Supporting Information for

ESAMP: Event-Sourced Architecture for Materials Provenance management and application to accelerated materials discovery†

Michael J. Statt,[a,*] Brian A. Rohr,[a] Kris Brown,[a] Dan Guevarra,[c] Jens Hummelshoej,[b] Linda Hung,[b] Abraham Anapolsky,[b] John M. Gregoire,[c,*] and Santosh K. Suram[b,*]

1 Detailed Schema Discussion

In this section, we describe the entities and their corresponding relationships present in our schema (Figure (1)) in detail.

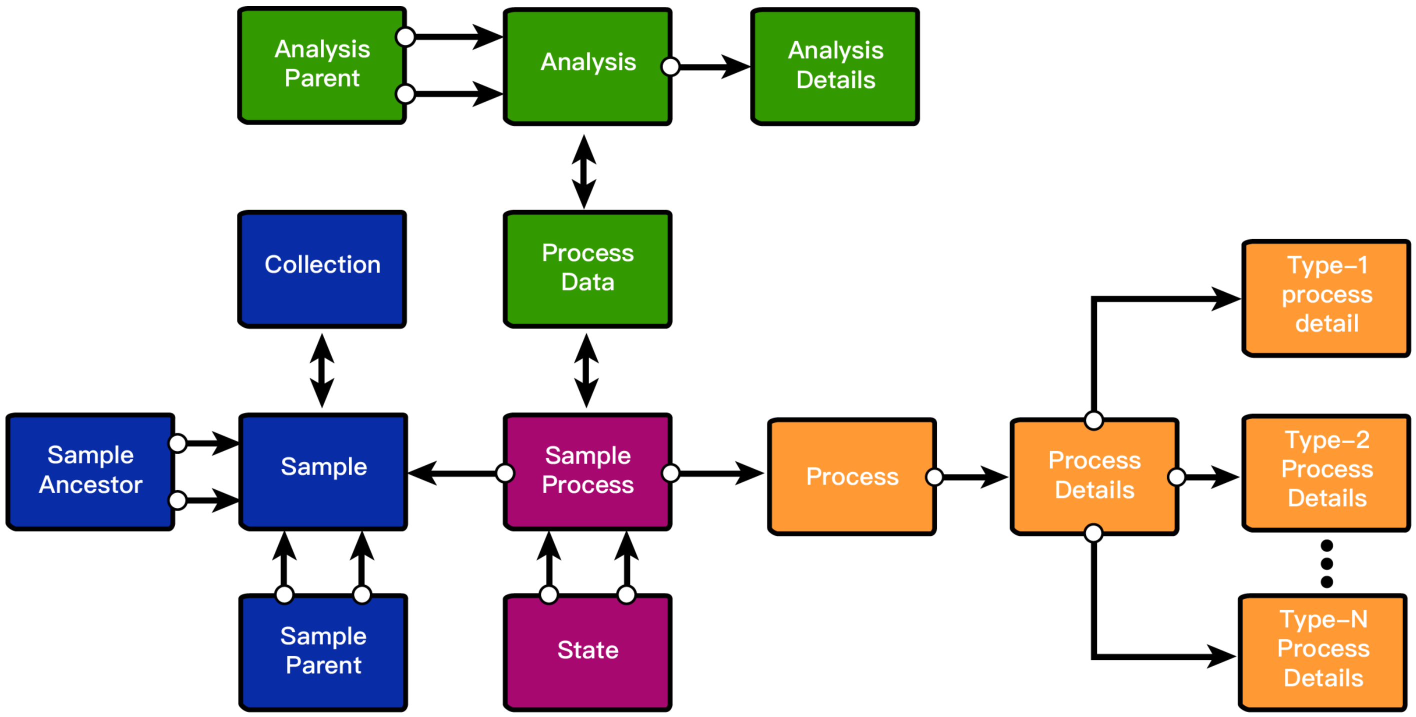

Fig. 1 Entity-relationship diagram for our database schema

1.1 The sample table

Each row in the sample table should correspond to exactly one object in the lab that a researcher prepares and makes measurements on. The concept of a sample is meant to be intuitive to a researcher; a lab that makes and tests solar cells will have one entry in this table per solar cell. The sample table has three columns:

- label (varchar): Oftentimes, samples already have labels or "IDs" in research laboratories. The intent of this column is to store that label.
- type (varchar): e.g. "anode" or "catalyst"
- details (jsonb): a json dictionary that can store any other attributes of the sample (e.g. "thickness" or "position_on_plate").

[a] *Modelyst LLC, Palo Alto, CA, 94303, United States*
[b] *Accelerated Materials Design and Discovery, Toyota Research Institute, Los Altos, CA, 94040, United States*
[c] *Division of Engineering and Applied Science, California Institute of Technology, Pasadena, CA 91125, United States*
* Corresponding authors: Michael J. Statt <michael.statt@modelyst.io>, John M. Gregoire <gregoire@caltech.edu>, Santosh K. Suram <santosh.suram@tri.global>

It is common for researchers to group samples. For example, in high throughput experiments, thousands of samples may exist on the same chip or plate, and in these cases, researchers need to be able to keep track of and make queries based on that information. We call these groups of samples "collections," which are described in more detail below.

In real scientific research settings, samples may be created or destroyed, so it is important for any schema to support that use case. For example, an anode and a cathode may be combined to create a battery, at which point, it may be most useful to the scientist to think of and label the battery as a new sample. This would result in the creation of a new sample (the battery) and the destruction of two other samples (the anode and the cathode). Similarly, a solution may be divided into ten aliquots, at which point, it may be most useful for the scientist to think of and label the ten aliquots as new samples. This would result in the creation of ten samples and the destruction of one sample. As samples are combined to yield new ones, it is important to keep track of the full lineage of each sample. Otherwise, it would be impossible to answer questions like, "which anodes gave the best battery performance?" In this schema, that information is tracked in the parent table and in the ancestor table, which are discussed below.

1.2 The collection table and the sample_collection table

In the schema presented in this work, a collection is a user-defined group of samples.

It is clear from the previously-mentioned example that many samples can (and almost always do) belong to one collection. It is also important that we allow for the same sample to exist in many collections. For example, a researcher may want to group samples by which plate or wafer they are on, which high-level project they are a part of, and which account they should be billed to all at the same time.

In addition to the common columns, the collection table has three columns:

- type (varchar): a short string that specifies the type of the collection (e.g. "plate," "project", or "account")
- name (varchar): the name of the collection (e.g. plate "1000," project "oxygen reduction catalysis," or account "Toyota")
- details (jsonb): a dictionary that can store any other necessary information about the collection. For example, if the collection is type "plate," then "thickness" may be a property that is stored in the details dictionary.

Since many samples can belong to one collection, and one sample can belong to many collections, there is a many-to-many relation between samples and collections. In order to represent this many-to-many relation, we must add a mapping table between the collection table and the sample table. We call this mapping table sample_collection, and its columns are simply:

- sample_id (integer): a foreign key to the sample table
- collection_id (integer): a foreign key to the collection table

For example, if a sample with ID 1 belongs to collection with ID 2, then the row (1,2) would be inserted into the sample_collection table.

1.3 The parent table

Whenever one or many samples undergo a process and new samples are created as a result of that process, we call the sample(s) that went into the process "parents" of the sample(s) that resulted from the process. The role of the parent table is to keep track of these parental relationships and thereby enable us to use queries to answer questions like, "which anode and cathode were combined to create this battery?"

The parent table has two columns:

- parent_sample_id (integer): a foreign key to the sample table specifying the ID of the parent sample
- child_sample_id (integer): a foreign key to the sample table specifying the ID of the child sample

For example, if a sample corresponding to a cathode has ID 1, and a sample corresponding to an anode has ID 2, and the two are combined to yield a battery with sample ID 3, then the following two rows would be inserted into the parent table:

parent_sample_id	child_sample_id
1	3
2	3

parent_sample_id	child_sample_id
1	3
2	3
3	5
4	5

1.4 The ancestor table

The role of the ancestor table is simply to make it easier to query for the full history of any sample, including any of its parents, parents' parents, etc. Similar to the parent table, the ancestor table has the following columns:

- ancestor_sample_id (integer): a foreign key to the sample table specifying the ID of the ancestor sample
- child_sample_id (integer): a foreign key to the sample table specifying the ID of the child sample

Let's walk through an example that illustrates the utility of the ancestor table. A substrate with sample ID 1 and a metal oxide powder with sample ID 2 are combined to yield a battery anode, which is assigned sample ID 3. Then, that anode and a cathode with sample ID 4 are combined to make a battery with sample ID 5. The following rows would be inserted into the parent and ancestor tables:

In this example, samples 1 and 2 are not parents of sample 5, but they are ancestors of sample 5. The ancestor table does not store any unique information; it can be calculated from the parent table alone. However, it does make queries much simpler. For example, with the ancestor table, the following question can be answered with a SQL query, "what is the full set of samples that were used to create the battery with sample ID 5?" The query is simply, "select child_sample_id from ancestor where ancestor_sample_id = 5." Without the ancestor table, to answer the same question, one would need to write a recursive loop checking to see if each parent had more parents.

At first glance, it may seem that the ancestor table contains all of the information in the parent table and more, which would obviate the need for the parent table; however, this is not the case. In the example above, if only the ancestor table existed, it would be impossible to tell which samples were combined to yield sample 5. In other words, the fact that samples 1, 2, 3, and 4 were all used in the preparation of sample 5 would be known, but information regarding which of samples 1-4 were combined with which, and in what order, would be lost. So, the parent table is needed in order to preserve the exact lineage of each sample, and the ancestor table is needed to simplify queries regarding a sample's history.

Additionally, the ancestor table stores the rank of the ancestor relationship. This is an integer that denotes the number of generations between the descendant and the ancestor. A value of 0 indicates a parent-child relationship, while a value of 1 indicates a grandparent-grandchild relationship. This is helpful when evaluating the degree of connectivity between an ancestor and child relationship, such as constraining queries on a sample's provenance to only include those samples which meet a certain degree of ancestry. For example, the question "Select the samples that are at most two generations away from a given sample" is easy to query using the ancestor table.

1.5 The process table

Each row in the process table represents one experimental procedure (e.g. a synthesis or characterization step) that is applied to a sample. For example, if an XPS spectrum is measured on a sample, that results in the addition of one row in the process table. It is important to understand that, in this schema, if the same type of process is performed twice, that

ancestor_sample_id	child_sample_id
1	3
2	3
3	5
4	5
1	5
2	5

results in the addition of two rows in the process table. For example, if an XPS spectrum is measured on the same sample twice, that yields two rows in the process table. The process table has four columns:

- timestamp (timestamp): the date and time at which the process was run
- category (varchar): the broad category of the process (e.g. "electrochemistry," "spectroscopy"
- name (varchar): the specific name of the process (e.g. "chronoamperometry", "XPS")
- ordering (integer): the order in which processes with the same timestamp were run. Ideally, such processes would have different timestamps; however, in practice, sometimes, high throughput experimental equipment runs a series of sequential, automated processes and records the time at which the user pressed "run" as the timestamp for all of the processes.
- process_detail_id (integer): a foreign key to the process_detail table

1.6 The `sample_process` table

There is a many-to-many relation between samples and processes, and the `sample_process` table is the mapping table that captures this relation. Some processes involve multiple samples (e.g. combining a cathode and an anode to create a battery), and an individual sample can (and almost always does) undergo multiple processes (e.g. one or many synthesis steps followed by one or many characterization techniques). This necessitates the many-to-many relation. Simply put, every time a sample undergoes a process, there a row is added to the `sample_process` table. This concept of a sample-process pair is important because data can be generated when samples undergo processes. How this table relates to the collection of raw data is described in the discussion of the measurement_group table below. The `sample_process` table has two columns:

- sample_id (integer): a foreign key to the sample table
- process_id (integer): a foreign key to the process table

This structure makes all of the following questions easy to answer with SQL queries: 1) How many processes have been run on sample x? 2) What is the full process history of sample x? 3) What is the full process history of sample x and any samples that went into the preparation of sample x? The queries that answer these questions are available in the next section.

1.7 The state table

Here, we define the concept of a sample's state, which is useful for some research applications. A researcher may want to declare that certain types of processes are "state-changing," meaning they substantially alter the sample. Whether or not a given type of process is "state-changing" is subjective and purely left to the discretion of the researcher who measures the data. As an example, a researcher may wish to indicate that when a catalyst sample's state is changed when it undergoes an electrochemistry experiment because electrochemistry experiments can alter the composition of the catalyst. However, if a sample is simply weighed and put back on the shelf, its properties are largely the same before and after that process, and it therefore may be most useful for the researcher to classify the weighing process as non-state-changing. The state table has three columns:

- start_`sample_process`_id (integer): a foreign key to the `sample_process` table specifying the `sample_process` event that created the state
- end_`sample_process`_id (integer): a foreign key to the `sample_process` table specifying the `sample_process` event that ended the state
- duration: a float specifying the amount of time that the state existed, or in other words, the amount of time that the sample was in that state.

This is useful for the construction of datasets for machine learning purposes. For example, let's consider the scenario where a research group wants to use machine learning to predict photocatalyst band gap from composition. The dataset could be created by querying for samples where both the composition was measured and the band gap was measured. However, if state-changing processes occurred between the time of the composition measurement and the band gap measurement, that simple query would include rows with data that would be deleterious to the training of the machine learning model. Instead, the researchers could use the state table to query for samples where both the composition and band gap was measured while the sample was in the same state. This would yield a clean dataset.

1.8 The process_detail table

The purpose of this table is to capture the metadata for each process. For example, a given research team almost invariably runs many different types of processes (possibly XPS experiments, electrochemistry experiments, deposition processes, etc.), and it is obviously important to keep track of which type of process is being run. Additionally, the same type of experiment can be run with different parameters. For example, a cyclic voltammogram can be collected at a different sweep rates, voltage range, etc. This metadata must also be captured. There are many ways to adequately capture this information. Three possible design decisions are discussed here: 1) using a "triple key store" table, 2) using a single, wide "dimension" table, or 3) many process-specific dimension tables with fewer columns.

The key-value table would have the following columns:

- process_id (integer): a foreign key to the process table.
- key (varchar): a short string that identifies the meaning of the value in the "value" column (e.g. "sweep rate", or "experiment type").
- value (varchar): a string containing the name of the file. The string may be converted to a different data type using the information in the last column.
- data_type (varchar): A string that specifies the data type of the data in the "value" column

This choice of architecture has several advantages. Most importantly, this design choice allows for new keys to be added without adding any columns to the database. For example, if a research group starts to do a new type of experiment or capture additional metadata, new rows with new, unique keys need to be added to this table, but not new columns need to be added. Additionally, no null values being added to the database.

However, in certain cases, the key-value architecture has some disadvantages as well. First, it is clumsy to query if a user is frequently interested in looking up the value for many of the keys at once. For example, let's analyze the case where a user wants to answer the question, "for process ID 1, what was the minimum voltage, maximum voltage, sweep rate, and solution pH?" The SQL query to answer this question would involve four subqueries in the select clause or four joins to different, aliased copies of the process_detail table, both of which are quite clumsy.

Furthermore, the case where a large number of key-value pairs of metadata are captured for each process, and a given process is frequently run with the exact same set of parameters and therefore the exact same set of key-value pairs, this architecture results in the addition of a large number of rows to the process_detail table. Let's analyze a concrete example to make this point clear. If a high-throughput research group runs cyclic a voltammogram experiment with the exact same set of 10 parameters (e.g. sweep rate, minimum voltage, etc.) on 100,000 samples, this would result in the addition of 1,000,000 rows to the process_detail table.

Using the dimension table architecture design choice, the process_detail table instead has a column for each type of metadata that is captured, and the process table has a foreign key to the process_detail table. In other words, if one were to switch from the key-value architecture to the dimension architecture, there would be a column for each distinct key in the key-value table. For example, for a group that does XPS and cyclic voltammogram experiments, the columns may include:

- experiment_type (varchar): a string like 'XPS' or 'CV' that specifies the type of experiment
- sweep_rate (float): a short string that identifies the meaning of the value in the "value" column (e.g. "sweep rate", or "experiment type").
- Xray_intensity (float): a string containing the name of the file. The string may be converted to a different data type using the information in the last column.

This design choice has the opposite set of advantages and disadvantages. The disadvantages are the following. Whereas the key-value architecture allows for the capture of new types of metadata only by adding new rows, the dimension table architecture requires the addition of a new column for every new type of metadata that is captured. Additionally, whereas the key-value architecture never results in null values, the dimension table architecture results in many null values and a much wider table.

The advantages of the dimension table architecture are the following. Answering the previously mentioned question, "for process ID 1, what was the minimum voltage, maximum voltage, sweep rate, and solution pH," with a SQL query becomes very easy. There is one join in the from clause, and the desired information can be readily selected.

Additionally, let's revisit the case above in which a given process is run with the same set of 10 parameters on 100,000 samples. Using the dimension table architecture, this would result in the addition of 1 row to the process_detail table instead of the 1,000,000 for the key-value architecture. Each of the 100,000 entries in the process table would have a foreign key to this one row in the process_detail table.

Finally, a third option is to store a thin, dense dimension table for each type of process. This design choice is very similar to the previously described "one wide dimension table" option but with the following key differences. The main benefit of the this design choice is that the number of nulls is greatly reduced. The main drawback is that an additional join is introduced into most queries regarding process details. A visualization of these three options is shown in Figure 2.

The design choice that was chosen in this work was to use multiple, thin, process-specific dimension tables because there are many types of processes, each with a large number of metadata columns, and running processes with the same set of metadata parameters is a common use case.

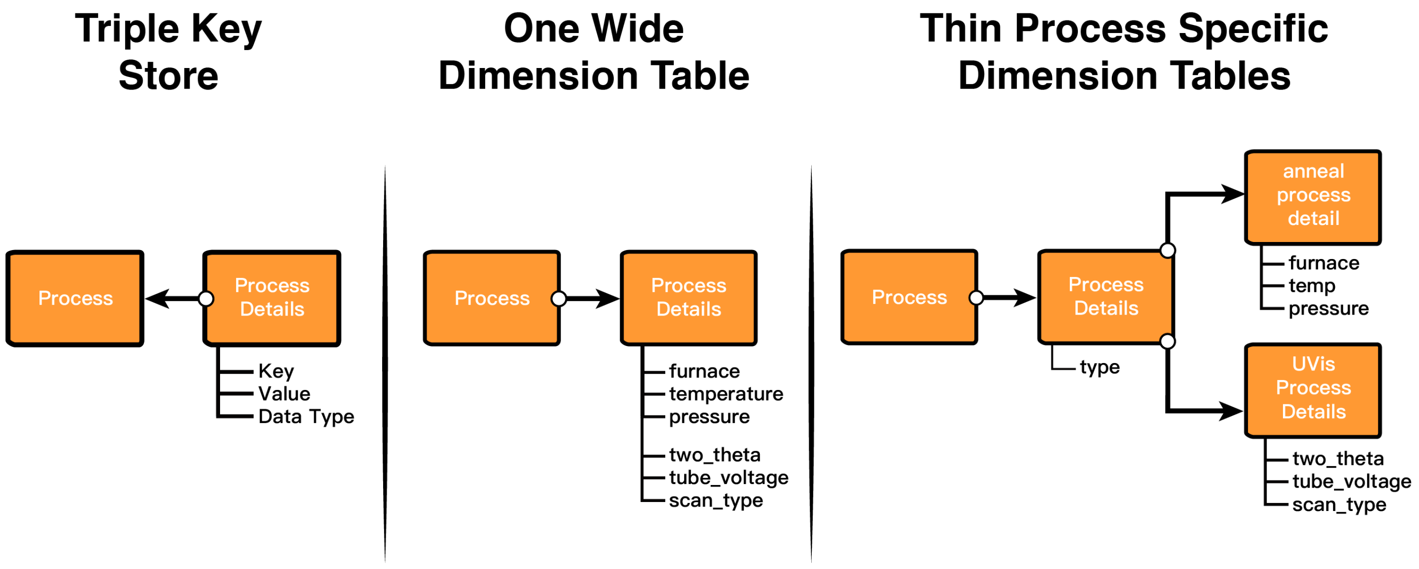

Fig. 2 An illustration of the three strategies for storing the the details of a process.

1.9 The `process_data` and the `sample_process_process_data` table

Often, when samples undergo processes, this results in the creation of one or many raw data files. These raw data files are tracked in the `process_data` table.

In the simplest case, one sample undergoes one process, and data is recorded. However, in high-throughput experimentation, more complicated cases often arise, and it is imperative that the schema can handle these use cases. Specifically, in high-throughput experiments, it is common for equipment to analyze an entire plate or chip full of samples simultaneously or in rapid succession. In these cases, some of the raw data that is captured pertains to all of the samples (e.g. the temperature of the plate), and some of the data applies only to individual samples (e.g. the XPS spectrum for each sample). In the general case, an individual process can produce many raw data files, each of which can pertain to any subset of the samples that underwent that process. For that reason, there is a many-to-many relation between sample-processes and raw data files.

The `process_data` table has the following columns:

- raw_data_json: a json string containing the raw data that the experiment recorded
- file_path: a string containing the full path to the file

The `sample_process_process_data` table is simply a mapping table to allow for the many-to-may relationship between sample-processes and process data files. It has the following columns:

- process_data_id: a foreign key to the `process_data` table.
- sample_process_id: a foreign key to the `sample_process` table.

1.10 The analysis, analysis_detail, and analysis_parent tables

Almost always, data that is captured in the lab is processed to arrive at higher level results, and the purpose of the analysis table is to store these higher level results. For example, if a current-voltage curve is measured for a solar cell, it may be useful to send that data through a function to identify the open circuit voltage.

The `analysis` table has the following columns:

- analysis_name: a short string identifying the user-defined name of the function.
- input: a jsonb dictionary containing the inputs to the function
- output: a jsonb dictionary containing the outputs of the function
- analysis_detail_id: an integer foreign key to the analaysis_detail table, which is described below

There is a many-to-many relation between the the `analysis` table and the `process_data` table because it is possible for many pieces of process data to be used in the same analysis, and it is possible for the same piece of process data to be used in many analyses. There is a mapping table called `process_data_analysis` to allow for this many-to-many relation.

Sometimes, the same function can be run with different parameters, and the purpose of the `analysis_detail` table is to store the parameters that were used when the analysis function was run. For example, a smoothing function may be run with different window sizes. The `analysis_detail` table has the following columns:

- analysis_name: a short string identifying the user-defined name of the function.
- details: a jsonb dictionary containing key-value pairs that capture the choice of parameters used when running the analysis.

In the most complex case, an analysis function may use the output of a different analysis as an input. For example, when calculating the fill factor for a solar cell, the open circuit voltage and the short circuit current density must already be determined. This parent-child relationship between analyses is conceptually similar to the parent-child relationship between samples. The `analysis_parent` table captures this information. It has the following columns:

- parent: an integer foreign key to the analysis table containing the ID of the parent analysis
- child: an integer foreign key to the analysis table containing the ID of the child analysis

2 Queries

2.1 How many processes have been run on sample x?

```
select
    count(*)
from
    sample
join sample_process on
    sample_process.sample_id = sample.id
where
    sample.label = 'x'
```

2.2 What is the full process history of sample x?

```
select
    process.timestamp,
    process_detail.type,
    process_detail.technique
from
    sample
join sample_process on
    sample_process.sample_id = sample.id
join process on
    sample_process.process_id = process.id
join process_detail on
    process.process_detail_id = process_detail.id
where
    sample.label = 'x'
```

2.3 What is the full process history of sample x and any samples that went into the preparation of sample x?

```
select
    sample.label,
    process.timestamp,
    process_detail.type,
    process_detail.technique
from
    sample
join sample_process on
    sample_process.sample_id = sample.id
join process on
    sample_process.process_id = process.id
join process_detail on
    process.process_detail_id = process_detail.id
where
    sample.label = 'x'
    or sample.id in (
    select
        ancestor.ancestor
    from
        ancestor
    join sample on
        ancestor.child = sample.id
    where
        sample.label = 'x')
```

The complete database schema for our Materials Provenance Store database is shown in Figure 3.

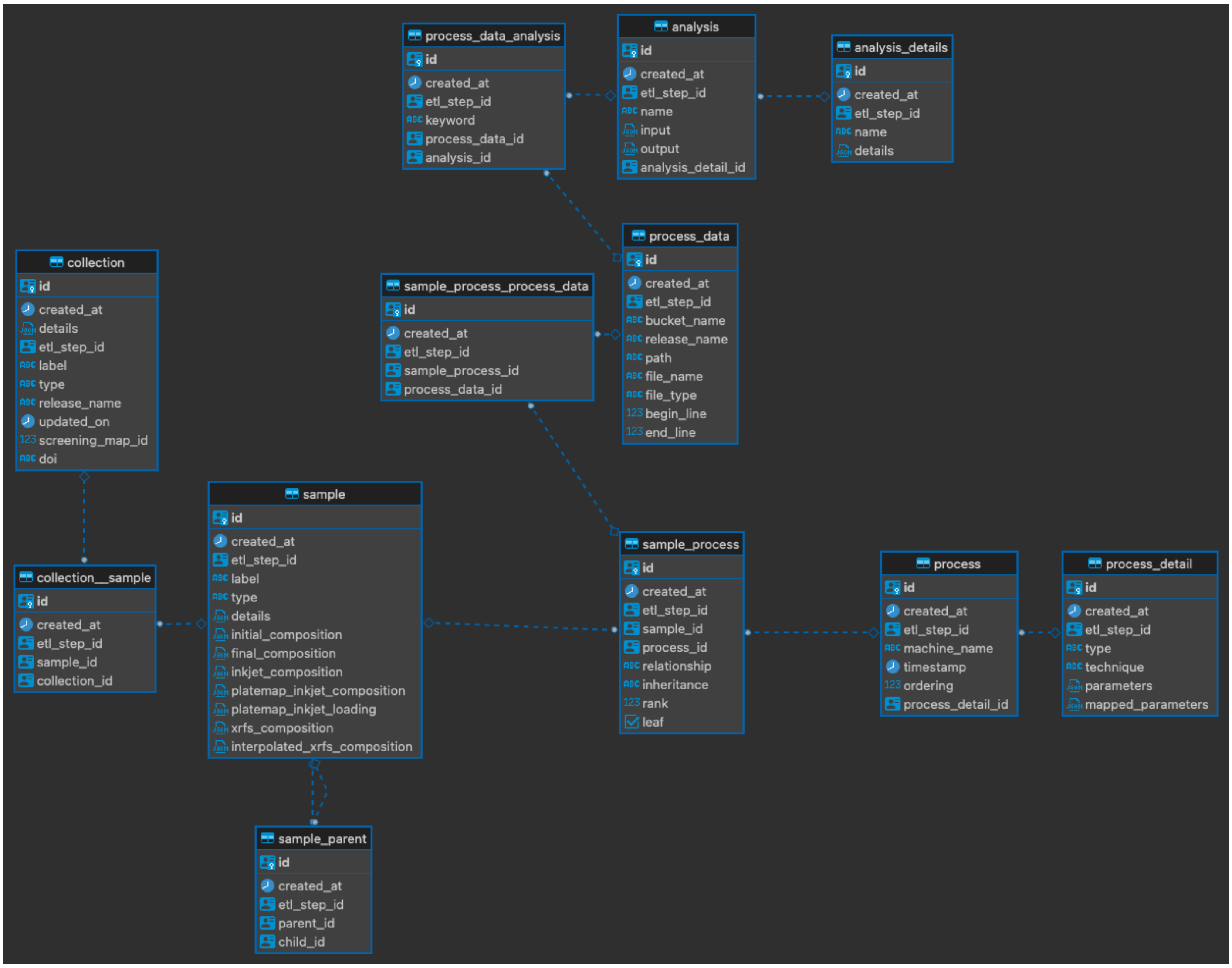

Fig. 3 Database Schema for the Materials Provenance Store database