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

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 two 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").

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 two columns:

- type: a short string that specifies the type of the collection (e.g. "plate," "project", or "account")
- name: 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.

[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>

parent_sample_id	child_sample_id
1	3
2	3

parent_sample_id	child_sample_id
1	3
2	3
3	5
4	5

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:

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:

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

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.

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 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 supplemental information.

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 three ways to adequately capture this information: 1) using a "key-value" table, 2) using a "dimension" table, or 3) using a table with a jsonb dictionary column. 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, the third option is to use a table with a jsonb dictionary column. This table would have the following column:

- details (jsonb): a json dictionary containing the details relevant to that type of process

This design choice has some of the advantages of each of the previous two design choices. Like the key-value table, the jsonb table has no nulls, and like the dimension table, it does not result in repeated entries for the same set of process details. The disadvantage is a lack of structure. Using the dimension table design choice, each column (e.g. "sweep rate") and its data type is well defined. In contrast, with the jsonb design choice, there is no technical guarantee that a given key (e.g. "sweep rate") appears in the dictionary, and if it does appear, there is no guarantee that the corresponding value can be cast as a float. These risks should be mitigated by careful, programmatic data ingestion.

To summarize, the key-value architecture should be used if:

- A very large number of different types of metadata that need to be stored.
- Completely new types of metadata are added to the database frequently.
- Querying for many, specific key-value pairs is an uncommon use case.
- It is uncommon for many processes to be run with the same set of metadata.

The dimension table architecture should be used if:

- Only a small number of different types of metadata need to be stored.
- Completely new types of metadata are added to the database infrequently.

- Querying for many, specific key-value pairs is a common use case.
- It is common for many processes to be run with the same set of metadata.

The jsonb architecture should be used if neither of these scenarios applies. For example, in the case of the database presented in this work, we wish to be able to capture a large number of different types of metadata, new types of metadata will be added frequently, and it is common for many processes to be run with the same set of metadata. For that reason, the jsonb architecture was chosen.

Oftentimes, a research group is not sure which case applies to them until after the data is loaded into the database. In this case, it is safest to start with the key-value architecture because it is the most flexible, and this design choice can be revisited after ingesting the data and running simple queries to determine how many distinct keys exist in the database, and how frequently new, distinct keys are added.

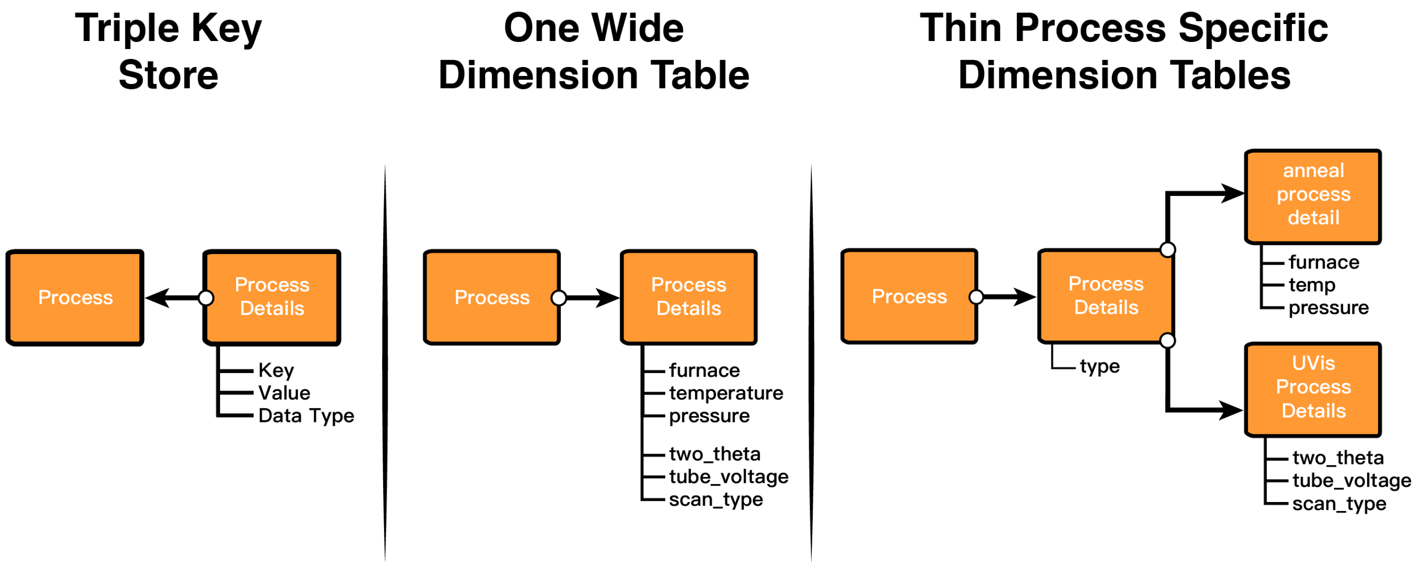

Fig. 1 SI FIGURE: An illustration of the three strategies for storing the the details of a process.

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. In the simple 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.

We define the measurement_group table to record these sets of samples so that any piece of raw data that is recorded is explicitly linked to exactly the samples it pertains to. Specifically, each row in the measurement_group table represents a group of one or many sample-process pairs that raw data files pertain to. The example above demonstrates the need for a many-to-many relation between sample-process pairs and measurement groups. We call the mapping table that captures this many-to-many relation the `sample_process`_measurement_group table because it maps rows in the `sample_process` table to rows in the measurement_group table. Below, we define the columns on the measurement_group table and explicitly show the example described above for clarity.

Each measurement_group is completely defined by the sample-process pairs that it is comprised of. Since measurement groups have no other properties, the measurement_group table has no columns other than the columns that are common to all tables in the schema.

The `sample_process`_measurement_group mapping table has the following columns:

- sample_process_id (integer): a foreign key to the sample_process table

- measurement_group_id (integer): a foreign key to the measurement_group table

As mentioned previously, it is common for processes to yield raw data; the purpose of the process_data table is to capture this raw data in the database. Since one or many data files may apply to the same measurement group, but an individual raw data file can never pertain to multiple measurement groups, there is a many-to-one relation between process_data and measurement_group. The process_data table has three columns:

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

- raw_data: a string containing the raw data that the experiment recorded

- file_name: a string containing the name of the file

- file_type: A string that specifies what type of data is in raw_data (e.g. ’json’, ’csv’, etc.).

Let us walk through the example above of a plate with four samples all undergoing a process which heats the plate, records the temperature of the plate (and therefore all the samples) as well as records an XPS spectrum of all four samples. In the real database, all primary keys are integers; however, in this discussion, we have changed some of the primary keys to letters for clarity. The sample_process table records the fact that the four samples (with IDs 11-14) all went through this heated XPS process (which has ID 10).

Table 1 sample_process

id	sample_id	process_id
A	11	10
B	12	10
C	13	10
D	14	10

There is a measurement group for each XPS spectrum (with IDs 1-4 in this example) and one for the temperature measurement (with ID 5 in this example). The mapping table between measurement_group and sample_process stores which entries in the sample_process table are members of which measurement group.

Table 2 measurement_group_sample_process

measurement_group_id	sample_process_id
1	A
2	B
3	C
4	D
5	A
5	B
5	C
5	D

The process_data table contains the data and links it to the appropriate measurement group.

Table 3 process_data

measurement_group_id	filename
1	XPS1.csv
2	XPS2.csv
3	XPS3.csv
4	XPS4.csv
5	Temperature.csv

1.10 The analysis and analysis_group table

Almost always, data that is captured in the lab is processed to arrive at 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 purpose of the analysis table is to store these higher level results. Each distinct analysis_name represents a pre-defined data processing function that will be run many times as new data is recorded. Each row in the analysis table represents an application of one of these functions to data in the database. For example, if the function that finds the open circuit voltage of a current-voltage curve is called "get_VOC," and that function were run on five different current-voltage curves, five rows would be inserted into the analysis table with the same analysis_name but with different inputs and different outputs. The analysis table has the following columns:

- analysis_name (varchar): the name of the analysis function
- input (jsonb): a json dictionary that contains all of the inputs to the function
- output (jsonb): a json dictionary that contains all of the outputs of the function
- analysis_group_id (integer): a foreign key to the analysis_group table

Since the same group of data is often analyzed using many different analysis functions, we define the concept of an analysis_group. Every row in the analysis table is linked to an analysis group, and an analysis group represents the set of measurement groups that contain all the raw data necessary to run the analysis functions (rows in the analysis table) that link to it. Each analysis_group is entirely defined by the measurement groups that it is comprised of. Since analysis groups have no other properties, the analysis_group table has no columns other than the columns that are common to all tables in the schema. In the simplest case, an analysis group is linked to just one measurement group. For example, consider the case where a current-voltage curve for a solar cell is recorded, and the goal is to store the open circuit voltage, short circuit current density, maximum power point, and fill factor in the database. One analysis group would be created, and it would only be linked to the measurement group that has the current-voltage curve of interest. Four rows in the analysis table would be created - each would have a different analysis_name (e.g. "jsc," "voc," etc.) and each would have the same measurement_group_id. Since data from a given measurement group can be used in many analyses, and the data from many different measurement groups can be used in the same analysis, there is a many-to-many relationship between measurement groups and analysis groups. By convention, we name the mapping table measurement_group_analysis_group, and it has the following two columns:

- measurement_group_id (integer): a foreign key to the measurement_group table
- analysis_group_id (integer): a foreign key to the analysis_group table

1.11 RDMS implementation

There are three standard ways to adequately capture the process detail information: 1) using a "key-value" table, 2) using a single wide dimension table, or 3) using thin process-specific dimension tables. Figure **??** shows these three architectures. Each strategy offers advantages and disadvantages specific to the research workflows that are being modeled. The general trade off that underlies the design choice is flexibility versus query speed and data type constraints. The "key-value" type framework is the most flexible, but cumbersome to query and with weak type constraints on the stored data. Whereas the thin process-specific dimension tables require all columns to be defined in the framework prior to ingestion, but offer advantageous query structure, times and type constraints. Additionally, while one could store all possible process detail keys on a single wide dimension table, scientific research processes usually have few overlapping parameters between process types. For example, very few input parameters are shared between a UVis-type process and an anneal process. This would result in many `process_detail` entities having null values for all parameters that belong to the other types of processes. It is also important with a single wide dimension table to prevent name collisions as two parameters may share the same name from process type to process type, but their meaning and units can be drastically different, such as "scan speed" for a XRD and cyclic voltammetry experiment. Thus, when the number of process types grows it makes sense to separate the parameters of each process type into their own process-specific entity, such as `uvis_process_detail`. This removes the possibility of name collisions, removes the need to populate all null values for non-valid parameters, and makes it facile to determine the process keys that are valid for a given process.

The choice of architecture may also be limited based on the type of relational database management system (RDMS) being used. Many NoSQL and graph RDMS's utilize javascript object notation (JSON) documents to store all entities, which makes the key-value type architecture the only viable choice. Additionally, many modern SQL RDMS's also offer powerful JSON type columns that can reduce the query speed and complexity for the "key-value" strategy. However, for most scientific data type-constrained values can provide important data integrity checks that reduces the burden on any consumers of the database downstream. For a complete discussion of the advantages and disadvantages of these three architectures please refer to the SI.