Using Object-Relational Mapping to Improve Sustainability of a Research-ready Clinical Cancer Data Platform

Georgina Kennedy[1,2,3], *Tim Churches*[1,2]
[1] *Faculty of Medicine & Health, UNSW Sydney,* [2]*Ingham Institute of Applied Medical Research, Liverpool, Sydney,* [3]*Maridulu Budyari Gumal (SPHERE) Cancer Clinical Academic Group, Australia*

Background

The CaVa data platform has been established to support streamlined health-services research into the causes and effects of ***Ca***ncer care ***Va***riation. It takes a two-pronged approach of (1) making data 'Researcher Ready', by harmonizing and normalizing the full breadth of historical clinical cancer data to the OHDSI OMOP CDM with Oncology Extension [1], as well as (2) making researchers 'Data Ready' by offering a library of templates to support stereotypic analyses, bootstrapping new analyses, allowing teams to spend more time focusing on the unique aspects of their research. This will improve both efficiency and quality of analyses.

The inclusion of multiple distinct source data systems – currently MOSAIQ (Elekta) and PowerChart (Cerner), with a roadmap towards Aria (Varian) as well as linked administrative data sources – is a high-complexity target. These heterogenous data sources pose one obvious and considerable design challenge, but if the platform is to be able to provide near-real time updates for prospective data, changes within an individual source system must also be accounted for. Any solution that fails to consider the drift of data source configuration, code sets, and supporting clinical and business processes will be brittle and ultimately unsustainable.

Methods

The CaVa data processing pipeline leverages an Object Relational Mapping (ORM) paradigm [2] to create a layer of abstraction between the raw data source and the target CDM format. This uses the concepts of object-oriented (OO) programming (in this case, in Python) to decouple lower-level data elements from the business processes required to support their inclusion in the target model. This means that CaVa does not issue any SQL statements directly, instead using SQLAlchemy to map Python classes to the data models and then querying data and managing database connection and transactions through the ORM and Core APIs respectively.

Example model definition and data validation

Figure 1 shows the SQLAlchemy model definition for a portion the CONDITION_OCCURRENCE table, including the specification of relationships to the PERSON and CONCEPT tables, and the generation of IDs from sequences. It is possible to add methods within the class definition to provide validation and support conventions such as acceptable domains or relationships. Two different validation methods are demonstrated here – for simple conventions across columns within the target table, the `@validates` decorator will enforce consistency with data validation rules, raising an exception prior to mutating the attribute. For complex validation rules that apply across related objects, the session-level event hook `before_flush` is used instead. The attribute is updated before the check is applied, however it can be reverted within the validation function itself. The ability to implement convention validation at load-time across many classes is a clean and fluent way to enforce consistency of design decisions across implementations.

```
class condition_occurrence(Base):
    __tablename__ = 'condition_occurrence'
    condition_occurrence_id: Mapped[int] = mapped_column(Integer,
                                                         index=True,
                                                         primary_key=True)
    condition_start_date: Mapped[Optional[Date]] = mapped_column(Date)
    condition_start_datetime: Mapped[Optional[DateTime]] = mapped_column(DateTime)
    condition_end_date: Mapped[Optional[Date]] = mapped_column(Date)
    condition_end_datetime: Mapped[Optional[DateTime]]  = mapped_column(DateTime)

    condition_type_concept_id: Mapped[int] = mapped_column(BigInteger,
                                                           ForeignKey('concept.concept_id'))

    condition_type_concept: Mapped[concept] = relationship(lazy='joined',

foreign_keys=[condition_type_concept_id])

    def convention_validation(self, sess):
        domain = sess.query(concept.domain_id
                            ).filter(concept.concept_id==self.condition_type_concept_id
                            ).one_or_none()
        if domain and (domain[0] != 'Type Concept'):
            raise AssertionError(f'Cannot assign condition_type_concept_id of domain {domain}')

    @validates('condition_start_date', 'condition_start_datetime')
    def validate_start_dates(self, key, field):
        # pairwise comparison of either start date field against either end date field
        for comparator in [self.condition_end_date, self.condition_end_datetime]:
            if isinstance(comparator, datetime):
                if comparator < field:
                    raise AssertionError(f"{key} cannot be later than the condition end date")
        return field

@sa.event.listens_for(db_session, 'before_flush')
def _convention_check(session, flush_context, instances):
    for target in [*session.new, *session.dirty]:
        # this listner will be triggered for all new and dirty objects
        # but will only run validation for mapped classes where a 'convention_validation'
        # method has been created - only called if fields have actually been updated
        method = getattr(target, 'convention_validation', None)
        if callable(method) and session.is_modified(target):
            method(session)
```

Figure 1: Condensed code snippet illustrating the mapping of a Python class to reflect the nature of the underlying data model definition as well as validation of relationship conventions through a session-level event listener.

```
def get_table_links_from_ORM(Model):
    # build graph for traversing ORM
    edges = defaultdict(list)
    for tablename, tableobj in inspect.getmembers(Model):
        if isinstance(tableobj, sa.orm.decl_api.DeclarativeAttributeIntercept):
            for colname, colobj in inspect.getmembers(tableobj):
                if isinstance(colobj, sa.orm.attributes.InstrumentedAttribute):
                    for fk in colobj.expression.foreign_keys:
                        reference = str(fk.column).split('.')
                        ref_tab, ref_col = reference[0], reference[1]
                        edges[tablename].append((ref_tab, ref_col, colname))
                        edges[ref_tab].append((tablename, ref_col, colname))
    return Graph(edges)
```

Figure 2: Example convenience function using object properties to reduce complexity of queries to end users

Data source changes

When onboarding a new data source, it is possible to kick-start development with model generation scripts that can automatically create an ORM definition to reflect the source model. The CaVa application uses an interim schema definition in csv format that can be defined and updated by non-technical users at the source to generate this code, however libraries also exist to generate directly from the database if preferred (e.g. sqlacodegen). Using class inheritance, it is also possible to make some changes in mapping transparent to the main ETL pipeline, thus reducing the amount of code that must be updated to handle each change.

More than initial setup (which can be burdensome however is only required once per source) a significant benefit also comes from the ability to implement comprehensive version control of the database (including conversion scripts) with the alembic library. Having data validation functionality sit within the class definition itself further improves maintainability through integration with alembic updates.

Changing relational database engine backend

Using dialects, SQLAlchemy can support multiple database types with the same interface code. It is still possible to introduce database-specific behaviour in some cases (e.g. auto-incrementing sequences), however this is easily abstracted away from end users.

Automatic query building

Figure 2 is an example function that is used as the basis for dynamic query generation from an ORM definition, traversing the model definition to create a graph of table relationships. It is thus possible to allow non-technical users to specify their desired source and target table, together with a set of columns, and from this generate queries (including validly defined relationships) to produce a flattened version of output data. Note that due to self-references and polymorphism within the CDM, it is not possible to generate these deterministically and instead a list of candidate query options are generated at this time.

Attributes, properties, and methods

For commonly-used queries or definitions, methods can be implemented to ensure consistency of definition and interpretation. A cancer-specific example would be the definition of a `stage` function within the condition_occurrence class to return a value representing diagnostic stage in a fixed manner according to chosen conventions and defaults (such as selecting between first vs. most recent 'stage' modifier associated to this record), or throw an exception if applied to a non-cancer diagnosis. It is possible to implement arbitrarily complex definitions that can return results of python functions and/or SQL queries, or some combination of both through the use of hybrid attributes.

Conclusion

There are valid criticisms of the use of ORMs, typically around performance as well as the fact that they are not truly a one-size-fits all abstraction and yet they can tend to be treated as such and in doing so can lead to poor design practices and antipatterns [3]. In this case, however, the benefits of being able to treat the clinical records as conceptual objects with defined properties and behaviours strongly outweigh these issues, and in fact that there are plenty of pragmatic design choices already baked into the OMOP CDM that in isolation would be similarly considered to be a deviation from best practice (e.g. polymorphic keys, naïve trees), which are nonetheless required to balance the varied requirements of the user base in a maintainable and usable fashion.

References

1. Belenkaya R, Gurley MJ, Golozar A, Dymshyts D, Miller RT, Williams AE, et al. Extending the OMOP Common Data Model and Standardized Vocabularies to Support Observational Cancer Research. JCO Clin Cancer Inform. 2021;5:12-20.
2. Torres A, Galante R, Pimenta MS, Martins AJB. Twenty years of object-relational mapping: A survey on

patterns, solutions, and their implications on application design. Information and Software Technology. 2017;82:1-18.
3. Karwin B. SQL Antipatterns: Avoiding the Pitfalls of Database Programming: Pragmatic Bookshelf; 2010.

Acknowledgements

Dr Kennedy is a Maridulu Budyari Gumal (SPHERE) Cancer CAG Senior Research Fellow and is supported by a Cancer Institute NSW Research Capacity Building Grant (2021/CBG003).