CHAPTER 4

A GENERALIZED DATA EXPLORATION (GDE) MODEL

The generalized data exploration (GDE) model is a framework for describing the data exploration process, devoid of any reference to a specific database structural model or data exploration *task environment* (cf. Section 3.1). An instance of the data exploration process is called the *data exploration session*, and in this chapter we develop a formal definition.

The GDE model is built around metadata, data entities and data derivations. Metadata describes numerous aspects of a data set, data entities model the data objects under investigation, and data derivations model the mappings between sets of data entities. Higher-level semantic structures built from collections of data entities and derivations describe different aspects of the data exploration process. *Derivation sequences* emphasize the temporal, linear nature of data exploration. Graph-based structures, *data derivation graphs* and *data lineage graphs*, portray non-linear relationships among data entities.

We begin this chapter by defining the primary structural components of the GDE model: metadata, data entities and data derivations. We then define higher-level structures: sequences that model temporal relationships and graphs that model derivation relationships among data entities. Special constructs and techniques are developed to compose data entities into *congruence* and *similarity* classes, based on their metadata.

New types of derivations and graphs are then developed to accommodate these new constructs. The graph structure is a focal point for the remainder of this thesis.

4.1 Data and Metadata

Before defining the GDE model, we must first define what we mean by *data*, how it is structured, and how it is used. Data is the focus of a scientific inquiry, but is meaningless without additional information to describe it and enable its manipulation. Such descriptive information is called *metadata*.

4.1.1 Data

Data has many definitions, depending on the context in which it is used. Data can be broadly defined, however, as a set of values that approximate some theoretical or physical phenomena. Base data or primary data are terms often used to describe data sets that are the focus of scientific inquiry. Structural data characteristics have been addressed in Treinish (1991, 1992), where the following data attributes are defined: *dimensionality, parameters, data type, rank, mesh structure* and *aggregation*. *Dimension* relates to the number of independent variables. *Parameter* refers to the actual data values themselves that depend on the dimensions. *Data type* describes how the data is stored. *Rank* defines the number of values per parameter. The *mesh structure* defines a base (physical) geometry for data mappings and visualizations. *Aggregation* combines smaller data organizations into larger organizations. Though Treinish's focus is on continuous explicitly-structured physical science data, his concepts are generalizable to discrete, unstructured data sets such as relational databases.

Data can also be described as a function from a domain (physical) space D to a value (approximation) space V; $f : D \rightarrow V$. In Kao, Bergeron and Sparr (1995), a functional definition of a data set on f is given as:

$$S_f = \left\{ \left(x, f'(x)\right) \mid x \in D' \subseteq D \right\},$$

where f' is an approximation of f and D' is finite. The data set S_f is a finite approximation of f', and therefore a finite approximation of f, the mapping from domain to value space. The goal of a scientific inquiry, such as data exploration, is to derive features and characteristics of f from S_f. This work also develops formal definitions for data geometries and storage topologies.

4.1.2 Metadata

Closely associated with the data set are descriptive attributes called *metadata*. Like data, the term metadata is context sensitive, making it highly overloaded and often contentious. Metadata is a summarization, a compacted representation of a data set. This compactness often makes metadata more desirable for manipulation than the data set it describes. For the GDE model, we define four subclasses of metadata, *identification* metadata, *structural* metadata and *process* metadata and *knowledge* metadata (Lee and Grinstein 1995).

Definition 4-1. Metadata

Given a data set S, metadata is a 4-tuple M = (M_I, M_S, M_P, M_K), where M_I contains identification attributes, M_S contains structural attributes, M_P contains process-related attributes and M_K contains knowledge that describes S.

Identification metadata is a name or other identifier that places the data set into a specific context. Identification metadata is very important because metadata is meaningless

without knowing the data it describes. Possible examples of identification metadata include data set name, data set creator, creator affiliation, creator email and internet home page, date created, a timestamp, domain description, and instruments used. *Structural metadata* describes data organizations. Possible structural metadata include data set cardinality, number of data samples per element, data element type, physical storage structure, connectivity between samples and data element access interfaces. *Process metadata* describes actions such as query or visualization activations, and aggregates of these actions. Some simple examples of process metadata include execution time, resource utilizations, frequency count of usage and annotations describing insights gained into the data set. *Knowledge metadata* contains descriptions of data set *values*, such as statistical summaries, mathematical models that fit the data or analyst insights and observations.

Since metadata is a form of data, it has its own internal structure, or *format*, in addition to its values. This metadata structure is different than structural metadata, which describes the structure of some data set. We will use the term *metadata format* to mitigate the ambiguities that may arise when talking about the structure of metadata. Because metadata is inexorably tied to the data it describes, its format is highly data-dependent. Thus, Definition 4.4 is purposely vague, and actual metadata format can be defined once the data it describes is known.

4.1.3 Metadata Precision

Just as data sets are approximations to some theoretical or physical phenomena, metadata is an approximation to a data set, and therefore has a certain degree of precision. For example, consider a table of values that can be defined by a textual database query statement. The query statement (identification and process metadata) identifies attributes retrieved, the database tables referenced, and possibly conditions that hold true among

certain attributes. Additional statistical (knowledge) metadata, such as attribute value distributions, can be determined from the table of values. Both metadata descriptions (query text and statistical information) describe the same data, but the statistical version is possibly more precise than the textual version, because it may contain additional information about the distribution of data values, and exact boundaries on the value ranges not expressed in the query. The query might indicate that some attribute $x \geq 0$, while the statistic might show $x \geq 1.5$. Thus, a data set may have several redundant metadata descriptors associated with it, of varying degrees of precision.

Metadata precision affects whether or not changes to metadata are reflected in the data being described, or vice-versa. In the previous example, the query text representing identification metadata may be changed to more accurately reflect the value distributions representing structural metadata, while still describing the same table of values. Whether or not this is desired by the analyst is a separate issue. If the query is altered such that it describes a new table of values, then the structural metadata is likely to change, and this likelihood is affected by the metadata precision. In general, when data being described is altered, metadata that is more precise (such as the statistical information above) is more likely to change than metadata that is less precise (such as the query text above).

Metadata precision also affects comparisons made between different metadata entities. Not only might the metadata representation be different (a textual query vs. a numeric summary, for example), the precision of the representation might also be different. Thus, comparisons are not always straightforward, possibly requiring translation between representations, and resolution of the precision between representations. Comparisons based on metadata entities may ultimately require comparisons between the data sets they describe.

Metadata can be analyzed as data. Consider a data set described by a wavelet. The wavelet is a summary or abstraction of the data set, but it can also be analyzed and visualized like the data it describes. Its precision affects the accuracy of further analyses.

The mechanisms and effects of metadata precision are open problems, and are not within the scope of this research. This research only points to places in the analysis of data exploration sessions where metadata precision plays an important role.

4.1.3 Fundamental Metadata Relationships

Two metadata entities are equal if each of their elements are equal. This is a very strict and limiting definition, since two distinct metadata entities can describe the same data set, but with different precision and having different formats. Furthermore, it may be desirable to allow metadata values to be within some predefined range, and consider them to be in the same *equivalence class*.

Thus, to allow more flexibility in comparing metadata entities, we define two equivalence classes of metadata, *congruence classes* and *similarity classes*. These classes are related in their structural and knowledge metadata only. Identification and process metadata are not considered in the comparison, because they add more external context to the data set, and do not describe its internal structure.

A *congruence class* of metadata describes a set of metadata entities whose metadata are *congruent*. Two metadata entities are congruent if their structural metadata is equal, their knowledge metadata has the same format and has values within some fixed range. An example of structural metadata being equal is if two distinct query results possess the same attributes. An example of knowledge metadata having the same format (i.e., structural equality) is if two distinct statistical summaries contain the same types of

information, such as table size, mean and standard deviation. An example of having metadata values within a fixed range is if the table size is between 1000 and 2000 tuples, the mean of each attribute is between +/-100.0, and the standard deviation of each attribute is between 5.0 and 10.0. Congruence is written as $M_1 \cong M_2$ *iff* $M_{1S} = M_{2S} \wedge M_{1K} \cong M_{2K}$. Congruence, as we have defined it, is an equivalence relation because structural metadata being equal is an equivalence relation, knowledge metadata having the same format is an equivalence relation, and knowledge metadata having values equal to a fixed range of values (i.e., a predetermined set of values) is also an equivalence relation.

A *similarity class* of metadata describes a set of metadata entities whose metadata are *similar*. Two metadata entities are similar if their structural metadata is equal, their knowledge metadata has some format in common, and where knowledge metadata formats are common, knowledge metadata values may be within some fixed range. Similarity is written $M_1 \approx M_2$ *iff* $M_{1S} = M_{2S} \wedge M_{1K} \approx M_{2K}$. For the same reasoning as congruence, similarity is an equivalence relation. An example of two similar data entities would be if one had additional knowledge metadata describing some data set than the other, such as if one metadata entity contained the mean of a data set while another metadata entity contained the mean and standard deviation of the same data set.

4.2 Data Entities

At its core, the GDE model is built around the *data entity*, an abstract data type that contains a data set, a key value and associated metadata. In this section, we define three types of data entities: *d-entities*, *c-entities* and *s-entities*. They differ in how accurately their metadata describes the data set. The most accurate description is given by the d-entity.

Definition 4.2a - d-entity:

*A **d-entity** is a triple: e_d = (S, k, M), where S is a data set, k is a key value and M is metadata describing S.*

The data set *S* may consist of a collection of data points, either observed, sampled, or computed, in some value space that approximates a domain space, as defined in Kao, Bergeron and Sparr (1995). Associated with the data set is key value used as a name for the data entity. Finally, the metadata object *M* contains the identification, structural, process and knowledge metadata describing *S*.

One example of a d-entity is a database query result, which is a table of values. The data elements would be a set of tuples containing attributes that could be, among other things, textual, numeric, categoric, decimal (for currency) and temporal (a timestamp). Decimal and temporal data types are common in most commercial database systems. Metadata as found in the database system catalog could include items such as attribute names and types. Additional metadata that could be computed might include cardinality (the number of tuples), ranges and statistics for numeric attributes (as is done in *Exbase* (see Section 3.3.3.3)) and frequency counts for categoric attributes. A special kind of data entity is the *database*, which can be thought of as a universal collection of data sets, and is labeled E_U. The database has an arbitrarily-defined structure.

Another example of a d-entity is a data visualization. The data set is a table of values. The metadata, describing how the data is displayed, contains the type of visual representation, a mapping specification between each data attribute and each visual primitive in the representation, and a set of visualization display settings (as is done in *Exbase* (see Section 3.3.4)). The representation type is often based on the type of data to be displayed, or task to be performed, such as an isosurface that highlights a single

(nonspatial) value in the volumetric data set. The data-to-visual mapping is often two parallel arrays, matching attribute identifiers with visual primitive identifiers. The visualization display settings are a set of configurations for aspects of the display, such as colormaps, reference locations, lengths and angles, spatial randomness, etc.

Two d-entities are equal if all of their elements are equal. In database exploration, d-entities may contain large and complex data sets, making such comparisons unattractive to perform in real-time, as the d-entities are created. Furthermore, in the data exploration context, this definition is too limiting. As mentioned earlier, it is possible that two *data sets* may be equal but have different metadata descriptions, or vice-versa. It would be advantageous to compare d-entities based on their metadata alone, sacrificing precision in computing the comparison for speed of computation and the ability to infer associations among d-entities.

Two d-entities are *congruent* if their metadata are congruent. Thus, d-entities can be collected into congruence classes, based on their metadata. This will become useful in capturing certain qualities of the data exploration session that are discussed in Section 4.5.2. Congruence classes can be represented by the d-entity data structure in what we call *c-entities*, where the data set has a specific structure common to the class (structural metadata is equal) and values within some fixed range. The metadata defines the fixed ranges of values.

Definition 4.2b - c-entity:

Given a set of d-entities E_d that are congruent, a ***c-entity*** *is a triple $e_c = (S_c, k_c, M_c)$, where S_c is a set $\ni \forall\ e_d = (S_d, k_d, M_d) \in E_d$, $S_d \in S_c$, k_c is a key value and M_c is describing S_c, where M_I contains a set of key values $K_c \ni \forall\ e_d \in E_d$, $k_d \in K_c$, and the M_K format is expressed as fixed ranges of values.*

Two data entities are *similar* if their metadata are similar, and similarity classes can be composed as well. Similarity classes can also be represented by the d-entity structure, in what we call *s-entities*, where the data set has a specific structure common to the class (structural metadata is equal) and values within some acceptable range. The metadata defines, for each member of the similarity class, the portion of the knowledge metadata format that overlaps, and the fixed range of values for the overlapping portions.

Definition 4.2c - s-entity:

Given a set of d-entities E_d that are similar, an ***s-entity*** *is a triple $e_s = (S_s, k_s, M_s)$, where S_s is a set ∋ ∀ $e_d = (S_d, k_d, M_d) \in E_d$, $S_d \in S_s$, k_s is a key value and M_s is describing S_s, where M_I contains a set of key values K_s ∋ ∀ $e_d \in E_d$, $k_d \in K_s$, and the M_K format is expressed as $\bigcup_{i=1}^{|K_s|} M_{iK}$, with fixed ranges of values.*

In both c-entities and s-entities, the data set *S* is a set whose elements are the data sets of d-entities that are congruent and similar, respectively. The metadata of these new types of data entities contains information such as the keys of all congruent or similar d-entities, and the parameters of the congruence or similarity (within the knowledge metadata), such that individual d-entities can be identified in the future, if necessary.

4.3 Data Derivations and Sequences

In the GDE model, the fundamental data exploration task is a *data derivation* operation, applied by the analyst, on a set of data entities. In general, a data derivation is an operation that *derives* a set of data entities from some other set of data entities in some finite amount of time. In this section we first define the fundamental data derivation, the *d-derivation*, and then the *d-derivation sequence* which is the basis for the *data exploration*

session. Based on the congruence and similarity classes of d-entities, we then define *c-derivations* and *s-derivations*, respectively.

4.3.1 D-Derivations

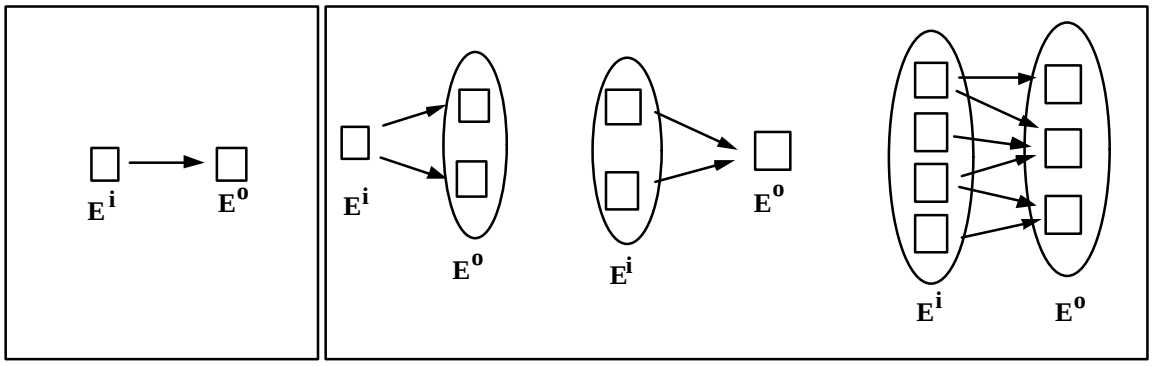
The d-derivation is the fundamental type of data derivation that models the mapping of some input set of d-entities to some output set of d-entities.

Definition 4.3 - d-derivation:
Given two sets of data entities E^i and E^o where $E^i \cap E^o = \varnothing$, a ***d-derivation*** *is a triple d_d = (S, t, M), where S is a binary relation over $E^i \times E^o$, $t \in \mathbf{N} > 0$ is a timestamp value and M is metadata describing S.*

In a d-derivation, the data set *S* is restricted to be a set of ordered pairs of d-entities that is a mapping from a set of input d-entities E^i, called the *preimage* of *S* , to a set of output d-entities E^o, called the *image* of *S*. The timestamp *t* serves to order the derivation with respect to time, such that the d-derivation can be identified by the timestamp *t* in the following manner: d(t). Metadata can describe the mapping, such as derivation issuer, execution time and derivation structure (e.g., a parse tree or expression tree), etc.

The sets E^i and E^o are disjoint because timestamps that are part of d-entity identification metadata complicate comparisons. Specifically, if a d-derivation were to reproduce its input as output, then the timestamps on input and output would also have to be equal. Since in practice the d-derivation requires some minimum execution time (i.e., the time required for the computer to perform the action specified by the d-derivation), the timestamps on input and output data entities *must* be different. This is one of two

important restrictions on d-derivations imposed by timestamps. The second restriction is discussed in Section 4.3.2. Figure 4-1 shows several kinds of d-derivations graphically.



a) simple derivation **b) compound derivations**

Figure 4-1. Various kinds of d-derivation mappings.

The two main types of d-derivations shown in Figure 4-1 are *simple* and *compound*. In simple d-derivations, the input and output sets of data entities are singletons. In compound d-derivations, the input and output sets can have multiple members. The d-derivation is not necessarily a *single-valued mapping*, because one input can derive multiple outputs (such as a visualization transform that simultaneously produces multiple, linked visualizations).

The image E^o of any d-derivation is also called the *data exploration state*. The d-derivation is responsible for changing the data exploration state by producing a new set of data entities for the analyst to focus on. This updating of the data exploration state is consequence of the d-derivation, so the derivation is said to be an *implicit state change*.

4.3.2 The D-Derivation Sequence

In the chapter introduction, we defined the *data exploration session* as an instance of the data exploration process. We can formalize this definition by using the data objects defined thus far: the data exploration session can now be defined as a temporally-ordered set (i.e., a sequence) of d-derivations that operate over a set of d-entities. Thus, the data exploration session is the *d-derivation sequence*.

Definition 4.4 - d-derivation sequence:
Given a database E_U, and a set of d-entities E_d, a ***d-derivation sequence*** *is a sequence of d-derivations* $\Theta_d = \langle \mathrm{d}(\mathrm{t}_1), \mathrm{d}(\mathrm{t}_2), \ \ldots \ \mathrm{d}(\mathrm{t}_1) \rangle$, $\ni \forall\ d(t_n) = (S_d, t_n, M_d)\ (1 \le n \le l)$, $\forall(e^i, e^o) \in S_d, e^i \in E_d$ *where* $t_{e^i} < t_n \vee e^i = E_U$, *where* t_{e^i} *is the timestamp on* e^i.

The d-derivation sequence emphasizes the linear progression of data entities, from unrefined base data in the database, to more evolved d-entities that can be interpreted as knowledge about the database. This sequential representation focuses on a set-of-data-entity view of the data exploration session, as data entities belonging a d-derivations image or preimage are clustered together. During the session, data sets are refined and knowledge is extracted. Such sequences are often used in practice in log files for operating systems, database management systems and world wide web browsers. The Exbase system described in Chapter 3 also uses a sequential log of derivations.

For the same reason that individual d-derivations cannot produce outputs that are equal to their inputs (the uniqueness of timestamp values), d-derivation sequences *cannot produce d-entities that already exist*. This is the second restriction that d-entity timestamps impose on the GDE model. In accepting this restriction, each d-derivation output is still unique, regardless of the data set or metadata contained within the d-entity. In the following section, we devise a methodology to address these restrictions.

Having the data exploration session (i.e., the d-derivation sequence) defined places all d-entities and d-derivations into a well-defined temporal context. Data entities can now be logically grouped with respect to time into sets having a specific meaning aside from congruence or similarity equivalence relationships. We already have the concepts of derivation preimages and images. Other special sets of d-entities, based on their timestamp values, with respect to some d-derivation sequence over a set of d-entities E_d include

E $(\lfloor t \rfloor)$ - the set of all $e \in E_d$ having timestamp $\leq$ t.

E $(\lceil t \rceil)$ - the set of all $e \in E_d$ having timestamp $\geq$ t.

E (t) - the set of all $e \in E_d$ having timestamp = t.

E [t_1 .. t_2] - the set of all $e \in E_d$ having timestamp $t_1 \leq t \leq t_2$.

Note we have adapted the floor and ceiling functions to partition a set of data entities based on the timestamp value.

In general, data entities appearing further down in the sequence (i.e., those having higher timestamp values) would be expected to be more processed, and therefore potentially closer to discovered knowledge. This is not always true, however, because at any given time, the analyst may perform derivations whose preimages include data entities from the very beginning of the sequence. This new derivation would be appended to the end of the sequence, though its result might not be very refined with respect to data entities occurring before it in the sequence.

Any type of d-derivation whose preimage does not include the entire image of the previous d-derivation is said to contain an *explicit state change*. In such cases, the analyst abandons all or part of a derivation output by deriving from some other data entity or set of data entities. For example, consider the following d-derivation sequence consisting of

fourteen d-derivations, starting from a database labeled E_U. For illustration purposes, each d-derivation is issued one second after the previous derivation. Figure 4-2 shows the sequence graphically; both are read left-to-right, and then top-to-bottom.

$$\langle \big(\{(\mathrm{E}_U, e1)\},1, M_1\big), \big(\{(e1, e2)\},2, M_2\big), \big(\{(\mathrm{E}_U, e3)\},3, M_3\big), \big(\{(e3, e4)\},4, M_4\big),$$
$$\big(\{(e4, e5)\},5, M_5\big), \big(\{(e2, e6),(e4, e6)\},6, M_6\big), \big(\{(e3, e7)\},7, M_7\big),$$
$$\big(\{(e3, e8)\},8, M_8\big), \big(\{(e8, e9)\},9, M_9\big), \big(\{(e7, e10)\},10, M_{10}\big), \big(\{(e10, e11)\},11, M_{11}\big),$$
$$\big(\{(e11, e12)\},12, M_{12}\big), \big(\{(e9, e13)\},13, M_{13}\big), \big(\{(e13, e14)\},14, M_{14}\big)\rangle.$$

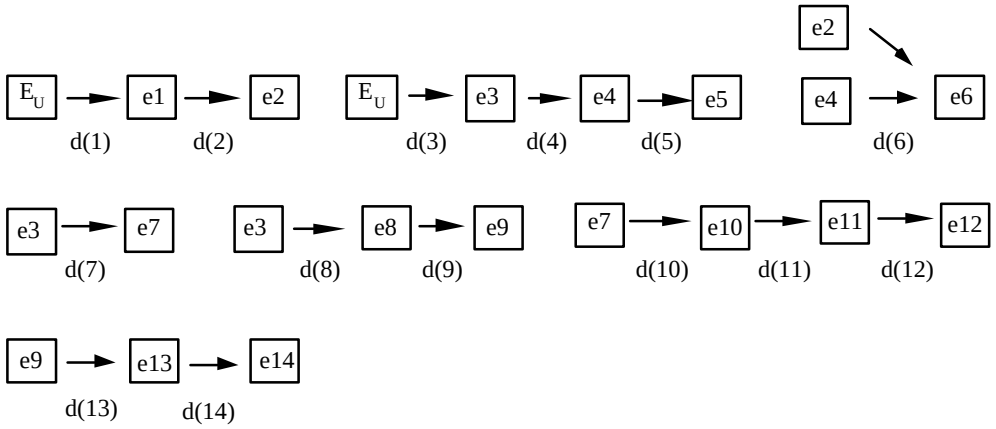


Figure 4-2. A graphical representation of a d-derivation sequence.

Some d-derivation inputs consist solely of the database, such as d(1) and d(3). Other d-derivation inputs consist of the image of the immediate predecessor element in the derivation sequence, such as d(2), d(4) and d(5). Still other d-derivation inputs are *not* from the immediate predecessor d-derivation in the sequence (i.e., explicit state changes), such as d(3), d(6) and d(7).

Accounting for explicit state changes in the session representation requires a graph-based solution to eliminate the redundancies in derivation inputs (such as E_U and e3 in Figure 4-2). *Data derivation graphs*, described in Section 4.3.3, address this issue.

Before we define them, however, we must define two other kinds of derivations over data entity congruence and similarity classes that data derivation graphs rely on.

4.3.3 C- / S- Derivations and Sequences

Because congruence and similarity classes may be created during the data exploration session, they must be accounted for in the GDE model. It might be beneficial to collect data entities that are "almost equal" without regard for temporal and other environmental information, thereby allowing certain kinds of derivations to have "self-loops", or certain kinds of derivation sequences to have "back edges". To do this, we first draw a correspondence between c-entities and d-entities, in the context of a data exploration session.

> *Given a set of d-entities E_d belonging to some data exploration session (cf. Definition 4.4), there exists a set of c-entities E_c corresponding to E_d such that* $\forall e_c = (S_c, k_c, M_c) \in E_c, \forall e \in S_c, e \in E_d$, *that are the set of congruence classes with respect to E_d.*

A similar correspondence can be drawn between s-entities and d-entities. This says that each element of each set of congruence (similarity) classes on a set of d-entities has a corresponding d-entity instance in the set of d-entities. Having this correspondence, we can now define new derivations that involve congruence (c-derivations) and similarity classes (s-derivations).

Definition 4.5a - c-derivation:

Given a set of c-entities E_c, a ***c-derivation*** *is a triple d_c = (S, t, M) where S is a binary relation over $E_c \times E_c$, t is a timestamp and M is metadata describing S.*

Definition 4.5(a) states that c-derivations contain data sets where each ordered pair contains two c-entities. The definition for s-derivations is very similar.

Definition 4.5b - s-derivation:

Given a set of s-entities E_s, an ***s-derivation*** *is a triple d_s = (S, t, M) where S is a binary relation over $E_s \times E_s$, t is a timestamp and M is metadata describing S.*

Having defined c-derivations and s-derivations, we can define a correspondence between c/s-derivations and d-derivations, in the same manner as we did for data entities above. We do this for c-derivations only.

> *Given a set of d-entities E_d belonging to some data exploration session (cf. Definition 4.4), a set of corresponding c-entities E_c , and a set of d-derivations D_d on E_d, there exists a set of c-derivations on E_c such that $\forall d_c = (S_c, t_c, M_c) \in D_c$, $\forall (e^i, e^o) \in S_c$, $(e^i, e^o) \in S_d$, in some d_d = (S_d, t_d, M_d) $\in D_d$, that are the set of congruence derivations with respect to D_d.*

As before, the same general definition can be used for similarity derivations with respect to a d-derivation sequence. Essentially, these correspondence definitions allow us to relax the criteria for collecting d-entities, while preserving the same amount of information about the session. Thus, we can view a single data exploration session as a d-derivation sequence, a c-derivation sequence, or an s-derivation sequence without losing any information. The sets that make up the data entity congruence classes can be singletons, so any d-entity not collected into a congruence class with other d-entities can be made into a congruence class unto itself.

Creating congruence classes and derivations are required for creating the data derivation graphs that are defined in the next section.

4.4 Data Derivation Graphs

In this section, we define alternate representations of the data exploration session: *d-graphs* that only contain d-derivations, *c-graphs* that only contain c-derivations and *s-graphs* that only contain s-derivations. D-graphs are useful for modeling a simplistic view of the data exploration session, where there is no attempt to collect d-entities into congruence or similarity classes; each data entity is uniquely defined by its timestamp. C-graphs model data exploration sessions based on congruence classes and s-graphs model sessions based on similarity classes. We also define three special-case *component graphs* that are defined over s-graphs: *forward, backward* and *identity*.

4.4.1 D-Graphs

The reader will find it easier to comprehend the graphical representation of Figure 4-2, shown in Figure 4-4, than its sequential listing. The *d-graph* is a collection of vertices representing d-entities, and directed edges representing d-derivations, emphasizing relationships between individual pairs of data entities.

Definition 4.6 - d-graph:
Given a d-derivation sequence Θ_d*, a* ***d-graph*** *is a connected, directed graph* $G_d = (\boldsymbol{V}, \boldsymbol{E})$*, where* $\forall\ d_d = (S, t, M) \in \Theta_d \wedge \forall (e^i, e^o) \in S,\ e^i \in \boldsymbol{V} \wedge e^o \in \boldsymbol{V} \wedge (e^i, e^o) \in \boldsymbol{E}$.

In the d-graph, d-entities are represented by graph vertices, and d-derivation mappings are represented by directed graph edges. The d-graph contains a special vertex to represent the database, the *source vertex*. Though in general, a graph can consist of a single vertex, we do not allow the d-derivation graph to consist solely of the source vertex, because it is supposed to show the derivations that make up the data exploration session. We also require the d-graph to be *connected* because derivations overlap each

other, i.e., they share data entities, and because the existence of any d-entity implies the existence of a sequence of derivations to produce it. Figure 4-3 shows the d-derivation sequence of Figure 4-2 as a d-graph.

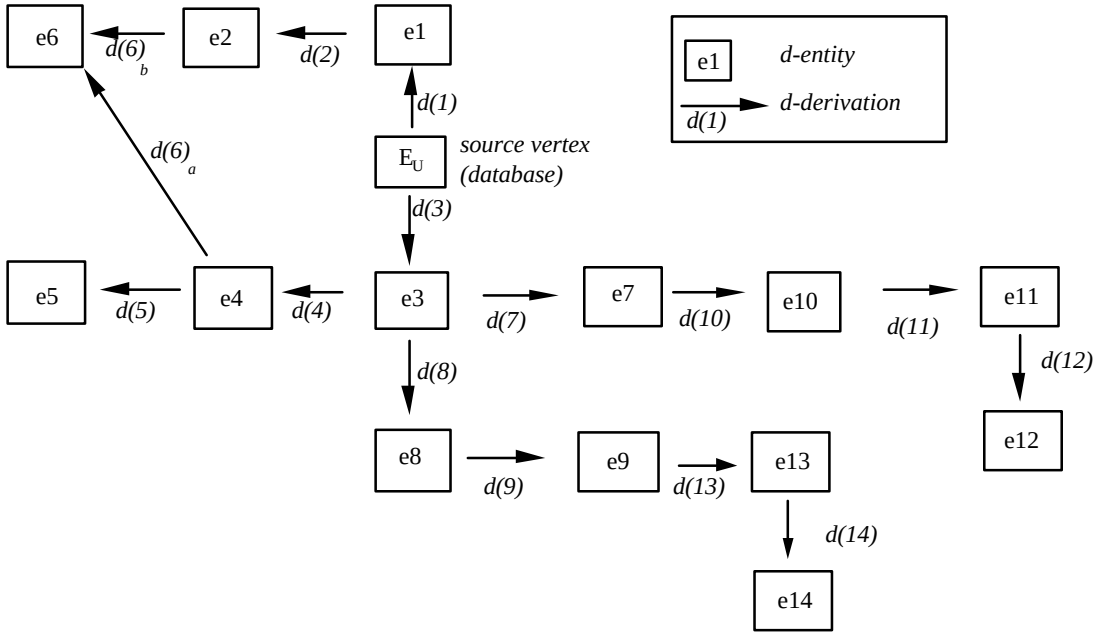


Figure 4-3. Graphical representation of a d-graph.

In Figure 4-3, the d-graph shows d-entity vertices connected by directed d-derivation edges. Vertices may contain multiple incident and exiting edges; the source vertex representing the data origin of the session only contains exiting edges. A *terminal vertex* contains no exiting edges, such as e5 and e6. The derivation d(6) contains two input data entities, e2 and e4. Derivations d(4), d(7) and d(8) are all distinct derivations over data entity e3. No derivations are shown producing multiple output data entities. Explicit state changes are readily apparent in this representation.

In terms of the d-graph, d-derivations cannot create “self-loops” and d-derivation sequences cannot create “back edges”, because of the restrictions imposed by timestamps.

Thus, important information about the session is hidden within d-entity instances. These types of graph patterns occur for two reasons: (1) the analyst knowingly applies a derivation that will create either structure, or (2) the analyst does not know that a derivation will produce either structure. In the first case, the analyst "knows" more that the computer, *a priori*, and might be able to direct the computer to re-organize the graph. In the second case, the computer must determine the relationship after the derivation has been issued (most likely during a post-processing session graph analysis phase). Determining both cases via the computer, however, is important.

4.4.2 C-Graphs and S-Graphs

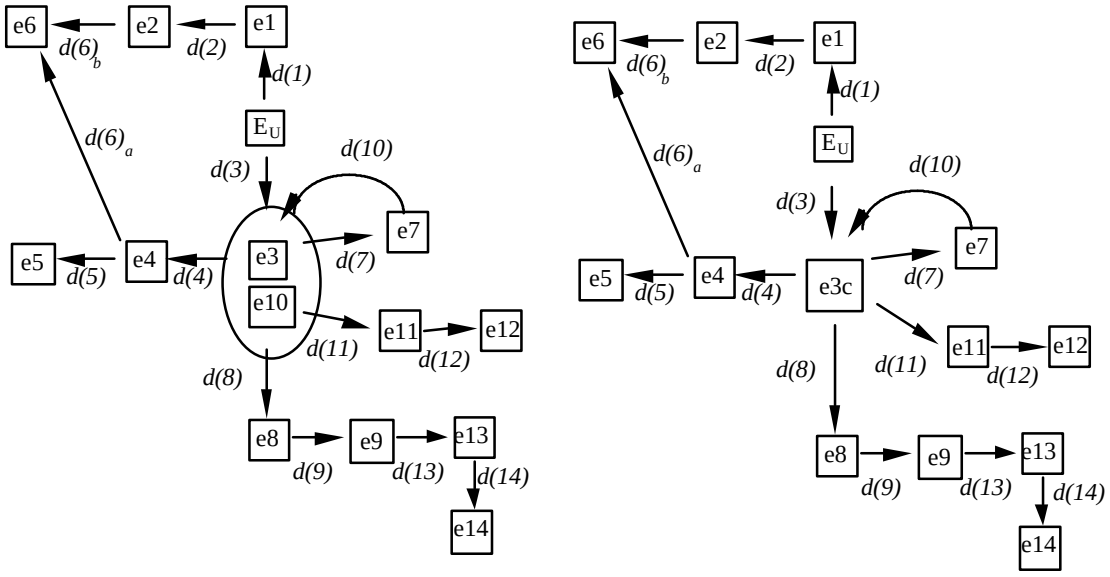
In this section, we describe how to *congruence graphs* (c-graphs) and *similarity graphs* (s-graphs) are constructed from a d-graph. Congruence and similarity graphs explicitly represent the looping and cyclic structures that are part of the data exploration session. We use the congruence and similarity classes of data entities to collect d-entities that are determined to be congruent or similar, respectively.

For example, suppose in the graph of Figure 4-3 the sequence <d(7), d(10)> are data scaling operations that progressively increase. Suppose derivation d(11) then performs a scaling that is *almost* the inverse of the previous sequence, such that e11 has the same structural metadata and knowledge metadata format as e3, but knowledge metadata values within some (very small) fixed range that also satisfies those in e3. We can create a congruence class that contains e3 and e11, and add a back edge (a c-derivation) from e10 to e3 in the new graph.

In addition, suppose that derivation d(14) does not modify its input data set in an way; it only creates additional knowledge metadata (e.g., statistics) that were not

computed before. By creating a similarity class containing e13 and e14, the new graph will now have a self-loop (an s-derivation) at e13.

By collecting congruence classes, the c-graph version of the d-graph can be created. This is based on user directives, algorithmic inference, or both. The net result is shown in Figure 4-4, where d-entities e3 and e10 are collected into a congruence class having two elements. All other congruence classes are singletons and all derivations are c-derivations.



a) collecting congruence classes **b) compressing the graph**

Figure 4-4. Creating a congruence graph.

In Figure 4-4(a) e3 and e10 are collected into a congruence class, and e10's subgraph is moved to branch off the new congruence class. In Figure 4-4(b) the congruence class is replaced by a new c-entity vertex (e3c) to create the c-graph.

In Figure 4-5, the similarity graph is created from the c-graph (though it could have been done directly from the d-graph) by collecting vertices e13 and e14 into a similarity class. All other similarity classes are singletons and all derivations are now s-derivations.

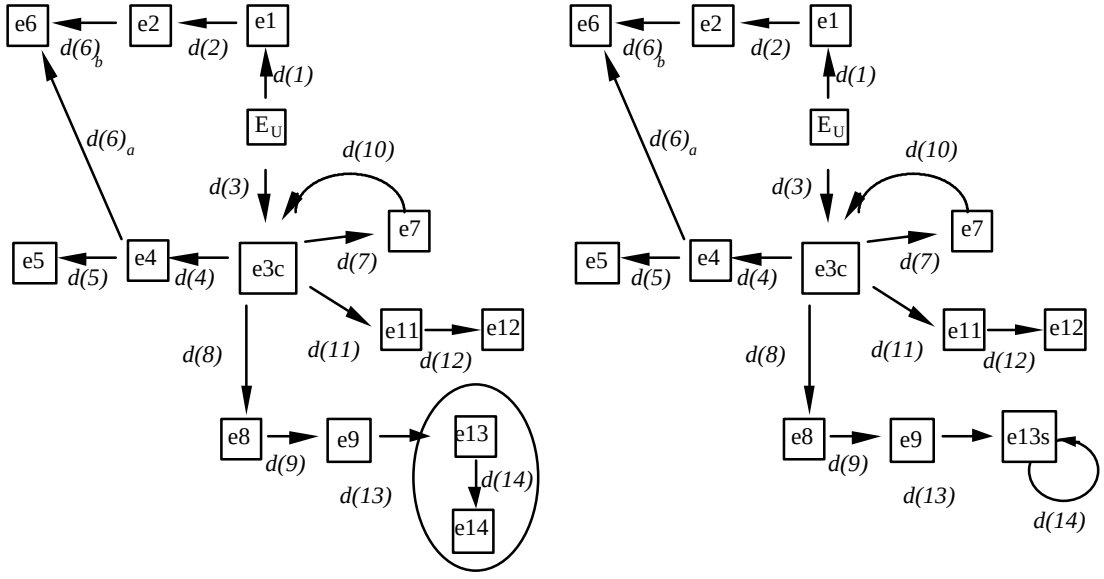


Figure 4-5. Creating a similarity graph.

In Figure 4-5(a) the similarity class containing vertices e13 and e14 also contains a derivation between them. This is preserved as a self-loop in Figure 4-5(b).

The definitions for c-graph and s-graph are very straightforward provided the congruence and similarity classes have been collected, respectively.

Definition 4.7 - congruence graph (c-graph):

Given a set of c-entities E_c and a set of c-derivations D_c corresponding to a d-derivation sequence Θ_d, a ***c-graph*** *is a connected, directed graph G_c = (**V**, **E**) where **V** = E_c and* $\exists(v_1, v_2) \in E_c, v_1 \in V \wedge v_2 \in V$.

Definition 4.8 - similarity graph (s-graph):

Given a set of s-entities E_s and a set of s-derivations D_s corresponding to a d-derivation sequence Θ_d, an ***s-graph*** *is a connected, directed graph G_s = (**V**, **E**) where **V** = E_s and* $\exists(v_1, v_2) \in E_s, v_1 \in V \wedge v_2 \in V$.

The d-graph is the most restrictive graph, while the c-graph is less restrictive and the s-graph is the least restrictive in terms of the precision at which data entities model their data sets. Explicit state changes, back edges and self loops are made apparent in c- and s-graph in return for this relaxed precision in comparing data entities.

4.4.3 Derivation Component Graphs

We can distinguish three types of *derivation components* of compound derivations (cf. Figure 4-1) that are meaningful in the context of the data exploration session: *forward*, *backward* and *identity* components. A forward derivation component is a subset of a derivation's data set where the image is not congruent or similar to any existing data entities in the session. A backward derivation component is a subset of a derivation's data set where the image is congruent or similar to some existing data entities that are not part of the derivation preimage. An identity derivation component is a subset of a derivation's data set where the image is congruent or similar to the preimage. Thus, a compound derivation can contain all three types of derivation components, in general.

In the following definitions, we use the term *derivation graph* to mean the d-graphs, c-graphs and s-graphs, collectively, and *derivation* to mean d-derivation, c-derivation and s-derivation, collectively, as a notational convenience.

Definition 4.9a - forward derivation component:

*Given a derivation graph G = (**V**, **E**), a **forward derivation component** of a derivation d = (S, t, M) ∈ **E** is a subset of S where for each $e^o \in E^o$, $e^o \notin E(\lfloor t \rfloor)$.*

The forward derivation component creates data entities that were not seen by the analyst yet. They are most important, because they show the "forward progress" of the exploration. The timestamp on the data entity is used to determine forward derivation components for simple c-derivations and s-derivations. For c-entities and s-entities that represent multiple data entities, the timestamp can be equal to the minimum timestamp of all instances of the class.

All derivation components of a d-derivation (Definition 4-3) are forward. The remaining two types of derivation components are defined over compound c- and s-derivations, those that model self-loops and back-edges in the c- or s-graphs.

Definition 4.9b - backward derivation component:

*Given a derivation graph G = (**V**, **E**), a **backward derivation component** of a derivation d = (S, t, M) ∈ **E** is a subset of S where for each $e^o \in E^o$, $e^o \in E(\lfloor t \rfloor) - E^i$.*

The backward derivation component recreates some existing data entities that are not part of the input to the derivation. Again, the timestamp is useful in determining backward derivation components. Backward derivation components may or may not have been consciously planned by the analyst. The d-graph has no backward derivation components.

Definition 4.9c - identity derivation component:

*Given a derivation graph G = (**V**, **E**), a **identity derivation component** of a derivation d = (S, t, M) ∈ **E** is a subset of S where for each* $e^o \in E^o$, $\exists\, e^i \in E^i \ni e^o \cong e^i \vee e^o \approx e^i$.

Identity derivation components do not utilize timestamps, but rather the definitions of congruence and similarity in their determinations. As with the backward derivation, identity derivation components may or may not have been consciously planned by the analyst. The d-graph has no identity derivation components.

Derivation component graphs can be constructed to show all the forward, backward and identity derivations that occurred during the session. Of these, we focus on the *forward derivation component graph*, or G_{fwd}, as it shows the "essence" of the data exploration session, a summarization showing only those data derivation components that proceed towards the isolation of important data for the analyst.

Definition 4.10 - forward derivation component graph:

*Given a derivation graph G = (**V**, **E**), a **forward derivation component graph** is a directed graph* $G_{fwd} = (\mathbf{V_f}, \mathbf{E_f})$, *where* $\mathbf{V_f} = \mathbf{V}$ *and* $\mathbf{E_f} \subset \mathbf{E}$ *where each data derivation* $d_f = (S, t, M) \in \mathbf{E_f}$ *is a forward derivation component of some derivation d ∈ **E**.*

The forward derivation component graph contains all of the data entity vertices in the generalized derivation graph since only forward derivation components create new data entities. It only shows a subset of all derivation components, however, so the number of edges are fewer. This has the effect of simplifying the c- or s-graph, to potentially make it more comprehensible. Figure 4-6 shows a graphical representation of the forward derivation component graph of the s-graph given in Figure 4-5(b). The vertex and edge labeling scheme is similar to that earlier example.

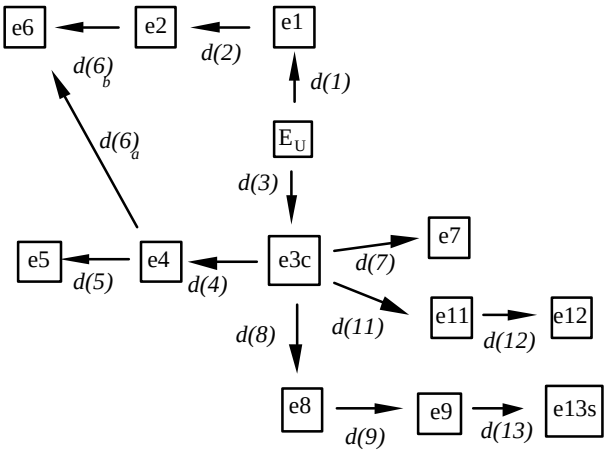


Figure 4-6. A simple forward derivation component graph.

The forward derivation component graph of Figure 4-6 is very similar to the s-graph of Figure 4-5(b), except that edges creating cycles and self loops are eliminated. Creating the forward derivation component graph has the effect of representing some implicit state changes (self-loops and back edges) as explicit state changes. For example, derivation d(10) in Figure 4-5 was represented as a back edge, and thus was an implicit state change. In Figure 4-6, edge d(10) was removed, so we know nothing about the derivation. By analyzing the metadata associated with e3c, we only know that e3c somehow resulted from deriving from e7. Thus, there had to be an explicit state change because there was no actual derivation edge.

The forward derivation component graph representation causes some information to be lost, specifically loops and back edges. In most cases, the data lost may be considered “noise” that interferes with the otherwise logical progression of isolating, identifying, analyzing and describing the data being explored. On the other hand, this noise may be very useful information in describing the experience of the analyst with the exploration environment and the data. It might also say something about the data itself - it

may be an easy data set to explore, or a difficult one, depending on how many cycles or self-loops are observed. This information is extracted from the forward derivation component graph in Chapter 5.

Identity and *backward derivation component graphs* restrict the c- or s-graph to identity and backward components, respectively, creating a forest of subgraphs showing places in the session where the analyst re-derived some data entity equivalence class. These component graphs show the implicit cycles in the session, which may indicate some noteworthy issue concerning the analyst, data, tool, etc. They do not show, however, all data entities, as does the forward derivation component graph, because they do not show the creation of new data entities. Thus, they are limited in their expressiveness.

The forward derivation component graph is actually a directed acyclic graph, or DAG, because all self-loops and back edges are omitted from the c- or s-graph. The forward derivation component graph is *weakly connected* because each pair of vertices is joined by a *semipath* (Harary, 1969). A semipath requires the definition of a *path*, an important construct that helps to determine reachability and evaluate certain costs among graph elements.

A path of length k from a vertex u to a vertex u' in a graph G = (V, E), where V is a set of vertices and E is a set of edges, is a sequence $\langle v_0, v_1, \ldots, v_k \rangle$, $v_i \in V$ of vertices such that $u = v_0$, $u' = v_k$, and the edge (v_{i-1}, v_i) exists for i = 1, 2, ... , k. Vertex u is called the start vertex, vertex u' is called the terminal vertex, and each vertex is distinct.

A *semipath* in the graph is a path in which each edge may be either (v_{i-1}, v_i), or (v_i, v_{i-1}). This relaxes the concept of direction in the path, such that two vertices may be reachable from some intermediate vertex in the semipath (which is common in the forward derivation component graph). Figure 4-7 shows graphical representations of a path and

two semipaths in a graph. The semipath of Figure 4-7(c) is a structure that is common in the forward derivation component graph, where the semipath is configured such that all vertices are reachable from a single vertex along the semipath.

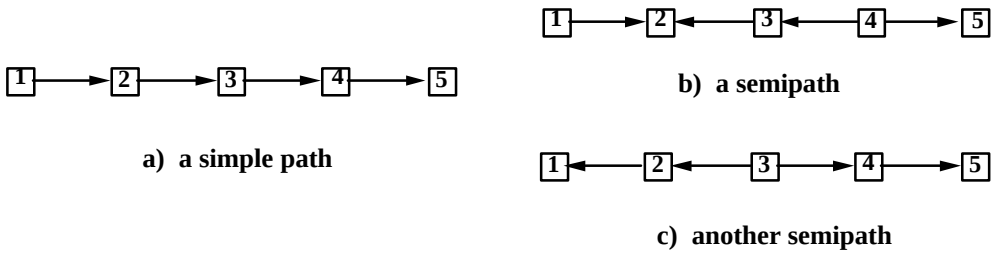


Figure 4-7. Graphical representations of a path and two semipaths.

Forward derivations are chained together to form paths, and explicit state changes give rise to semipaths. The precedence relationships exhibited by forward derivations are embodied in the terms *predecessor* and *successor* vertices.

Definition 4-11a - predecessor vertex:

Given a forward derivation component graph $G_{fwd} = (V_f, E_f)$*, and two vertices* $v_1, v_2 \in V_f$*,* v_1 *is a* ***predecessor vertex*** *to* v_2 *if* $\exists\, d = (S, t, M) \in E_f \ni (v_1, v_2) \in S$.

Definition 4.11b - successor vertex:

Given a forward derivation component graph $G_{fwd} = (V_f, E_f)$*, and two vertices* $v_1, v_2 \in V_f$*,* v_2 *is a* ***predecessor vertex*** *to* v_1 *if* $\exists\, d = (S, t, M) \in E_f \ni (v_1, v_2) \in S$.

Using Definitions 4.11(a) and (b), we can define the terms *ancestor* vertex and *descendant* vertex.

Definition 4.12a - ancestor vertex:

*Given a forward derivation component graph $G_{fwd} = (V_f, E_f)$, and two vertices v_1, $v_2 \in V_f$, v_1 is an **ancestor vertex** to v_2 if v_1 is a predecessor to v_2, or there is some $v' \in V_f$ such that v_1 is a predecessor to v' and v' is an ancestor to v_2.*

Definition 4.12b - descendant vertex:

*Given a forward derivation component graph $G_{fwd} = (V_f, E_f)$, and two vertices v_1, $v_2 \in V_f$, v_2 is a **descendant vertex** of v_1 if v_2 is a successor of v_1, or there is some $v' \in V_f$ such that v_2 is a successor to v' and v' is a descendant of v_1.*

Another way of phrasing the above is if a simple path exists from vertex v_1 to vertex v_2, then v_1 is an ancestor to v_2, and v_2 is a descendant of v_1. In Figure 4-6, E_U, the database, is an ancestor to all other vertices. Vertex e12 is a descendant of e3c. Vertices e4, e7 and e8 are successors to e3c. Vertices e2 and e4 are predecessors to e6.

4.5 Forward Derivation Paths

A *forward derivation path* is a generalization of a path within the forward derivation component graph. Whereas a simple path is a linear sequence of adjacent vertices specified between two vertices, a forward derivation path may incorporate additional vertices and edges because derivations may be compound (cf. Figure 4-1), Thus the forward derivation path is specified between two *sets* of vertices representing data entities.

Figure 4-8 shows the general concept of the forward derivation path. Instead of connecting two vertices, the forward derivation path connects two sets of vertices, a *source* vertex set and *sink* vertex set.

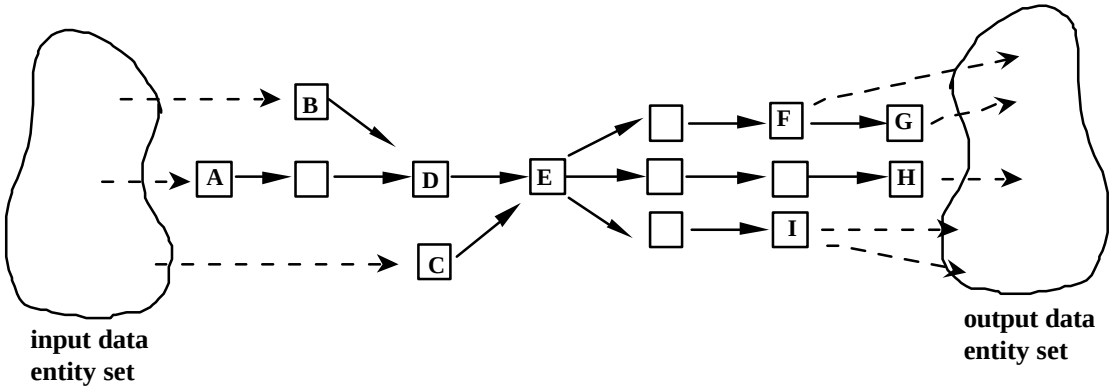


Figure 4-8. A forward derivation path between two sets of data entities.

Figure 4-8 shows that the forward derivation path contains multiple, overlapping simple paths. Note that an entire forward derivation component graph can be a forward derivation path if the source vertex set is the database and the sink vertex set contains all terminal vertices.

Definition 4.13 - forward derivation path:

Given a forward derivation component graph $G_{fwd} = (V_f, E_f)$*, a set of source vertices* $V' \subseteq V_f$*, and a set of sink vertices* $V'' \subseteq V_f$*, where* $V' \cap V'' = \{\varnothing\}$*, a* ***forward derivation path*** *from* V' *to* V'' *is a subgraph* $P_{fwd} = (V_p, E_p)$ *of* G_{fwd}*, where* $V_p = V' \cup V'' \cup \{all\ v \subseteq V_f \ni v \text{ is on a simple path from some } v' \in V' \text{ to some } v'' \in V''\}$*, and* E_p *contains all simple paths between* V' *and* V''*.*

4.5.1 Partial Forward Derivations

Though it may not make sense to consider only a portion of a compound forward derivation, such considerations *are* possible during the course of analyzing the forward derivation component graph. Any edge encountered on a simple path may be only a part of a forward data derivation, since, in general, derivations can be compound (cf. Figure 4-

1). Thus, simple paths may be *incomplete* or *partial*, because they neglect portions of (temporally atomic) forward derivations.

A *partial* forward derivation is a forward derivation whose data set is a proper subset of some other forward derivation, with respect to the other forward derivation. It normally is a single edge, encountered on a simple path.

Definition 4.14 - partial forward derivation:
Given a forward derivation component graph $G_{fwd} = (V_f, E_f)$, and a forward derivation $d_f = (S_f, t_f, M_f)$ where $S_f \subset E_f$, a ***partial forward derivation*** *is a forward derivation $d_p = (S_p, t_p, M_p)$ where $t_p = t_f$ and $S_p \subseteq S_f$.*

In Figure 4-9, the *completion* of a forward derivation along a simple path is shown. Data entities $e3_a$, $e7_a$ and $e7_b$, along with their derivation component edges, are added to the derivation path defined by the simple path $\langle \ldots e3_a, e5, e7_b, \ldots \rangle$. This accounts for all other predecessor data entities that participate in the construction of vertex e5, and all successor data entities created from e5. The data derivation sequence created is:

$$\langle \left(\left\{ \left(e3_a, e5 \right), \left(e3_b, e5 \right) \right\}, 5, M_5 \right), \left(\left\{ \left(e5, e7_a \right), \left(e5, e7_b \right), \left(e5, e7_c \right) \right\}, 7, M_7 \right) \rangle$$

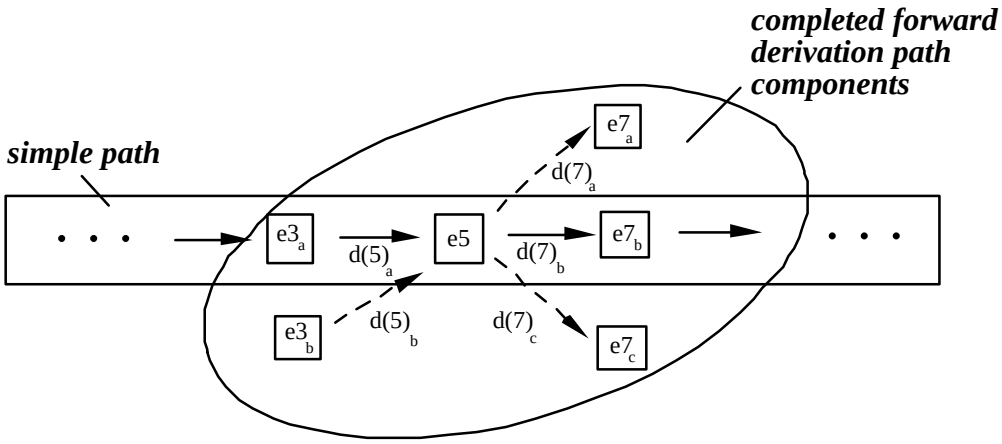


Figure 4-9. Completing a forward derivation.

Completing a forward derivation path between two graph vertices has the effect of adding additional vertices and edges to the simple path between the two vertices. This creates the set of simple paths of the forward derivation path.

4.5.2 Multiple Derivations per Vertex

Each vertex in the forward derivation component graph can have multiple distinct derivations associated with it. Additional derivation edges associated with each vertex can be considered for inclusion in a forward derivation path only if

1) some ancestor vertex is present in the original source vertex set, or

2) some descendant vertex is present in the original sink vertex set.

The process of completing the forward derivation path from a set of simple paths requires accounting for both complete forward derivation components and possibly multiple distinct derivations, for each vertex considered. The ability to determine ancestor and descendant vertices is therefore an important operation. While determining descendant vertices is a straightforward exercise in following graph edges, determining ancestor

vertices is harder to accomplish, given the acyclic nature of the forward derivation component graph. Data lineage graphs, discussed next, remedy this problem.

4.5.3 The Data Lineage Graph

The *data lineage* of a data entity is an ordered listing of all its ancestors. The data lineage of a forward derivation component graph vertex is a path from the vertex to the data origin vertex. This path is not necessarily linear, for the same reason that the forward derivation path is not necessarily linear. Since derivations may be compound (cf. Figure 4-1), vertices may have multiple incident edges, and tracing backwards potentially results in multiple overlapping paths. This is especially true if we account for complete forward derivations while tracing the data lineage. The *data lineage graph* shows the ancestral vertices for any graph vertex. It similar to the forward derivation component graph, but the edges are in the reverse direction.

Definition 4.15 - data lineage graph:
Given a forward derivation component graph $G_{fwd} = (V_f, E_f)$, a ***data lineage graph*** *is a directed graph $G_l = (V_l, E_l)$ where $V_l \subset V_f$ and for each $(v_1, v_2) \in S$ in some $d_f \in E_f$, there exists $(v_2, v_1) \in S$ in some $d_l \in D_l$.*

Figure 4-10 shows the data lineage graph for the forward derivation component graph of Figure 4-6.

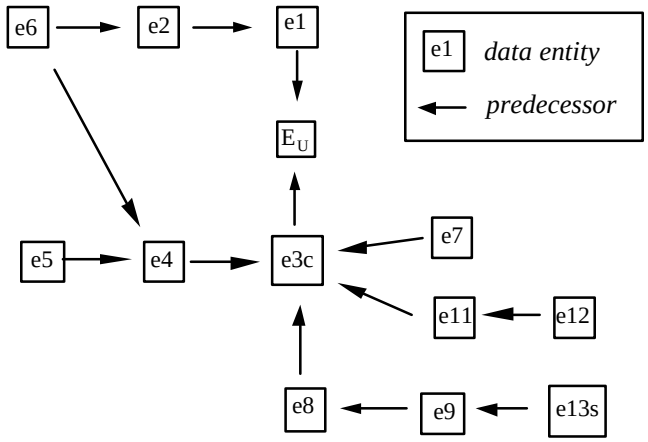


Figure 4-10. The data lineage graph corresponding to Figure 4-6.

It is evident that the data lineage graph is useful for determining ancestor vertices.

4.6 GDE Model Summary

This chapter defined a generalized model for interactive data exploration sessions. At the lowest level of abstraction, the GDE model has provisions for the elemental data structures that represent data sets and the transformations between them. At higher levels of abstraction, the GDE model has provisions for aggregate structures that represent the sequential, hierarchical and cyclical nature of the data exploration process.

One way of looking at a data exploration session is the linear, sequential application of derivations. The d-derivation sequence models this aspect of the session, but is clearly limited because it suppresses the hierarchical and cyclical aspects of the session. The various graphs defined in this chapter alleviate this problem. The graphs not only show branching patterns based on explicit state changes, but also cyclic and additional branching patterns resulting from the collection of congruence and similarity classes of data entities. By considering only certain graph components (forward, backward and identity), graphs can be made less complex, and possibly more understandable. Forward

derivation component graphs are important because they minimize the number of edges while showing all vertices, and they show only those derivations that created new, unseen data entities with respect to a data exploration session.

The forward derivation component graph stores analyst actions in derivation edges and system responses in data entity vertices. Thus, the forward derivation component graph stores the user-data interactions that make up a data exploration (cf. Section 2.2). Since most of the time, the system response is an unknown quantity (the analyst does not know the effect of the derivation until the system response is created and seen) interactions affect the execution of the plan that implements the tasks conjured up by the analyst to satisfy the goals of the exploration. Plans, tasks and goals evolve over the course of the exploration, based on the user-data interactions. In the following chapter, we look into how exploration sessions can be measured from this interaction-level retention of the data exploration process.

When considering the applicability of classical graph algorithms to the forward derivation component graph, it would seem that *connected components* and *transitive closure* provide the greatest utility. Connected components serves to identify vertices derived from a target vertex, and transitive closure, with certain application domain information, enables graph compaction. Because the forward derivation component graph is directed and acyclic, and has a temporal ordering among its vertices, the application of classical graph algorithms is limited. Additionally, accounting for complete forward derivations complicates any graph traversal. Operations that require complete forward derivations must take additional measures to properly traverse the graph.