Using ORMs 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, Australia
2 Ingham Institute of Applied Medical Research, Liverpool, Sydney, Australia
3 Maridulu Budyari Gumal (SPHERE) Cancer Clinical Academic Group, Sydney, Australia

Background

The CaVa data platform supports streamlined health-services research into the causes and effects of Cancer care Variation. 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, 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.

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 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.

We illustrate examples for handling model definitions, data validation methods, onboarding new data sources, database backend changes, automated query building, as well as the implementation of attributes, properties and methods that can be used to ensure consistent definition and interpretation of commonly-used queries or analyses. 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.

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

For commonly-used queries, methods can be implemented to ensure consistency of definition and interpretation. A cancer-specific example would be a stage function within the class condition_occurrence returning a value for diagnostic stage in a fixed manner according to conventions and defaults (e.g. selecting between first vs. most recent 'stage' modifier associated to 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.

Examples

```
class condition_occurrence(Base):
    # SQLAlchemy model definition for a portion the CONDITION_OCCURRENCE table,
    # including specification of relationships to the PERSON and CONCEPT tables

    __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])

    # Methods within class can include definitions for validation & enforce conventions
    # e.g. acceptable domains / relationships

    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 decorator enforces data validation rules, raising exception prior to
    # mutating the attribute

    @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

# For complex validation rules across related objects, session-level event hook before_flush
# used instead. The attribute is updated before check applied, however can be reverted within
# the validation function

@sa.event.listens_for(db_session, 'before_flush')
def _convention_check(session, flush_context, instances):
    for target in [*session.new, *session.dirty]:
        # this listener 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)

def get_table_links_from_ORM(Model):
    # Example convenience function using object properties to reduce complexity of
    # queries to end users. This forms the basis for dynamic query generation from
    # an ORM definition, traversing the model definition to  create graph of table
    # relationships. It is thus possible to allow non-technical users to specify
    # their desired source and target table, and from this generate queries (including
    # validly defined relationships) to produce a flattened version of output data

    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))

    # 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.
    return Graph(edges)
```

Conclusions

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. 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.

Contact: georgina.kennedy@unsw.edu.au @gkenno