Tabular Data Archetypes

JD Margulici :: linkedin.com/in/jdmargulici/ :: March 10, 2020

Introduction

This paper puts forth a set of ideas about the engineering design of data-intensive software systems, particularly those that model or interface with the physical world -sometimes referred to as cyberphysical systems, and largely overlapping with the class of applications known as the Internet-of-Thing (IoT). The concept of data archetype proposes a taxonomy for data sets, which aims to raise the level of abstraction suitable to specify and program these systems. Such an attempt necessarily carries tradeoffs: on the one hand it can speed up development and lower maintenance costs while improving reliability; on the other hand, every layer of abstraction reduces control and flexibility, and can negatively impact computing resources utilization. Our purpose is not to immediately settle these tradeoffs, but rather to present key concepts with the hope that they can stimulate further ideas and encourage progress, in whatever shape it materializes. The overarching theme is the use of explicitly defined information semantics in computer programming.

Context and assumptions

For this paper, we will exclusively focus on tabular data. The concepts extend naturally to data cubes, and it might be possible apply them to other forms of data (e.g. graph structures), but the simplification is warranted for the sake of clarity and conciseness. Tabular data covers a great deal of use cases, most notably time series, which arguably form the backbone of cyberphysical information.

In describing information captured in a system, we distinguish between “dimensions” or “entities”, and “facts”. In business intelligence lingo, dimensions refer to sets of labels that serve to categorize and analyze data, such as products, locations and times, whereas facts are the measurements and metrics that pertain to those dimensions -e.g. temperature measurements or sales numbers. In the context of cyberphysical systems, we assume that dimensions include things such as sensing devices, vehicles, buildings, as well as people, and the term entities seems overall more adequate. We will henceforth use the terms dimension and entity somewhat interchangeably.

Typically, entities will be stored in a relational database and be linked through relationships such as association and aggregation. Facts may also be stored in a relational database, but the technology landscape offers many other options such as specialized analytics stores, column-oriented databases and data lakes. Fact data tends to be considerably more voluminous than entity data and more often warrants distributed storage and denormalized topologies optimized for large read queries and map-reduce jobs.

In a typical architecture, facts are organized into data sets. We define a *data set* as a collection of homogeneous records whose most common system embodiment would be a database table – say for instance a set of temperature measurements from a network of weather stations. Here “homogeneous” essentially means that the records share the same schema. Document-oriented and column-oriented databases may challenge that notion a bit by allowing flexibility in the design of individual record attributes. Yet even so, records belonging to the same data set will share at least some of their structure, and that shared structure is what we refer to as their schema.

We further define a *data object* as a structured container for a finite set of facts. For instance, a data frame in R or Python Pandas would qualify as a data object, as would the data in a spreadsheet. Even though we associated data sets with database tables in the previous paragraph, data sets are best understood as abstract constructs that exist independently from their implementation – weather station measurements form a conceptual data set whether the data is stored or not. By contrast, data objects are concrete informational objects that are embodied in a program's memory. A straightforward relationship between the two concepts is that a query on a data set results in a data object. Like a data set, a data object follows a schema – for instance the set of columns in a data frame. Having said this, a data object is not necessarily instantiated from a single data set since it can also be formed with a join query – in which case it still would have a schema, but not one that corresponds to a stored data set.

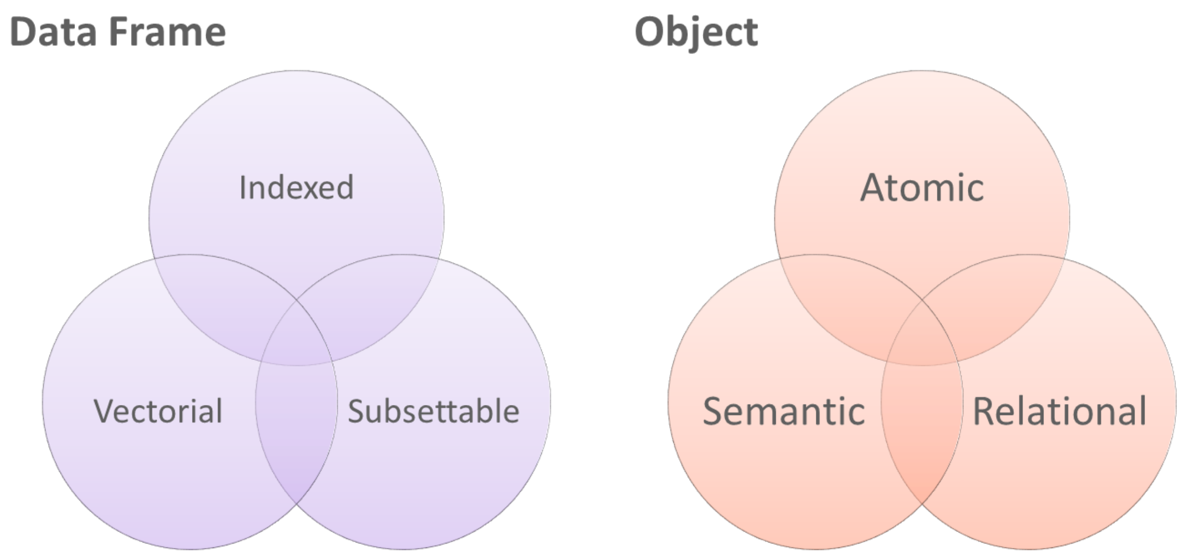
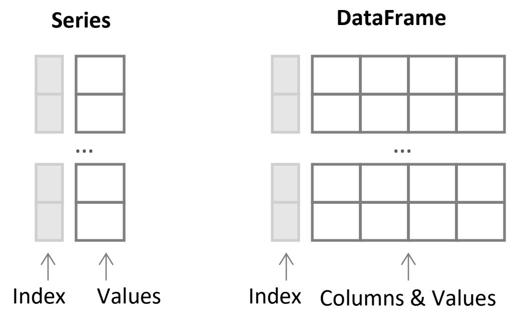


Figure 1 - Respective merits of data frames and objects

Different software publishers use the terminology data object for various purposes, and our definition is by no means authoritative. The underlying intent is a sort of synthesis between object-oriented programming and data analytics, leading to a type system for fact data. At a coarse level, the respective merits of data frames and objects are summed up in Figure 1. Data Frames are designed for efficient vectorial computations and manipulations. They were popularized by the R programming language and primarily intended for interactive sessions or focused data processing tasks. By contrast, application development usually relies on object-oriented programming, which enables large data ontologies and sophisticated logic by breaking down complexity through encapsulation and maintaining in-memory relationships between objects. In a data-intensive system where numerous data sets coexist, there may be value in marrying the two concepts by handling tabular data as first-class application objects that also feature the convenient functions offered by Data Frames.

At the implementation level, analytics software packages treat data as objects. For instance, Python Pandas features both a DataFrame class and a Series class. The former has columns while the latter wraps a one-dimensional array. Further, the Series serves a dual purpose as a singular-column data container as well as the result of a DataFrame's single row selection. Figure 2 describes the Series and DataFrame concepts as well as their topological relationships. A single row of a DataFrame is a Record with one row and several columns, thus conceptually distinct from a Series, though the Pandas implementation treats them as a Series whose index is the list of columns.



Data topologies

	One column	Many columns
One row	Scalar	Record (Series)
Many rows	Series	DataFrame

Figure 2 - Topology of tabular data

Since everything that occupies memory in Python is an object, it is only natural that classes exist to implement Pandas' data analysis functionalities. Utilizing singular DataFrame and Series classes allows these classes to behave as swiss-army knives of data slicing and aggregation and sidesteps complex problems of type determination in data transformation operations. However, that flexibility comes at the cost of semantics. It makes intuitive sense to deem that two DataFrames that share the same index and column definitions are objects of the same type, whereas a sales report and a log of temperature measurements are obviously distinct semantic entities, but there is no type system that readily captures this reality.

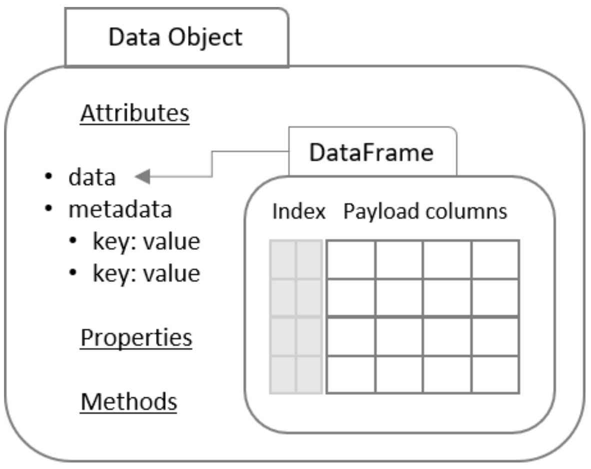


Figure 3 - Conceptual Data Object Implementation

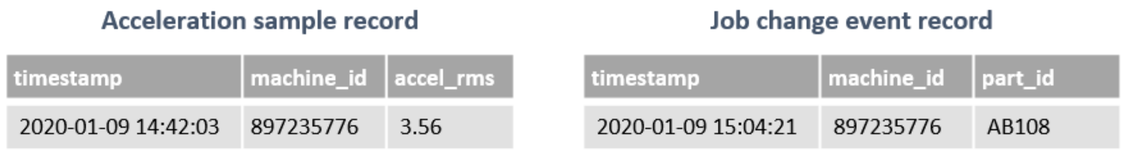
In the remainder of this paper we will make inferences about what a semantic type system for data objects might look like and examine what benefits and drawbacks would exist as a result. Practically, DataFrames expose metadata about column types and names that allows analysts to reason about their structure. However, they do not provide a structure to store metadata information such as data provenance or column properties beyond name and data type. The concept of data object aims to close that gap by allowing various properties to exist alongside a data core. At the implementation level, this can take the form of typed objects that expose a DataFrame by association and contain additional attributes. This structured is represented conceptually on Figure 3.

Having made those remarks, we can define a *data archetype* as an abstract base type for data objects. Our description of a data archetype will proceed in three steps that focus on different properties of data objects. The first step is to explore schema typing, since the similarity between two data objects necessarily involves similarity between their schemas. The second step will delve into data topology. As we will describe, two data objects may share the same schema yet still carry structural differences because of distinct index shapes. In the third step we introduce additional properties that arise when a data object is considered in the context of an underlying data set from which is was generated.

Schema Types: an Illustrative Example

We kick off our presentation of schema types with a concrete example intended to illustrate the possible payoffs. Let us consider two time series generated by a manufacturing machine. The first time series samples the vibration energy of a rotating spindle thanks to an embedded accelerometer, which serves to monitor the operating condition of the machine. High-frequency data from the accelerometer is windowed and averaged into a single root mean square value pulled every 10 seconds. The second time series is a set of discrete events that correspond to a job change on the machine. Here a "job" means a machine production batch for a given part. Each job may produce tens or hundreds of parts and may last anywhere from a couple of hours to a few days. Each job change event record hence contains a part identifier that specifies the type of part being produced. Figure 4 shows an example record from each series.

Based on our definitions, each series form a data set with a straightforward schema. Assuming we index records by their timestamp and machine identifier, the only difference between those schemas is that the data types of the payload column are different: the acceleration samples are float while an event record could be a string or a categorical variable depending on the application's design. In the next paragraphs we turn to the semantic interpretation of that difference and its design implications.



Acceleration sample record

timestamp	machine_id	accel_rms
2020-01-09 14:42:03	897235776	3.56

Job change event record

timestamp	machine_id	part_id
2020-01-09 15:04:21	897235776	AB108

Figure 4 - Example records for two time series

The sample series has a nominal sample rate of 10 seconds. This means that we generally expect that for a given machine id, we will find records timestamped every 10 seconds. This expectation may not hold consistently because records can be lost at any point along the data processing chain. Yet if there are large gaps between records, it is a likely indication that such losses occurred, which also provides relevant information about the system. By contrast, job changes are sparse and infrequent. We still have some expectations about the time interval between two job changes, but these expectations are statistically weak. For instance, the distribution of job changes could be assumed to follow a Poisson process whose mean can be estimated from past observations. This still leaves room for high variability in the time gap between two events, and hence the information content derived from a single gap observation is low. This characteristic difference between the two series has multiple implications for application development, which we detail below.

1. **Processing and summarization**: Samples are produced regularly and hence don't represent an actionable event unless their values express a meaningful condition. Conversely, job changes may trigger specific workflows. These range from updating a production dashboard, to alerting a customer, or even feeding back specific monitoring thresholds for the series of vibration samples -the theory being that different jobs employ different machining parameters.

   Summarization will also require different, yet somewhat predictable operations. Samples summarization will likely resort to descriptive statistics such as mean and standard deviation over meaningful time intervals -where meaningful is driven by a) the sampling rate of 10 seconds, and b) the kind of reporting frequency that plant operations can act upon. Job change summarization will logically group events by job identifier. Relevant statistics may include a pie chart showing the proportion of time a machine spent producing a given part type, or job duration statistics factored by part id.

2. **Querying**: A salient, yet probably underappreciated difference between the two data sets appears when performing a time range query -that is, a query that specifies a time interval of interest. In the case of samples, the data consumer will typically expect to receive all the samples whose timestamp lie between the bounds of the query interval. In the case of job changes, the data consumer may rather seek a time-ordered sequence of jobs handled by a given machine over the time interval. This is not as trivial as it may sound. Consider the case in which no job change took place over the query interval, which simply means that the machine worked continuously on a single job over that time range. If there is no record in the query range, then we cannot a priori know what that job was. There are various implementation solutions to address this use case, but the key point is that the semantics of the two data sets influence query patterns in a predictable fashion.
3. **Representation**: Another difference between the two data sets is how we may choose to represent them for interpretation by an application user, particularly in the graphical realm. Samples are the quintessential time series, which would be represented by a curve along the time axis. By contrast job change events signal a change of state. A more apt representation might be in the form of colored blocks along with a legend. This difference merely results from plotting a float variable vs a categorical variable, but data processing frameworks do not provide built-in routines that automatically direct each data type to a meaningful plotting method.
4. **Storage and indexing**: As already noted, the samples set features a dense, homogeneous distribution of data along the time axis, whereas the events set is sparse and heterogeneous. While many data storage technologies will easily handle both types of data, they will hardly be optimal in both cases due to the difference in record distribution.

This example points out that two data sets with quite similar schemas will receive different treatments in the process of designing, implementing and using a software application. But the real question is the converse: are there robust, predictable patterns of design and usage that can be assigned to classes of schemas and hence justify a type system? If so, one could imagine declaring application schemas as instances of a certain type. In turn, the schema type could be used by different parts of the application as described above: summarization analytics, event propagation, storage and querying, and user interfaces.

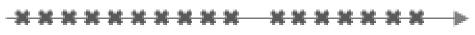
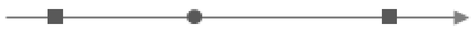
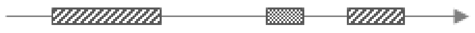
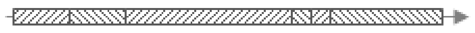
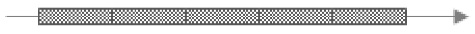
To elucidate this question, we propose a taxonomy of five time series schema types. These all share the same indexing scheme, i.e. a combination of an entity identifier (machine in the above example), also referred to as a key, and timestamp. In other words, for a given entity identifier, the records form a sequence based on the indexing time stamp -even though as we will see, the scope of some of these records is a time interval rather than a single point in time. This indexing scheme is commonplace in pub/sub messaging systems and column-oriented databases and can generally be considered to embody the core functional capability of event stream engines that underlie cyberphysical systems.

Time Series: Semantic Schema Types

The five time series schema types are summarized in Table 1. The table lists the five proposed types, a pictorial description meant to make each concept immediately accessible, distinct features associated with each type, and a real-world example to illustrate the application.

It is important to emphasize once again that the intended value of this exercise is to create useful semantic classes -as we already pointed out, each of these schema types can be made to look identical at the implementation level since they all share the same index, composed of a key and a timestamp. Also, it is worth noting that there may exist use cases of time series that don't fit neatly in this taxonomy. One example would be a data set that allows overlapping intervals for a given key, as the sequentiality between intervals becomes blurry. We would contend that such cases are really either variations or compounded versions of the five basic types presented here, and we include a discussion in the relevant subsections. However, it must also be recognized that the indexing scheme is inherently limiting for time intervals. In a database system that features multiple indexing capabilities, indexes would be created for both the start and end times of a time-spanning record. When that is the case, overlaps don't pose any problem. Ultimately, what we are considering here are different stages in the data life cycle: naturally occurring time series in cyberphysical systems are spun from events and will fit in the taxonomy, yet summarized data designed for analytics storage doesn't necessarily obey a sequential ordering and generally requires indexing on multiple columns.

Table 1 - Time Series Schema Types

Schema	Pictorial Description	Features	Example
Sample		Near-periodic timestamps	Temperature measurements
Event		Non-periodic timestamps Typed or untyped events	Alerts from monitoring system
Session		Start + end times Typed or untyped sessions Non-overlapping sessions	Machine production logs
State		Time-partitioning	Parking space availability
Period		Strictly periodic timestamps Windows may be sliding or tumbling	Daily sales reports

Sample Schema Type

The sample schema type assumes that payload values are collected at regular time intervals. This type embodies the common notion of what a time series is. The payload values are most often floats, as would be the case with sensor measurements, but other data types are possible. For instance, the payload values may sample a state (think windshield wipers activation on a connected vehicle), or maybe an entire array of sensor measurements coming from a device with multiple capabilities. The common semantic assumption is that each sample provides a snapshot of variables of interest at the time of collection. By contrast, the state schema type that will be described below only reports changes from one state to another. Another feature of the sample schema type is that the sampling regularity is only a loose expectation. In practice, the periodicity may be approximative only due to implementation details, and samples may be lost, either in a punctual fashion or over extended periods of time in case of a disconnection. This distinguishes the sample schema type from the period schema type, which by design provides a record for every period -even if some of those records are empty.

Event Schema Type

The event schema type captures discrete events that take place with no a priori regularity, i.e. the timestamps are generally assumed to form a sparse set. The payload for each event may be as poor or as rich as the application requires it, spanning from no payload at all (in which case the timestamp provides the entire information content) to multiple descriptive columns and/or semi-structured (say a mapping with ad-hoc keys) or unstructured data (such as an image). However a particular, optional payload attribute is an event type, which is a categorical variable that allows grouping of events by type. Consider for instance transit fare card transactions, where the card identifier is the key, and transaction events can be typed with the station identifier. The event schema type is selected for what we would call “hard” events, which are events that are considered to take place at a singular point in time, in contrast with “soft” events that have a definite duration, which we call sessions. Note that in the above transit example, either schema type may be suitable depending on whether the fare card is used only for checking in, or both at check-in and check-out.

Session Schema Type

The session schema type fits defined, non-overlapping time periods -also referred to as soft events. Except for the fact that they feature a start and end time, sessions are semantically like events -their distribution in time is a priori sparse, their payload content is infinitely flexible, and they can be qualified with an optional type. A critical property of this schema type is the absence of overlap between sessions that are attached to the same key. Said another way, sequentiality between sessions must be maintained unambiguously. Examples in the physical world are many: vehicle trips, phone calls, worker shifts, etc. Where the non-overlapping rule might seem like a constraint, the underlying cause is probably that multiple types of sessions are being intermingled. This would be the case of a self-diagnostics system on a connected device, which may be reporting multiple, simultaneous failure modes. For instance, a time period of low battery operations may overlap with a loss of network. A dataset that reports on all the failure events for the device would hence feature overlapping sessions. Nonetheless, the low battery sessions and the loss of network sessions individually follow the non-overlapping pattern. Such a use case can be addressed with individual topics for each failure mode, possibly merged downstream in the analytics pipeline into either an unqualified failure session dataset, or a state dataset that marks the boundaries of the various failure modes.

The sequentiality of sessions means that they can be stored with a single time index. A query range must select all sessions whose interval falls within the range, and look up the first preceding session, if it exists, whose interval may end within the range. Overlapping sessions would require that both start and end times are indexed for queries to succeed. This works in an analytics data store but not in a sequential dataset.

State Schema Type

The state schema type tracks changes in state. Hence records mark transitions from one state to another, and the payload describes the new state. Most typically, that new state is captured by a categorical variable, but it is possible to also imagine more complex state descriptors such as arrays or nested structures. The summary table suggests parking occupancy as an example, which is a binary state (though this may be tweaked to include a third, “not available” state when the information is not known). A log of prices in a dynamic roadway tolling application is another example. Assuming that each price level defines a state, the number of categories grows much greater.

The state schema type basically assumes an optimal representation of state changes, in that only transition events are part of the data set. This means that, by design, consumers of this data will assume that state doesn’t change between two events -whether such an assumption is safe and to what extent is a function of the system’s overall reliability. There are other ways to track states, such as regular sampling as pointed out in the description of the sample schema type. In practice the lines between these two semantic types can blur: the state schema type does not preclude events that indicate a transition between two identical states, and if such events are recorded at regular time intervals, then the data can end up identical to sample data. This underscores the fact that schema types are employed to materialize a semantic intent. If a state is sampled on a regular schedule, then there is no guarantee that the state remained constant between two events, and the exact transition points remain hidden. This may be fine for many applications, because knowledge of the exact transition points has little or no value and the sampling is frequent enough for the use cases of the data. However, the state schema type will always be more economical storage-wise since it generates sparse sets.

Period schema type

The period schema type is designed to capture regular reports. These can range from publishing average sensor reading values over sliding windows to complex data structures such as complete daily usage statistics for a website. Both the sample schema type and the period schema type assume data periodicity. A key semantic difference between the two types is that the sample type represents a present value or state at the time of collection. By contrast the period type is meant to carry data that summarizes information for time intervals. Further, loss of samples and therefore accidents in periodicity are expected with the sample schema type. On the other hand, the period schema type suggests that at the implementation level, records be created systematically, whether upstream data can be fetched or not.

Because only timestamps are used for indexing, a window specification must be provided for a complete interpretation of the data. Windows may either be tumbling, as shown in Table 1, or sliding. Either way, a single parameter expressing the duration of the windows suffices. We can assume by convention that the indexing timestamp points to the beginning of the period corresponding to each record, and hence the period’s end is determined by adding the window duration. This suggests that in addition to schema types, we can define schema parameters -in this case, the window duration. In similar fashion, the sample schema type could accept a sampling rate parameter, though it is not as critical to the interpretation of data sets as the period schema’s duration parameter is. Of course, schema parameter is basically a fancy term for metadata, but just like the schema type system in the first place, our purpose is to introduce more precise concepts and terminology.

With periodic reports we enter the realm of traditional data analytics, in which sequential storage may still make sense, but will often be complemented or supplemented by multi-indexing capabilities. Suppose for instance that reports provide daily sales reports for a chain of retail stores. One view of the data is as a set of sequences keyed by store id. However, there are likely many query use cases for this data, involving features of the stores or the merchandise sold, as well as filters on the data itself, all functionalities that may not be optimally covered in a column-oriented store. Developing schema types with multiple indexing capabilities is theoretically possible, but out of scope for this paper.

Data Topologies

Now that we have explored the notion of schema types, the second step toward the definition of data archetypes is to examine data topology. We already provided a hint of what that entails in Figure 2. Let us revisit this concept in the context of the time series schema types presented in the previous section.

Table 2 - Data Topologies for Time Series Schemas

Timestamp ↓	**Entities →**	Single value	Set of values
Single value		Record	Array
Range of values		Sequence	Log

As explained previously, a natural way to generate a data object is to issue a query on a data set. The time series schema types feature two indexes: the entity identifier, or key, and the timestamp. In either domain, a selection query can take one of two forms: either a strict equality constraint or a range. Let us enumerate the four possible kinds of selection queries that result from these combinations, which are laid forth in Table 2. Note that there are degenerate cases in which a query returns empty or partially empty results, but they don't qualitatively impact the topology. We can simply assume that the result from each query is a data frame that may contain zero to many rows, and that the data frame is wrapped into a data object featuring additional metadata attributes as proposed in Figure 3.

The four topologies are depicted in Figure 5 for clarity. Each object shows indexing as well as other "payload" columns, and metadata attributes in boxes. Indexing columns that become optional because they host uniform content are represented with a lighter shade.

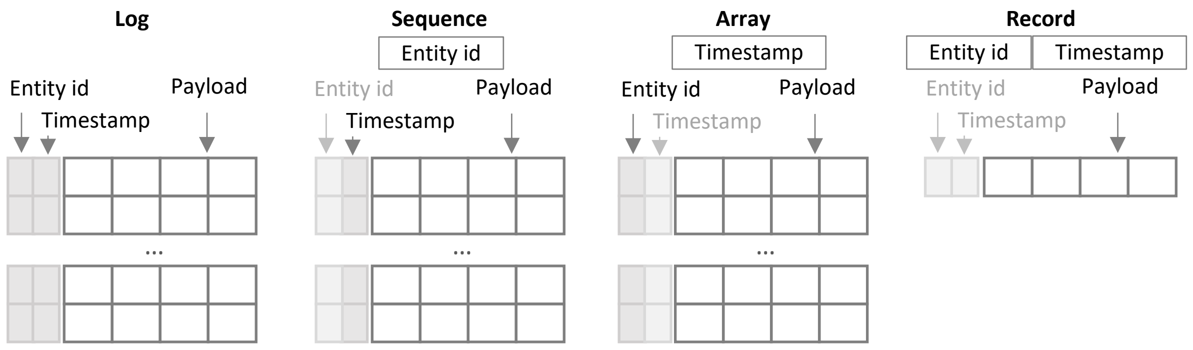


Figure 5 - Time Series Data Topologies

em

The topologies are as follows:

- Timestamp range and entity set: the data object is a **Log**, a priori mixing multiple records from multiple entities and combining indexes for timestamps and entities.
- Timestamp range and single entity: the data object's records form a natural **Sequence**, and timestamps provide the only required index, whereas the entity identifier column becomes

semantically unnecessary as it contains a single value which can be stored as a scalar metadata attribute.

- Single timestamp and entity set: the data object is an **Array**, consisting of multiple records all sharing the same timestamp. Hence the entity identifier provides the only required index and the timestamp column uniformly features a single value that can become a scalar metadata attribute.
- Single timestamp and entity: the data object is a single **Record**, which may be empty in the degenerate case. The timestamp and entity identifier may be kept in the Record's data but they can equivalently be considered metadata.

At this juncture, we ought to disambiguate what constitutes data and metadata since we present them as somewhat interchangeable. Sticking to our time series schema family, we can firstly remark that every record features an entity identifier and a timestamp. Whether these indexing fields constitute data or metadata is a bit of a philosophical debate: on the one hand they stand apart from the payload data and give it context, on the other hand they are an inseparable part of the observation that each record captures. When dealing with a single record, the question doesn't really need to be settled: both the entity identifier and the timestamp are attributes, whether considered data or metadata. At the level of a collection of records, however, different notions may become confounding:

- By definition, the entity identifiers in a Sequence are identical for every record. At the implementation level, the column may be preserved in the object's data frame for the sake of consistency, but the identifier is semantically an attribute of the Sequence itself. It's again a matter of terminology whether that attribute is data or metadata, but if we store it outside the data frame then it would typically be considered the latter.
- At the same time, a Log could be created by slicing a data set for multiple entity identifiers in a given time range, but feature only a single entity identifier: this would be the case if no records exist for the other entities over the selected time range. What this means is that the data in the Log is identical to that of a Sequence of records for that single entity over the same time range. However, there is a semantic difference since the Log also indicates the absence of data for other entities. In particular, it would be misleading to make the unique entity identifier that shows up in the data frame's index an attribute of the Log in the same way we could do it for the Sequence, since the scope of the data is actually greater. In order to fully represent the information, the Log object would need to feature the list of entity identifiers that were queried in order to generate it, irrespective of whether they show up in the data.

The last point is significant in that it illustrates how two data frames with the exact same content can express different levels of information based on the context that generated them. In other words, there can be information in data as well as in the absence of data. In the next section, we introduce additional contextual considerations that participate in the proposed definition of data archetypes.

Contextual Properties

The previous section already hinted at the nature and information value of a data object's contextual properties: a data frame may contain data for only a subset of the entities that were queried on a data set, and if so the list of queried entities indicate that the data set features no data for the missing entities -a piece of information in itself.

Another way to think about contextual properties is to regard a data object as always being an excerpt of some underlying data set that is unbounded. Since that excerpt is typically obtained by slicing the data set, the slicing parameters must be known for a full interpretation of the data. This remains true even if a data object is created from a join query: even though the application may not concretely store an underlying data set, there exists a virtual data set that results from the same join operation applied at the level of data sets.

We provided a strong example in the case of missing entities, and a more nuanced case arises when a time range query returns sparse results. Let us for instance consider a fleet of connected vehicles that each transmit position fixes over a wireless network at regular intervals. According to our taxonomy, the data set sourced from these vehicles has a schema of the samples type. However, the vehicles only transmit data while driving. This means that a sequence of data samples for a given vehicle over a specified time range will be nominally made up of subsequences of regularly spaced samples, separated by gaps that correspond to periods of inactivity. The specified time range is necessary to interpret the data object, since the periods of inactivity may occur at the edges of the time range, resulting in a loss of information inside the data frame. As an example, Figure 6 shows the speed of a vehicle against time. The vertical lines indicate the extent of the queried time range, and the shaded areas represent periods during which no data was collected. As a result, the range of the data appears shorter than the query range.

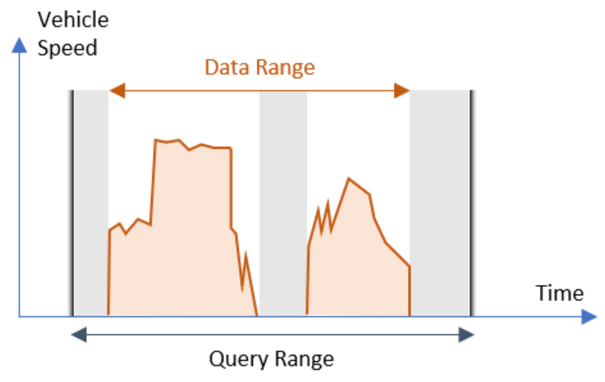


Figure 6 - Vehicle speed against time. The data range is a subset of the queried time range.

In this example, the data is naturally bundled into sessions, and each session may be called a vehicle trip, with some subjectivity remaining as to what constitutes a good time separation constant. For instance, a gap of 2 minutes or less means that a stop at a gas station will break a trip into multiple, independent legs. Conversely a longer gap of 30 minutes will ensure that all legs join into a single trip but may also group together segments of data that correspond to distinct trips with different purposes. Regardless of the desirable parameter, there is a natural grouping of data samples into sessions that are locally maximal, in that they contain no large gap, and that appending any additional sample from the data set either prior or posterior would result in a large time gap between consecutive records. These sessions form a particular subset among the set of possible subsequences of samples for a given vehicle, and we call these trips. Hence from an information standpoint, "is a trip" is a property of a vehicle fix sequence that may be true or false. The "is a trip" property of a data sequence provides a paramount example of a contextual property, since it is wholly dependent upon information contained in the underlying data set but inaccessible once the data has been reduced to a data frame.

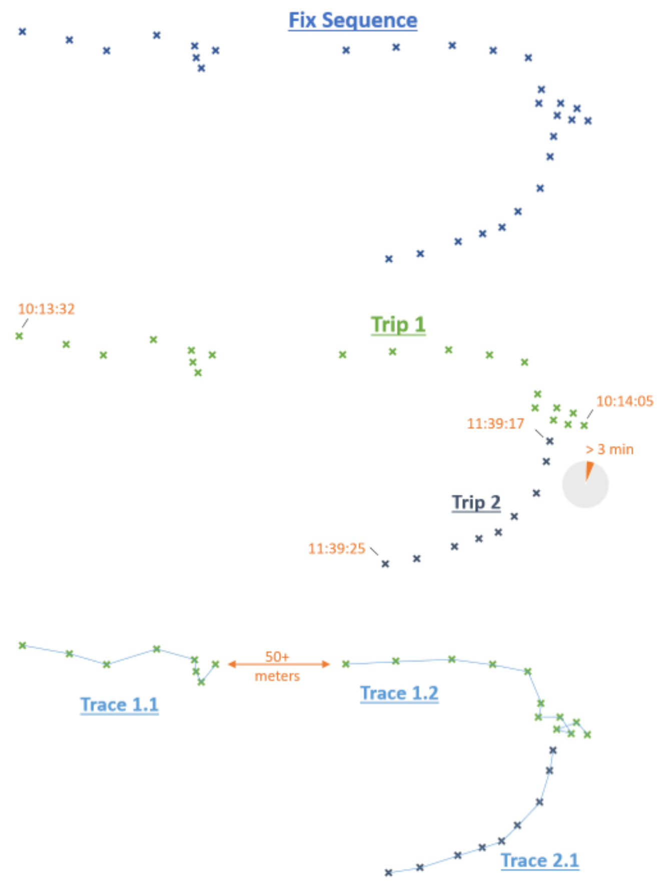


Figure 7 - Fixes, Trips and Traces

In the same vein, we can build additional context by measuring the spatial distance between consecutive fixes. When that distance is small, the line segment connecting the two fixes approximates the vehicle's trajectory. However, as that distance grows the approximation gets worse. Hence within a trip sequence we can build nested subsequences consisting of groups of fixes not separated by more than a threshold distance, which we call traces. The line string connecting fixes within a trace represent fragment of the vehicle's trajectory. Isolated fixes or disconnected traces inform us about the vehicle's itinerary but not its spatial trajectory. The "is a trace" property is another contextual property that may be affixed to a sequence of fixes, and an application could translate that property to the spatial representation of the sequence as either a linestring or multipoint, to use GIS nomenclature. Figure 7 displays example data under the perspectives of a simple sequence, a grouping by trips with a time constant of 3 minutes, and a grouping by trace with a spatial constant of 50 meters.

This example also hints at the extension of the time series schema types to the spatial domain. To be specific, we can define the Fix schema type with indexing fields for a moving object identifier, a timestamp, and a fix made up of a latitude and longitude values in a chosen spatial referencing system.

Summary

We can now summarize the concept of a data archetype, after which we provide a discussion of its possible benefits and applications.

A data archetype is an abstract base type for tabular data objects that are embodied by data frames. The archetype provides structural, semantic and contextual elements that lend predictability to the data object so that it can be handled programmatically. These elements are:

- A *schema type*, which specifies an indexing strategy and some core columns, but allows concrete schemas to feature additional columns.
- A *data topology*, which we define in terms of the shape of indexing values -for instance, a Sequence is made up of records with the same key and contiguous timestamps within a time range.

- Optionally, *contextual properties* that relate an archetypical data object to the underlying data set from which it is extracted, such as whether a Sequence is locally maximal.

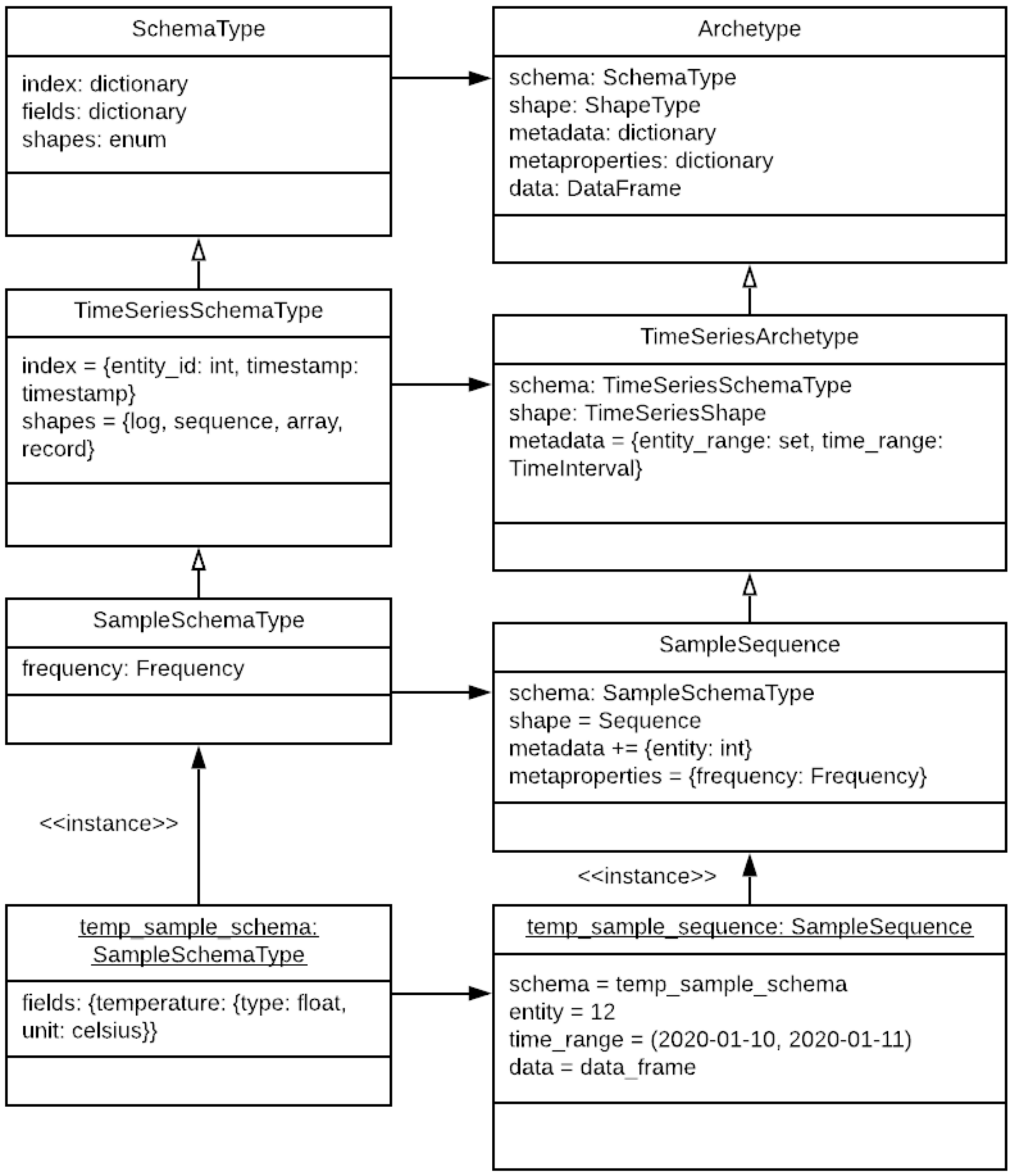

SchemaType
index: dictionary
fields: dictionary
shapes: enum
Archetype
schema: SchemaType
shape: ShapeType
metadata: dictionary
metaproperties: dictionary
data: DataFrame
TimeSeriesSchemaType
index = {entity_id: int, timestamp: timestamp}
shapes = {log, sequence, array, record}
TimeSeriesArchetype
schema: TimeSeriesSchemaType
shape: TimeSeriesShape
metadata = {entity_range: set, time_range: TimeInterval}
SampleSchemaType
frequency: Frequency
SampleSequence
schema: SampleSchemaType
shape = Sequence
metadata += {entity: int}
metaproperties = {frequency: Frequency}
<<instance>>
<<instance>>
temp_sample_schema: SampleSchemaType
fields: {temperature: {type: float, unit: celsius}}
temp_sample_sequence: SampleSequence
schema = temp_sample_schema
entity = 12
time_range = (2020-01-10, 2020-01-11)
data = data_frame

Figure 8 - Structural relationships between schema types, archetypes, and their respective instances

Figure 8 provides a glimpse of the structural relationships between schema types and archetypes. The description isn't formally rigorous but depicts the core concepts in the framework of object-oriented programming diagrams. There are parallel inheritance chains applying to schema types as well as archetypes, both of which get instantiated concretely into a schema and a data sequence, respectively.

The attributes of the base Archetype include a schema, a shape (provisionally of type ShapeType, although details are omitted), data in the form of a DataFrame, metadata that supplements the data, and metaproperties. There is a soft boundary between metadata and metaproperties, but the design intention is that metadata are usually object-specific whereas metaproperties may be attributes of a specialized archetype. On this note, the subtyping of Archetype involves a combination of specializing the attribute types (e.g. the schema type of a TimeSeriesArchetype is a TimeSeriesSchemaType), and fixing these attributes (e.g. the shape of SampleSequence is the value Sequence).

Discussion

At the highest level, the concept of tabular data archetypes proposes to establish a type system for tabular data objects. Assuming a practical implementation that builds on DataFrames, programmers find themselves equipped with semantic properties and explicit structural descriptors that favor automation, code reuse and safer interfaces, generally pointing to improved productivity and systems reliability. On the flip side, the proposal adds a layer of abstraction that carries its own overhead, further obstructs machine-level operations and may be difficult to properly generalize.

Set in those terms, the discussion belongs to the realm of ontological research. As exemplified by the taxonomy of time series schemas, data archetypes anchor tabular data into a descriptive framework that links data representations to real-world, empirically tangible concepts such as sequentiality, regularity, or density. One source of inspiration for this work is the object-process methodology (Dori, 2002), which one can describe as an attempt to establish a system engineering discipline for information modeling and processing. The object-process methodology provides rich concepts to model entities, relationships, states and transformations. One of the abstract object types in the methodology is the informational object, recognizing the existence of immaterial records that relate to but also contrast with physical entities. A log of temperature measurement records provides a prime example of such an informational object, which we have called data object in this paper. Data archetypes expand the description and semantics of informational objects by identifying discrete classes of presumably universal character.

Another body of work that relates to data archetypes is the concept of level of measurement, pioneered by Stanley Smith Stevens and later extended by Nicholas R. Chrisman (Chrisman, 2002). The core idea behind levels of measurement is to recognize semantic features of data representations that cannot be captured by the traditional data types used in computer science. For instance, integers may be used for counts in one context, and for categorical variables in a different context. Programming languages will allow addition of integers regardless of that context, even though such an operation is meaningful in the former context but not in the latter. Similarly, data archetypes attach empirical meaning to data representations, beyond the requirements of computer storage and processing.

On a more practical level, how can one leverage the concepts presented here, and to what benefits? Here we shift perspective and reason about contemporary application architectures. In recent years, message-oriented middleware has seen tremendous growth in popularity, a trend most visible with the success of the Kafka framework. This growth can be linked to other computing trends, namely distributed computing and service-oriented architectures, or micro-services. In Kafka and other publish-subscribe messaging systems, messages are organized by topics, within which they are indexed by a key and timestamp. This simple architecture proves remarkably potent for handling a wide range of use cases -as demonstrated by Kafka's popularity, and its publisher's claims that the platform can be employed not only for messaging but also as a distributed logs system, a storage system and a streams processing system (Kreps, 2013) (Confluent, 2020). The power of this abstraction is such that it has spawned novel ideas on event-driven computing, and the duality between database tables and change logs that are influencing contemporary system design (Kleppmann, 2017).

The time series schema types that we presented in this paper all follow the same indexing strategy comprising of a key and a timestamp. Hence the schema taxonomy is compatible with pub-sub messaging and could serve as the basis for declaring topic schemas. This supposes a schema registry in which schemas can be assigned metadata attributes. This architectural pattern, while not required to get a system up and running, is actively encouraged and is currently a topic of research and development at leading software companies (Lin, 2016) (Confluent, 2020).

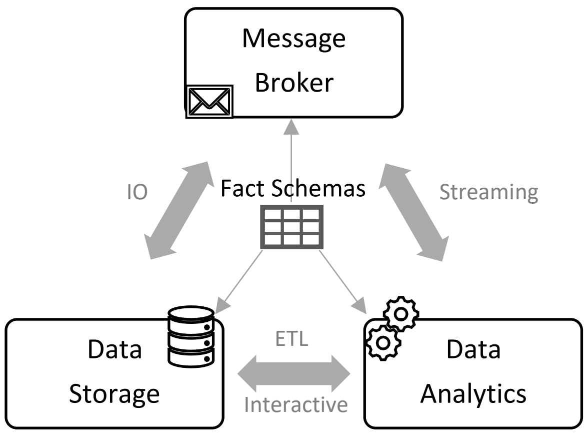


Figure 9 - Backend tech blocks sharing common schemas

The benefits of a schema repository for data topics range far and wide in a distributed, service-oriented system. Data may flow from a message broker to a database for long-term storage. On the way, it may be transformed through a streaming or batch pipeline, and an analyst may query measurements into an interactive computing framework operating data frames such as R or Python Pandas. These technologies may all share a common schema declaration that can be used to validate messages, specify transformations, create and query the database table, as well as provide predictability to the column structure of data frames. Figure 9 provides an abstracted view of this blueprint. Because of technology fragmentation, this is surprisingly not the norm, unless one uses a tightly integrated application framework. However, it is feasible to develop relatively simple interfaces to realize loose couplings between the different components by using the schema registry as the pivot. This suggests the following meta-architecture:

- Declare entity types and relationships to model the application's operating environment.
- Declare fact types as tabular record schemas all sharing a common index made up of a key and timestamp.
- Assign a metadata tag containing an entity type name to each fact schema, such that an individual record can be linked to an entity by combining its schema's entity type tag with its key, which is always be an entity identifier.
- Schemas can further be typed (another metadata attribute) and assigned additional attributes (say a sampling frequency in the case of a sample schema).

This model has the advantage of making it possible to introduce a semantic layer in the system without constraining any of the components to rely on it. Functionalities that take advantage of the data semantics can then be added incrementally, while realizing the general benefits of a schema registry. In turn, the availability of data semantics across the data plane offers the following benefits:

- **Automation and code reutilization**: as we have seen, archetypes present a predictable structure, hence their consumption and transformation can be standardized, which makes it possible to specify and program a system at a higher level of abstraction.
- **Rapid application development**: standardization and automation make it possible to develop a highly efficient application framework for rapid application development -which may sacrifice optimal resource consumption at scale but minimize the implementation and maintenance efforts required to get the system up and running.
- **Interoperability**: a major hurdle in the development of the Internet-of-Things ecosystem is the challenge of interoperability across systems (Noura, Atiquzzaman, & Gaedke, 2019). Part of the challenge lies in the diversity of communication protocols, but in truth that is a rather straightforward problem to solve, especially in a service-oriented environment in which translation services can be appended to existing systems. The harder problem is that different applications have disparate data models and semantics -there is never an easy fix to bridge such a gap. Publishing messages with explicitly declared schema types can help a great deal there. This is not to suggest that schema typing can magically cross application boundaries, but rather that type declaration imposes a common discipline and documentation norm in application design, which can at least get you part of the way there.

This last point is a good segue to examine the relationship between data archetypes and the development of the semantic web (Aberer, 2004). The core principle of the semantic web is to describe system entities through triplet relationships of the form subject–predicate–object. In theory, this is a technology-agnostic and infinitely extensible specification -and hence presented as the ultimate solution to cross-system interoperability. The obvious disadvantage is that the triple store bears no resemblance to how software systems get developed, forcing costly translation efforts. The semantic web is also brittle in that it relies on inferential chains that must connect perfectly to provide its usefulness.

Data archetypes don't have much in common with the semantic web architecture, except for the fact that they address data semantics. Within that realm, the differentiation is that the focus of data archetypes is semantic *patterns* rather than straight semantics. What is meant here is that archetypes assign a type and therefore a combination of properties to each data object. For instance, a sampling of external temperature records and a set of overheating events have different data archetypes. This flags a structural difference between these objects, which translates into different methods for interpretation and summarization. However, the same could be said of a sampling of medical patient temperature records and fever events -data archetypes don't capture the fact that meteorological predictions and medicine have different ontologies. What data archetypes do offer is a set of standards for describing structural semantics to facilitate data representation and processing. And in fact, the semantic web community has relatively recently published standards specifically addressing spatiotemporal relationships, such as the Time Ontology in OWL (World Wide Web Consortium, 2017).

Given that archetypes establish a link between information structure and semantics, the next question is to consider how archetypes relate to relational databases' Structured Query Language (SQL). In the introduction we presented data objects as an attempt to marry tabular data and object-oriented programming. In a way, this is reminiscent of the object-relational impedance mismatch that governs object-relational mapping technology. Relational databases remain a mainstay for storing tabular data, and they provide rich semantics and powerful query capabilities to exploit those semantics. Yet the definition of data archetypes introduces concepts that supplement SQL relationships:

- **Schema types**: there is no concept of schema type in SQL, as every schema is completely described by its columns and relationships. A type system may be implicitly created by templating tables, but there is no direct way to leverage templating relationships in applications that consume table data.
- **Data topologies**: data topologies apply to query results, and again exist implicitly in SQL -Table 2 shows the direct mapping between query arguments and topologies. We elaborate on this topic below.
- **Contextual properties**: interestingly, the main example of contextual properties that we provided, i.e. local interval maximality, directly relates to SQL window functions, which are a relatively recent addition to SQL. Again, and thanks to the addition of window functions, SQL makes it possible to query locally maximal data objects such as trips, but the language doesn't explicitly express the existence of such a concept.

The above enumeration reinforces the notion that data archetypes add a layer of semantics on top of tabular data applications: the concepts are already inherent in the use of existing technologies, but only implicitly. The case of data topologies is particularly interesting. Our description of data objects involves storing time intervals and sets of keys as metadata. These are basically the query arguments used to extract the data object from the underlying data set. While this presents the advantage of summing up the structural semantics of the data, it is also quite simplistic in that it only applies to the simplest range queries. We did not attempt to describe what happens with more complex queries, say ones that add attribute filtering, or with chained queries. Another way to look at the data object's metadata is as a trail of data provenance. Seen in this way, that metadata can be captured more completely by expressing it as an SQL query, or a sequence of queries if multiple selections or transformations have been applied. On the flip side, an SQL query encodes a set of procedures but doesn't provide a direct reading on the shape of the resulting projections. For this we would need to apply relational algebra on the query arguments themselves. In cases where the query arguments express a known topology (e.g. a simple time interval), the query result can then be expressed as a data archetype (e.g. a Sequence). In the more general case, the query result is not a known archetype. A complete semantic interpretation of the corresponding data object against the underlying data set (or data sets in case of a join) would require that the query arguments be carried as metadata, but it defies classification. Hence at the end of the day data archetypes are useful to the extent that they are representative of the most common data operations performed in a typical application, and in that sense constitute a system of patterns, in the same way that design patterns leverage the benefits of object-oriented programming.

The last point of discussion is a more open-ended hypothesis regarding the potential value of archetyping for artificial intelligence. Arguably, we cannot speak of intelligence in the absence of semantics. In other words, there is nothing truly intelligent about a neural network that successfully classifies images into predefined buckets -though the design and engineering of that network did require plenty of human ingenuity. On the other hand, if a machine can successfully weave together multiple concepts and make inferences from their arrangement, it at least simulates intelligence. This is what makes natural language processing applications like chatbots or digital assistants impressive. This intuitively suggests that naturally occurring concepts such as aggregations, associations, time ranges or spatial extent need to be baked into the fabric of artificially intelligent systems to interpret and synthesize information about the physical world.

Conclusion

In this paper, we introduced several observations regarding the semantics of tabular data sets, especially as these relate to the physical world -meaning that they pertain to things, people, places and times. These observations suggest a system of types and attributes for tabular data that existing computing tools only handle implicitly. Whether there are benefits to employing the concept of archetypes at either the application design stage or the systems implementation stage is an open question. On the one hand, doing so would not lead to computing performance improvements, and in fact would probably do the opposite. On the other hand, one can predict that information systems are generally moving toward higher levels of abstraction as their scope increases. A software developer in the 1970s could have needed to write assembly code to sort a list of values. Today that developer is more likely to call a method on an instance of a list object. Although the method might not be optimal for the developer's use case, it allows far greater productivity overall -and the method is still quite efficient! As more and more information systems get deployed to model and interact with the physical world, common patterns may eventually be captured in the programming toolkit, with the ultimate benefits of improving design quality, boosting productivity, and facilitating interoperability. For example, time series databases are a relatively new category, at least as a mainstream technology. Likewise, streaming analytics technology is a rapidly evolving field for which several new frameworks have appeared in recent years. But to be fair, the adoption of these newer technologies appears primarily driven by performance considerations -in other words, the framework is beneficial because it handles a certain class of computations very efficiently. On the flip side, a concept like levels of measurements, although profoundly relevant to the description of the physical world, has not been adopted in computing technology. Yet as a last word, we contend that if there is such a discipline as data science, then the concepts presented in this paper ought to be captured by the scientific theory in some shape or form.

References

Aberer, K. (2004). Emergent Semantics Systems. In G. C. Bouzeghoub M., *Semantics of a Networked World. Semantics for Grid Databases. ICSNW 2004. Lecture Notes in Computer Science, vol 3226* (pp. 14-43). Berlin: Springer.

Chrisman, N. (2002). *Exploring Geographic Information Systems.* Wiley.

Confluent. (2020, March 4). *Schema Registry*. Retrieved from confluent.io: https://docs.confluent.io/1.0/schema-registry/docs/intro.html

Confluent. (2020, March 4). *What is Kafka?* Retrieved from Confluent.io: https://www.confluent.io/what-is-apache-kafka

Dori, D. (2002). *Object-Process Methodology: A Holistic Systems Paradigm.* Springer Science & Business Media.

Kleppmann, M. (2017). *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems.* O'Reilly Media, Inc.

Kreps, J. (2013, December 16). *The Log: What every software engineer should know about real-time data's unifying abstraction*. Retrieved from LinkedIn Engineering:

https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying

Lin, C.-C. (2016, August 11). *More Than Just a Schema Store*. Retrieved from Yelp Engineering: https://engineeringblog.yelp.com/2016/08/more-than-just-a-schema-store.html

Llanes, K. R., Casanova, M. A., & Lemus, N. M. (2016). From Sensor Data Streams to Linked Streaming Data: a survey of main approaches. *Journal of Information and Data Management*.

Noura, M., Atiquzzaman, M., & Gaedke, M. (2019). Interoperability in Internet of Things: Taxonomies and Open Challenges. *Mobile Networks and Applications*, 796–809.

Whitehouse, K., Zhao, F., & Liu, J. (2006). Semantic Streams: a Framework for Composable Semantic Interpretation of Sensor Data. In R. K., K. H., & M. F., *Wireless Sensor Networks. EWSN 2006. Lecture Notes in Computer Science, vol 3868.* Berlin: Springer.

World Wide Web Consortium. (2017, October 19). *Time Ontology in OWL.* Retrieved from W3C: https://www.w3.org/TR/owl-time/