Categorical data integration for computational science

Kris Brown
Ryan Wisnesky
David I. Spivak

I'm a Stanford PhD student working in Chemical Engineering.
What I'm about to present has been written up in a paper with the same title here, and the goal of that paper was to outline how CQL could be useful in particular to scientists like myself.

Outline

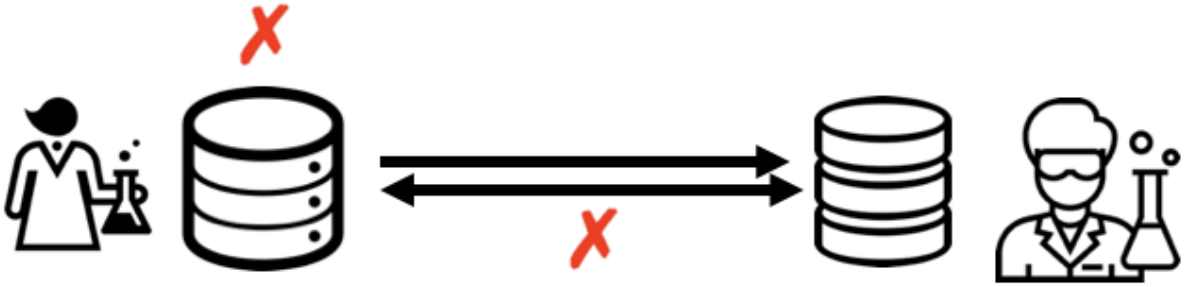
- Background
- Algebraic database schemas
- Functorial data migration
- Applied example

2

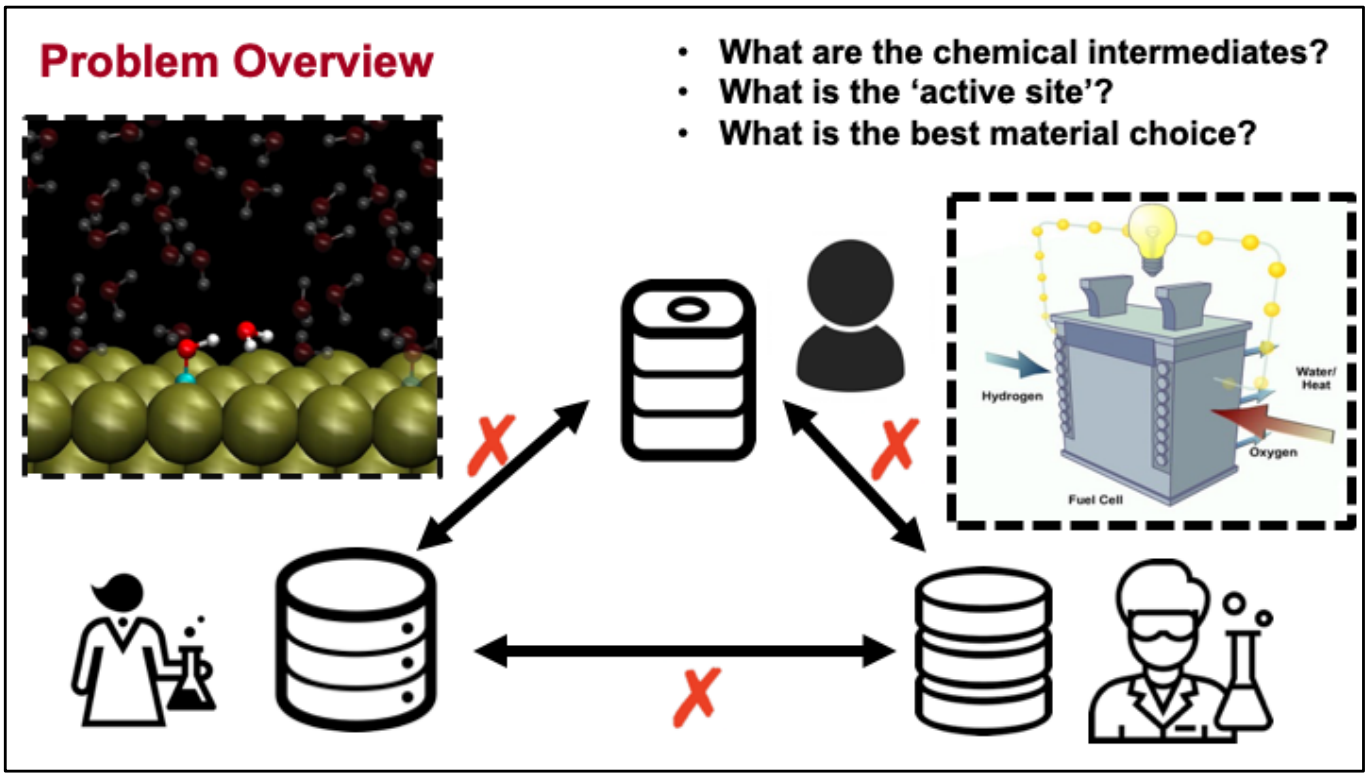
So we'll go over Algebraic databases and Functorial data migration in the context of some toy examples,
and then talk about a bigger integration that I was able to accomplish using CQL,

Problem Overview

- Many scientists do not work with structured data
- Difficult to communicate the nuances of data that is structured
- Lack of tools to combine information in disparate datasets

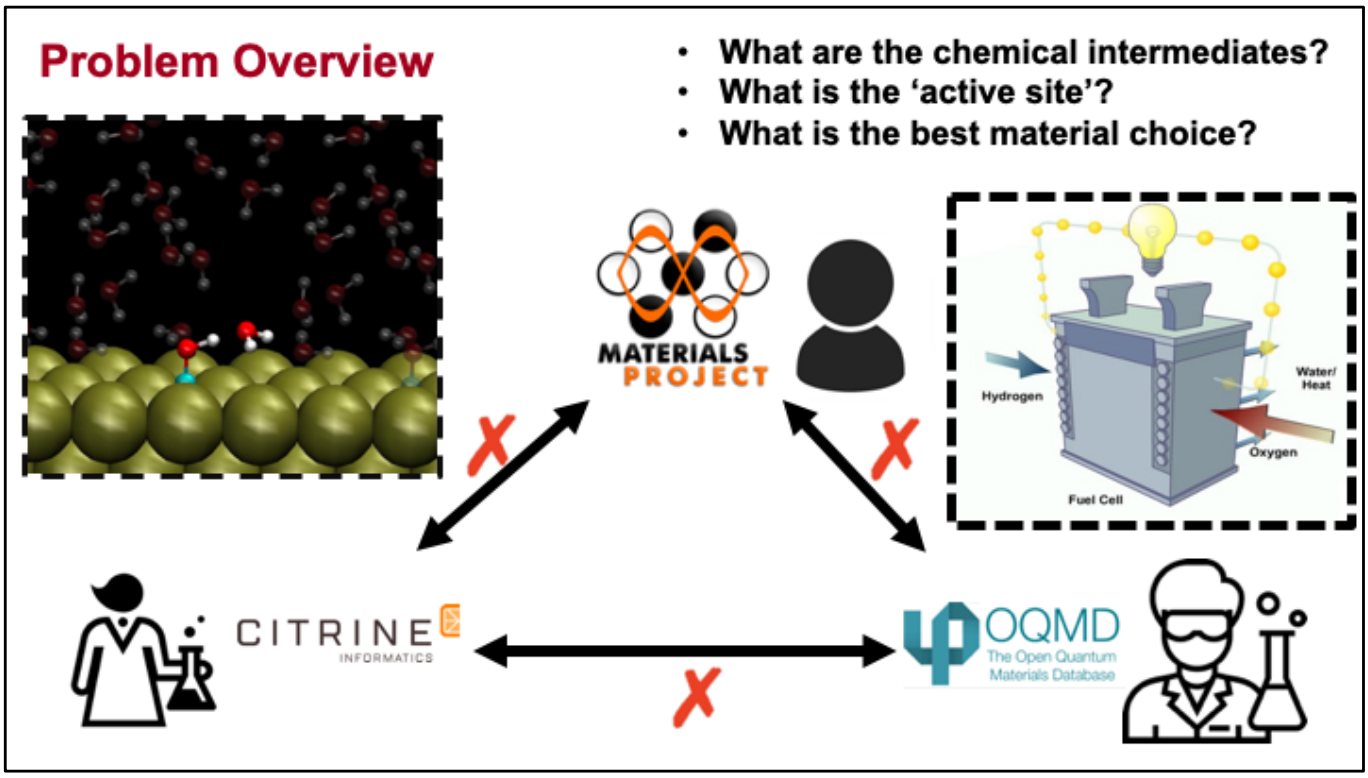


Let me describe the situation that led to an interest in categorical databases:
Data is the currency for scientific hypotheses and theories, and naturally we would like to be able to freely exchange it. However:
<1> <2> <3>
These problems are amplified for scientists over software engineers because basic scripting and data engineering tasks are even more challenging and error-prone.



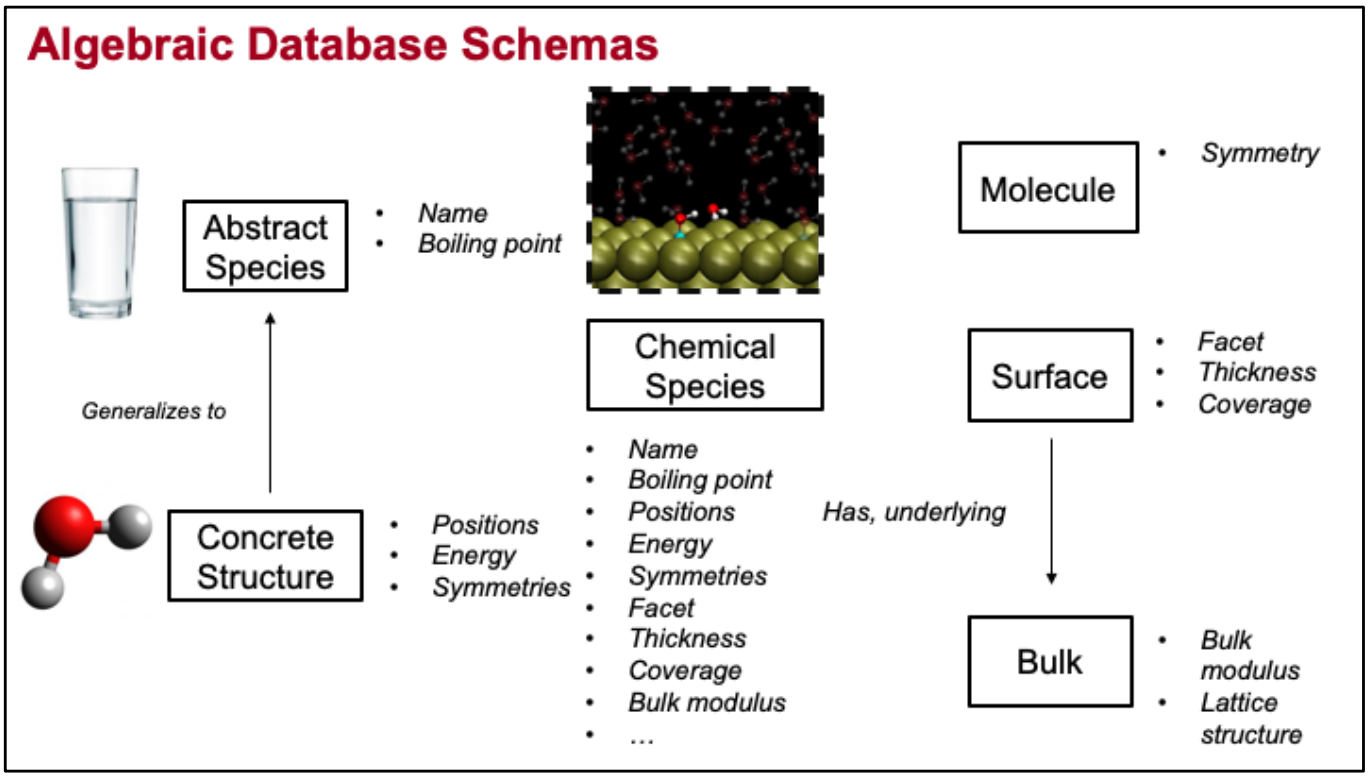
So our reality is that different research groups model the same world in very different ways. To make things concrete, I'll share more about my field:

- We are interested in catalysis, in particular, we look at energy technology, such as a fuel cell which requires catalysis to most efficiently convert chemical energy into electric energy.
- Computational chemistry models the catalysts, which you can think of as flat surfaces with molecules impinging from above, in finite cells with atoms at particular points. We cannot directly model the entire process, but with a lot of computational resources we can learn a few things about a few snapshots of these tiny cells and use theoretical tools to infer things about how the the macroscopic process should operate.
- Some example questions we may want to ask, given these modeling tool



So now we want to consider the real world, where there do exist databases that try to represent these types of models and their analyses.

We'll approach it from the perspective of a researcher in this field who has their own picture of the world and wants to migrate in from one of these sources to augment their data.

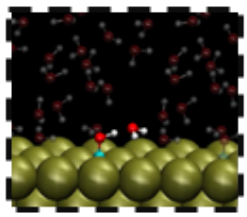


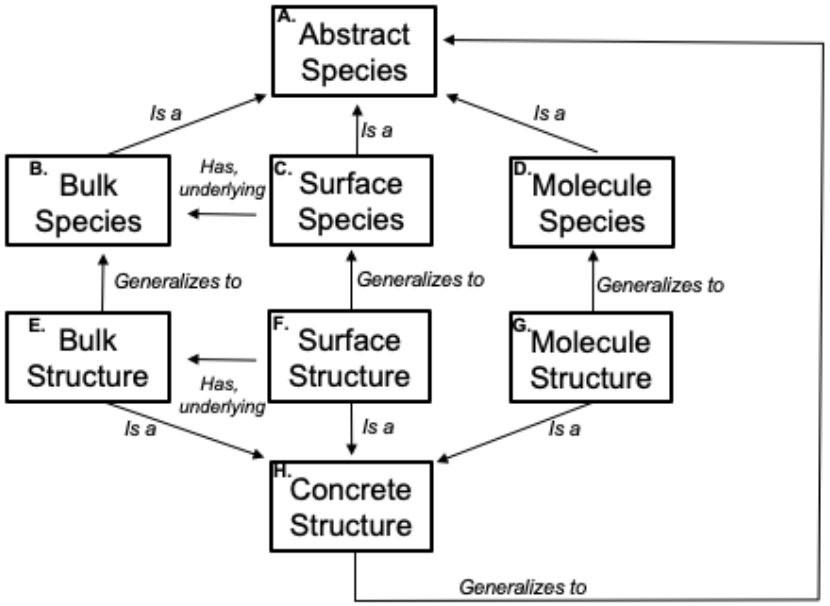
So suppose we want to model this domain. We could think of every one of our "simulation boxes" as a chemical species. And there are lots of things we can say about chemical species

However, there would be a lot of redundant, denormalized information here without distinguishing, say, the concept of water (H2O, boils at 100 oC, etc.) and a particular box with H1 at this coordinate, H2 and O at these other coordinates. This is actually a many-one relationship.

Likewise, we can recognize that some attributes are really only fitting for talking about surfaces, which are infinitely periodic in 2D, bulks, which are infinitely periodic in 3D, and molecules, which are not periodic at all.

Algebraic Database Schemas





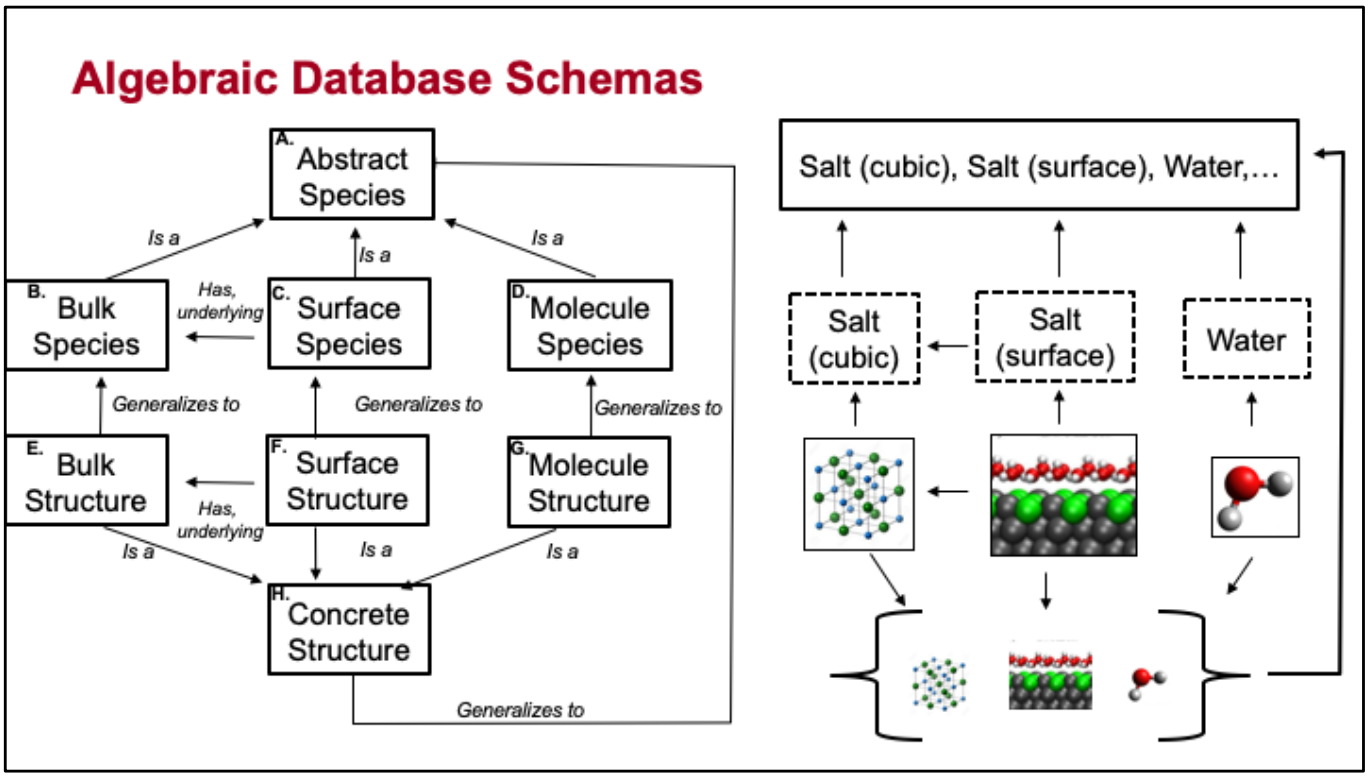
Data Integrity Constraints

So our conclusion of this analysis is that to talk about our simulation boxes, it would be better to have 6 categories.

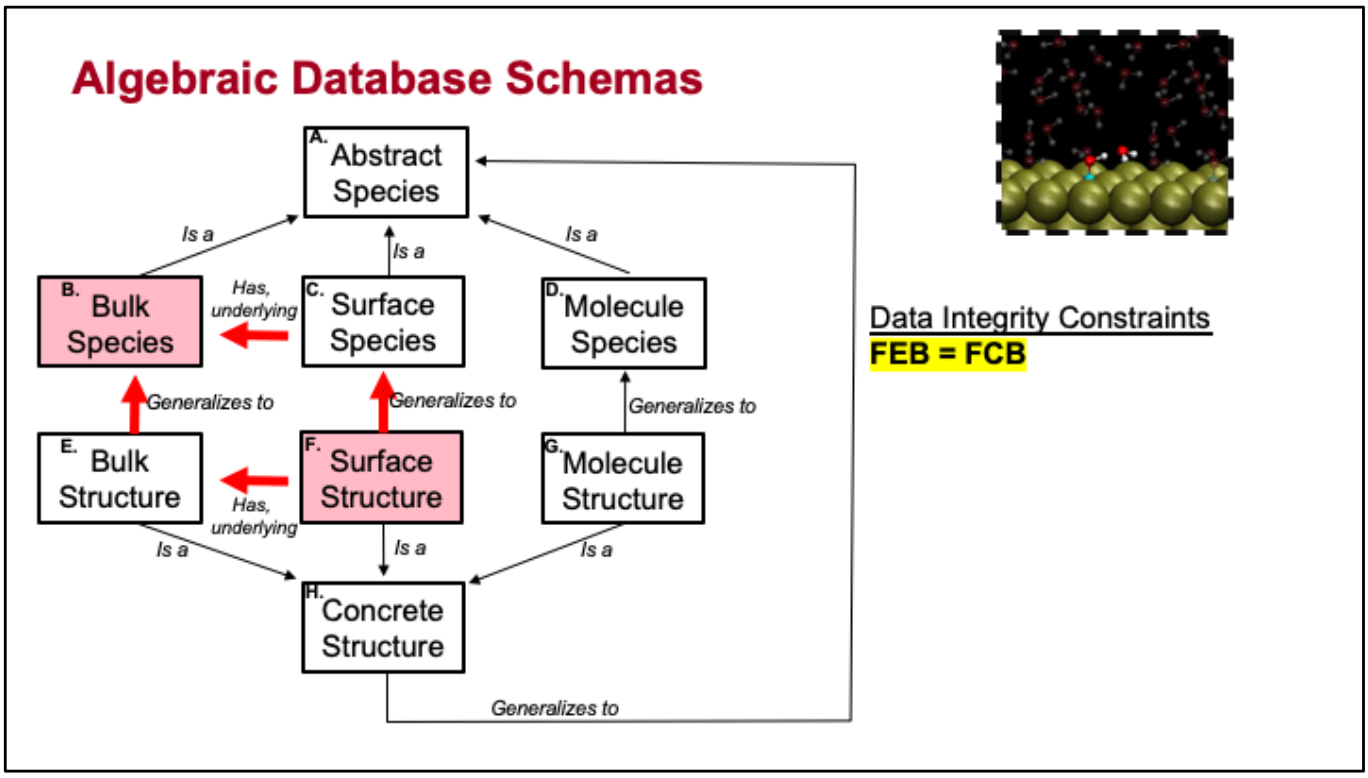
There are certain properties of species and structures independent of bulk/surface/molecule, so we can collect those here with a subset many-one relation.

And we can include the many one relations of generalizing from a concrete structure to an abstract species too.

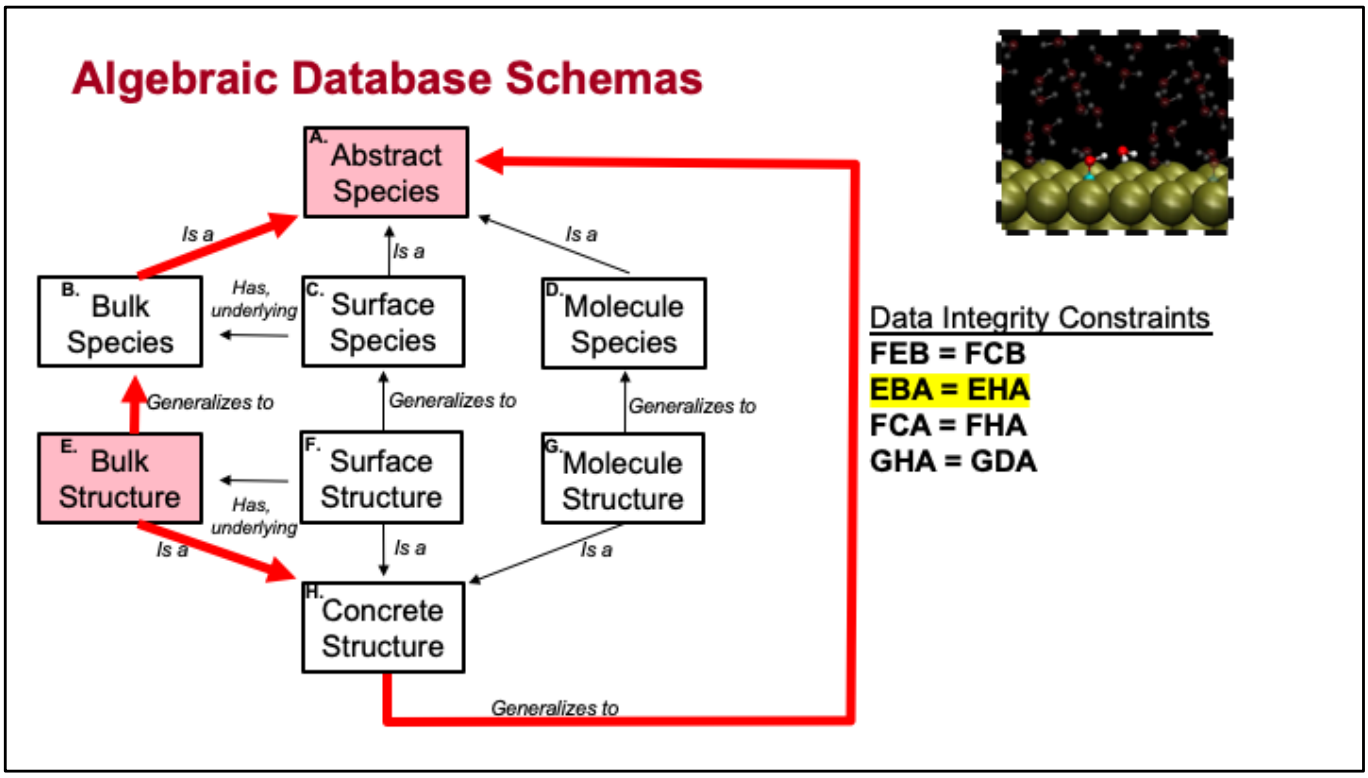
Up to this point, everything I said could have been about standard relational databases. When we view this as a schema for an algebraic database, nothing changes, yet we can add richer constraints to enforce the meaning of these objects and relations.



Here I try to make these tables more concrete by giving examples what records in each table look like.

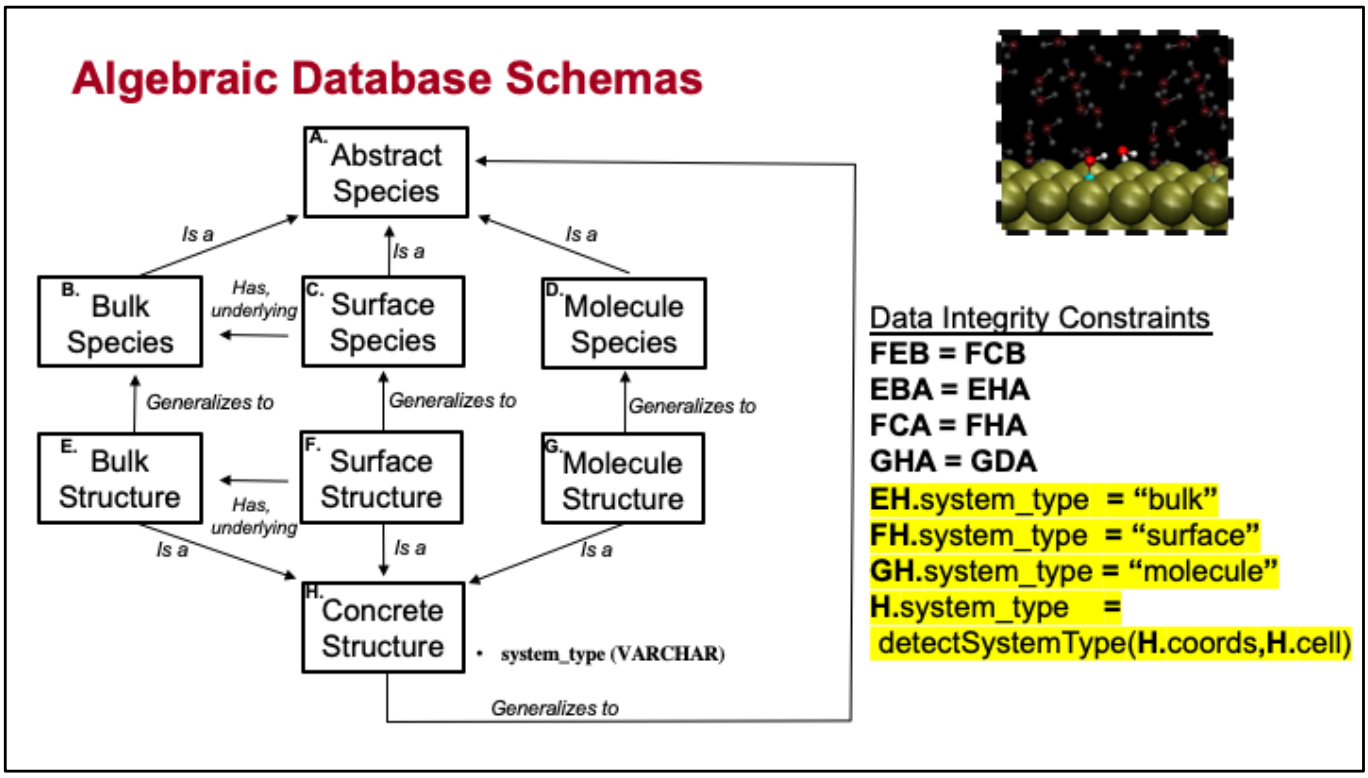


For example, there are many categories of surfaces which are identified by a number, e.g. 111 or 110, and there are many categories of bulk materials depending on how atoms are packed, e.g. FCC or HCP. Now, if we have a concrete Cu-fcc-111 surface structure, that generalizes to the concept of Cu-fcc-111, which has the concept of Cu-fcc underlying it. We want to enforce that we get the same result no matter which path we take.



Likewise we have three constraints which say it doesn't matter in which order you discard the information about whether you are bulk, etc. and the concrete structure information.

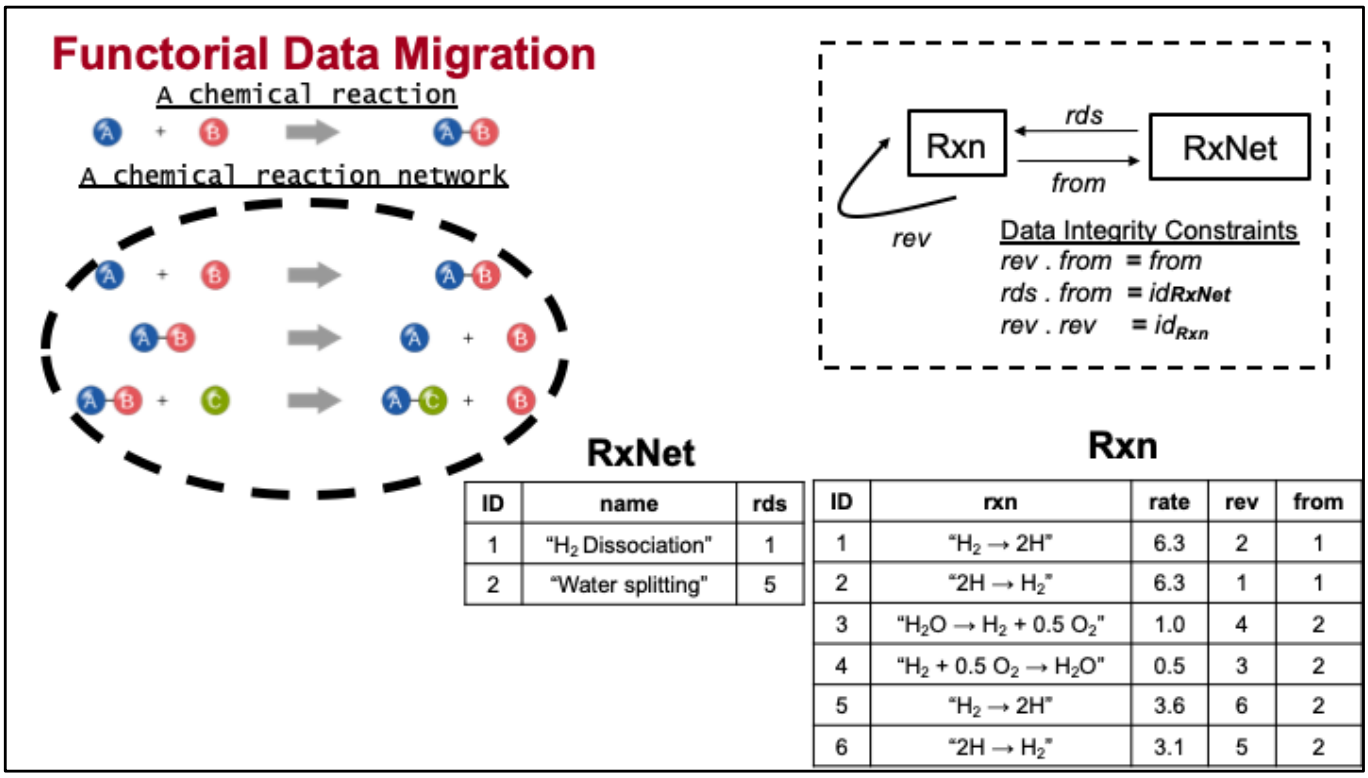
Just because I use the same string "generalizes to" or "Is a" does not tell the computer that these relationships 'mean' the same thing. That is something that I need to encode as path equalities if I want it to be machine-enforced.



An interesting feature is the ability to not simply equate paths but to also integrate arbitrary functions from a programming language.

Assuming you have some routine which looks at the simulation box and the positions of the atoms and classifies it as bulk/surface/molecule, you could enforce that the data in your schema is consistent with this routine.

Now the *meaning* of "Bulk" as a table title is machine-enforced, rather than just a string that relies on human interpretation to use correctly.



Now a new domain to model to ground discussion of the mapping between these algebraic databases:

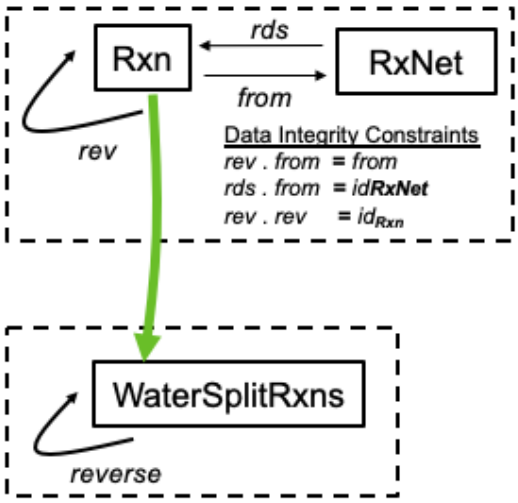
We learned in high school that chemical reactions are things like “A and B combine together to form AB at some quantifiable rate”

If we know the set of reactions that are possible, we can construct a reaction network and compute things like the equilibrium/final rate of each reaction. So there is a subset relation from reaction (within some network) to the network. To make the schema more interesting, note that every reaction can be thought of as having a reverse reaction (note the constraint that rev.rev is identity) and there is a distinguished reaction in the set which is a “rate determining step”.

The other two constraints say that the rds of a reaction network belongs to that reaction network, and the reverse of a reaction belongs to the same network.

Here’s some example data with two different reaction networks, noting that sometimes the same reaction can appear in multiple networks.

Functorial Data Migration



- Rxn ↦ WaterSplitRxns
- rev ↦ reverse
- rxn ↦ rxname

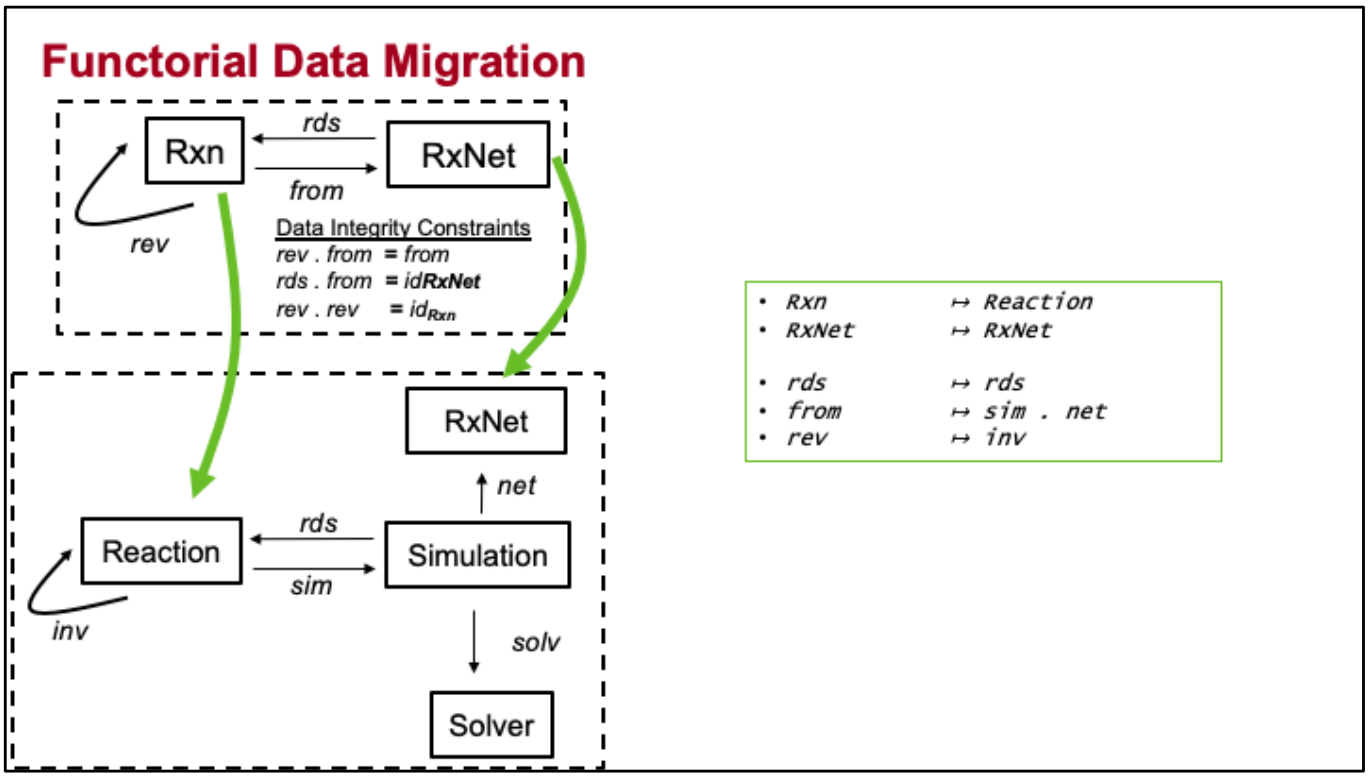
WaterSplitRxns

ID	rxname	rate	reverse
1	"$H_2O \rightarrow H_2 + 0.5\ O_2$"	1.0	2
2	"$H_2 + 0.5\ O_2 \rightarrow H_2O$"	0.5	1
3	"$H_2 \rightarrow 2H$"	3.6	4
4	"$2H \rightarrow H_2$"	3.1	1

Now, consider an alternate model of the same system. Before, we were parameterizing reactions by which network they came from, but consider a scientist who only considers a particular network, such as water splitting reactions. They might be interested in data from the first schema but want to import it into their simpler schema. Suppose this data gets transferred:

This is a simple example, but there are a lot of things that can go wrong in general during data migration. This resulting database is in principle a possible result by migrating with SQL and Python scripting, but we can tell there is something wrong with it, even though the target schema has no data integrity constraints. This is because we can interpret the reverse of reverse being identity from the source schema in the target schema (this is functorial data migration respecting constraints in source and target).

Conceptually, this is a really nice property to have for someone making their database public to outsiders without having to trust them as much to not accidentally misuse the data.



Now we can consider another limiting example, where our source schema is actually embedded into a larger schema, which not only parameterizes by reaction network but also by a particular numerical solver that computes properties of the reaction network.

Specifying this requires us to map the objects and arrows, and again there are many possible mistakes that could be made in the new schema that we can forbid merely by interpreting the constraints from the source schema.

Specifying relation between databases

1. Which tables / paths correspond to the same meaning?
   - What functions are needed to relate them?
2. What implicit attributes are present?
3. How may we want to filter the data records?
4. What is sufficient to determine equality for each object?

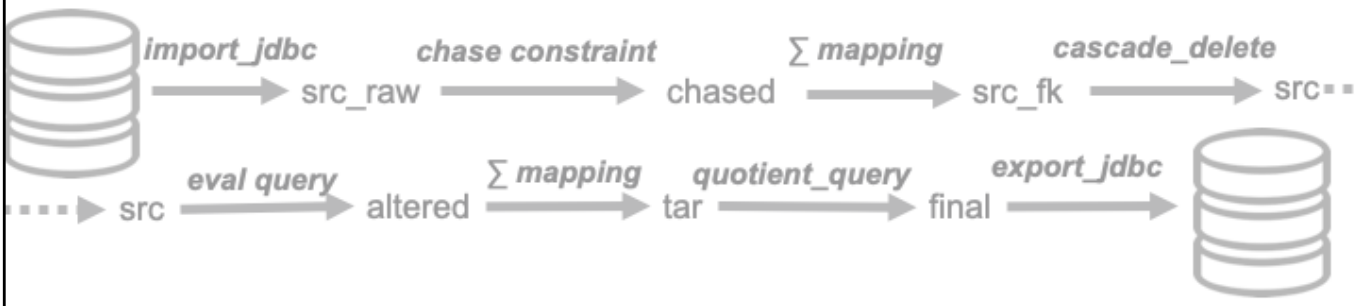
SUNCAT
CENTER FOR INTERFACE SCIENCE AND CATALYSIS

So in general, the interface I want to expose to a scientist (which is also sufficient for CQL to correctly migrate data) are these points here.

Example of 2: SUNCAT has an attribute to declare which simulation software is being used, yet OQMD is entirely generated from a single software (therefore that info is nowhere in the database…rather to be found on the webpage). Need a way of injecting information as well as synthesizing new information from old.

Note that this is a very declarative interface.

Migration process

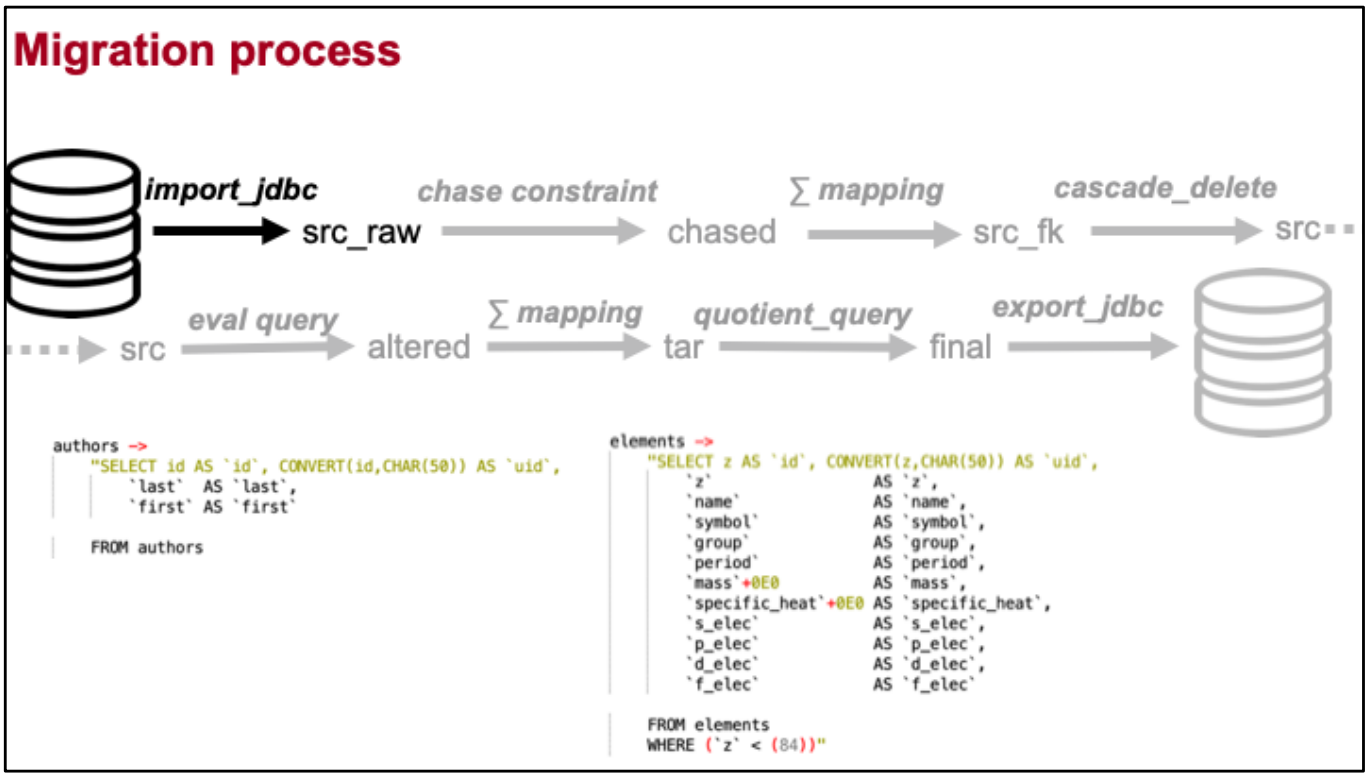


Real world data

- Violated primary key constraints
- NULL foreign keys
- Violated path equalities
- Duplicate records

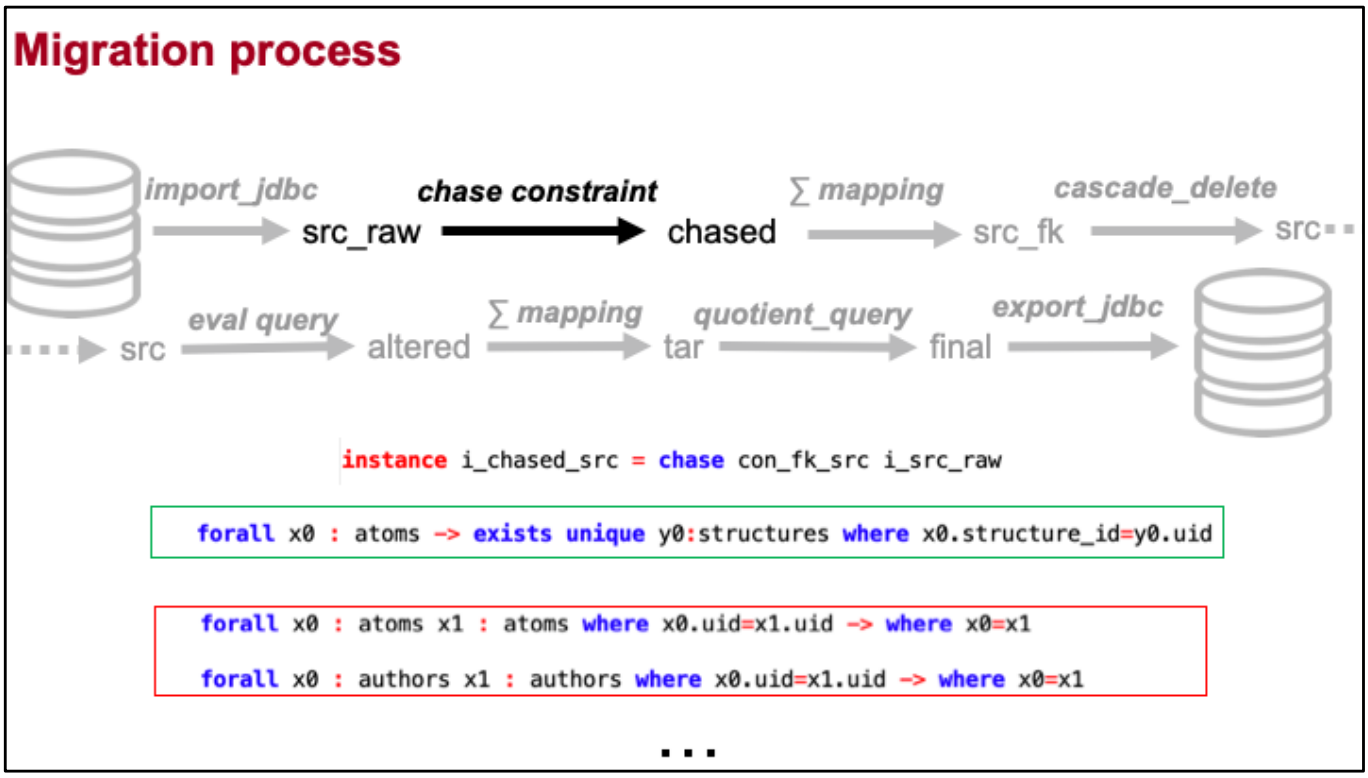
There are a lot of built in primitives in CQL that can do an entire migration in one step, but dealing with real, unclean data from the external world made this a multi-step process.

There are also building blocks that let me customize a process for my exact needs which, given how many decisions I had to make, are probably not optimal for everyone.



The first step involves interfacing with the external world to read in a MySQL database.
This requires writing a query for each object in the resulting schema
Note that CQL is particular about the datatypes and checks these upon landing the data. OQMD has some columns with inconsistent datatypes (e.g. **mass** here is coerced to always be float by adding a floating 0.0 to everything)

We're loading data into a pseudoschema that has attributes instead of foreign keys and primary keys. We want to be able to load all the data before cleaning it.

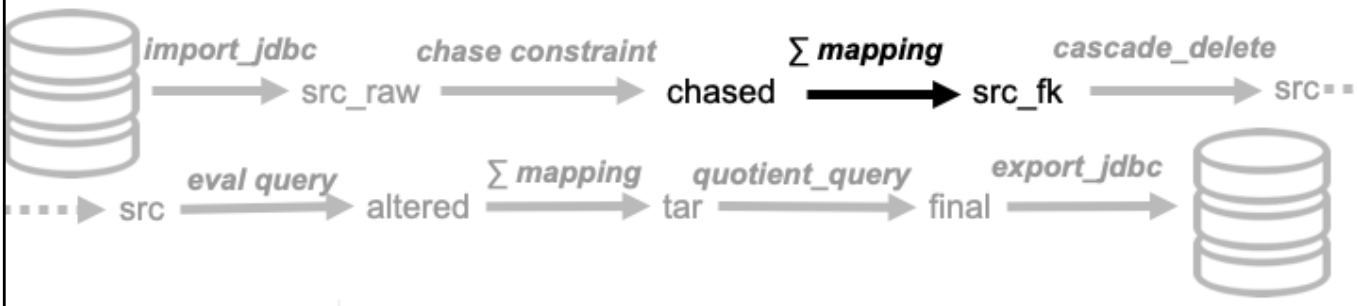


We then make this instance satisfy foreign key constraints, (by creating a new record where previously there was a null FK)

And primary key constraints. (by collapsing records with the same unique ID).

This is all stuff that is typically done under the hood in CQL (there is no need for a user to think about meaningless ID numbers) but in this special case I am working with IDs at a data level to repair the messy input data.

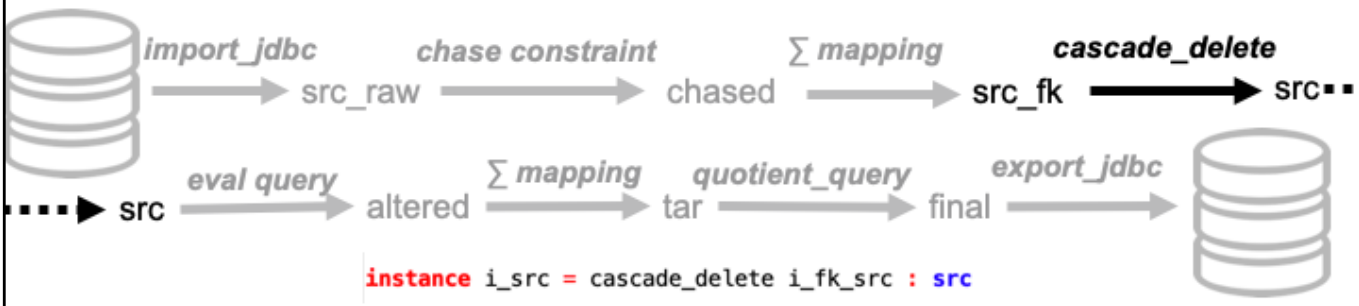
Migration process




```
instance i_fk_src = sigma M_fks_src i_chased_src


  entity
      structures     -> structures

  attributes
      entry_id       -> entry_id . uid
      spacegroup_id  -> spacegroup_id . uid
```

This next step brings us to a more normal CQL schema, where there are real foreign keys and we no longer explicitly carry around VARCHAR uid attributes.

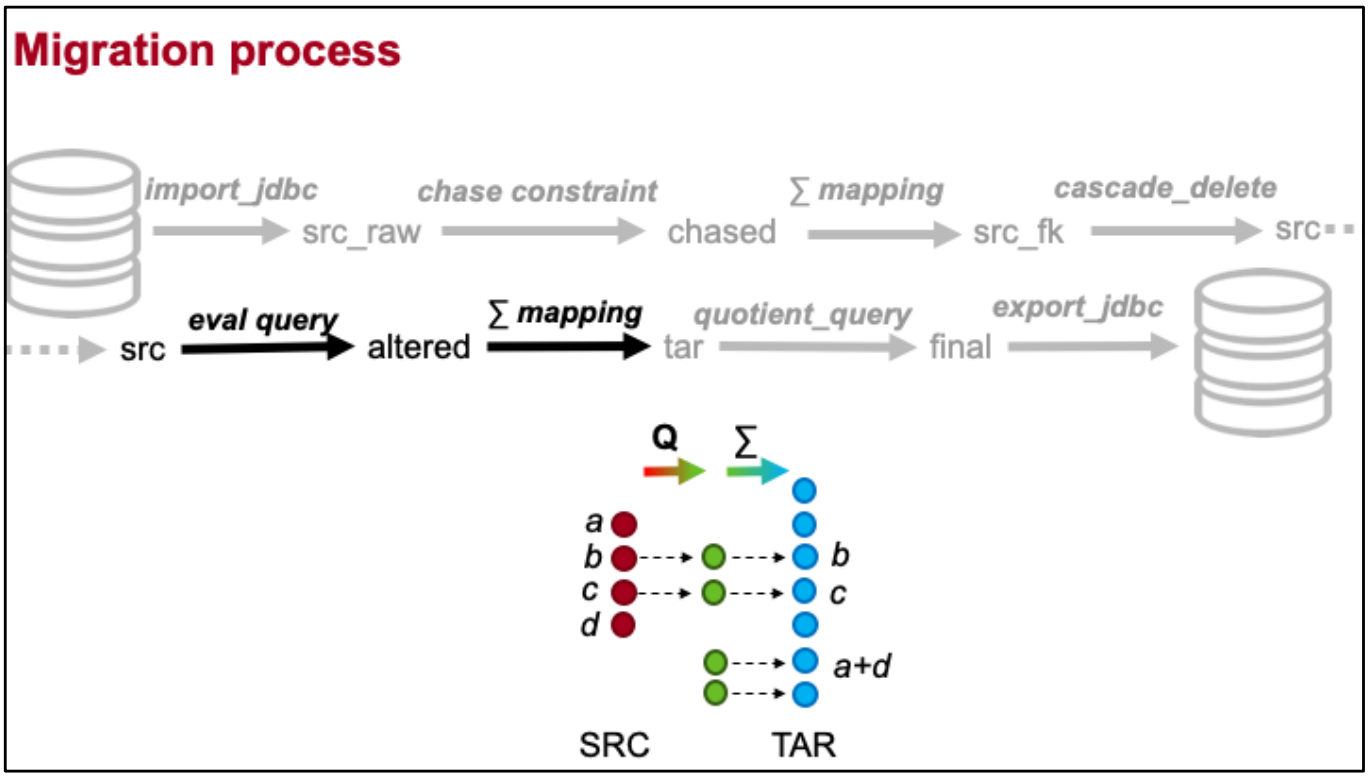
Migration process




```
instance i_src = cascade_delete i_fk_src : src
```

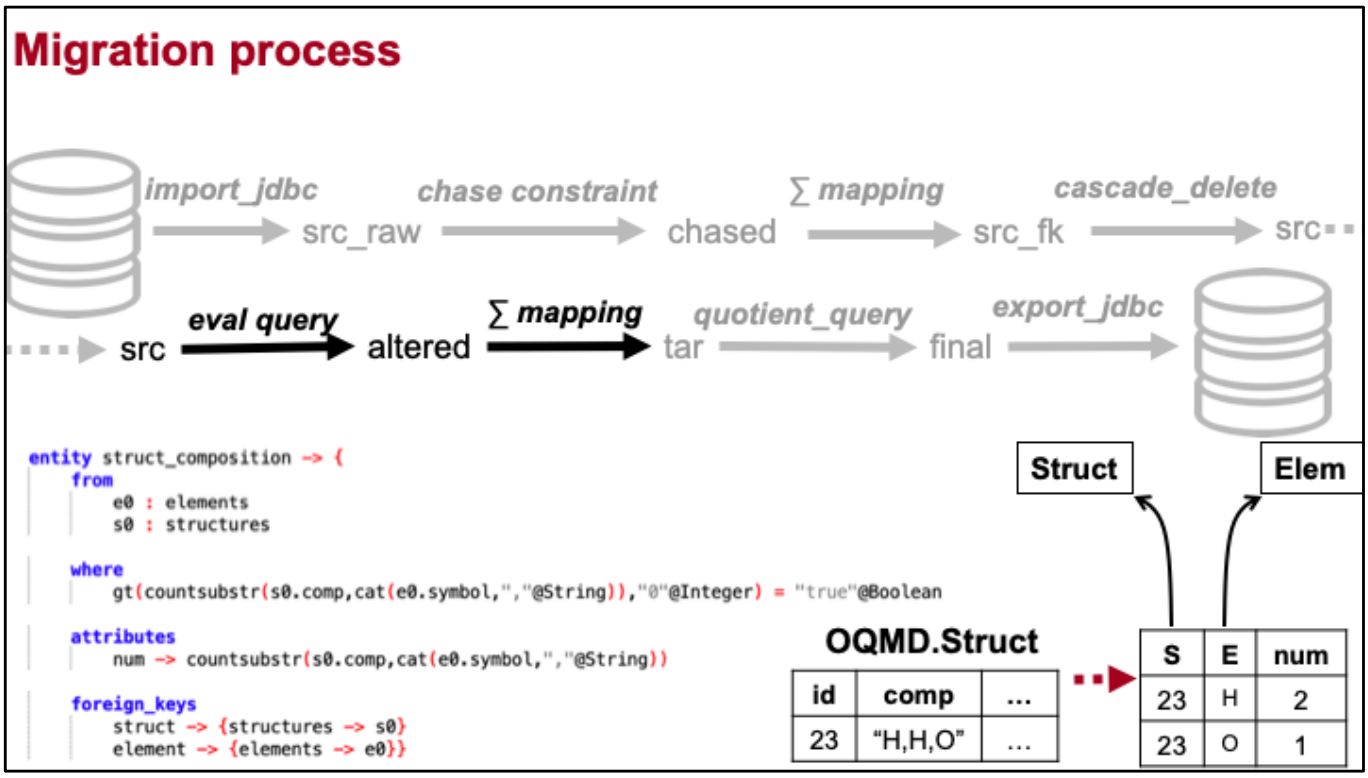
```
observation_equations
//------

    forall st0 : struct   . phase(st0.species)        = st0.system_type
    forall a0  : atom     . gteq(a0.x,"0.000"@Double) = "true"@Boolean
    forall m0  : molecule . system_type(m0.struct)    = "molecule"@String
    forall b0  : bulk     . system_type(b0.struct)    = "bulk"@String
    forall s0  : surface  . system_type(s0.struct)    = "surface"@String
```

The final step to getting a clean source database is to recursively delete records which do not abide by path equalities or observation equations that we expect the source to have.



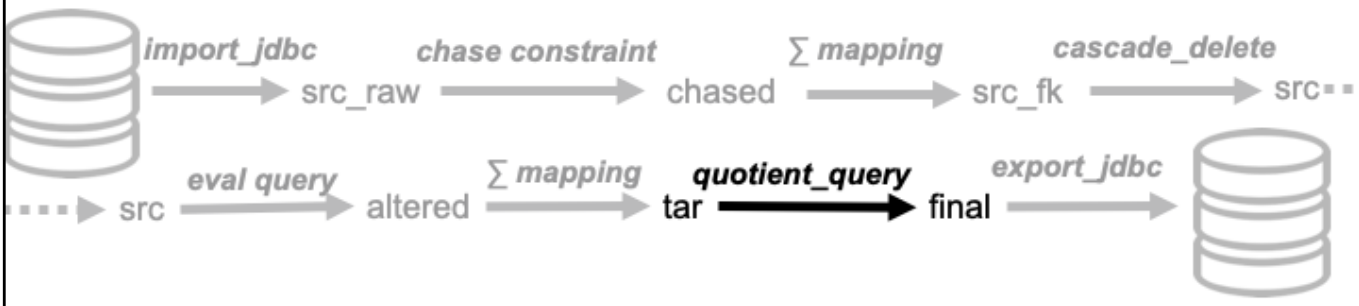
The CQL query operation has a SELECT-FROM-WHERE syntax – but it's a generalization of a SQL query, insofar as you can produce multiple, linked, tables at the same time.

In general, these two steps can be seen as the query disregarding information in the source which is not mappable to the target as well as deriving additional information which can be mapped to the target. This may not be enough to fully account for everything in the target, so it's really an intermediate schema that gets populated with only the objects and attributes we know how to map into the target. Then the mapping process is straightforward from that into the target schema.



In this example I create a brand new mapping table which says which elements (and how many) are in each chemical structure.

Migration process

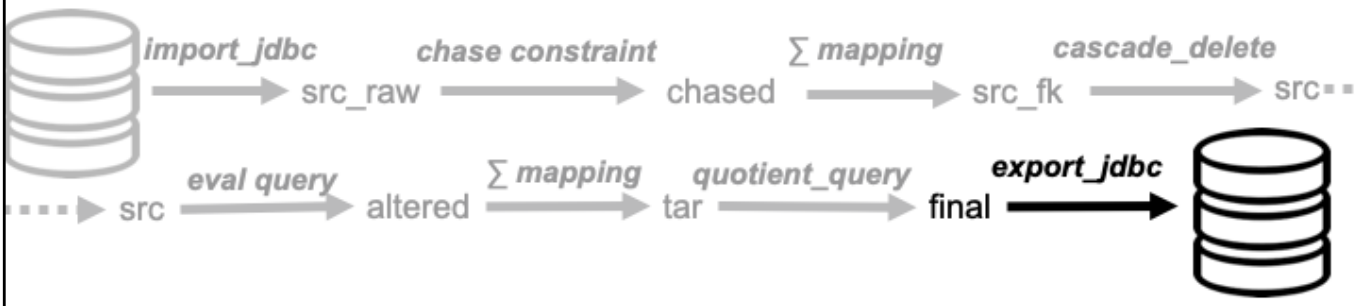



```
instance i_final = quotient_query i_merged {

    entity job -> {from a:job b:job where a.stordir=b.stordir}

    entity element -> {from a:element b:element where a.atomic_number=b.atomic_number}
```

The original messy source data may have unwanted duplicates, and some of the record generation in the query and in chasing the constraints is very conservative about freely generating records unless there is an explicit reason to collapse multiple records into one, so the final cleanup step is to quotient tables by the user-defined "identifying information"

E.g. Chemical elements differ from each other in hundreds of attributes, but what it MEANS for one to differ from another is to have a different atomic number.

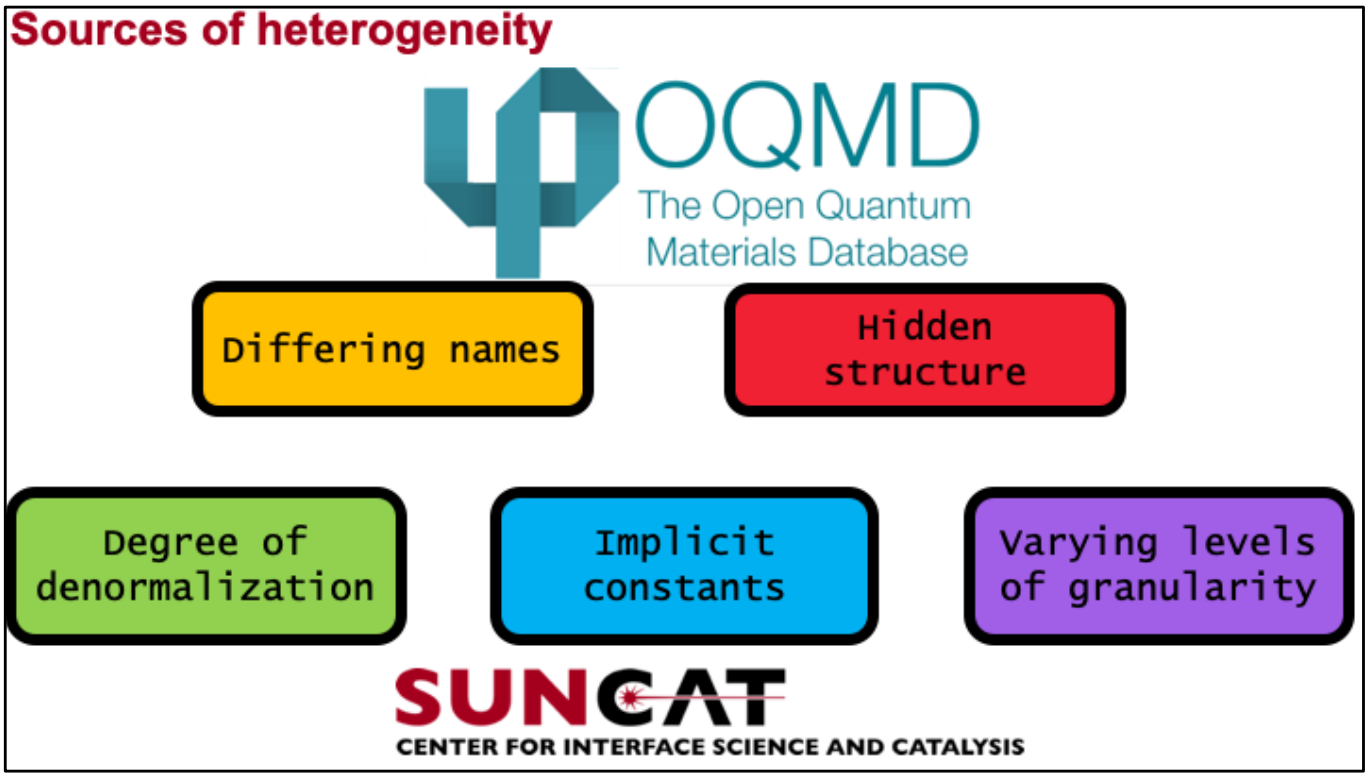
Migration process




```
command cmd_merged = export_jdbc_instance i_final "jdbc:mysql://127.0.0.1:3306/integrated?user=ksb&password=ksb" ""
```

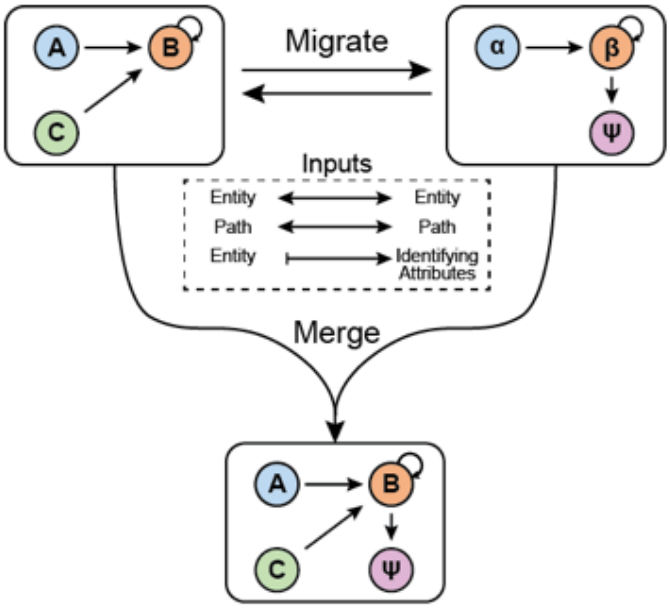
Lastly we have a simple command to reconnect everything that has been happening in the safe, algebraic database world back into the real world.

Note we lose information (e.g. labeled nulls become SQL nulls) in this process. But now we have a pipeline which can operate as a black box for someone working with MySQL databases + has no knowledge of CQL internals.



On a more concrete level, a large majority of specifying the migration process fell into one of these five categories, which I give details about in the paper. While there are some complicated things which don't fall into these categories, such as the derivation of the mapping table that existed in the target schema but not the source, mostly it was one of these things which had straightforward approaches to addressing in CQL.

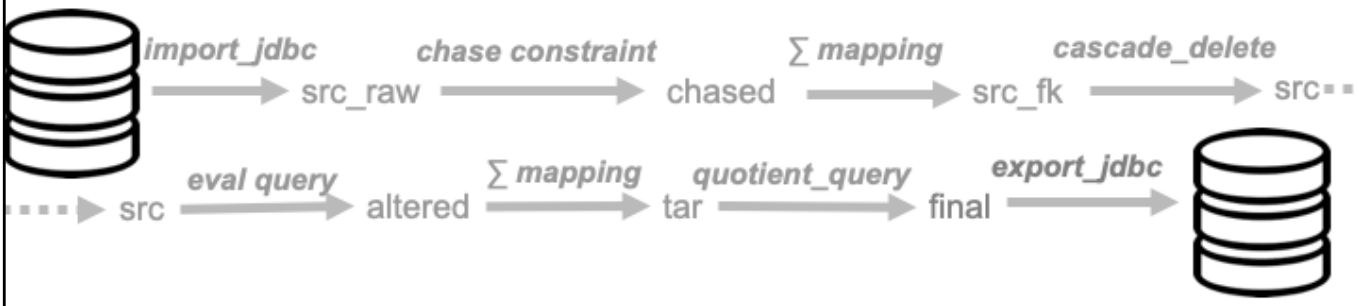
Resulting interface



So the net result, beyond the particular migration that was done, was really an interface that takes in these declarative inputs that try to be as intuitive as possible for a domain expert

And be able to leverage the theory underlying CQL to get for free a safe tool which can migrate and merge together multiple databases of interest.

Possible interface

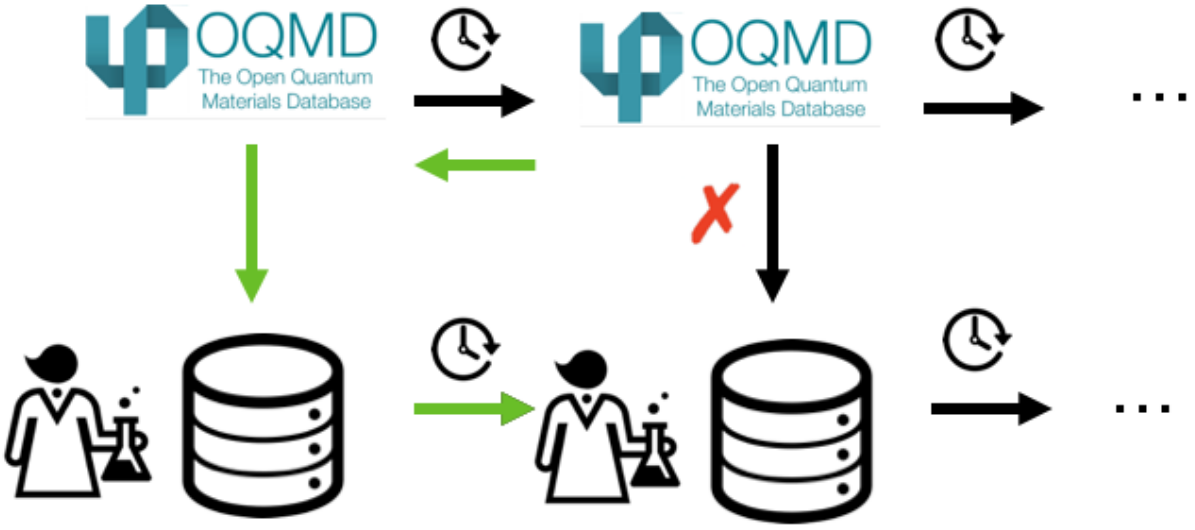



```
from cdi import Migrate

m  = Migrate(src = src, tar = tar, overlap = overlap, funcs = [count_words,Len,plus,matches,cat])
fi = m.file(src = isrc, tar = itar)
with open('cdi/library_example/lib.cql','w') as f:
    f.write(fi)
```

The overall CQL file for this process was thousands of lines long because there was so much detail to specify, but also there was a lot of redundancy that could be abstracted. Also making one change in the overall schema would require dozens of coordinated changes throughout the file, so mainly for refactoring I found it easy to work with a python wrapper that I wrote which has constructors corresponding to the basic CQL operations (like Migration, Schema, Query, Mapping, Java function, etc.). The generated file would be type checked by CQL and more or less always know if something went wrong in the code generation process.

This is the kind of thing I imagine would be needed for a scientist interfacing with CQL, but in general my experience was positive using it directly so I think this isn't necessary for, say, a software engineer.

Other use cases



As a final thought, I wanted to share a more immediate use case for this technology. The one I had been talking about naturally relies on people benefiting from an entire community agreeing to using algebraic databases, and that kind of systemic change is really not something that happens on its own or easily. For this other use case, consider that sometimes the person who designed that database that doesn't exactly match the one you want is in fact yourself, some time ago.

We can compose the migration we wrote to our old database with our new migration in order to get OQMD into the new database.

True story: OQMD changed substantially since the start and end of undertaking this project (luckily older versions are still available). While we could write a new migration from new OQMD to our new SUNCAT database, the least amount of work would be to migrate new OQMD to old OQMD and then compose all three migrations.

Categorical data integration for computational science (2019)

- Value of category-theoretic database tools for computational scientists
  - Efficient (compile-time) checking of constraint satisfaction
  - Data *sharers* can better convey meaning + restrict data misuse
  - Data *receivers* protect against incoming data not to specification.

- A strategy for the problem of data sharing
  - No “universal standard” needed
  - Data migration made rigorous/ergonomic
  - Composable migrations reduce amount of work needed