GDL: A High Level, Access-Path Oriented Language

Hideko S. Kunii
J. C. Browne
Umeshwar Dayal

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

March 10, 1981 TR-175

This research was partially supported by a grant from NASA Langley Research Center, Number NSG1446.

GDL: A High Level, Access-Path Oriented Language

H. S. Kunii, J. C. Browne and Umeshwar Dayal

Department of Computer Sciences

The University of Texas at Austin

Austin, Texas 78712

ABSTRACT

We present a high level data language called the Graphical Data Language (GDL) for network structures. The manipulation statements in the GDL are formulated as algebraic operations on labelled directed graphs including link (arc) operators. Logical access paths are visible to the user and may be defined dynamically in terms of links (arcs). Thus the user has flexible control over execution efficiency while retaining full data independence. The expressive capability of GDL includes transitive closure, grouping and dynamic type creation for records as well as links. Its query representation is structured and concise.

1.0 INTRODUCTION

It has become clear that no single one of the popular existing data models [TSC76a, CDD70,72, DBT71, TYL76] is a suitable or adequate basis for all types of applications. The ANSI-SPARC [ANS75] multi-schema approach is partially motivated by this problem. The severe difficulties in designing and implementing effective schema mappings [DYL79] make it worthwhile to continue the search for more effective integrated data models.

An ideal data model would possess all of the following properties.

1. Representational power combined with a usable interface
2. A formal algebraic foundation for data relationships and the operations on data and data relationships
3. Practical data independence. The structure and efficiency of data base operations should be independent of mappings between physical representations.
4. Specifications of processing patterns at a logical and abstract level consistent with practical data independence. This is necessary to obtain efficiency with modest resources.

This paper introduces a directed graph based data model and a set of formal operations on data and data relationships which possesses all of these properties to at least a practical degree.

Records are represented as nodes characterized by attributes and holding values. Data relationships and access paths are represented by directed arcs (links). The set of operations and their combination rules form a DML and the data model together with the DML forms a graphical data language (GDL). The operations include both record and link manipulations. Each operation is algebraically defined. The composition rules yield a high level DML without loss of functionality. The operations of the DML include definition and deletion of types as well as value occurrences. Definitions of existing record types and link types can be dynamically changed. The GDL can express transitive closure and grouping.

The definition of GDL is at a preliminary stage. Facilities for integrity management and for aggregate operations (such as summations or averages) are not yet defined. This paper presents an overview of data definition and manipulation statements and the algebraic graph operations into which a manipulation statement is decomposed. A graph based data model allows specifications of expected access paths through definition of arcs in the graph. This introduces transparency into the data model at an abstract level. Transparency in a data language is a crucial factor to avoid disastrous performance of queries as in general purpose programming languages [WRT74]. It avoids the imposition upon the DMBS of complex tasks such as the design of access paths and query optimization to obtain reasonable performance. It is also not desirable to require a user to know about physical details of access paths and storage structures. Our approach is to have the user handle access paths at an abstract level in terms of graph

manipulation. This means the user specify an information base on processing patterns which can be used to generate efficient processing structures.

Data independence in GDL is based upon two factors. The first and previously mentioned factor is that relationships and access paths are specified only in abstract representations. The second factor is that GDL allows dynamic type creation and type alteration so that execution environments can be tailored to processing requirements.

The representational power of GDL (full directed graphs) can be shown to be greater than that of any of the currently popular data models. The visual and intuitive representation of relationships and access paths by graphs aids schema definition and construction. The high level nature of GDL DML statements simplifies programming and unifies embedded and free-standing use of the data base.

2.0 RELATED WORK

GDL can be used to implement other data models such as the relational model and the network model. (It might be a powerful candidate for a conceptual schema data model in the 3 level schema system). The entity-relationship model [CHN76] may be naturally mapped onto GDL.

The Link and Selector Language [TSC76] has a similar approach in the sense that LSL also intends to solve the performance problems by making access paths visible to users. GDL is a much higher level

language than LSL. GDL is expressed in terms of graphs and the power of record and link selection is much more powerful than in LSL. LSL was intended as a target language for DML compilers while GDL is a complete language system.

An extended owner-coupled set data model proposed by J. Bradley [BRD78] developed a non-procedural language on a CODASYL-like graph data structure. However, it is a very restrictive data model because linkages between record types are allowed only if corresponding connection fields exist in those record types.

NQUEL [DYL79] is a non-procedural language for a generalized network model. We adopt NQUEL-like notation for link operators in GDL. However, GDL differs from NQUEL in that it makes access paths explicitly visible to users, while NQUEL treats links merely as logical relationships. Also, GDL is algebraic (i.e., queries are constructed by composing elementary operations), while NQUEL is non-procedural and predicate calculus-based.

The Unified Database Language (UDL) developed by C. J. Date [DAT80] is claimed to accommodate both record-at-a-time and set-at-a-time manipulations on network data structures which are simpler than CODASYL data structure. (Relational and hierarchical data models are treated as subsets of the network model.) Record-at-a-time manipulations are made clear in UDL by the use of a "cursor" concept. The Set-at-a-time manipulations of UDL are not sufficient to express arbitrary queries on the network structure. GDL is a more dynamic language because of the operations for

creating record types and link types as a part of DML program execution.

3.0 DATA DEFINITIONS

The data structures of GDL are directed graphs. The nodes represent records and the arcs represent links. Each node (arc) is labelled with the type of the record (link) that it represents.

A schema S, which is a logical description of a database, is described by $\underline{R}$ a set of record types, and $\underline{L}$ a set of link types, where

$$\underline{R} = \{R_i \mid i=1,l\}$$

and $\underline{L}=\{L_j \mid j=1,m\}$.

$\underline{L}$ can be an empty set. An attribute set A_i is associated with each record type R_i:

$$A_i = \{A_i^k \mid k=1,n_i\}$$

Associated with each attribute A_i^k is a domain of the occurences (values), denoted as domain (A_i^k). Further, if $X = \{A_i^k \mid k = 1, p\}$ and $X \subset A_i$, then we define domain (X) as the Cartesian product of domain (A_i^k), $1 \leq k \leq p$. An occurrence of a given record type is a type of values $V_i^k \in$ domain (A_i^k). Then, let r[X] denote the value of r on domain X, where $r \in R_i$ and $X \subset A_i$.

An initial node record type R_i and a terminal node record type R_i are defined for each link type L_j:

$L_j = (R_i, R_i')$, $\quad L_j \in \underline{L}$, $\quad R_i, R_i' \in \underline{R}$.

We allow self-loop link types in which the initial node record type is the same as the terminal node record types. $L_j = (R_i, R_i)$. This kind of link type is called a recursive link type. There can be more than one link type between a given pair of an initial node record type and a terminal node record type. $L_j = (R_i, R_i')$, $L_j' = (R_i, R_i')$. Link types describe the binary relationships among record types which can be visualized by Bachman diagrams [BCH69].

Actual occurrences of records and links determine $\underline{S}$, a database state for a given schema S (an extension of schema S). Let us denote $\underline{R}_i$ as a state of record type R_i and $\underline{L}_j$ as a state of link type l_j:

$$\underline{R}_i = \{r_i \mid r_i \in R_i\}$$

$$\underline{L}_j = \{l_j \mid l_j \in L_j\}$$

where r_i and l_j denote an occurrence of record type R_i and and occurrence of link type L_j respectively. Note that for a given link occurrence, its initial and terminal node record occurrences must exist. Then, we can write $\underline{S}$ as follows:

$$\underline{S} = (\underline{R}, \underline{L})$$

where $\underline{R} = \{\underline{R}_i \mid i=1,l\}$

and $\underline{L} = \{\underline{L}_j \mid j=1,m\}$

Ordering among records of a given record type can be expressed by a recursive link type. The GDL may undertake to automatically update

such link occurrences whenever the states of associated record types change. Unlike the CODASYL DBTG network model [DBT71], the functionality of a link is not imposed in GDL data structure. (It may be defined as one of the constraints on the data structure.)

An example of a schema definition in GDL is given in Fig. 1.

4.0 GDL ELEMENTARY OPERATIONS

The manipulation statements of GDL described in the next section are translated into one or more GDL elementary operations. These operations are classified into conventional set operations, and database operations on record types and link types. The first type of operation includes Union ($\cup$), intersection ($\cap$), and difference ($-$). In this section, we describe typical operations of the second type (not all of them) to give an overall picture of the DML or GDL language. There are three creation operations (record, link, and unique record identification (URI)-tuple) among GDL's elementary operations.

Record Concatenation

This operation creates new record occurrences by concatenating a pair of record occurrences from two record types connected by a given link type L:

$$\{(r_1 \frown r_2) : r_1 \in R_1 \wedge r_2 \in R_2 \wedge \exists l \in L \quad (l=(r_1 , r_2)) \},$$

where R_1 and R_2 are the initial and the terminal node record type of L, $(r_1 \frown r_2)$ denotes the concatenation of record occurrences

```
     define schema                ACADEMIC - RECORD
record COURSE
        attribute          CNO      integer
        attribute          CNAME    character      10
        attribute          CORE     character       3

record ENROLLMENT
        attribute          CNO      integer
        attribute          SNO      integer
        attribute          GRADE    character       2

link REPORT
        initial node                COURSE
        terminal node               ENROLLMENT
end
```

Fig. 1 An example of a schema definition

r_1 and r_2. The attribute set of the result record type is the union of A1 and A2. If the user wants to concatenate two unconnected record types, then a link type which connects the above two record types must be created before record concatenation can take place.

Link Creation

This operation creates new link occurrences from a given initial node record type R_1 to a given terminal node record type R_2:

$$\{(r_1, r_2) : r_1 \in R_1 \wedge r_2 \in R_2\},$$

where (r_1, r_2) denotes the link occurrence from r_1 to r_2. It creates every possible link occurrence from R_1 to R_2.

URI-tuple Creation

Each record occurrence may be represented by a unique record identifier URI. The URI for a record occurrence is equivalent to the record occurrence itself when values from the record occurrence are not used in processing. It is often the case that links may be followed without the use of the actual values of record occurrences. URI's may be used in the formulation and execution of link traversal in place of actual record occurrences. Operations on URI's are a convenient basis for expressing the semantics of link traversal.

A URI-tuple is a tuple of the URI's for record occurrences. (It may also be visualized as a recursively linked record type whose attributes are record type URI's.) URI-tuple creation can be done on a single record type R or two record types connected by a link type as in the record concatenation. For the first case, it returns:

$\{<URI(r)> : r \in R\}$. For the second case, it returns:

$$\{<URI(r_1), URI(r_2)> : r_1 \in R_1 \wedge r_2 \in R_2 \wedge \exists_{l \in L} (l = (r_1, r_2))\}$$

The next two operations (projection and intra-record restriction) are similar to those in the relational model. URI-tuple projection, URI-tuple expansion and the reference operation on URI-tuples are rather unique in data languages although implementations must implicitly use these operations.

Projection

It extracts the values of a given attribute set, for a given record type and constructs a new tuple consisting of the above values, i.e., it returns $\{r[X]\}$, where $r[X]$ is defined in the previous section.

Intra-Record Restriction

This operation corresponds to the restriction in the relational algebra [CDD72]. It selects the record occurrences of a single record type which satisfy a given intra-record restriction. (e.g. SALARY >1000).

URI-tuple projection

This operation extracts a set of URI's of given record type(s) from a given set of URI-tuples, similar to the projection above. For example, we may extract

$\{<URI(r_1)>\}$ from $\{<URI(r_1), URI(r_2)>\}$ by the URI-tuple projection.

URI-tuple expansion

This operation adds one more URI type to a given URI-tuple set. For example, if we want to add URI's for record type R3 to $\{<URI(r_1), URI(r_2>\}$, then the URI tuple expansion returns

$\{<URI(r_1), URI<r_2), URI(r_3)>\}$.

Reference

It retrieves actual values of record occurrences for a given set S of URI tuples. For example the reference to $\{<URI(r_1), URI(r_2)>\}$ returns $\{<r_1, r_2>\}$.

Group

Some queries deal with properties of aggregrates of record/link occurrences rather than properties of individual record/link occurrences. For example, we may want to find students (SNO) who took all the core courses. (The schema diagram is shown in Fig. 2.) To answer this question we need to group the record occurrences of ENROLLMENT by SNO and test whether each group contains all the core courses. We incorporate this grouping concept into link operators. We define a group which is an equivalence class on Ri:

$g_{Ri}(x,r) = \{r' : r' \in Ri \wedge r'[x] = r[x]\}$

Thus, record occurrences of a record type for a given database state can be partitioned into disjoint groups. We denote this collection of groups as $G_{Ri}(X)$:

$G_{Ri}(X) = \{g_{Ri}(X,r) : r \in Ri\}$

Further, we can also form the same equivalence class (group) by specifying the value x of r[X]:

$$g_{Ri}(X,x) = \{r : r \in Ri \wedge r[X]=x\}$$

where $x \in$ domain (X)

$$G_{Ri}(X) = \{g_{Ri}(X,x) : x \in R[X]\}$$

Link operators, which we describe next, are performed on this collection of groups. If no grouping is specified, then link operators are performed on individual record occurrences. Quotient relations in [FRT77] are also a collection of equivalence classes called blocks on a relation. These authors define an algebra of quotient relations to add a set-processing capability to the relational approach. On the other hand, we develop an algebra based on graph data structures.

Link Operators

GDL provides several link operators to enhance the expressive power of the language. Five types of link operator are described in this paper. Each link operator has four variations, each of which returns occurrences of a different type as the result:

1. record occurrences of either the initial or the terminal node record type

2. concatenated record occurences of the initial and terminal node record types

3. URI's for record occurrences

4. tuples of URI's for the record occurrences

Which variant of a link operator should be employed is determined by the context in which the operator is used. For example, if a link operator is used in the context of deriving record occurrences from more than one record type then the second variant is chosen. In the following descriptions of link operators we deal with only the first variant, since the remaining ones are easily inferred from it.

i) existential link operator

R1--->/L/(A2)--->R2 or R1--->(A1)/L/--->R2

A2 specifies a subset of attributes of R2 by which a grouping is made. The grouping is optional. If the result to be returned is the terminal node record type R2, then this operator returns:

$$\{\ r2 : r2 \in R2 \wedge \exists r1 \in R1\ \exists r2' \in g_{R2}(A2, r2)\ \exists l \in L\ (\ l = (r1, r2')\ )\ \}$$

That is, it returns the occurrences of R2 which are in the groups having at least one link occurrence from record type R1. Fig. 3 shows an example of this operator. Its schema diagram is shown in Fig. 2. [CNAME = "DB"] is an intra-record restriction as presented above.

If the initial node record type R1 is to be returned, then the result is:

$$\{\ r1 : r1 \in R1 \wedge \exists r2 \in R2\ \exists r1' \in g_{R1}(A1, r1)\ \exists l \in L\ (\ l = (r1, r2')\ )\ \}$$

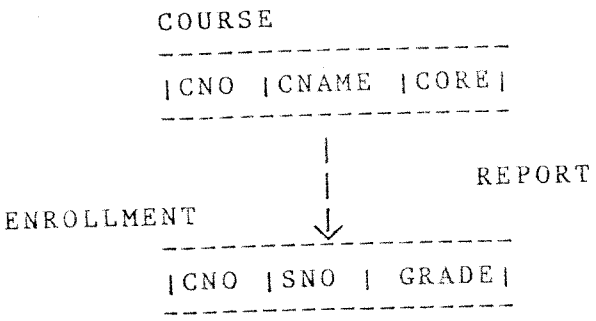
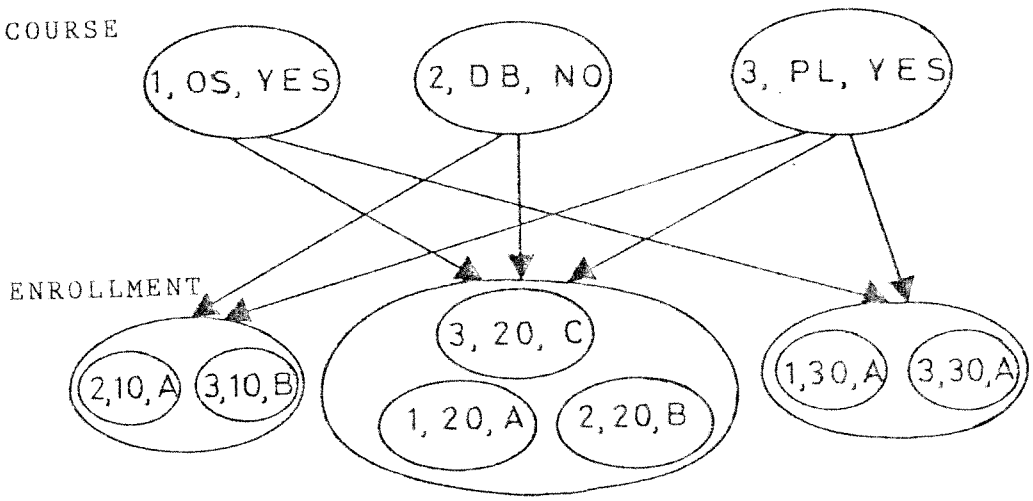


Fig. 2 An example of a schema

State:



Intent: Retrieve all the ENROLLMENT occurrences of the students who took DB course.

GDL link operator:

COURSE [NAME = "DB"] --->/REPORT/(SNO)---> ENROLLMENT

returns

{ (2, 20, B), (1, 20, A), (3, 20, C), (2, 10, A), (3, 10, B)}

Fig. 3 An example of an existential link operator

ii) negative existential link operator

R1-+->/L/(A2)-+->R2 or R1-+->(A1)/L/-+->R2

This is simply the negation of the existential link operator. It returns:

$$\{r2 : r2 \in R2 \wedge \neg(\exists r1 \in R1\ \exists r2' \in g_{R2}(A2,r2)\ \exists l \in L\ (l = (r1,r2'))\}$$

or

$$\{r1 : r1 \in R1 \wedge \neg(\exists r2 \in R2\ \exists r1' \in g_{R1}(A1,r1)\ \exists l \in L\ (l = (r1, r2'))\}$$

iii) universal link operator

R1===>(A1)/L/(A2)===>R2

A1 and A2 are subsets of attributes of R1 and R2. They specify how the groupings are to be made on R1 and R2. The grouping specifications are again optional. If the terminal node record type R2 is the result type, then this operator returns:

$$\{r2 : r2 \in R2 \wedge \forall g_{R1} \in G_{R1}(A1)\ \exists r1 \in g_{R1}\ \exists r2' \in g_{R2}(A2,r2)\ \exists l \in L\ (l=(r1,r2'))\}$$

That is, this operator first partitions record occurrences of R1 and R2 according to the grouping specifications. Then, it returns the record occurrences in the group g_{R2} such that for every group g_{R1}, there is at least one link from any record occurrences in g_{R1} to any record occurence in g_{R2}. One example is shown in Fig. 4. The division operation of relational algebra can be easily simulated by this operator.

Intent: Retrieve all the ENROLLMENT occurrences of the students who took all the core courses.

GDL link operator:

COURSE[CORE = "YES"] ===> /REPORT/(SNO) ===> ENROLLMENT

returns

{ (2,20,B), (1,20,A), (3,20,C), (1,30,A), (3,30,A)}

Fig. 4 An example of a universal link operator

If the initial node record type is to be returned, then the result is:

$$\{r1 : r1 \in R1 \wedge \forall g_{R2} \in G_{R2}(A2)\ \exists r2 \in g_{R2}\ \exists r1' \in g_{R1}(A1, r1)\ \exists l \in L\ (L=(r1', r2))\}$$

iv) negative universal link operator

R1===>(A1)/L/(A2)===>R2

This is a negation of the universal link operator, and returns either of the following, depending on the desired result type:

$$\{r2 : r2 \in R2 \wedge \neg(\forall g_{R1} \in G_{R1}(A1)\ \exists r1 \in g_{R1}\ \exists r2' \in g_{R2}(r2, A2)\ \exists l \in L(l=(r1, r2')))\}$$

or

$$\{r1 : r1 \in R1 \wedge \neg(\forall g_{R2} \in G_{R2}(A2) \exists r2 \in g_{R2}\ \exists r1' \in g_{R1}(r1, A1)\ \exists l \in L\ (l=(r1', r2)))\}$$

v) transitive closure link operator

R1 --->n /L/ n---> R1

This is a specialized link operator which applies only on recursive link types. We call it an (irreflexive) transitive closure link operator. It returns record occurrences each of which is connected from one of the given record occurrences of the initial node record type by a sequence of up to n link occurrences (n is a positive integer). If + is specified, then the result is all record occurrences that are connected from the given record occurrences of the initial node record type by any sequence of the occurrences of the recursive link type. For example, we can retrieve all the superiors (i.e., managers at every level in the hierarchy) of specified employees by using this link operator.

Note that this link operator allows the retrieval of those record occurrences which are connected to themselves, although the implementation must take care of infinite loops.

Inter-record Restriction

This operation is similar to the intra-record restriction except that its operands are the attributes of two different record types, e.g., R1.F1 = R2.F2. These two operands are provided by some link operators, since we do not allow this operation on two unconnected record types. This constraint is imposed for the sake of efficiency in processing queries. It does not reduce the power of this language since GDL provides the capability of creating links dynamically.

Note that the join operator in the relational model can be translated into a sequence of GDL elementary operations: a link traversal by some link operator, an inter-record restriction, and a record concatenation.

5.0 DATA MANIPULATION

There are six major data manipulation statements for both records and links: select, derive, add, delete, remove and update. Certain link manipulations may not be needed by a DML user. They are, however, essential when the GDL is used for developing higher level language interfaces. Before introducing these statements, we need to describe synonym statements, and path and record selection expressions which are used in many data manipulation statements.

Synonym

Synonym statements may be used when the user wants to reference a single record type more than once in a single statement:

SYNONYM of E is E1, E2

Otherwise, record type names and link type names themselves are used to denote qualified sets of records and links of the types.

Path

It is not easy for non-mathematical users to express complex queries in terms of relational calculus [CDD71]. It is also not simple to write queries in a procedural language as the CODASYL DBTG DML. GDL eases the difficulties of writing queries by providing non-procedural and structured language constructs.

In any meaningful query for selecting occurrences from a record type (let us call this record type a source record type), the record types referenced in the query must have some relationships to the source record type. (Here we use term "type" loosely. To be more accurate, we mean by "type" a set of some occurrences of the type.) Then, those record types are in some way connected to the source record type in the GDL graph when all the relationships between record types are expressed by links. Some link types and record types can be temporarily created. Whether record or link types are temporary or not does not make a difference at the representation level of queries.

To express the structure of connected record types, let us define the concept of "path" which is extended from the definitions in [HRW76] to our model. A path is a finite alternating sequence of record types and link types beginning and ending with record types, such that for each link type, the preceding record type is the initial node record type of the link type and the following record type is the terminal node record type of the link type. Let us now expand the notion of a path by including operations on record types and link types on a path, which we call a path expression. A path expression is a definition of a sequence of record selections and link traversals (in terms of link operators).

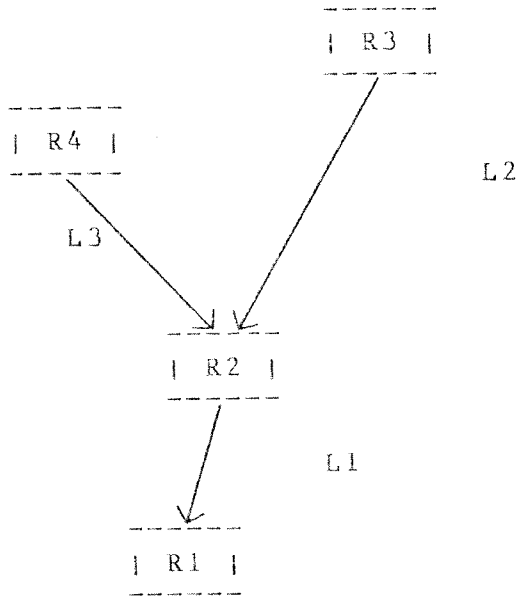
<path expression> ::= <path expression> ---> <link operator>
---> <record selection expression>|
<record selection expression> ---><link operator> --->
<record selection expression>

Examples of paths are shown in Fig. 5. Rectangles denote record types and arrows denote link types. There are five possible paths in the graph. Link type names are enclosed within / / to distinguish them from record type names. Any path expression is translated into a sequence of elementary operations along the direction of the path. For example the path expression in Fig. 5 is executed in the following sequence when R1 is the source record type:

RESULT1 := R2 [R3 --->/L2/ --->R2]

RESULT := R3 [RESULT1 --->/L1/ --->R1]

Schema:



Path expression:

path R3 ---> / L2/ --->R2--->/L1/--->R1

Fig. 5 An example of a path expression

Every path must follow the direction of its link types. If bi-directional traversals are needed, different link types whose directions are opposite should be defined between record type pairs. This inhibition of inverse traversals improves the readability of selection specification and simplifies the language, without decreasing the capability of selection.

Record Selection Expression

A record selection expression (RSE) specifies the record occurrences of a given source record type which the user wants to manipulate. The expression is translated into a sequence of elementary operations described in the earlier section. We illustrate only major features of record selection in this paper.

<record selection expression> ::= <source record type> [
<record selection clause>]

<record selection clause> ::= <intra record restriction> |
<network specification>

An intra-record restriction may be specified by itself or be embedded in an inter-record selection specification which we call a "network specification". The network specification describes the relationships between a given record type and other record types. A network specification is one or more complex path expressions which may be connected by set operators (∪, ∩). Each complex path expression is composed of a list of inter-record restriction expressions and a path expression described before. Any record type specification appearing in the path specification can have a

qualification which is either simply an intra-record restriction or another network specification. An inter-record restriction is performed on the record types of the RSE as the types are encountered along the path. Examples of GDL statements are given later in this section. Note that negation has to be represented by negative link operators.

Select Record

There are two manipulation statements which may be used for record retrieval: "select record" and "derive record". The select record statement retrieves record occurrences from a single record type, while the derive record statement selects record occurrences from more than one record type and composes occurrences into a new record type (analogously to the join operation in relational algebra). (This procedure can also be used for insertion.) The reason for two record retrieval statement types is to distinguish simple record retrieval (select record) from a compound operation of record retrieval and record composition (derive record). The latter is discussed later. The syntax for the select record statement is:

select record <target record type> := <record selection expression> <projection> with <accompanied link list>

The target record type specifies the record type into which retrieved record occurrences are to be brought. (We assume that all the record types and the link types are defined before being referenced.) The <record selection expression> specifies qualification of the source record type. The <projection>

containing a list of attribute names indicates that a projection operation should be performed on the retrieved record occurrences. If this part is absent, the whole record is copied to the target record type.

A list of associated link types, which is optionally specified in the with <accompanied link list> part, allows the user to select occurrences of the associated link types at the same time.

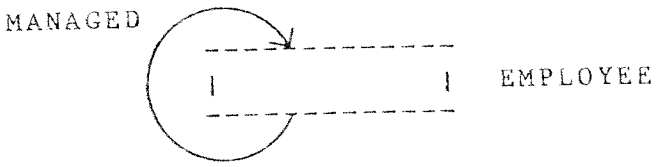
The initial node record types or terminal node record types must be the source record in the select record statement.

<accompanied linked list>::= {<target link type> := <source link type>}

The target and the source have similar meaning as in records.

The first example of a select record statement is shown in Fig. 6. The diagram illustrates (a part of) a schema definition in which there is one record type called EMPLOYEE (rectangle) and one recursive link called MANAGED (arrow). Our query is to select the names of those employees who directly manage employee KUNII. We are concerned with two types of EMPLOYEE here declaring E1 and E2 as the synonyms of EMPLOYEE: E1 denotes the employee called KUNII and E2 denotes the managers of E1. We can express this query easily by a single path. An intra- record restriction (i.e., NAME="KUNII") is specified for E1. (NAME) in the second GDL statement specifies a projection on E2. MANAGERS is the target record type. If the query is modified to select all the superiors of EMPLOYEE KUNII, then the path expression will have a transitive closure link operator instead

Schema:



Intent:

Select, into MANAGERS, the names of those who directly manage employee KUNII

GDL statements:

```
synonym of EMPLOYEE is E1, E2
select record MANAGERS := E2 [path  E1 [ NAME = "KUNII"] --->
       /MANAGED/ ---> E2 | (NAME)
```

Fig. 6 An example of a record selection statement

of an existential link operator:

path E1[NAME=KUNII]--->+ /MANAGED/+ --->E2

This example with transitive closure (--->+ / /+ --->) cannot be expressed in the relational model. The select-record statement in Fig. 6 is translated into the following:

1. intra-record restriction executed on E1

E1' := E1[NAME = "KUNII"]

2. existential link operator executed on MANAGED, E1 and E2

E2' := E1'--->/MANAGED/--->E2

3. projection executed on E2'

MANAGERS := E2'(NAME)

(The preceding sequence of elementary operations is simplified. An actual implementaion may utilize the aforementioned facility of URI-tuples to follow the link.)

Our second example of a record selection statement shown in Fig. 7 contains a nested path expression. The schema definition is the same as in Fig. 6. We want to retrieve those employees who are managers at the lowest level. The inner path expression specifies that the employees in E1 are not managers of any level. Their managers are, thus, at the lowest level of an enterprise.

The next example, Fig. 8, illustrates how to specify an inter-record restriction accompanied with a link selection. The

Intent:

Select, into LOWEST-MANAGER, the employees who are managers of the lowest level.

GDL statements:

```
synonym of EMPLOYEE is E1, E2, E3
select record LOWEST-MANAGER := E3
      [ path E1[path E2 -+-> /MANAGED/ -+-> E1] --->
               /MANAGED/---> E3 ]
```

Fig. 7 An example of a nested path expression

Intent:

select the managers (LOW-PAY-MANAGER) whose salaries are lower than at least one of their direct subordinates, and also the link occurrences of MANAGED (CHEAPLY-MANAGED) connected to the above managers.

GDL statements:

```
synonym of EMPLOYEE is E1, E2
select record LOW-PAY-MANAGER := E2 [
   E1.SALARY > E2.SALARY
        path E1-->/MANAGED/-->E2 ]
              with CHEAPLY-MANAGED := MANAGED
```

Fig. 8 An example of an inter-record restriction

schema is again the same as Fig. 6. The restriction is done between record type E1 and E2 connected by the link called MANAGED.

Fig. 9 shows an example of two paths connected by a set operator. We are, in this example, retrieving those students who are now taking "CS386" but who have not actually taken its prerequisite courses. This query is decomposed into two path expressions connected with an inter section operator . This is not the only way to express the above query in GDL. We can, for example, decompose the GDL statement into two statements to avoid redundant evaluation of the intra-record restriction on STUDENT.

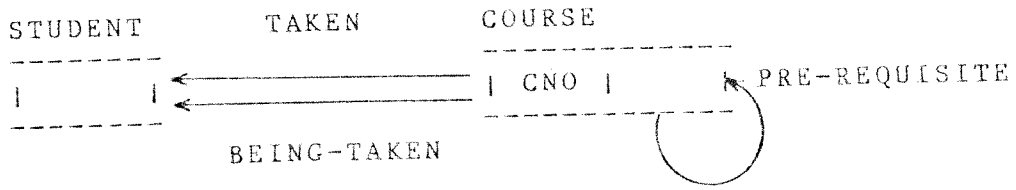
Select Link

The select link statement retrieves the link occurrences of an existing link type which satisfy the qualification. (The derive-link statement creates new link occurrences according to the qualification specified.) The syntax for the select-link statement is:

select link <target link type> := <source link type> [<link selection clause>]

The <link selection clause> is a special case of the network specification described in the record selection expression. That is, the outermost path expression is specified as the path along the source link type. For example, we can select link occurrences for a link type called LOW-PAY-MANAGED from link type MANAGED similar to the example in Fig. 7.

Schema:



Intent:

select the students, into SKIP-STUDENT, who are now taking "CS386" but who actually have not taken its prerequisite courses.

GDL statements:

```
select record SKIP-STUDENT := STUDENT
       [ (path COURSE [CNO = "CS386" ] ---> /BEING-TAKEN/
          ---> STUDENT )ⁿ
         (path COURSE [CNO = "CS386" ] ---> /PRE-REQUISITE/
          ---> COURSE =ǂ=> /TAKEN/ =ǂ=> STUDENT) ]
```

Fig. 9 An example of two paths connected by an intersection operator

```
select link LOW-PAY-MANAGED := MANAGED
    [E1.SALARY > E2. SALARY
            path E1-->/MANAGED/-->E2 ]
```

In general, we can qualify the initial node record type and the terminal node record type.

Derive Record

This statement composes new record occurrences for a given target record type. Its syntax is:

derive record <target record type> := (<derived attribute list>) [<network specification>]

<derived attribute list>::=<target attribute>:=
<source record type>.<source attribute> {,<target attribute>:=
<source record type>.<source attribute>}

The <derived attribute list> specifies the correspondence between target attributes and source attributes. With this statement, we can create a record type consisting of an employee name and his/her manager name from the schema in Fig. 6. (See Fig. 10) If the target record type is not an empty record type, then created record occurrences are added to the existing value sets for the record type.

Derive Link

This manipulation statement creates new link occurrences for a given link type. The syntax is:

derive link <target link type> := (<initial node record type>,<terminal node record type>) := [<link derivation specification>]

<link derivation specification> ::= <product specification>|
<network clause>

<product specification>::=<inter record restriction list> on <record selection expression> * <record selection expression>

There are two cases of this statement. The first case is that there is no path connecting the two record types between which link occurrences are to be created. The second is that there is a path connecting the record types which consists of more than one link type. In the former, we need to produce a Cartesian product of a given pair of initial node record type Ri and terminal node record type Rt: { (ri, rt) : ri ∈ Ri∧rt ∈ Rt }. In the latter case we construct the URI tuples of the initial and terminal node record type by processing a (complex) path expression. The two record selection expression in the product specification also may have qualification as in the network specification. Fig. 11 shows an example of this statement. We want to construct link type MANAGE from record type EMPLOYEE to record type PROJECT.

If the target link type is not empty, the created link occurrences are inserted to the type. Note that the derive link statement permits the user to create a new access path.

Add Record

Intent:

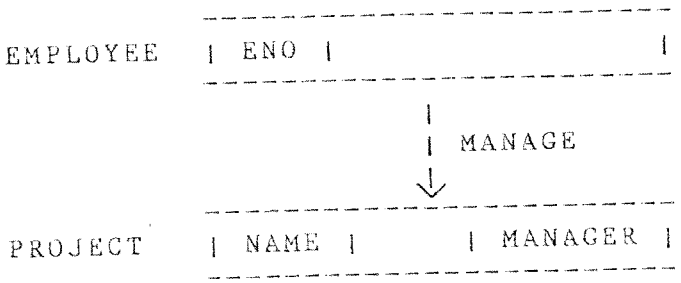
get all employee name/direct manager name pairs

GDL statements:

```
derive record EMP-MNG  :=  (ENAME:=E1.NAME,MNAME:=E2.NAME)
       [  path E1-->/MANAGED/-->E2 ]
```

Fig. 10 An example of a derive record statement

Schema:



Intent:

Construct a link type called MANAGE which connects EMPLOYEE to PROJECT if the employee is the manager of the project.

GDL statement:

```
derive link MANAGE  :=  (EMPLOYEE,PROJECT)
          [EMPLOYEE.ENO=PROJECT.MANAGER
            on EMPLOYEE * PROJECT ]
```

Fig. 11 An example of a derive link statement

To copy the occurrences of a record type into another record type, we have an add record statement. Its syntax is:

add record <source record type> to
<target record type>

Record occurrences of the source record type are copied to the target record type. The former can be input by the user or established from the current database by using select/derive-record manipulation. The source record type and the target record type should be compatible. (Compatible means having the same data structure.)

Note that if the target record type in the derive-record statement is one of the record types in some schema, the derive record statement does an insertion to the schema. Also note that this statement does not change the state of the source record type.

Add Link

This statement allows the user to copy the occurrences of a link type to another link type. The syntax is:

add link <source link type> to <target link type>

As in the add-record statement, this operation can be used to input link occurrences. The derive link statement can also be used for the insertion of link occurrences. Let us now define compatible link -types: If two link types have both compatible initial node record types and compatible terminal node record types, then they are called compatible link types. The source link type should be compatible with the target link type. The add-link statement is

successfully executed only for the occurrences of the source link type whose initial and terminal node record occurrences actually exist. It does not change the state of the source link type.

Delete Record and Delete Link

The syntax is:

delete record <source record type> from
<target record type>

delete link <source link type> from <target link type>

Both statements delete the record/link occurrences of the source record/link type from the target record/link type, i.e., a difference operation. The source type and the target type should be compatible. They do not have any effect on the source record or link type. If the record occurrences are initial or terminal node record occurrences of some link occurrences, then those link occurrences are also deleted.

Remove Record and Remove Link

These statements provide another form of deletion for a record type or a link type. They delete the record occurrences or the link occurrences which satisfy a given qualification. The qualification is specified in the same way as in the select-record manipulation or in the select-link manipulation. The syntax is:

remove record <target record type>

[<record selection clause>]

or

remove link <target link type>

[<link selection clause>]

The record selection clause is the same as record selection expression except that it contains only the qualification part, i.e., it does not specify the source record type because it is the same as the target record type. As in the delete-record statement, the execution of the remove-record statement may invalidate the existence of the link occurrences whose initial or terminal node record occurrences are removed by the statement.

Update Record

This manipulation updates values of some attributes of a record type. Record occurrences to be updated are qualified by the <record selection clause> previously described. The syntax is:

update record <target record type>
(<update attribute list>)
[<record selection clause>]

<update attribute list>::=<attribute name>:=<value>
{,<attribute name>:=<value>}

Update Link

This manipulation updates qualified link occurrences. The syntax is:

update link <target link type> <update node>

[<link selection clause>]

<update node>::= initial node := <record selection expression> | terminal node := <record selection expression>

The <update node> specifies a change to be made either on the initial node record type or on the terminal node record type. If more than one occurrence of the record type to be updated qualifies and these are connected to the same occurrences, then the redundant occurrences are deleted. This statement is like the reconnect statement in CODASYL DML and NQUEL. Fig. 12 shows an example of an update link statement. Its schema is the same as in Fig. 11.

As we have seen, record types and link types can be dynamically created and deleted. Thus, the constraint that record types appearing in a query must be connected by some path does not reduce the power of the language, since the user can create necessary link types "on the fly".

6.0 SUMMARY

We have introduced a Graphical Data Language (GDL) for data structures that are labelled directed graphs. The nodes of the graphs represent records and the arcs represent access paths. The language makes access paths and data relationships explicitly visible at an abstract level. These specifications are important in

Intent:

update link MANAGER such that REAGAN (ENO=1) manages PROJUSA instead of CARTER (ENO=2)

GDL statement:

```
update link MANAGE (initial node := EMPLOYEE [ENO=1])
  [path EMPLOYEE [ENO=2]-->/MANAGE/-->PROJECT
                        [ NAME = "PROJUSA" ] ]
```

Fig. 12 An example of an update link statement

securing efficient performance with modest resources. Manipulation statements are described in terms of elementary algebaic operations on graphs which is fine enough to capture the user's intention. We have developed extensive operations on links including existential, universal and transitive closure liuk operators.

GDL is a high level language in which the structure of a schema is visually expressible. Therefore, GDL is not only easy to implement, but also easy to formulate queries in. The expressive power of this language is greater than that of the relational model since it accommodates irreflexive transitive closure and grouping.

References

1. [ANS75] ANSI/X3/SPARC, "Study Group on Data Base Management Systems: Interim Report, ANSI - 2/8/75.

2. [AST76] Astrahan, M.M., et al., "System R: Relational Approach to Database Management", ACM Trans. on Database Systems, Vol. 1, No. 2, Jun., 1976, 97-137.

3. [BCH69] Bachman, C.W., "Data Structure Diagrams", ACM Database, Vol. 1, No. 2, 1969, 4-10.

4. [BRD78] Bradley, J., "An Extended Owner-Coupled Set Data Model", ACM Trans. on Database Systems, Vol. 3, No. 4, Dec. 1978, 385-416.

5. [BRW80] Browne, J.C., Kunii, T.L., Kunii, H.S., Takahashi, K., Katayama, O. and Oyanagi, K., "An Evolutionary Data Base Management System", Proc. IEEE COMPSAC 80, Oct. 1980, 320-326.

6. [CHN76] Chen, P.C., "The Entity-Relationship Model - Toward a Unified View of Data", ACM Trans. on Database Systems, Vol. 1, No. 1, Mar. 1976, 9-36.

7. [CDD70] Codd, E.F., "A Relational Model of Data for Large Shared Data Banks", CACM Vol. 13, No. 6, Jun., 1970, 377-387.

8. [CDD72] Codd, E.F., "Relational Completeness of Data Base Sublanguages", Data Base Systems, Courant Computer Science Symposia Series Vol. 6, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1972, 65-98.

9. [DBT71] Data Base Task Group of CODASYL Programming Language Committee, Final Report, Apr., 1971.

10. [DAT80] Date, C.J., "An Introduction to the Unified Database Language (UDL)", Proc. Sixth Int. Conf. on Very Large Data Bases, 1980, 15-32.

11. [DYL79] Dayal, U., "Schema-Mapping Problems in Database Systems", Dissertation, Harvard University, Cambridge, MA,

Aug. 1979.

12. [FRT77] Furtado, A.L. and Kerschberg, L., "An Algebra of Quotient Relations", ACM-SIGMOD, Proc. Int. Conf. on Management of Data, Toronto, Aug. 1977, 1-8.

13. [HRW76] Horowitz, E. and Sahni, S., Fundamentals of Data Structures, Computer Science Press, Inc., Potomac, Maryland, 1976.

14. [RST74] Rustin, R., ed., "Data Models: Data-Structure Set Versus Relational", Proc. ACM-SIGFIDET Debate, Ann Arbor, 1974.

15. [TSC76] Tsichritzis, D., "LSL: A Link and Selector Language", ACM-SIGMOD, Proc. Int. Conf. on Management of Data, Washington D. C. Jun., 1976, 123-133.

16. [TSC76a] Tsichritzis, D. C. and Lochovsky, F. H. , "Hierarchical Data Base Management" Computing Surveys 8, 105-124 (1976).

17. [TYL76] Taylor, R. W. and Frank, R. L. , "CODASYL Data Base Management Systems" Computing Survey 8 67-104 (1976).

18. [WRT75] Wirth, N., "On the design of Programming Languages," Proc. IFIP, Aug. 1974, 386-393.