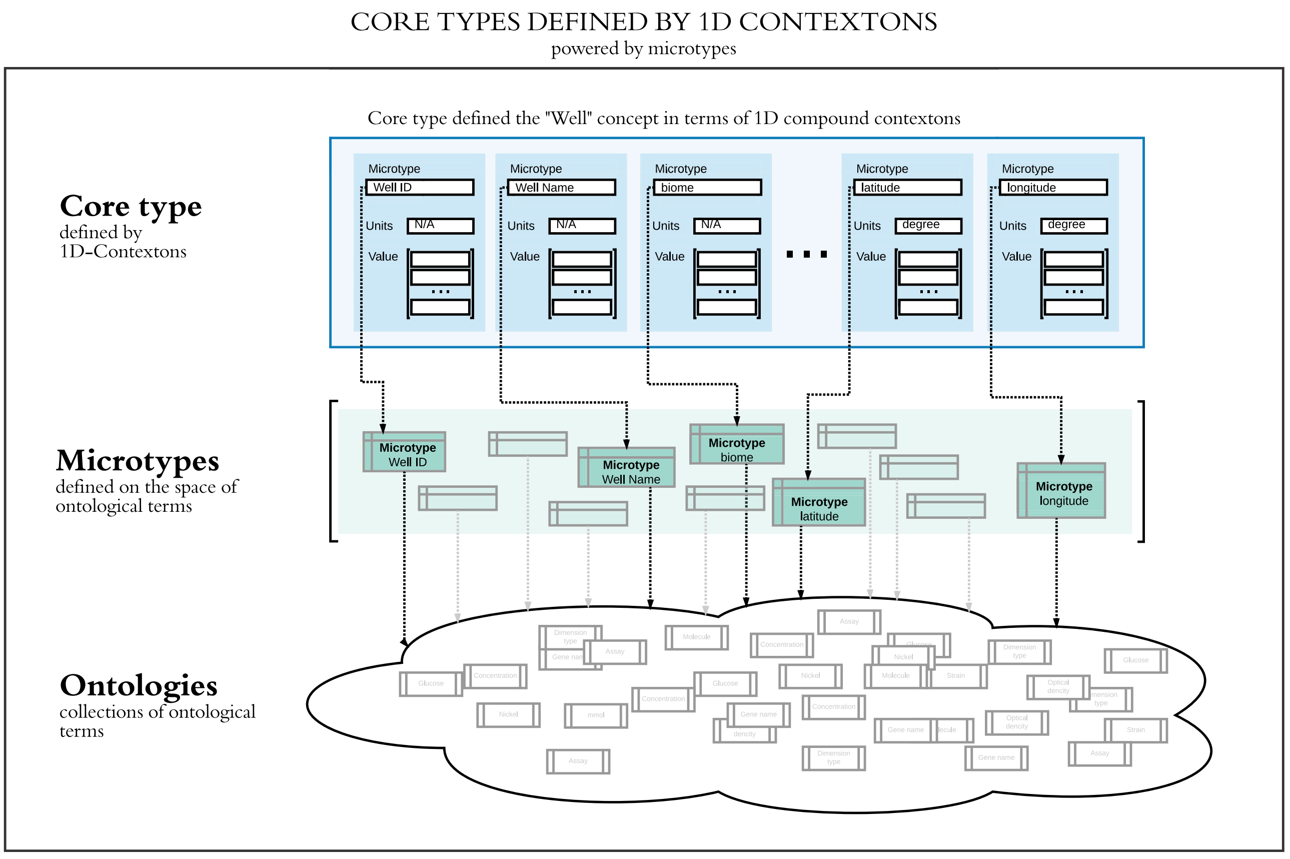

Figure 4: Definition of the static core type "Well"—a location in which environmental samples are taken by ENIGMA field researchers. A number of attributes, such as the name, biome, latitude, and longitude, are relevant to describing each well. Well names are free text, but the biome must be chosen from among a collection of biome terms described in the ENVO ontology. Microtypes are used to ensure that each well's latitude and longitude are specified in degrees.

3. Context for the data values. The values themselves are stored as N-dimensional contextons, which specify the microtype (what is being measured) and the units of measurement. To accommodate heterogeneous data measured under the same conditions, our data model allows multiple sets of homogeneous data values within the same NDArray. Although all such datasets must have the same dimensions, each can include data with a different microtype and units of measurement. These contextons usually correspond to the dependent variables in an experiment.

Each dimension in the NDArray must be labeled by at least 1 contexton. The microtype for this contexton may be the same as the overall *Dimension Type*. To provide a rigorous formal definition of the data object, the contextons that describe dimensions should be restricted by instance administrators to an informative subset of microtypes. Microtypes that link to core types are included in this set. Other microtypes, such as "comment" (a free text string), are not allowed as dimension contextons in the absence of other informative microtypes, because they do not formally document the data.

It is up to the data generator how best to model their data using this system: for example, a 2-dimensional NDArray with 2 contextons describing each dimension could also be represented as a 4-dimensional NDArray, with 1 contexton along each dimension. Often, this decision reflects experimental design: if factorial design is used, it usually makes sense for each independent variable to be a single dimension; on the other hand, if a sparse subset of conditions is chosen in the experiment, it may be more convenient to combine several independent variables into 1 dimension. Null values (representing sparse datasets or missing data) are allowed.

Data provenance

Dynamic type definition is a powerful tool that introduces freedom and flexibility to the system. However, at the same time, it can result in isolated "data islands" where data are not sufficiently well documented to determine their provenance (i.e, the inputs and methodology used to produce data). A second key design principle of CORAL is all data must be linked in a network that formally annotates the provenance of these data. This is important both for *Findability*, as datasets that are unrelated to other data in the system may be difficult to find, and also because storing the complete provenance of each object is a key aspect of *Reusability*.

We achieve this goal by requiring that all data objects in CORAL must be linked to core types, either through direct references to core objects or to another dynamic object that ultimately links to a core object. Links between static and dynamic data types are created via the reference mechanism implemented in microtypes, since the same microtypes are used to build context in both static and dynamic data. This concept is similar to foreign keys in SQL. All static and dynamic types are linked in a directed graph, in which the edges document the processes by which one data type is derived from another. Thus, the core types form a "backbone" to which all data must be anchored, and the provenance of every object in the system is clear.

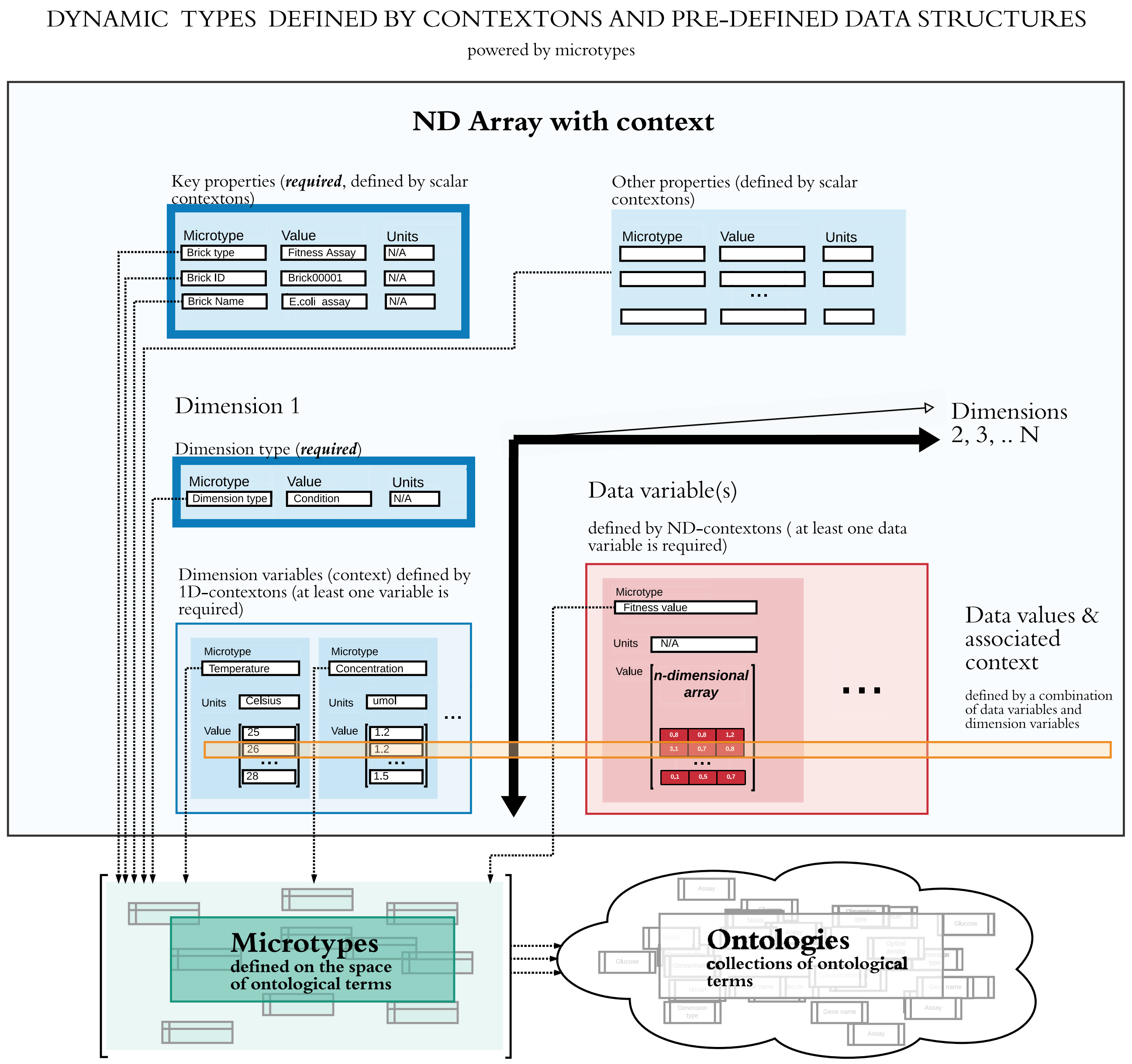
DYNAMIC TYPES DEFINED BY CONTEXTONS AND PRE-DEFINED DATA STRUCTURES
powered by microtypes
ND Array with context
Key properties (required, defined by scalar contextons)
Microtype
Value
Units
Brick type
Fitness Assay
N/A
Brick ID
Brick00001
N/A
Brick Name
E.coli assay
N/A
Other properties (defined by scalar contextons)
Dimension 1
Dimension type (required)
Dimension type
Condition
Dimensions 2, 3, .. N
Data variable(s)
defined by ND-contextons (at least one data variable is required)
Fitness value
n-dimensional array
Data values & associated context
defined by a combination of data variables and dimension variables
Dimension variables (context) defined by 1D-contextons (at least one variable is required)
Temperature
Celsius
25
26
28
Concentration
umol
1.2
1.2
1.5
Microtypes
defined on the space of ontological terms
Ontologies
collections of ontological terms

Figure 5: Definition of a dynamic data type to describe the results of a fitness assay. The first dimension specifies the conditions under which researchers measured fitness values of different genes. The second dimension might specify genes, while a third dimension might specify replicates. Because particular fitness datasets may have any of these dimensions or more, they are more readily modeled as dynamic datasets rather than as static core objects.

In conclusion, the CORAL data model ensures that data meet all 4 of the FAIR principles. Data are *Findable* by searching for ontological labels as well as links to other data and *Accessible* because all versions of all data, microtypes, and ontologies are given permanent IDs and stored in an underlying document store. Our focus on baking *Interoperability* and *Reusability* into our data model greatly facilitates adherence to these principles as well, without the design overhead of prior general-purpose data modeling approaches.

Functionality

In addition to storing data, CORAL's design enables rich functionality to make the system useful for data analysis, visualization, and managerial oversight. Major features are described in the Methods section. They include a user-friendly upload wizard that guides data providers through the process of formally documenting and uploading datasets, without needing to master technical details such as ontological terms; a powerful search engine that can find datasets according to their provenance as well as searching the contents of the dataset; a plotting wizard that allows nontechnical users to visualize their datasets using plot types that are relevant to the dimensionality of the dataset; a remote data access API; and a management dashboard that gives an overview of all data in the system and allows users to drill down to explore particular datasets of interest. CORAL also includes a Python API to enable powerful computational analyses of datasets in Jupyter notebooks [18]. This API provides access to the search engine, retrieves individual datasets and their provenance, simplifies merging of linked datasets to perform integrative analyses, and automatically tracks the provenance of datasets as computational operations (e.g., dimension reduction, clustering, and data transformation) are applied to them. Details of these features are provided in the Methods section, and examples of the User Interface (UI) are shown in the supplementary information.

Discussion

For most fields of business or research, information typically flows between 3 groups: from data generators who create data, to data scientists who analyze the data, and to results consumers who use the analyses created by the data scientists in order to make decisions. In ENIGMA, data are generated by wet labs and analyzed by computational biologists, and the results are used by project management to plan future experiments. Each of the 3 groups has its own language—terms that are critical to documenting data in a *Reusable* and *Interoperable* way. Data scientists who perform analyses typically are more concerned about the mathematical content of the data than about preserving the context provided by the data generators, so conventional tools for data analysis (e.g., R, pandas) focus on the numeric data structure and often lose context.

A benefit of the CORAL data model is that it requires that all context be formally documented by data generators, and it also ensures that this context is not lost during data analysis. Lower-level libraries typically used by data scientists make it easy to lose such context, but CORAL ensures that such context is maintained throughout the data analysis pipeline. Maintenance of context throughout the entire information processing pipeline is critical for the end users of the data (results consumers) to make informed and correct decisions. Users in particular fields of business can build custom solutions that maintain relevant context, but CORAL is a drop-in solution for each industry, once the ontologies and microtypes relevant to that field are chosen or created.

Choosing an appropriate set of microtypes to describe all data in a CORAL instance is analogous to choosing a language for that instance and must be done by instance administrators prior to uploading data. Additional microtypes can be added to an instance as needed, for example, if additional types of measurements will be performed. Microtype names as well as other ontological terms are arranged in hierarchies, with more specific types and terms linked to more general concepts. Well-designed microtypes and ontologies provide a single controlled vocabulary term to describe a synonymous group of concepts, thus limiting ambiguity. Requiring all contextual labels to be built from contextons rather than free text also eliminates the possibility of errors due to misspelling.

The CORAL approach of building complex data types dynamically using a limited number of mathematical data structures labeled with contextons has several additional benefits:

- The number of mathematical data models is small, and thus many tools and libraries can be implemented in a "generic" way, so that each can operate on many data types in the system. For example, tools to analyze or visualize matrix data could operate on all complex data types that are fundamentally matrices. This is an advantage even over the approach of using fully specified schema for each data type, as such data types are heterogeneous and thus require custom tools.
- Any user (not just "power users," such as instance administrators) can add new ad hoc dynamic data types to the system. This is possible because the microtypes and ontologies defined by instance administrators act as a common language for each CORAL instance. Much of the design cost (e.g., data validation) is already built in at the microtype level. Users can construct an enormous variety of complex data types from a few simple, mathematical data models and well-defined contextons, analogous to the enormous variety of objects that can be built out of LEGO bricks.
- Complex dynamic data types made using our model are readily understandable to end users, compared to fully specified schema. Once users are familiar with the language (set of microtypes and ontological terms) used on each instance, the limited number of mathematical data structures underlying our complex types makes the data easy to grasp and thus improves *Reusability*. Furthermore, by allowing users to define ad hoc data types as needed, users can store their data in a format that more naturally reflects their experimental design, rather than shoehorning data into rigidly specified predefined schema.

In conclusion, CORAL is a framework that organizations can use to rapidly deploy a powerful, integrated database to catalog, store, and analyze all their data. Large organizations typically generate or use hundreds of types of data, each with a different, often customized, format, including slightly differencing variations on the same data types (e.g., a time series vs. a condition series or with vs. without replicates). The traditional method of building a bespoke system for data management is time-consuming and expensive, requiring a team of data scientists and database experts. CORAL enables an organization to easily create and reuse templates that capture the unique aspects of each of their hundreds of data types while also clustering conceptually similar types in order to organize the data and make them easy to understand. Together, these features provide a uniquely powerful tool to enable integrative cross-dataset analyses, generating deeper insights than are possible using traditional analysis tools.

Methods

Details of microtype structure

Each microtype has the following properties:

- Name—(required) name for this microtype; must be an ontological term, which is unique within a particular CORAL instance.
- Description—(required) a formal definition for this microtype. It should not include units; those are provided by a separate "unit" part of each contexton.
- Scalar type—(required) microtype values are restricted to a single scalar type. For example, values for a microtype such as "Concentration" could be restricted to being floating point numbers. Values for a microtype such as "Chemical" could be restricted to being ontological terms.

Use of ontological terms rather than strings as a scalar type for values is highly recommended for 2 reasons. First, as with the ontology of microtype names, collapsing synonyms into a single term eliminates ambiguity. Second, using ontologies as scalar types enables a powerful form of data validation (e.g., the designer of a data type may require that a user populate a field by choosing between a limited set of possible ontological terms).

For scalar types that are numeric, the microtype designer must specify options for valid units of measurement. As described below in "Details of contextons," units *must* be specified as ontological terms. The microtype designer may also specify 1 or more parent classes of ontological terms as valid units (e.g., "concentration unit") in order to assert that all child terms (e.g., "mg/ml") in the ontology are valid units for the microtype.

- Validator—(optional) allows designers to limit the allowed values for this microtype. For example, a string could be required to match a particular regular expression, a numeric type could be input to a function to determine its validity (e.g., to restrict values to nonnegative numbers), and ontological terms are restricted to particular ontologies or parts of ontological hierarchies. For example, a "chemical" microtype may be defined so that its allowed values are ontological terms from the Chemical Entities of Biological Interest ontology [19].
- Synonyms/aliases—(optional) synonymous terms for this microtype. This facilitates mapping of input data to microtypes while limiting ambiguity in contextual descriptions. For example, "ASV," "ESV," "Exact Sequence Variant," and "sub-OTU" are all synonyms for the same concept: a single amplicon sequence that was inferred to be present in a sample prior to the introduction of amplification and sequencing errors.

For all aspects of microtypes that are defined as ontological terms (e.g., name, synonyms, values, and units of measurement), microtype designers should endeavor to use community standard ontologies wherever possible. This will enable *Interoperability* with other advanced data science applications that use ontologies, since these ontologies can be developed by wide communities and used across diverse data systems.

System microtypes

To set up a new instance of CORAL, system administrators must define a set of microtypes called "system microtypes," which are mandatory parts of data objects. These include the *Data Type*, *Dimension Type*, *Values Type*, and *Unit Type* microtypes. The allowed values of each of these microtypes is a set of ontological terms. These sets of terms are specified by instance administrators, which allows them to specify broad categories of both static and dynamic data types that may be built in each instance of CORAL. This process is analogous to choosing a language for that specific CORAL instance (see Discussion). When a dataset is uploaded, the specific values of each of these microtypes must be chosen by users from among the allowed terms, in order to formally document the dataset.

Details of contextons

Values for contextons contain either a single scalar or an array of the same type of scalars; valid scalar types may be defined in the corresponding microtype. The dimensionality of the value or array of values depends on what the contexton is providing context for. See Fig. 6 for examples.

Units for contextons *must* be specified as an ontological term, to eliminate ambiguity. By default, unit values may be chosen from the terms in the community standard PATO Units ontology [20], although CORAL also allows instance administrators to define custom units that are relevant to their domain of interest, as defined by the *Unit Type* system microtype.

Compound contextons

In some cases, context is best described through a combination of atomic microtypes. For example, to express the concept "concentration of nitrate," multiple contextons are combined to build a single *compound contexton*. The primary contexton includes the microtype that indicates the main property being measured (concentration), while any number of additional contextons may be used to modify the first (e.g., indicating that the molecule whose concentration is being measured is nitrate or that a series represents technical replicates rather than biological replicates). An example is shown in Supplementary Fig. S5.

Implementation

A complete prototype implementation of CORAL, including documentation and installation instructions, is available at https://github.com/jmchandonia/CORAL. This prototype consists of 2 parts: a front end written in Angular and a back end written in Python. Both parts must be installed on a single host, and the front end may then be accessed from other hosts using a web browser. The prototype code repository also includes sample configuration files, microtypes, data, and installation instructions that enable the installation of an example CORAL instance containing a subset of published ENIGMA data.

In the prototype back end, CORAL's NDArray dynamic types are built on top of the widely used and well-supported open-source xarray library [10]. Each NDArray corresponds to a single xarray *Dataset* object, and reserved key-value pairs are used to store additional metadata (e.g., contextons) that are mandatory in CORAL's NDArray objects. The xarray library stores and loads each of these *Dataset* objects from the local filesystem in NetCDF-4 format [12]; this format implements a subset of the HDF5 data model [11].

In order to implement the FAIR principles of *Findability* and *Accessibility*, CORAL stores all datasets in the document store of the open-source, multimodel database ArangoDB. Ontological terms, metadata about dynamic objects, processes, and each type of static object are stored in an individual collection. As mentioned above, the dynamic datasets themselves are stored on disk in NetCDF-4 format, while the ArangoDB collection indexes metadata such as the ontological terms referenced in each dataset. ArangoDB represents all of the provenance (i.e., process) links between all core data objects and dynamic datasets using a graph database. As a multimodel database, ArangoDB allows queries of the graph database and document store simultaneously, simplifying implementation of the CORAL API. Every data object in CORAL is assigned a permanent identifier, and old versions of data will be marked as superseded by newer versions (with a new identifier) rather than deleted. This process ensures that all versions of all datasets in CORAL are retrievable by their ID at any time.

Dynamic type validation

As described above, each microtype includes custom validation methods to ensure that it stores only valid data. To maximize *Interoperability* and *Reusability* of our dynamic data objects, CORAL supports 2 additional types of data validation: upload templates and use templates. These are described below.

Upload Templates simplify the process of importing data into CORAL by specifying a default set of dimensions and other contextual data for a dynamic data type. When uploading data, the user can optionally add to or remove the default data from the template. For example, a template for microbial growth data might specify a default time series as 1 dimension and specify that the values in the matrix are, by default, optical density values with dimensionless units. The template also specifies how to handle invalid data in each underlying microtype or even specify more strict limits on validating these data. When new datasets are uploaded, they are validated against all the constraints imposed by their underlying ontological terms. For example, if a field is described by an ontological term that implies a numeric value, all such values are checked to ensure they are numeric before the data can be uploaded. Similarly, if a term indicates that a field is a foreign key linking to a core type, the presence of all linked

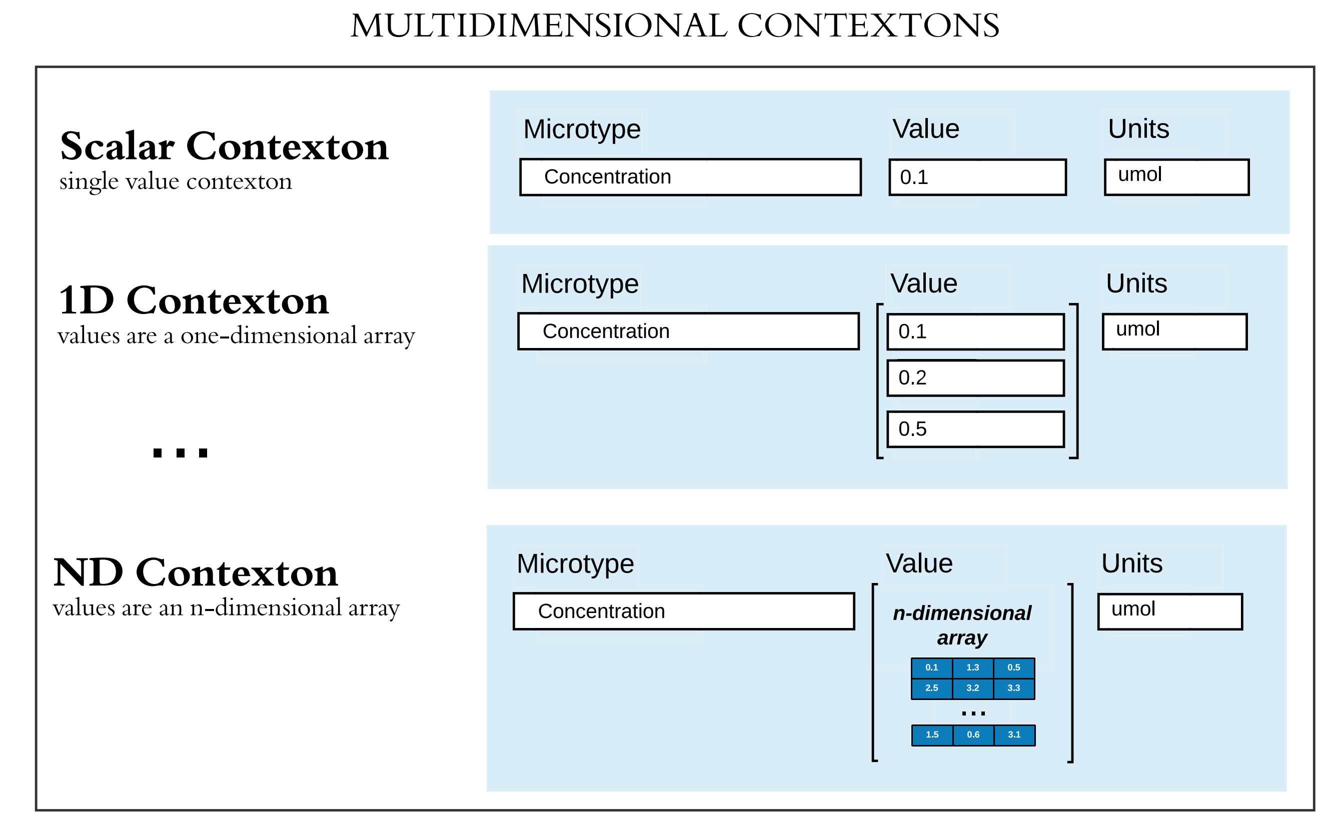

Figure 6: Examples of single-value (i.e., scalar), 1-dimensional, and N-dimensional contextons. Contextons are of the same dimensionality as the mathematical data object that they provide context for. For example, the data values in an N-dimensional array are N-dimensional contextons, while the values of variables on each dimension axis are 1-dimensional contextons with the same length as each dimension. Single-value contextons are used to provide context for an entire dataset and as modifiers in building compound contextons.

datasets is verified before the new data are accepted. By default, trying to import invalid data results in an error, preventing them from being uploaded. However, the user can choose, via the templates, whether to treat invalid data as missing (i.e., null values) or to adjust outliers to be within an acceptable range. In these cases, the user is warned of the changes when uploading, but the dataset is accepted.

Use Templates are applied when dynamic datasets stored in CORAL are retrieved for use. When writing a function to manipulate dynamic typed objects, the algorithm developer can specify, using a template, which dynamic objects the algorithm accepts. For example, if an algorithm only works on time-series data, the algorithm developer can specify that the algorithm only works with datasets that have at least 1 dimension labeled with the *Time Series* microtype. Use templates are therefore similar to interfaces in object-oriented languages such as Java and C++.

Search

The provenance graph (Fig. 2C), which links each dataset in CORAL with the process and input objects used to generate it, enhances both *Findability* and *Reusability* of static and dynamic datasets. Data are more *Findable* because they can be searched for using graph queries that are supported by the underlying database, in addition to any of the context data within the dynamic type itself (both the microtypes and the ontological terms used as values can be searched). These queries can be quite powerful, for example, identifying all data objects that ultimately resulted from work performed by a particular person or group. The graph of samples and core types also provides additional provenance, using formal ontological terms, for each dynamic type, which is a key aspect of enhancing *Reusability*.

Because ontologies and microtypes are both stored in tree-like hierarchies, CORAL enables users to search by ancestral terms as well (e.g., a search for “carbohydrate” will also find all data that refer to “glucose”).

Visualization

CORAL includes “wizards” to build powerful interactive visualizations of dynamic datasets. For multidimensional data, the user specifies which dimensions are to be graphed on which axes of the plot. CORAL then generates the plot, labeled according to the appropriate microtypes and ontological terms used in each dimension. Examples are shown in Supplementary Fig. S4.

Which plots are available depends on the number of dimensions in each dataset and the scalar types associated with the microtypes that provide context in each dimension. This feature requires the *Use Templates* functionality described above. For example, to visualize data as a 2-dimensional line graph, the dataset must have at least 1 dimension labeled with a microtype with numeric values and also have an overall values type labeled with a numeric microtype. The plotly library [21] is used to draw the plots.

If the number of dimensions in a dataset exceeds the number that can be plotted in a given plot type (e.g., a 4-dimensional dataset on a 2-dimensional line graph), the interface requires additional dimensions to be constrained by the user. Each dimension may be plotted as a series or averaged, or a single value from the dimension may be chosen.

Datasets that contain or are linked in the provenance graph to 2 specific microtypes, *latitude* and *longitude*, may also be plotted on a Google Map. Map pins may be colored according to any variable in the data brick.

Dynamic libraries

In addition to uploading data, CORAL users can perform analyses of the dynamic data types stored within the system, then save the resulting dynamic data objects. In this case, rather than asking the user to provide provenance for the objects, all provenance is tracked automatically. We wrapped a number of popular Python functions for manipulating data, such as slicing, merging, or other mathematical manipulations such as dimensionality reduction. When these functions are run on dynamic datasets, the object's "session provenance" stores all computational manipulations that have been performed on the data since the time they were retrieved from storage, including records of all methods called and the method parameters. Thus, if the user stores processed results in the CORAL data store, this session provenance is also saved, so that the same process can easily be replicated.

Our NDArray dynamic data type is particularly well suited for modeling observational and physical data. We focus on this data type because observational datasets are among the most heterogeneous and complex-structured data that have to be managed. In addition, observational data must be searchable, sliceable, and mergeable in such a way as to facilitate statistical and physical modeling, which is often a goal of computational data analysis.

Dynamic joins (step-by-step merging)

To improve *Interoperability* of dynamic data objects, we provide tools to easily merge data from other static and dynamic datasets that are linked in the provenance graph.

For example, the dynamic dataset of geochemical data is linked to the Sample core type and from there to the Location core type. A user wishing to correlate geochemical data with the sampling locations and/or depths could ask CORAL to automatically retrieve and merge all data from the *Sample* core type as additional variables in the dynamic geochemistry dataset. The user could then merge latitude and longitude data into their dataset by using the *Sample* identifiers to map each data point back to the *Location* core type. The resulting larger dataset could be saved as a new dynamic dataset in the CORAL data store or analyzed further using the wrapped Python libraries.

An example of Dynamic Joins is shown in Supplementary Fig. S6.

Authentication and Permissions

CORAL uses the OAuth 2.0 protocol to manage user access. CORAL instance administrators may add or remove access to individual OAuth identities; however, any user with access to a CORAL instance also has access to all the data stored in that instance. Instance administrators may specify whether a user has access to a bulk upload interface (which allows upload of preformatted large datasets, eliminating the Excel steps in the upload wizard described below), as well as a list of static data types they are authorized to upload.

Python API

CORAL provides a unified Python API to support many queries and manipulation of data objects. There are several types of CORAL users, each with their own API needs. Data providers (e.g., bench scientists or clerical workers) need a specialized interface to upload data, supported by API functionality to map strings to ontological terms, perform data validation, and add new objects to the system. Managers need a specialized interface that gives an overview of the system, with the ability to drill down into particular objects, and this is supported by a powerful search API. Finally, data scientists will manipulate datasets in Jupyter notebooks and need APIs for data manipulation, wrappers for popular Python tools, and powerful search tools.

Because the CORAL API is built using standard Python libraries for numeric analysis, such as pandas, NumPy, and xarray [10], it is simple for data scientists to use the API to export CORAL data in a number of formats, including simple tab-delimited tables to HDF5 [11] and NetCDF-4 [12].

Additional documentation of our Python API is provided in the supplementary information.

REST API and Remote Data Access

The prototype implementation of the CORAL back end includes a REST API that provides data to the front end. Authentication of this API is done using OAuth, with information about the logged-in user stored in JWT tokens. The same REST API also allows users who are not logged in but possess a preshared cryptographic token to remotely run the search API and retrieve the resulting static and dynamic datasets. This functionality provides both *Findability* and *Accessibility* of all data in CORAL. This feature also allows developers to build specialized websites to display particular types of data that are stored in CORAL, without requiring users to have accounts on the CORAL server. CORAL uses public-key cryptography to validate the tokens, which are generated in pairs, with 1 of the 2 tokens secured on the server and the other given to developers authorized to access the data. If the developer token is compromised, access may be revoked by removing the server-side token.

Documentation of our REST API, including our Remote Data Access API, is provided in the supplementary information.

Upload wizard

The uploader was designed to allow non–computer science users (e.g., bench biologists or other data providers) to upload data. We guide them through providing all the necessary context and provenance for each new dataset using a "wizard." When uploading new data, complete provenance must be provided: all input objects, processes, and other data are then connected in the ArangoDB graph database to the newly uploaded object. The wizard is shown in more detail in the supplementary information.

The upload wizard starts by asking the user to document all the data variables and dimensions in the data they are uploading. The interface allows the user to select from drop-downs of allowed data types, units, and so on. If similar datasets are uploaded multiple times, instance administrators can create an "upload template" for the corresponding data type, which provides default answers to these questions. Once the user specifies the structure of the data, the wizard creates an Excel template for the user to paste data into. Rows and columns in the template are labeled and colored to reduce the possibility of error. After the user uploads the filled-in template, CORAL interactively allows the user to check the data, then validates the data to ensure that all constraints are satisfied. For example, references to static objects must use the correct names of objects already uploaded into the system.

A final step requires the user to specify the process and personnel that generated the data. This allows CORAL to connect the newly uploaded dataset to the appropriate inputs in the provenance graph.

Management tools

CORAL provides several interfaces that display an overview of all data in the system, allowing users to drill down to view particular data objects. A management dashboard supports queries of particular interest to managers, such as who is producing which datasets, which datasets are most widely used, and so on.

A key aspect of the management dashboard is an overview of the provenance graph, showing all datasets in the system. This is the default view that is displayed to all users upon logging in to CORAL. Double-clicking on individual datasets displays them in the search interface. Checkboxes on the left side of the graph allow users to filter the display to show only a subset of the data (e.g., data produced by a particular project, lab, or person).

Data scientist view

Data scientists access the CORAL API through Jupyter notebooks, running in a shared directory of a server running JupyterHub. Default permissions are set by the organization. The default CORAL permission settings allow any user to read others' notebooks, but writing privileges must be explicitly granted (although a user can fork a writable copy of another user's notebooks). Because all notebooks and data are readable by any user with access to an instance of CORAL, every user may easily reproduce others' analyses.

Availability of Source code and Requirements

Project name: CORAL
Project homepage: https://github.com/jmchandonia/CORAL [22]
Operating system: The prototype back-end code has been tested on Debian GNU/Linux, CentOS Linux, and macOS. The prototype front end runs on any modern web browser.
Programming language: The majority of the prototype back-end code is Python, with some Java dependencies. The prototype front end is written in Angular (TypeScript).
License: GNU Affero General Public License (AGPL-3.0 license) or (at the recipient's option) a separate commercial use license.
RRID: SCR_0022711

Data Availability

Snapshots of our code and other data further supporting this work are openly available in the *GigaScience* repository, GigaDB [23].

Additional Files

Supplementary Information: Additional details concerning the CORAL user interface, features, and API documentation are available in the supplementary file "giac089_Supplemental_File", online at https://academic.oup.com/gigascience/article/doi/10.1093/gigascience/giac089/6762021#supplementary-data.

Supplementary Fig. S1. The provenance graph displays linked data as a directed graph of boxes. Blue ovals indicate static data types, while orange squares represent dynamic data types. Static data types with no external data inputs are shown near the top of the graph and highlighted in green as a starting point for browsing the data. Double-clicking on individual objects brings the appropriate datasets up in the search interface. Checkboxes on the left side (ENIGMA Campaigns and ENIGMA Labs/People) allow smaller subsets of the data to be shown. If any boxes are checked, only the data produced by a particular lab, project, or person, as well as inputs and outputs, are shown.

Supplementary Fig. S2. (A) Data generators can upload dynamic datasets using predefined templates or build a new data structure on the fly. A list of broad data categories (e.g., "microbial growth") is defined by the CORAL instance administrators, as are all the possible microtypes and ontological terms that can be used by bench scientists to describe their data. (B) Once a data category is chosen, data generators then define the variable(s) stored in their data object. These definitions indicate exactly what quantity was being measured and the units of measurement. The drop-down for units only displays appropriate units as defined in the microtype. All fields use autocompletion, with a drop-down that displays appropriate microtypes or units that are filtered using the characters typed by the data generator. (C) Generators next define the overall structure of the data: how many dimensions are in the data, and what varies along each dimension? Generators input these terms using auto-completing text boxes, and valid units for each variable must be selected from a menu. Valid dimension types and valid units for each microtype are defined by the CORAL instance administrators. (D) CORAL then generates a template spreadsheet into which data generators will paste their data. The template is of the appropriate dimensionality for the data type that was defined in the previous steps. All of the variable names and units are indicated in the template, so the user is clear on where to paste everything. (E) The data generator pastes their data (including the values of variables along each dimension) into the template, then uploads the file. (F) CORAL then gives the generator a preview of the data, so they can check whether everything was parsed correctly from the template. Generators may also add additional contextons that describe the context of the entire dataset. (G) Datasets are validated (via validators defined in the microtypes) and linked (via microtypes that refer to other data in the system) before they can be accepted. (H) As a last step, the data generator provides information about the process by which they created the data, which is used to build links in the provenance graph.

Supplementary Fig. S3. An overview of a 4-dimensional dynamic dataset containing metals measurements on 209 environmental samples.

Supplementary Fig. S4. (A) A wizard guides the user through plotting a 4-dimensional chemical measurements dataset as a vertical bar chart. (B) An interactive bar chart is shown in response to the user-specified options shown in Supplementary Fig. S4A. This logarithmic plot shows the concentration of nitrate in a number of environmental samples. The "Share Plot" button allows users to copy the URL, which they can share with other CORAL users to display the same plot again. (C) An example of the interface for plotting dynamic datasets on a map. (D) An example of the map resulting from Supplementary Fig. S4C, in which the concentration of nitrate from several samples is plotted on a map, according to the sampling location.

Supplementary Fig. S5. This figure shows the top level of the microtype tree browser, with the "ENIGMA" category opened to show several subhierarchies under it and the "Measurement" hierarchy opened to show some of the microtypes that represent measurements. The "Filter by keyword" box at the top also allows users to search all microtypes and their definitions; as the user types, the tree is immediately filtered to show only relevant microtypes and their parents.

Supplementary Fig. S6. Example of compound contexton. The concentration microtype indicates the primary property being measured. Additional modifying contextons indicate that the molecule for which the concentration is being measured is nitrate, that the measurement is being performed at 25 degrees Celsius, and that the values in the contexton are logarithmically scaled (log base 2).

Supplementary Fig. S7. Example of Dynamic Joins: the user starts with a Source dynamic dataset, which contains data that are linked to the core static type “Well” through the “Well ID” microtype, which serves as a primary key for Well objects. By using Dynamic Joins, the user can bring in fields from the linked (“Source”) Well objects, which become dimension variables in a new dynamic dataset, labeled Target. The Target dynamic object contains fields from both Source objects.

Abbreviations

CDE: Common Data Element; CORAL: Contextual Ontology-based Repository Analysis Library; KBase: DOE Systems Biology Knowledgebase; SDK: Software Development Kit.

Competing Interests

The authors declare that they have no competing interests.

Funding

This material by ENIGMA (Ecosystems and Networks Integrated with Genes and Molecular Assemblies) (https://enigma.lbl.gov), a Science Focus Area Program at Lawrence Berkeley National Laboratory, is based upon work supported by the US Department of Energy, Office of Science, Office of Biological and Environmental Research, under contract number DE-AC02-05CH11231.

Authors’ Contributions

PSN, JMC, and APAconceived the project. PSN and JMC designed the data model, with feedback and advice from APA. PSN wrote the prototype back-end code and supervised a contractor’s implementation of the prototype front end. JMC contributed bug fixes and new features to the prototype back end and supervised implementation of bug fixes and new features in the front end. JMC and PSN developed microtypes and contextons for the ENIGMA deployment of CORAL. JMC converted sample ENIGMA data from [24] to CORAL format. PSN, JMC, and APA tested the software. JMC drafted the manuscript, with feedback and edits from PSN and APA.

Acknowledgments

We thank Gavin Price and other DOE Systems Biology KnowledgeBase (https://kbase.us) team members for helpful discussions.

References

1. Wilkinson, MD, Dumontier, M, Aalbersberg, IJJ, *et al.* The FAIR Guiding Principles for scientific data management and stewardship. *Sci Data* 2016;**3**(1):160018. https://doi.org/10.1038/sdata.2016.18.
2. Codd, EF. Relational database: a practical foundation for productivity. *Commun ACM* 1982;**25**(2):109–17.
3. Pokorny, J. NoSQL databases: a step to database scalability in web environment. *Proc 13th Int Conf Inf Integr Web-Based Appl Serv.* pp. 278–83. Ho Chi Minh City, Vietnam: Association for Computing Machinery, 2011. https://doi.org/10.1145/2095536.2095583.
4. Griffin, PC, Khadake, J, LeMay, KS, *et al.* Best practice data life cycle approaches for the life sciences. *F1000Research* 2018;**6**:1618.
5. Baker, M. 1,500 scientists lift the lid on reproducibility. *Nat News* 2016;**533**(7604):452–4.
6. Nature: Challenges in irreproducible research. 2018. https://www.nature.com/collections/prbfkwmwvz.
7. Freedman, LP, Cockburn, IM, Simcoe, TS. The economics of reproducibility in preclinical research. *PLoS Biol* 2015;**13**(6):e1002165.
8. Arkin, AP, Cottingham, RW, Henry, CS, *et al.* KBase: The United States Department of Energy Systems Biology Knowledgebase. *Nat Biotechnol* 2018;**36**(7):566–9.
9. Goecks, J, Nekrutenko, A, Taylor, J; The Galaxy Team. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biol* 2010;**11**(8):R86.
10. Hoyer, S, Hamman, J. xarray: N-D labeled arrays and datasets in Python. *J Open Res Softw* 2017;**5**(1):10.
11. The HDF Group. *Hierarchical Data Format, Version 5*. HDF Group; 1997–2022. https://www.hdfgroup.org/solutions/hdf5/.
12. Unidata. *Unidata | NetCDF. Netw. Common Data Form NetCDF Boulder CO UCARUnidata*. 2020. http://doi.org/10.5065/D6H70CW6. Accessed 2022.
13. Gonçalves, RS, O’Connor, MJ, Martínez-Romero, M, *et al.* The CEDAR workbench: an ontology-assisted environment for authoring metadata that describe scientific experiments. *Semantic Web–ISWC Int Semantic Web Conf Proc Int Semantic Web Conf*. 2017. https://doi.org/10.1007/978-3-319-68204-4_10.
14. ISO. 14:00-17:00: ISO/IEC 11179-3:2013. https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/05/03/50340.html. Accessed 11 July 2022.
15. Grinnon, ST, Miller, K, Marler, JR, *et al.* National Institute of Neurological Disorders and Stroke Common Data Element Project—approach and methods. *Clin Trials* 2012;**9**(3):322–9.
16. Studer, R, Benjamins, VR, Fensel, D. Knowledge engineering: principles and methods. *Data Knowl Eng* 1998;**25**(1–2):161–97.
17. Gruber, T. Toward principles for the design of ontologies used for knowledge sharing. *Int J Hum Comput Stud* 1995;**43**(5–6): 907–28.
18. Kluyver, T, Ragan-Kelley, B, Pérez, F, *et al.* Jupyter Notebooks—a publishing format for reproducible computational workflows. In: F Loizides, B Scmidt, eds. *Position and Power in Academic Publishing: Players, Agents and Agendas*. IOS Press; 2016. https://doi.org/10.3233/978-1-61499-649-1-87.
19. Hastings, J, Owen, G, Dekker, A, *et al.* ChEBI in 2016: Improved services and an expanding collection of metabolites. *Nucleic Acids Res* 2016;**44**(D1):D1214–9.
20. Gkoutos, GV, Schofield, PN, Hoehndorf, R. The Units Ontology: a tool for integrating units of measurement in science. *Database* 2012;**2012**:bas033.
21. Plotly Technologies Inc. *Collaborative Data Science*. Montréal, QC, 2015. https://plotly.com.
22. CORAL GitHub repository. v0.1.2. 2022. https://github.com/jmchandonia/CORAL.
23. Novichkov, PS, Chandonia, J, Arkin, AP. Supporting data for “CORAL: A framework for rigorous self-validated data modeling and integrative, reproducible data analysis.” *GigaScience Database*. 2022. https://doi.org/10.5524/102249.
24. Smith, MB, Rocha, AM, Smillie, CS, *et al.* Natural bacterial communities serve as quantitative geochemical biosensors. *mBio* 2015;**6**(3):e00326–15. https://doi.org/10.1128/mBio.00326-15.