# IMPLEMENTATION CONCEPTS FOR AN EXTENSIBLE DATA MODEL AND DATA LANGUAGE*,**

D. S. Batory, T. Y. Leung,*** and T. E. Wise

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-86-24 February 1988

## Abstract

Future database systems must feature extensible data models and data languages in order to accommodate the novel data types and special-purpose operations that are required by nontraditional database applications. In this paper, we outline a functional data model and data language that are targeted for the semantic interface of GENESIS, an extensible DBMS. The model and language are generalizations of FQL [Bun82] and DAPLEX [Shi81], and have an implementation that fits ideally with the modularity required by extensible database technologies. We explore different implementations of functional operators and present experimental evidence that they have efficient implementations. We also explain the advantages of a functional front-end to $\neg$ 1NF databases, and show how our language and implementation are being used to process queries on both 1NF and $\neg$ 1NF relations.

---

*This work was supported by the National Science Foundation under grants MCS-8317353 and DCR-86-00738.

**Revision of TR-86-24, October 1986. To appear in *ACM Transactions on Database Systems.*

***Current address: Department of Computer Science, UCLA, Los Angeles, CA 90024.

## 1. Introduction

Nontraditional database applications, such as VLSI CAD, graphics, and statistical data processing, have very different requirements than traditional business-oriented applications. Their differences often demand unusual storage structures and query processing algorithms, and require data models and data languages to admit new constructs, such as novel data types and operations.

Rather than developing monolithic DBMSs for each specialized application, there is a growing interest in extensible DBMSs which can be customized by plugging or unplugging modules that provide desired DBMS capabilities [Sto83, Car84, Car86, Day85a, Sto85, Bat85b, Bat86, Sto86, Sch87]. Customization can be achieved in a matter of hours or days, rather than the man-months or man-years of effort that are required to modify existing (nonextensible) DBMSs. To realize an extensible database technology, however, requires a fundamental alteration in the way DBMSs are presently constructed.

Functional data models and data languages have been proposed as candidates for the logical interface of extensible DBMSs [Day85]. Functional data languages are more powerful than conventional relational languages and have characteristics which are ideally suited for nontraditional applications: new object types and new functions can be introduced, functions can be composed, recursive functions can be defined, and scalar-valued and sequence-valued functions can be treated uniformily.

The functional data model was first proposed by Sibley and Kerschberg [Ker75, Sib78]. Subsequent work by Buneman on FQL [Bun79, Bun82] and by Shipman on DAPLEX [Shi81] popularized the functional approach and clearly demonstrated its significance. Much research based on FQL and DAPLEX has followed since [Zan83, Gra84, Alb85, Hei85, Day85b, And86, Lyn86, Man86, Fis87]. Despite the potential and growing interest, there have been few serious attempts to discover how functional DBMSs can be implemented efficiently. Part of the problem is that implementations of systems in functional programming languages are notoriously slow; this may have discouraged active research. Possibly in an effort to avoid true functional implementations, many 'functional' data languages that have been built are essentially relational database interfaces in disguise; the basic capabilities that distinguish the functional approach from the relational (e.g., new object types, recursive functions, etc.) are barely present, if not completely absent.

There is, however, a more fundamental reason for the lack of general implementations of functional data languages. Our research reveals that FQL and DAPLEX, the two most important functional data languages yet proposed, do not express functional database concepts in their most elementary and extensible form. We believe their extra complexity frustrates a simple implementation.

In this paper, we present a variation of the functional data model that is based on productions (stream rewrite rules) rather than mathematical functions. The proposed model GDM and data language GDL are targeted to provide the semantic interface to GENESIS, an extensible DBMS [Bat86-87]. Although the formal specifications of GDM and GDL are still under design, the basic implementation concepts have been established. We explain and develop these concepts in the following sections.

A novel feature of GDL is that computations are expressed as streams of tokens, where a token is either a database object or a delimiter that signals the end of one computation and the start of another. This intermixing of objects and delimiters provides a simple and efficient means to standardize the packaging of DBMS algorithms. This standardization is especially important to GENESIS, as it promotes extensibility, module plug-compatibility, and provides a necessary step toward a building-blocks technology for DBMSs.

We progressively develop concepts that are needed for implementing productions. Basic implementation strategies are reviewed (eager v.s. lazy evaluation, demand-driven v.s. data-driven evaluation), and experimental results are presented to show that productions have efficient implementations.

Lastly, GENESIS supports ¬1NF relations. We show how functional models facilitate experimentation with 1NF and ¬1NF databases, and how we are using GDL to evalutate predicates on ¬1NF tuples. Thus, an important by-product of our research is a starting point for building a ¬1NF query processor.

## 2. A Functional Data Model and Data Language

The GENESIS functional data model (GDM) and the data retrieval portion of the GENESIS data language (GDL) are highlighted in this section. GDM closely resembles traditional functional models, with the notable exception of our replacement of functions with 'productions' and its attendant impact on GDL's model of computation. A formal definition of GDM and GDL is forthcoming [Bat88].

### 2.1 Data Model Concepts

A GDM plan of a database can be depicted as an **object graph**, where nodes denote object types and arcs are relationships between types. The object graph of Figure 2.1a represents a simple department-employee-child database. Dept objects are departments, Empl objects are employees, and Sibling objects are children.

OBJECT, INT, FLOAT, BOOLEAN, and CHAR(n) are among the object types that are system-defined. OBJECT is the set of all objects, and INT, FLOAT, BOOLEAN, and CHAR(n) are subtypes of OBJECT. CHAR(n) is the set of all character strings whose length is n. User-defined types, such as Dept, Empl, and Sibling in the department-employee-child database, can be declared as subtypes of system-defined types or as subtypes of other user-defined types. As a rule, all object types in a GDM database belong to a single hierarchy, with OBJECT as the root. Figure 2.1b shows the type hierarchy of the department-employee-child database.

Operations that can be performed on objects and relationships between objects are expressed as **productions** in GDM. The term 'function' is the conventional term for relationships or operations in functional data models, but in GDM it is misleading. Functional data model 'functions' are not mathematical functions but are actually productions or rewrite rules. The distinction will be made clear shortly.

Associated with every object type is a set of productions. INT, for example, has > and + for comparing and adding integers. The hierarchical relationship between object types permits subtypes to inherit productions from their supertypes. Production inheritance via type hierarchies is known as **specialization** [Shi81].

### 2.2 Data Language Concepts

Fundamental concepts in the computation model of GDL are sequences and streams. { $d_1$ $d_2$ $d_3$ } is a **sequence** of three objects '$d_1$', '$d_2$', and '$d_3$', where braces enclose a sequence. A **stream** is an encoding of a sequence. The simplest encoding treats objects and braces as tokens. Thus, the sequence { $d_1$ $d_2$ $d_3$ } is a stream of five tokens: a left-brace, three objects, and a right-brace. The distinction between sequences and streams lies in their relationship to GDM productions. (We note that the explicit representation of braces in streams is a novel and essential feature of the GDL computation model).

GDM productions are stream rewrite rules. The simplest productions are generators. They produce the sequence of all objects of a given type, and are given the same names as their object type. Thus, for an object type D, there is a generator D:→{*D}. The D production takes no input and produces the stream beginning with '{', ending with '}', and the stream of all D objects inbetween. (*D denotes a stream of D objects in D:→{*D}). Suppose D generates the stream { $d_1$ $\cdots$ $d_n$ }.

Consider the production F:D→*R. F transforms a stream by replacing each D object with its stream of related R objects. F does not rewrite left-brace and right-brace tokens; it simply leaves them as is. As an example, suppose F maps $d_1$ to the stream $r_1$ $r_2$ $r_3$, $d_3$ is mapped to $r_4$, and all other D objects are mapped to the null stream. F would rewrite the stream { $d_1$ $\cdots$ $d_n$ } as { $r_1$ $r_2$ $r_3$ $r_4$ }. Figure 2.2 illustrates this rewriting.

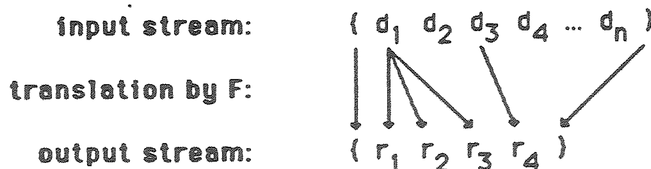
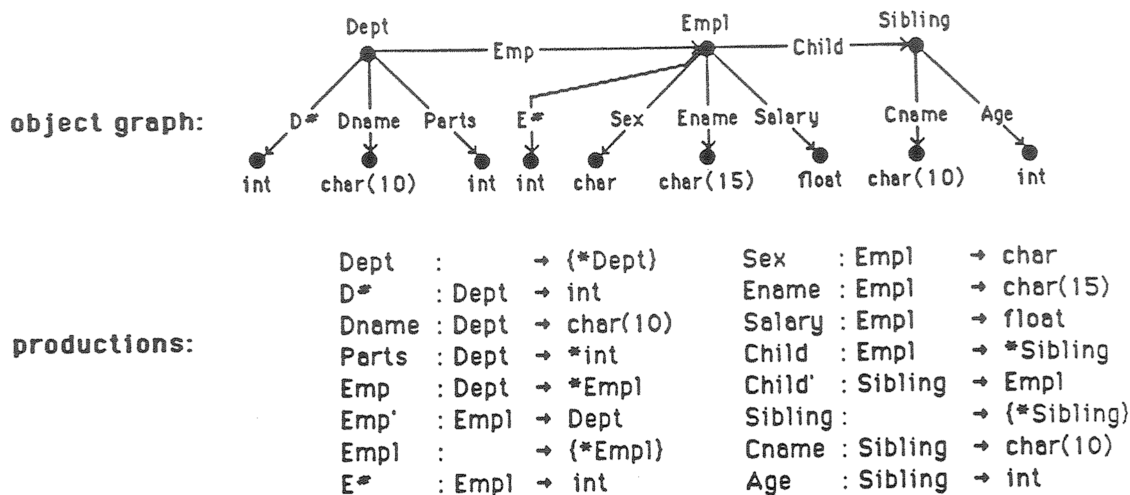
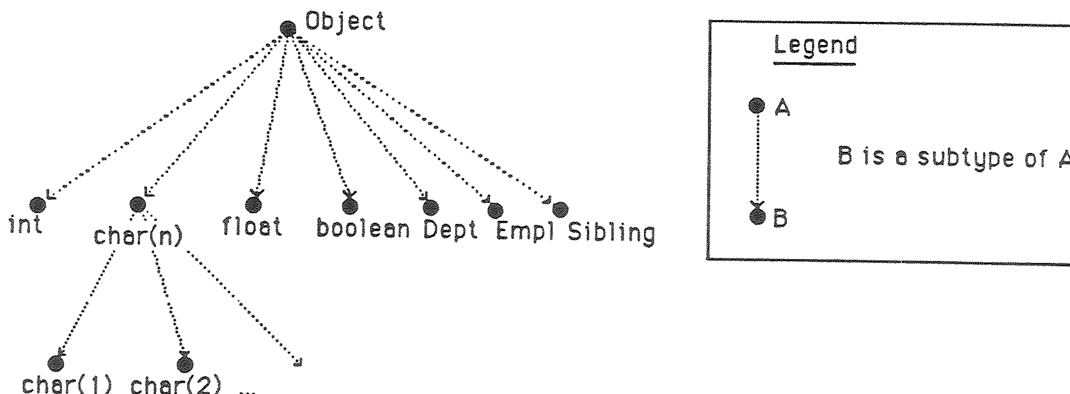



**Figure 2.2 The Rewrite of Stream { $d_1$ $\cdots$ $d_n$ } by Production F**




2.1a A Production Model of a Department-Employee-Child Database




2.1b Object Type Hierarchy in the Department-Employee-Child Database

**EMPL_REL**

| E# | Sex | Ename | Salary | D# |
|---|---|---|---|---|
| 105 | M | Smith | 40K | 1 |
| 202 | F | Jones | 43K | 1 |
| 347 | M | Blake | 35K | 1 |
| 150 | F | Clark | 27K | 2 |
| 251 | M | Adams | 36K | 2 |

**DEPT_REL**

| D# | Dname | Parts |
|---|---|---|
| 1 | Ceramics | { 1 4 7 } |
| 2 | Plastics | { 14 21 } |
| 3 | Pottery | { 8 34 62 } |

2.1c Partial Contents of Database in Relational Form
(with Set-Valued Attributes)

**Figure 2.1 A Department-Employee-Child Database**

Production composition (operator composition) is denoted in GDL by the dot '.' operator. D.F is the composition of productions D and F, and results in the stream { $r_1$ $r_2$ $r_3$ $r_4$ } as we saw above. Note that D.F is a reversal of the usual functional notation F(D), and is a notation borrowed from FQL. Also note the composition of the *definitions* D:→{*D} and F:D→*R is achieved by simple text substitution: F replaces the D in {*D} with *R to yield (D.F):→{**R}, which can be simplified to (D.F):→{*R}. The star '*' operator is idempotent; a stream of streams of R objects is indistinguishable from a stream of R objects.

An arc F from object type D to object type R in an object graph is normally represented by two productions: F:D→*R which replaces each D object with its stream of related R objects, and the **reverse production** F':R→*D which replaces each R object with its stream of related D objects. As a general rule, the reverse production F' is not the mathematical inverse of F. Consider the ancestory database of Figure 2.3a. Children:Person→*Person maps (parent) Person objects to (child) Person objects, and Children':Person→*Person maps children to their parents. If p is a Person, p.Children.Children is the stream of grandchildren of p, and p.Children.Children' is the stream of parents of the children of p. p usually does not equal p.Children.Children' (see Figure 2.3b). Again, this is a consequence of the fact that GDM productions are stream rewrite rules and not mathematical functions.

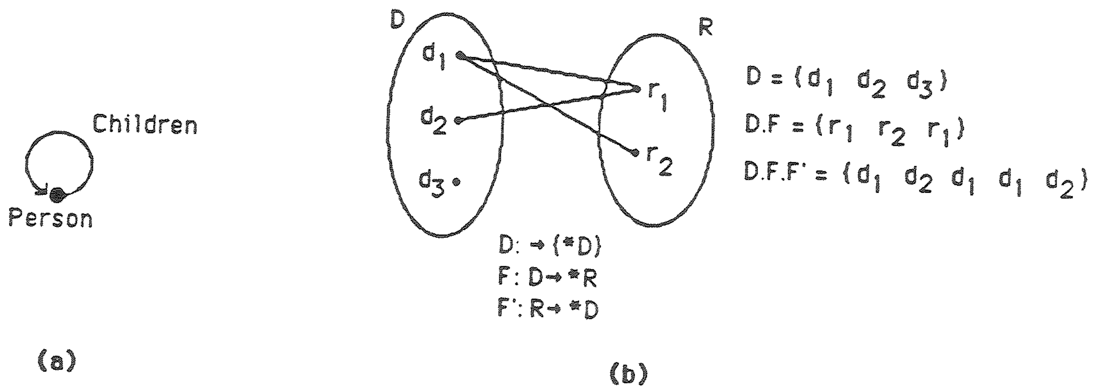



**Figure 2.3 Productions and Reverse Productions**

In the following sections, we survey some common productions and show how they can be used to query the department-employee-child database.

### 2.2.1 The Tuple, NORMALIZE, and Print Productions and the SELECT Command

Let $F_1 \cdots F_n$ be productions that map D objects. The production [$F_1, \cdots, F_n$] is a rewrite rule that replaces each object d in a stream of D objects with an n-tuple of the form [{d}.$F_1$, $\cdots$, {d}.$F_n$]. '[ , $\cdots$ , ]' is called the **tuple production.** [1]

Consider the expression Dept.[Dname, Emp.Ename]. Each department d in sequence Dept is replaced by the tuple [{d}.Dname, {d}.Emp.Ename]. Every tuple that is produced is an ordered pair of sequences; the first sequence contains a single object which is the name of department d and the second sequence contains the names of employees that work in department d. Figure 2.1c shows a portion of the department-employee-child database in relational form. DEPT_REL is the department relation and EMPL_REL is the employee relation. The stream that Dept.[Dname, Emp.Ename] produces is { [{Ceramics}, {Smith Jones Blake}], [{Plastics}, {Clark Adams}], [{Pottery}, {}] }. Note that the tuples are not in first normal form.

The NORMALIZE production replaces ¬1NF tuples with their stream of 1NF tuples. The 1NF_PRINT production prints a 1NF tuple. Thus, executing the expression:

[1] [...] is actually a higher-order production. It takes a list of productions as its argument and returns a production to replace objects with tuples as its result.

Dept.[Dname, Emp.Ename].NORMALIZE.1NF_PRINT

results in:

| Dname | Ename |
|---|---|
| Ceramics | Smith |
| Ceramics | Jones |
| Ceramics | Blake |
| Plastics | Clark |
| Plastics | Adams |

Note that normalization eliminates tuples that have null sequences (e.g., [{Pottery}, {}]). In cases where the elimination of these tuples is not desired, we rely on the extensibility of GDL by introducing a new production NORMALIZE_NV which retains (rather than discards) tuples with null sequences. [2]

The above expression is rather ungainly. Henceforth, we will write:

SELECT Dept.[Dname, Emp.Ename]

where SELECT is a macro which appends the .NORMALIZE.1NF_PRINT to the given expression if tuples are produced. An example where an expression doesn't produce tuples (and hence appending NORMALIZE.1NF_PRINT is not appropriate) is listing the numbers of all employees:

SELECT Empl.E#

In this expression, a .SEQ_PRINT production is appended to "Empl.E#" to print the sequence of employee numbers. In the case where a SELECT produces a single number or string, a .VAL_PRINT would be appended. The conditional appending of .NORMALIZE.1NF_PRINT, .SEQ_PRINT, or .VAL_PRINT provides a uniform way for displaying the results of GDL expressions.

### 2.2.2 The Comparison, EXISTS, and WHERE Productions

Comparison productions map sequences of objects to booleans. As an example, the less-than production LT(n):{*NUMBER}→BOOLEAN maps a sequence of numbers to true if any number in the sequence is less than n, where NUMBER is INT or FLOAT. Thus, {0 1 2 3 4}.LT(3) is true and {5 6 7}.LT(3) is false. Other comparison productions are EQ, NEQ, GT, GEQ, and LEQ. [3]

We simplify the syntax of comparison productions by dropping the parentheses and renaming the production with its symbolic counterpart (i.e., EQ(n) is the same as = n, and LT(n) is < n, etc.). This simplification allows GDL expressions to resemble their relational language counterparts, thus making them more readable.

A boolean expression produces a boolean result when evaluated. A **GDL predicate** is a collection of boolean expressions that are connected by the operators AND, OR, and NOT. Let $B_1, \cdots, B_n$ be boolean expressions, and let $P(B_1, \cdots, B_n)$ be a GDL predicate. An object d satisfies P if $P(\{d\}.B_1, \cdots, \{d\}.B_n)$ is true.

[2] Another possibility would be to introduce null values or null tokens. Doing so would require all productions to understand how to process or map them. We do not handle null values in our implementation, and will not examine null values further in this paper.

[3] LT(n) is actually a composition of two productions: lt(n) and ANY. lt(n) replaces each number in a sequence with a boolean indicating if that number is less than n. ANY forms the logical disjunction of a sequence of booleans. In this paper, we will treat LT(n) as a primitive.

The WHERE(GDL_predicate) production filters objects from a stream that do not satisfy GDL_predicate. Here are some examples:

*Query 1.* Print the E#'s of employees whose name is "Smith" and whose salary is greater than $40,000.

```
SELECT    Empl.WHERE( Ename = "Smith" AND Salary > 40000 ).E#
```

For each employee e, the predicate ({e}.Ename = "Smith") AND ({e}.Salary > 40000) is evaluated. For qualified employees, the E# is printed.

The EXISTS:{*OBJECT}→BOOLEAN production replaces nonempty sequences with true and empty sequences with false. Here is an example of its use:

*Query 2.* Print the names of departments that have at least one employee.

```
SELECT    Dept.WHERE( Emp.EXISTS ).Dname
```

For each department d, the predicate {d}.Emp.EXISTS identifies departments that have no employees.

*Query 3.* List the name of department #57 and the names of its employees that earn more than $40,000 and that have a child that is older than 4.

```
SELECT    Dept.WHERE( D# = 57 ).[Dname, Emp.WHERE( Salary > 40000 AND Child.Age > 4 ).Ename ]
```

In this example, WHERE productions are used both outside and inside the tuple production. The outside WHERE qualifies departments, while the inside WHERE qualifies employees.

### 2.2.3 Statistical Aggregation and the GROUP Production

COUNT, SUM, AVE, MAX, and MIN are statistical aggregation productions. COUNT:{*OBJECT}→INT is the rewrite rule that replaces a sequence of objects with the number of objects in the sequence. AVE:{*NUMBER}→FLOAT replaces a sequence of numbers with their average, where NUMBER is INT or FLOAT. The following queries illustrate their use.

*Query 4.* Print the total number of employees.

```
SELECT    Emp.COUNT
```

Emp generates the sequence of all employees and COUNT replaces this sequence with its length.

*Query 5.* Print the name of each department and the total number of its employees.

```
SELECT    Dept.[Dname, Emp.COUNT]
```

For each department d, a tuple [{d}.Dname, {d}.Emp.COUNT ] is produced. Note that COUNT counts the number of employees within a single department.

It is occasionally necessary to nest aggregations. This is accomplished by the GROUP production. GROUP:OBJECT→{OBJECT} replaces each object in a stream with the singleton sequence containing that object. Thus, $\{d_1 \cdots d_n\}$.GROUP equals $\{\{d_1\} \cdots \{d_n\}\}$. Here is an example of its use:

*Query 6.* Find the average salary of all employees in each department and print the maximum of these averages.

```
SELECT    Dept.GROUP.Emp.Salary.AVE.MAX
```

The subexpression Dept.GROUP.Emp.Salary.AVE produces a sequence of averages, one average for each department. (An average represents the average employee salary within a department). The concluding MAX finds the maximum in the sequence of averages.

It is instructive to see how streams are mapped in Query 6. Figure 2.4 lists the type of stream that is produced at intermediate points in its computation. Note that GDL productions *always* operate on the innermost sequences, i.e., a stream of type {*{*A}} is mapped by F:{*A}→B to a stream of type {*B}.

| Stream Rewrite Rule | Intermediate Production Composition | Intermediate Stream Type |
|---|---|---|
| Dept:→{*Dept} | Dept | {*Dept} |
| GROUP:OBJECT→{OBJECT} | Dept.GROUP | {*{Dept}} |
| Emp:Dept→*Empl | Dept.GROUP.Emp | {*{*Empl}} |
| Salary:Empl→FLOAT | Dept.GROUP.Emp.Salary | {*{*FLOAT}} |
| AVE:{*FLOAT}→FLOAT | Dept.GROUP.Emp.Salary.SUM | {*FLOAT} |
| MAX:{*FLOAT}→FLOAT | Dept.GROUP.Emp.Salary.SUM.MAX | FLOAT |

**Figure 2.4 A Summary of the Evaluation of the Evaluation of Query 6**

### 2.3 Distinguishing the Computation Models of FQL, DAPLEX, and GDL

The computation model of GDL was designed to capture the best features of the computation models of FQL and DAPLEX.

The computation model of FQL is based on sequences. No distinction is made between sequences and streams as is done in GDL. The FQL function F:A→$B replaces an A object with its sequence of B objects, denoted $B. Unlike GDL, F cannot map a sequence of objects directly. It must first be *extended* by the extend operator $. The function $F:$A→$$B maps a sequence of A objects to a sequence of sequences of B objects by applying F to each A object in the original sequence. This computation model frequently produces sequences that are nested deeply. For example, consider function G:B→$C. G can be composed with F only if it is extended. Thus, F.$G is a legal composition in FQL, while F.G is illegal. If F.$G is to map a sequence of A objects, it must be extended to $F.$$G. The multiplicity of extension operators quickly becomes burdensome as further compositions are made. GDL avoids extension operators altogether by its distinction of streams and sequences.

The computation model of DAPLEX handles function composition and extension similar to that of GDL. However, DAPLEX computations are based on sets, rather than streams or sequences. The ordering and duplication of objects within a set is permitted under special circumstances, but set nesting (e.g., set of sets) is not permitted. These restrictions cause difficulties in expressing aggregations. For example, to compute the average salary of employees in GDL is simple: Empl.Salary.AVE. The DAPLEX expression AVE(Salary(Empl)) does *not* compute the desired average, but instead computes the average of all *distinct* salaries. The correct expression, which retains duplicate salary figures, is AVE(Salary OVER Empl). Taking this one step further, queries involving nested aggregations, such as Query 6, are even more difficult to express because there is no simple way in which sets of aggregates can be formed [Yed86]. One way that Query 6 could be written is:

```
PRINT    MAX( i IN INTEGER  SUCH THAT
         FOR SOME d IN Dept : i = AVE( Salary OVER Empl(d) ) )
```

In general, DAPLEX expressions become more complicated as the nesting of aggregations increases. Both GDL and FQL avoid this problem by admitting nested sequences in their computation models.

## 2.4 1NF and ¬1NF Relational Databases

There is growing interest in ¬1NF relational databases [Ozs85, Sch86, Dad86, Rot86, Pis86]. ¬1NF relations are distinguished from their 1NF counterparts in that relation-valued attributes are admitted. These relations also can have relation-valued attributes, so that the nesting of relations can be arbitrarily deep. Figure 2.5 shows nested and unnested relation schemes for our Department-Employee-Child database. Note that these are just two of many possibilities.

Making the nesting of relations explicit has its drawbacks. Nesting relations in different orders can dramatically impact the way queries are expressed. As an example, SQL/NF is an extension of SQL to handle ¬1NF databases [Rot87]. In the context of 1NF schemes, SQL/NF reduces to SQL. Figure 2.6a-b shows SQL/NF expressions of Query 3 over the 1NF and ¬1NF schemes in Figure 2.5a-b. Note the dissimilarity in the way the same query is expressed for both databases.

GENESIS supports ¬1NF relations. One of the primary motivations for choosing GDM and GDL as the front-end to GENESIS was their ability to make the nesting of relations transparent. This is due to the fact that a view-like mechanism is embodied in the mapping of GDM databases to 1NF and ¬1NF relations. For this reason, GDM and GDL tend to simplify query formulation (cf. Query 3 with Figures 2.6a-b) and greatly facilitate experimentation with 1NF and ¬1NF storage schemes. The impact of nesting is confined to the implementation of productions, which is the subject of the remaining sections in this paper.

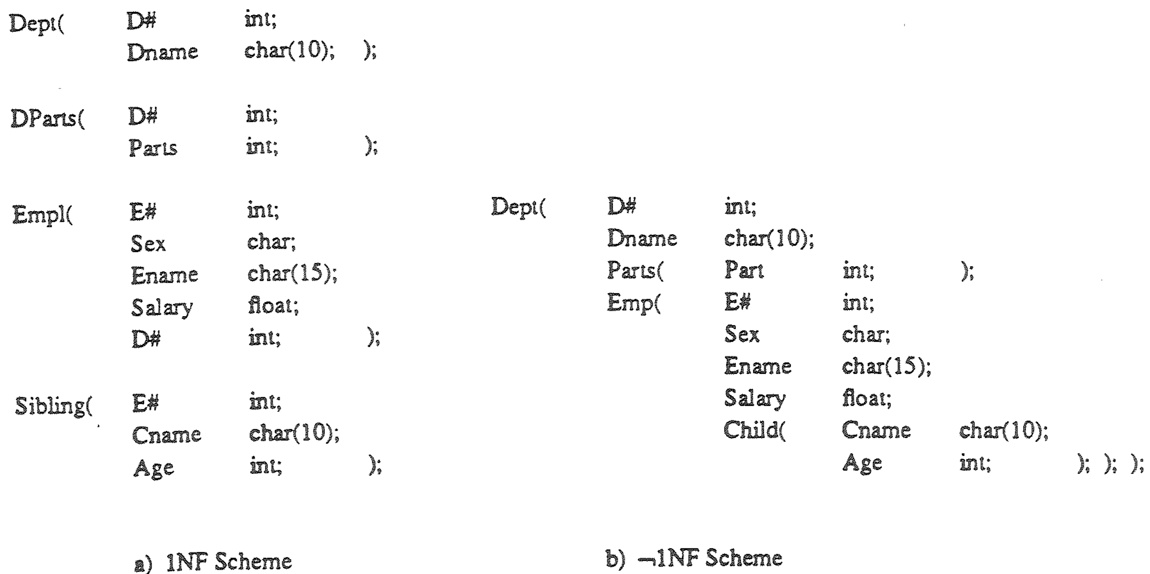



**Figure 2.5 1NF and ¬1NF Relation Schemes for the Department-Employee-Child Database**

```
SELECT  Dname, Ename
FROM    Dept, Empl, Sibling
WHERE   D# = 57
AND     Salary > 40000
AND     Age > 4
AND     Dept.D# = Empl.D#
AND     Empl.E# = Sibling.E#
```

a) 1NF Query

```
SELECT  Dname,( SELECT  Ename
                FROM    Emp
                WHERE   Salary > 40000
                AND     EXISTS( SELECT  *
                                FROM    Child
                                WHERE   Age > 4      ) )
FROM    Dept
WHERE   D# = 57
```

b) ¬1NF Query

**Figure 2.6 SQL/NF Expressions of Query 3 over 1NF and ¬1NF Relation Schemes**

## 3. The Big Picture

GDL expressions are not executed as they are written, but undergo an optimization similar to conventional relational queries. Although we are currently doing the optimization manually, the process can be automated. In the following, we explain how GDM and GDL fit into the 'big picture' of GENESIS, and how the GDL computation model forms a cornerstone of the GENESIS implementation.

GDM is an object-based front end to a relational-like storage system. GENESIS stores databases internally as networks of files and links, where a link is realized either as a CODASYL set or a join algorithm. The conversion from an object-based to a record-based representation is necessary for performance and comprehendability; algorithms for query processing, storage structures, recovery, etc. are traditionally expressed (and are best understood) in terms of records/tuples.

Query optimization in GENESIS involves a conversion between these representations and is accomplished in three steps: 1) mapping a GDL expression to an equivalent, but not optimized expression on files and links (henceforth called a record expression or r-expression), 2) optimizing the r-expression, and 3) evaluating the optimized expression. Figure 3.1 shows these steps and their intermediate results.

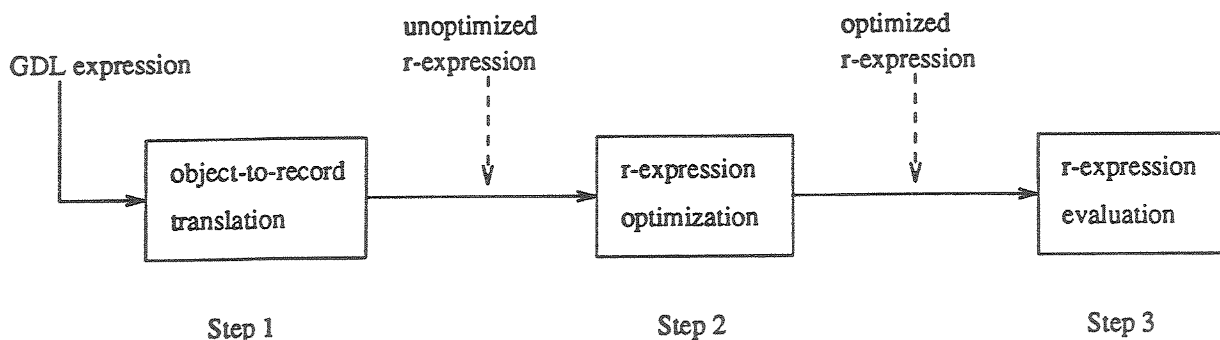



**Figure 3.1 Steps in GDL Expression Optimization**

The expression translation process (Step 1) relies on straightforward mappings between a GDM database and its network database counterpart. Many of the details of these mappings are not essential to this paper. Those that are important are explained in Section 5.1 where we show how records correspond to trees of objects.

The target of GDL expression translation and optimization is an r-expression which is a composition of operations on files and links. (The distinction between GDL expressions and r-expressions is similar to the differences between SQL statements and their relational algebra counterparts). What is significant here is that file and link operations are fundamentally no different than operations on object types. They too can be

realized as productions. The only distinction is that file and link operations map streams of *records*, rather than streams of *objects*.

As mentioned above, the correspondence of records to trees of objects permits GDL expressions to be applied directly to records. For this reason, it is common for file and link operations of r-expressions to use GDL expressions as arguments for record selection and attribute projection.

Consider the file retrieval production RET(F,Q,T). RET generates the sequence of records from file F that satisfy GDL boolean expression Q, and each qualified record is trimmed of unnecessary fields by the GDL tuple expression T. (This latter point is consistent with ¬1NF query languages where expressions, not attribute lists, are needed to express projections over ¬1NF relations [Rot87]).

To illustrate, recall the ¬1NF Dept relation of Figure 2.5b and consider the query to list the names of departments with the names of their employees that earn more than $30,000. The following are equivalent expressions:

```
GDL expression:     Dept.WHERE(D#=57).[Dname, Emp.WHERE(Salary>30000).Ename].NORMALIZE

r-expression:       RET(Dept, D#=57, [Dname, Emp.WHERE(Salary>30000).Ename].NORMALIZE)
```

That is, the r-expression realizes the GDL expression by selecting all Dept records that have D#=57 and applying the expression [Dname, Emp.WHERE(Salary>30000).Ename].NORMALIZE to the selected records.

Now consider the join production JOIN(F1,F2,J,T). JOIN generates the sequence of records that result from the join of record sequences F1 and F2 over join predicate J. For each record produced by the join, the GDL tuple expression T is applied to produce a stream of trimmed records. As an example, recall the 1NF database of Figure 2.5a and the query considered above. The following are equivalent expressions:

```
GDL expression:     Dept.WHERE(D#=57).[Dname, Emp.WHERE(Salary>30000).Ename].NORMALIZE

r-expression:       JOIN(   RET(Dept, D#=57, [Dname,D#].NORMALIZE),
                            RET(Emp, Salary>30000, [Ename,D#].NORMALIZE),
                            Dept.D#=Emp.D#,
                            [Dname, Ename] )
```

The r-expression realizes the GDL expression as a composition of RET and JOIN productions. The RET productions generate sequences of records from the Dept and Emp files, and the JOIN production joins these sequences to produce the desired result.

Three comments. First, the use of functional/production concepts to express both queries and internal DBMS computations provides a unifying theme to GENESIS; productions are a cornerstone of the GENESIS implementation effort.

Second, because r-expressions are similar to relational algebra expressions, they can be optimized (in Step 2) by traditional means (e.g., the System R algorithm [Sel79]) or by recently proposed rule-based optimizers (e.g., the EXODUS optimizer [Gra87]). Thus, our use of productions is consistent with existing research and enables us to build upon known results in query optimization.

Third, some productions take multiple streams as input to produce a single output stream. The JOIN production is an example. In principle, productions with multiple input streams are no more complicated than the single input stream productions illustrated in Section 2. (For example, one can define a MERGE production which merges two or more streams of objects. This production could be used to express the concatenation of streams resulting from several different GDL expressions).

It is worth noting that there are a few operations, such as join, that seem to require the rereading of portions of their input streams. For example, the join of two files over nonkey fields will often pair several records of one stream with multiple records of the other. Permitting portions of streams to be reread introduces significant inefficiencies and complications to the GDL computation model. We avoid this problem by requiring such productions to internally or externally buffer records/objects that might need to be reread. In this way, the mechanics for all but a handful of productions are kept simple.[4]

---

[4] Hash join algorithms read their input streams without rereading, but might require buffering of some portions of these

Finally, the last step (Step 3) in GDL expression optimization is to evaluate the resulting r-expression. The remaining sections in this paper deal with this topic. We show that it is easy to package algorithms as productions and to compose algorithms by hooking productions together. Although our discussions and examples are cast in terms of processing streams of objects, it is important to remember that algorithms that process streams of records (e.g., join and retrieval) can be implemented in a similar manner.

---

streams if multiple passes are needed [Sha86]. Sort merge joins would require buffering to avoid rereading. Nested loop joins don't require buffering at all. For each outer file record, an expression is evaluated to retrieve all inner file records that have the same join value as the outer file record. The same inner file record may appear in *different* streams (corresponding to different execution instances of the inner expression), but all streams are read sequentially, and 'backing up n records' is never performed.

## 4. Implementation Concepts

We explain in this section how productions can be implemented in software and their compositions executed on a single processor. We note that one of the potential advantages of functional (or production) data languages is the ease with which expressions can be mapped to a parallel, multiprocessor or multithreaded process environment. Each production of an expression, for example, could be executed on (or implemented by) a different processor/process. Although the potential for distributed and parallel implementations of GDL expressions is present, the first step toward more sophisticated realizations is to understand their implementation in a conventional setting.

We begin by presenting an encoding of GDL sequences which simplifies computations on streams. We show how productions can be realized as stream translators, and how translators can be linked together to evaluate GDL expressions. We then survey different strategies by which productions can be implemented. Finally, we present templates for packaging algorithms of stream translators.

### 4.1 Another Sequence Encoding

A GDL sequence contains elements of a single rank. An element can be an object (rank 0), a sequence of objects (rank 1), a sequence of sequence of objects (rank 2), and so on. Thus, sequences like { { a } { b c } { } } are legal, as each element is a sequence of zero or more objects. Sequences like { { a } b }, where elements are either sequences or single objects, are disallowed. The reason for this restriction lies in the definition of productions: they replace all elements of a common rank with zero or more elements of another.

As we noted in the Section 2, GDL sequences can be encoded as a stream of objects and braces. It turns out that there is a more convenient way to encode streams which takes advantage of the above-mentioned regularity. The benefit of this encoding is slightly shorter streams and simpler translation algorithms, both of which lead to more efficient implementations.

The encoding we use replaces brace tokens with initialization and separator tokens. A stream begins with an **initialization token** '#' to indicate the start of a stream. Start-of-sequence braces '{' are eliminated entirely. Each end-of-sequence brace '}' is replaced by an indexed **separator** '$|_i$' whose index i indicates the nesting depth of the end-brace. The innermost sequences are assigned index 0, while outer-level sequences have incrementally larger values. Indexing is required to differentiate null sequences from the end of an outer-level sequence. Here are some examples:

| | | |
|---|---|---|
| { a b c d } | is encoded as | # a b c d $|_0$ |
| { { a } { b c } { } { d } } | is encoded as | # a $|_0$ b c $|_0$ $|_0$ d $|_0$ $|_1$ |
| { { { a b } { c } } { { d } { } } } | is encoded as | # a b $|_0$ c $|_0$ $|_1$ d $|_0$ $|_0$ $|_1$ $|_2$ |

### 4.2 Stream Translators

A **stream translator** is a procedural definition of a production. It lists the actions that are taken to process a token (object, initialization, or separator) on an input stream. Figure 4.1 shows the stream translator for the production F:A→*B. Initialization and separator tokens are transmitted directly and A objects are replaced by their stream of B objects.

An overview representation of F is shown in Figure 4.2a. F translates a **primary input stream** of tokens to a **primary output stream**. This representation of F, as we will see in the following section, is useful in describing how different translators can be hooked together.

Many productions have stream translators that are similar to F. GROUP, aggregation productions (e.g., COUNT), generators (e.g., A:→{*A}), EXISTS, and comparison productions (e.g., EQ(n)) are examples. Their translators are shown in Figure 4.2b, all of which have straightforward interpretations.

The GROUP translator directly transmits initialization tokens. Each object 'a' that it receives is mapped to 'a $|_0$', an encoding of '{ a }'. Level i separators are mapped to level i+1 separators to indicate an increase in the level of nesting.

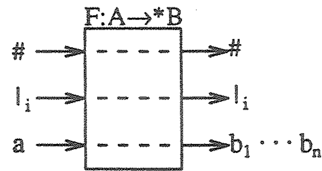



**Figure 4.1 A Stream Translator for F:A→*B**

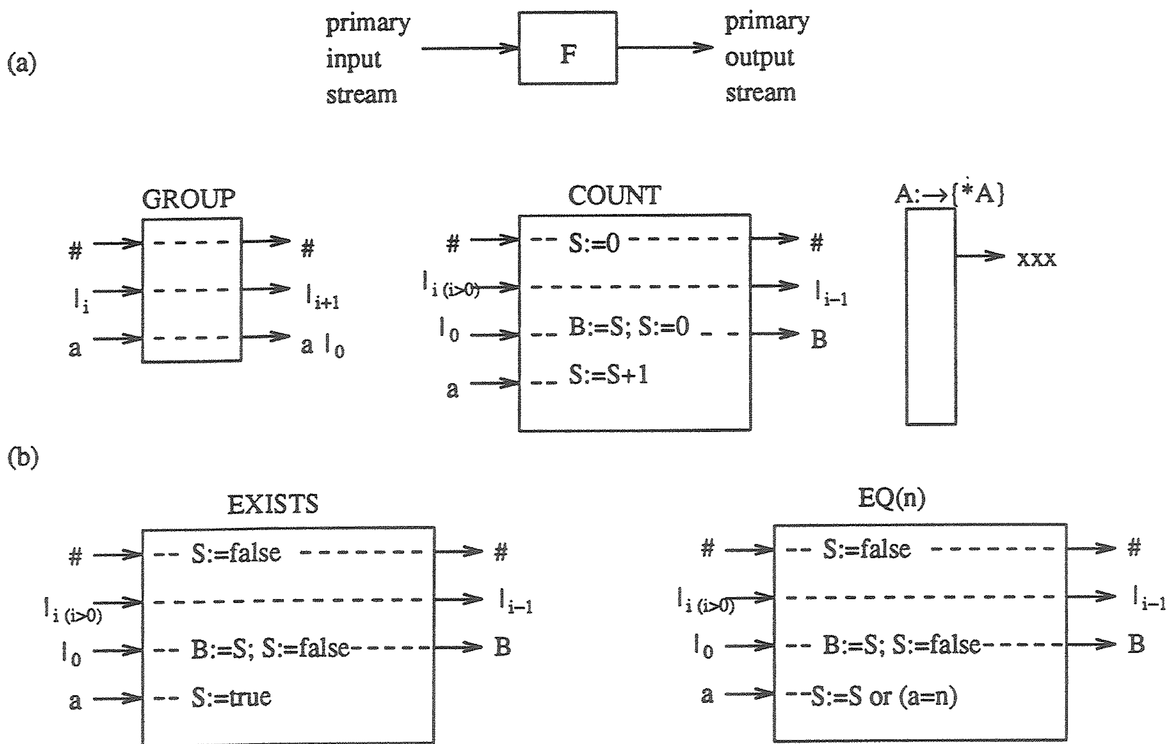



**Figure 4.2 Stream Translators Without Secondary Streams**

The COUNT translator, as another example, clears an accumulator (S) on reception of an initialization token and transmits the token. Each object 'a' that it receives increments the accumulator; no tokens are placed on the primary output stream. On reception of a level 0 separator, the contents of the accumulator are transmitted and the accumulator is cleared. Level i (i>0) separators are translated to level i-1 separators to indicate a decrease in the level of nesting.

Some productions, such as WHERE(), are more complicated as they allow expressions (rather than constants) as parameters. Figure 4.3a shows an overview representation of a production F that has n expression-valued parameters. Consider the expression A.F($E_1, \cdots, E_n$). For each object a of type A, the expressions {a}.$E_1, \cdots$, {a}.$E_n$ are evaluated before F can evaluate a. This means that when an object a is received by F on its primary input stream, a is placed (along with a level 0 separator) on its **secondary output stream** in order for it to be received by all arguments of F. The results of evaluating {a}.$E_1, \cdots$, {a}.$E_n$ are returned to F along F's **secondary input stream.** [5] On the basis of the returned results, F places its result on the primary output stream. Figure 4.3b shows the translators for WHERE, TUPLE, AND, and OR.

---

[5] Actually, there is one secondary input stream and one secondary output stream for each argument of F that is

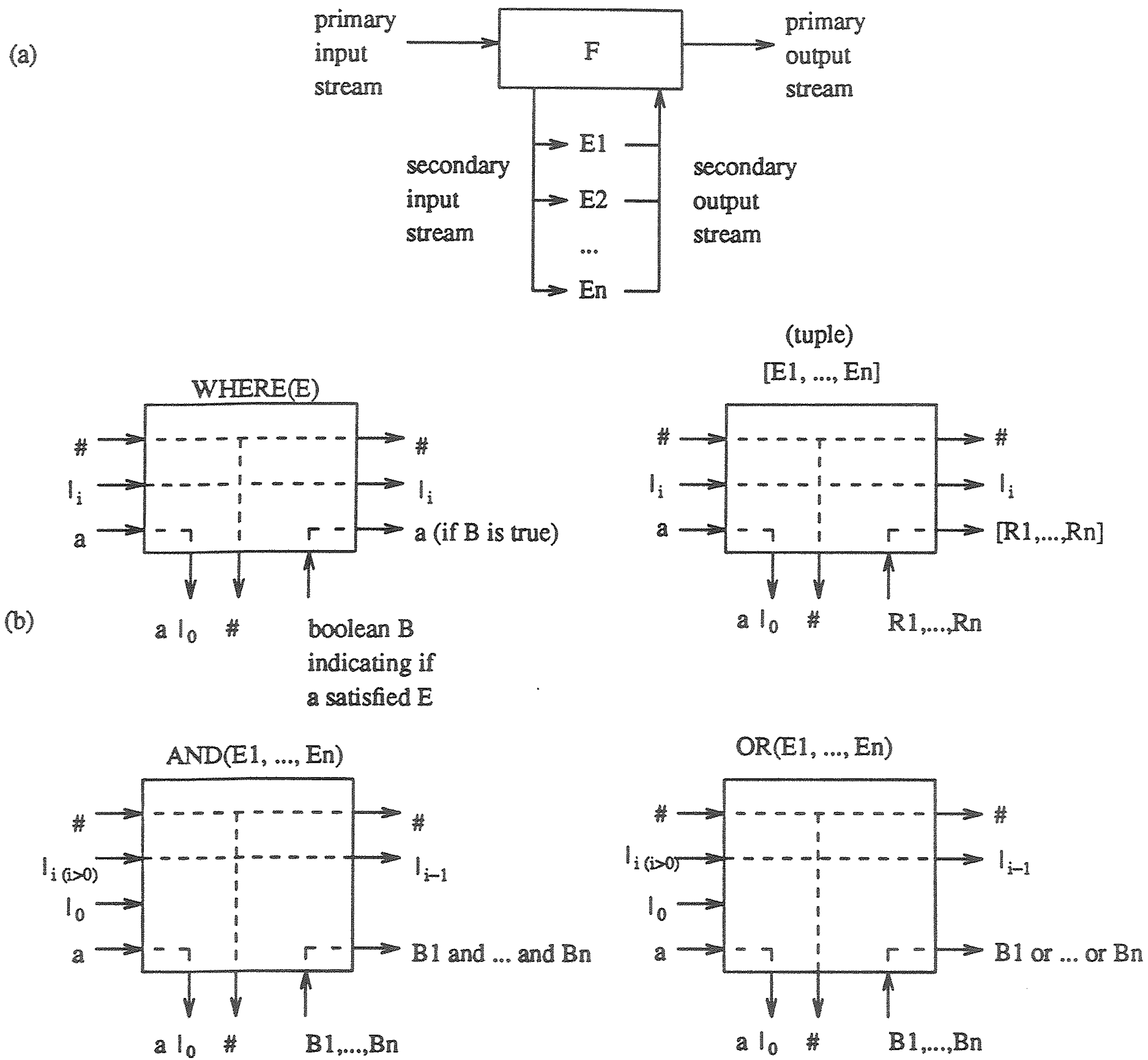



**Figure 4.3 Stream Translators With Secondary Streams**

expression-valued. This level of implementation detail is discussed in Section 4.5.

### 4.3 Pure Expressions and Translator Networks

As mentioned in Section 2, SELECT operations are abbreviations of GDL expressions that are actually evaluated. Specifically, we noted the implicit reliance on NORMALIZE, 1NF_PRINT, SEQ_PRINT, and VAL_PRINT productions, and the use of abreviated names for EQ (=), LT (<), etc. A GDL expression in which all productions are explicit, abbreviated names are excluded, and boolean expressions involving ANDs and ORs are written in operator form (i.e., A OR B OR C is written as OR(A,B,C)) is called **pure.**

A pure expression is a list of productions. A **translator network** of a pure expression is the linking of stream translators in the order in which their corresponding productions appear in the expression. Figures 4.4a-c show a simple GDL expression, its pure expression counterpart, and its translator network.

(a) SELECT A.WHERE( B=4 OR D<7 ).[E.F]

(b) A.WHERE( OR( B.EQ(4), D.LT(7) ) ).[ E, F ].NORMALIZE.1NF_PRINT

(c)

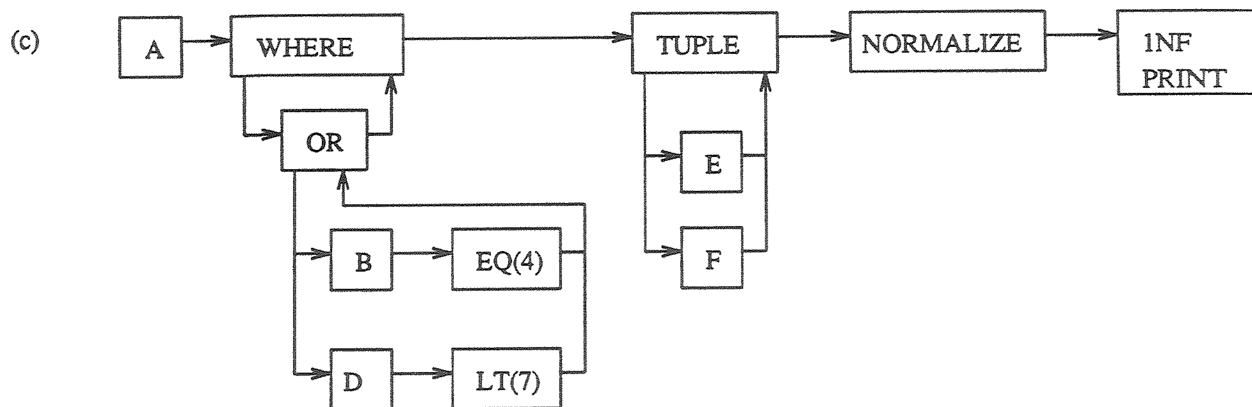



**Figure 4.4 Pure Expressions and Translator Networks**

### 4.4 Evaluation Methods

Translators can evaluate their input streams in an eager or lazy manner. **Eager evaluation** is used in APL and LISP where each function (translator) computes its output result (stream) in its entirety before execution passes to the next function (translator). Because there is considerable overhead in maintaining large lists in main memory and the attendant difficulties of garbage collection, eager implementations of translators for database processing do not seem practical.

**Lazy evaluation,** on the other hand, is more appropriate. Only the tokens of a stream that are actually needed are computed, and computations occur only on request. [6] The ubiquitous GET_FIRST and GET_NEXT in procedural or embedded database languages are the means by which records of a sequence are produced in a lazy manner.

Lazy evaluation can be achieved in a data-driven or demand-driven manner. Computations are triggered in **data-driven** implementations by the reception of a token [Rit74], whereas computations are triggered by the request for a token in **demand-driven** implementations [Hen80]. When implemented by a single-threaded process, both methods are quite similar as the following example shows.

---

[6] There are, of course, productions which cannot exploit lazy evaluation. The SORT production, which sorts a sequence of objects, may require the entire primary input stream to be present before sorting can proceed.

Consider the translator network for expression A.B.C. In a data-driven implementation, execution begins at translator A. A's output tokens flow, one at a time, to B. For each A token, B computes its output tokens, and they flow, one at a time, to C. Once C completes its computation on a B token, control returns to B for it to output its next token to C. When B no longer can output tokens, control returns to A.

In a demand-driven implementation, execution begins at translator C. C requests a token from B, which in turn, requests a token from A. As A produces tokens, B consumes them and outputs its tokens, on demand, to C. Once A has produced all of its tokens, and B has consumed them, control returns to C.

As this example suggests, data-driven and demand-driven computations are similar. In a functional context, A.B.C is equivalent to C(B(A)). Data-driven corresponds to an inside-to-outside evaluation of the expression; demand-driven corresponds to the usual functional (outside-to-inside) evaluation. We will see later that data-driven and demand-driven implementations generally have a comparable performance. Their primary distinction appears to be the difficulty or ease with which they handle multiple input streams and common subexpressions.

Consider the MERGE production which merges two sequences A and B (see Fig. 4.5a). Demand-driven implementations have no problem here; MERGE alternately demands tokens from A and B. In a data-driven context, scheduling difficulties arise when A and B should be executed. The problem is evident when a single-threaded process executes the network of Figure 4.5a. Either A or B begins execution, but not both. Before MERGE can process its input, both A and B must be executing. So in processing multiple streams, demand-driven has a simpler implementation.

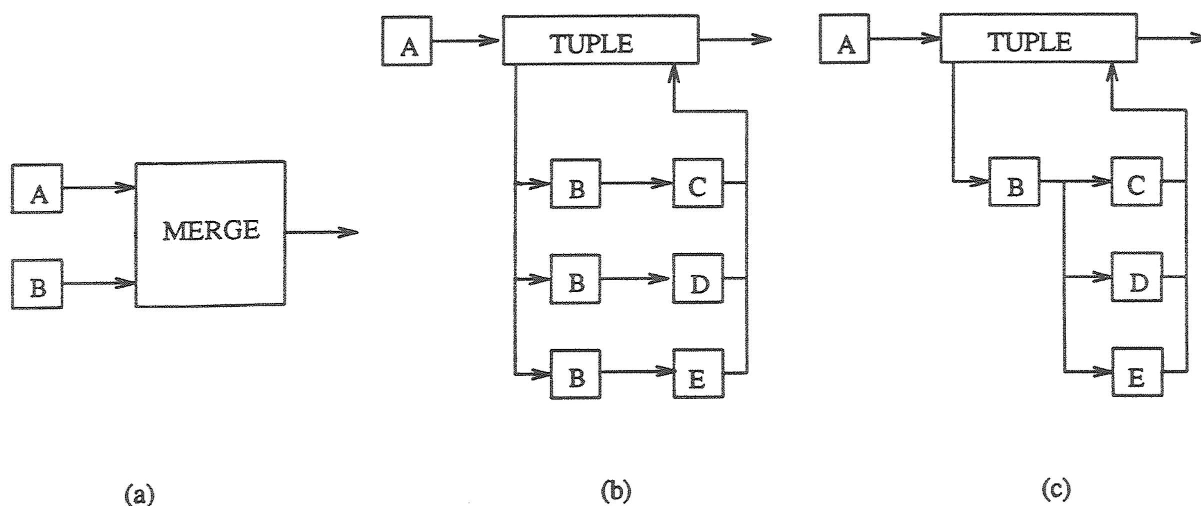



**Figure 4.5 Limitations of Demand-Driven and Data-Driven Computations**

A second difference, this one in favor of data-driven translators, concerns expressions that involve redundant computations. In the expression A.[B.C, B.D, B.E], the subexpression {a}.B is computed three times for each a in A (see Fig. 4.5b). Networks for data-driven translators can be rearranged so that redundant computations are eliminated (see Fig. 4.5c). That is, the result of the expression {a}.B is broadcast to the networks for C, D, and E. Networks for demand-driven translators cannot be rearranged in this manner; eliminating redundant computations in demand-driven implementations is known to be a difficult problem. The most common technique involves maintaining a list of tokens, and reclaiming tokens (nodes) on the list after all translators have processed it [Tur79]. As mentioned earlier, processing in-core lists is not appropriate for efficient database processing. For this reason, we do not attempt to eliminate redundant computations in demand-driven implementations. [7]

[7] At the time of writing this paper, we believe that it is possible to avoid redundant computations without retaining in-core lists. A paper describing the technique is forthcoming.

Although it may not be immediately apparent, common subexpression problems do *not* arise in a demand-driven evaluation of the expression X.[Y, Z, W]. Each time TUPLE gets an x object, it evaluates expressions {x}.Y, {x}.Z, and {x}.W in succession. TUPLE never demands another x object until all three expressions have been evaluated. Further, the result of each of these expressions is either a single object (resulting from an aggregation) or a sequence of objects (which is terminated by a $|_0$ token). In either case, TUPLE knows when to stop demanding tokens from each expression. The key to solving the common subexpression problem in a demand-driven environment is synchronizing the broadcast of objects *and* making certain that each subexpression is prepared to receive that object. A demand-driven TUPLE solves a simple subproblem of the more general problem.

These observations do not rule out or favor a data-driven or demand-driven implementation of stream translators. In Section 5 we explore their differences and efficiencies further by experimental means. Before we proceed to these results, we briefly describe the coding templates that we are using to implement translators.

### 4.5 Translator Templates

Lazy evaluation can be realized by coroutines [Knu73, Hen80]. Each translator is a coroutine. It receives tokens on its primary input stream from its **supplier** and sends tokens on its primary output stream to its **consumer**. Again consider the network for A.B.C. The supplier to B is A, and the consumer of B is C. A has no supplier and C has no consumer.

Translators communicate via tokens, object descriptors, and signals. A **token** is an output of a translator, and is one of the values #, obj (for object), $|_0$, $|_1$, ..., $|_n$. When a translator produces an object, it outputs the obj token and an **object descriptor**, which is a pointer to a buffer that contains the actual object. (Passing pointers to objects is much more efficient than passing the objects themselves). A **signal** is a value that is transmitted from a translator to its supplier; valid signals are NEXT and SKIP. NEXT requests the production of the next token. SKIP causes the remaining objects in the current level 0 stream to be skipped, causing the supplier to produce $|_0$ as its next output token.

SKIP is needed for lazy evaluation. Recall that EQ(n):{*NUMBER}→BOOLEAN maps a sequence of numbers to true if the sequence contains value n. Suppose value n appears in the sequence. The result of EQ(n) will be true, no matter how many values come after n in the sequence. By not generating these values, a substantial savings of unnecessary computation can result. Productions like EQ(n) can take advantage of this optimization by using the SKIP signal. Note that SKIP is useful for both data-driven and demand-driven translator implementions.

Tokens, signals, and object descriptors are exchanged between translators via mailboxes. A **mailbox** is a shared variable that has one reader and one writer. As shown in Figure 4.6a, a translator uses six mailboxes; three are used to communicate with its supplier, the other three with its consumer. A translator receives tokens and object descriptors from its supplier in its in_token and in_obj_des mailboxes, and signals from its consumer in its in_signal mailbox. The out_token, out_obj_des, and out_signal mailboxes are used to send tokens, object descriptors, and signals.

Mailboxes are shared among translators of a network (as indicated in Figure 4.6b) in the following way. When a network is initially created, each translator is allocated an activation record which contains its three input mailboxes (in_token, in_obj_des, and in_signal), local variables, and state information. Activation records are then connected via pointers to reflect the supplier/consumer relationships of the network. The output mailboxes of a translator are then found in the activation records of its supplier and consumer translator.

Every translator network has an activation record. This record contains the supplier mailbox (out_signal) for the first translator of the network, and the consumer mailboxes (out_token and out_obj_des) for the last translator. These mailboxes enable entire networks to be referenced as individual translators. Translators that have secondary streams (such as WHERE and TUPLE) use this arrangement to communicate with their subnetworks. In general, if a production has n expressions as parameters, its translator will manage n activation records, one for each of the n expressions (subnetworks) that are to be evaluated. The results of computations for different subnetworks will thus appear in different mailboxes.

When a network is evaluated, translators communicate by placing their results in mailboxes, and transferring control to the supplier or consumer translator to process its contents. Transfer of control is accomplished via input and output statements. Input transfers control to the translator's supplier; output transfers control to

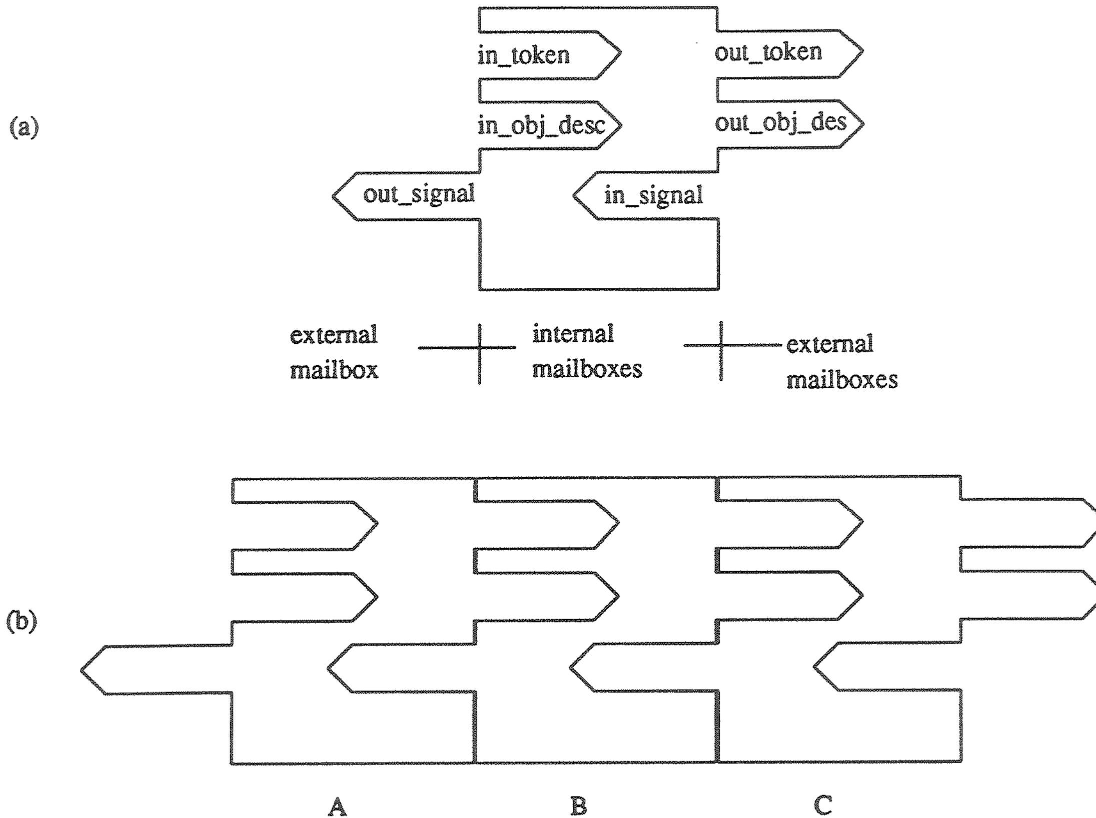



**Figure 4.6 Translator Mailboxes**

the consumer.

The internal structure of translators is quite simple: it is a single loop that encompasses a switch statement. Each cycle through the loop processes an input token. The switch directs translator execution to an appropriate action, based on the token received. There is one case for each token type. Figure 4.7 shows the templates for demand-driven and data-driven translators with one input stream and one output stream. Note that their only difference is the placement of the input statement (which requests input from the supplier). When a demand-driven translator is first called, it immediately demands input from its supplier. In contrast, a data-driven translator is already given its input (in its mailbox) and must process it before requesting additional input.

Templates for translators that process two or more primary input streams are not very different from the above. Figure 4.8 shows the translator for the MERGE(A,B) production, which merges ordered streams A and B into a single ordered stream. (We've simplified the MERGE code by eliminating the additional logic for processing SKIP demands). In the general case, each primary input stream has its own set of mailboxes. The number of cases within a switch statement equals the number of legal combination of states that different streams can be in. The definition of MERGE in Figure 4.8 requires it to operate on pairs of sequences. Thus, an illegal state is for a sequence to be present on one stream (i.e., the current token is an object) while no sequence present on the other (i.e., its current token is $|_1$).

Three comments. First, singleton sequences {a} arise frequently in translators that have secondary input and output streams. A way to process these streams efficiently is by pairing the obj and $|_0$ tokens to form a single token obj_$|_0$. The action associated with this token is a concatenation of the actions for obj and $|_0$.

**demand-driven translator template**

```
translator()
{  <local variables and local mailboxes>

   loop {
      out_signal = in_signal;
      input;        /* demand next token */

      switch( in_token ) {
      case #:       /* initialize translator */
      case obj:     /* process object */
      case I0:      /* process I0 */
      ...
      case In:      /* process In */
      }; /* end switch */

   }; /* end loop */
}; /* end translator */
```

**data-driven translator template**

```
translator()
{  <local variables and local mailboxes>

   loop {

      switch( in_token ) {
      case #:       /* initialize translator */
      case obj:     /* process object */
      case I0:      /* process I0 */
      ...
      case In:      /* process In */
      }; /* end switch */

      out_signal = in_signal;
      input;        /* wait for next token */
   }; /* end loop */
}; /* end translator */
```

**Figure 4.7 Demand-Driven and Data-Driven Coding Templates**

Second, packaging algorithms within translator templates is straightforward. Separators and initialization tokens merely serve as computation delimiters. The benefit of this packaging is a standarized interface and communication protocol for DBMS algorithms. As we will explain in Section 6, this standardization simplifies DBMS extensibility.

Third, the code that defines the structure of translator templates (e.g., a loop and a switch) introduces very little overhead. For computations whose run-times are in tens of milliseconds, the overhead imposed by templates is negligible. Certainly, productions that involve i/o, such as retrievals, joins, sorts etc. should exhibit the same performance as non-translator-packaged implementations of these algorithms. It is only in the case of very lightweight computations, such as those involved in GDL predicate evaluation where contents of individual fields are read and compared, will the overhead be noticable. In the next section, we experimentally measure these overheads in evaluating predicates.

```
MERGE() /* a demand-driven translator to merge ordered input streams A and B */
{   define A input stream mailboxes: in_token_A, in_obj_des_A;
    define B input stream mailboxes: in_token_B, in_obj_des_B;
    define output stream mailbox: in_signal;

    out_signal_A = NEXT;  /* demand a token from stream A and stream B */
    input_A;
    out_signal_B = NEXT;
    input_B;

    loop {
      switch( in_token_A, in_token_B )  {
      case (#, #):      out_token = #;
                        output;
                        out_signal_A = NEXT;
                        input_A;
                        out_signal_B = NEXT;
                        input_B;
                        break;

      case (obj, obj):  if (contents(in_obj_des_A) < contents(in_obj_des_B))
                        {     out_obj_des = in_obj_des_A;
                              out_token = obj;
                              output;
                              out_signal_A = NEXT;
                              input_A }
                        else { out_obj_des = in_obj_des_B;
                              out_token = obj;
                              output;
                              out_signal_B = NEXT;
                              input_B };
                        break;
     case (obj, |0):    out_obj_des = in_obj_des_A;
                        out_token = obj;
                        output;
                        out_signal_A = NEXT;
                        input_A;
                        break;
     case (|0, obj):    out_obj_des = in_obj_des_B;
                        out_token = obj;
                        output;
                        out_signal_B = NEXT;
                        input_B;
                        break;
     case (|i, |i):     out_token = |i;
                        output;
                        out_signal_A = NEXT;
                        input_A;
                        out_signal_B = NEXT;
                        input_B;
                        break;
     default:     /* error:: illegal state */
     }; /* end switch */
   }; /* end loop */
};  /* end translator */
```

**Figure 4.10 A Demand-Driven MERGE Translator**

## 5. GENESIS-Specific Implementation Results

We begin this section with a discussion of how records are currently represented in GENESIS. We then present preliminary experimental results on the performance of data-driven and demand-driven translator implementations, and compare the results to a more conventional means of implementation. Finally, we comment on the possibilities and advantages of compiling, rather than interpreting, GDL expressions.

### 5.1 Record Representation

A GENESIS record is essentially a COBOL record; the contents of scalar and repeating fields are concatenated to form a contiguous sequence of bytes. Every GENESIS record is conceptually viewed as an ordered tree of fields [Smi85], which in turn corresponds to a tree of object types in GDM. The root of a tree represents an entire record. Its siblings are the record's fields, their siblings are their subfields, and so on. The ordering of nodes reflects the ordering of the fields in the record. The leaves of a tree are scalar fields of a primitive type, such as CHAR, INT, and FLOAT. Non-leaf nodes correspond to repeating groups, where elements are treated as subfields.

Figure 5.1a shows a ¬1NF representation of a Dept record, and Figure 5.1b shows its tree representation. The schemes for both were defined earlier in Figure 2.5b. Beside each node in the tree representation is the name of the corresponding field and its ordinal number in parentheses. Under each leaf node is the data value that is contained in the field. The depicted record has Dname="Ceramics" and two employees, neither of which have children.

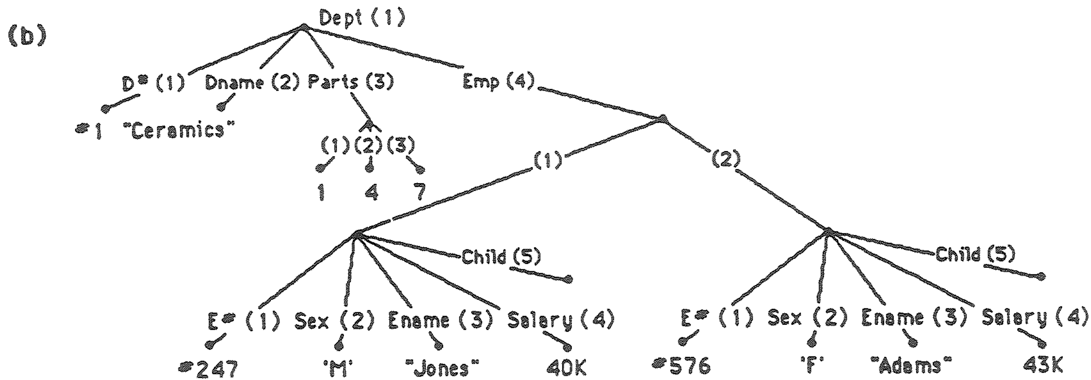



**Figure 5.1 ¬1NF and Tree Graph Representations of a GENESIS Record**

A **trace** is a GENESIS run-time data structure. It is used to identify the path from the root of a record to the node in question. The ordinal trace to the field containing "Jones" in Figure 5.1b is (1,4,1,3). Traces serve as cursors to fields of records in GENESIS.

Operations such as UP, DOWN, LEFT, and RIGHT, enable a trace to be repositioned on different fields within a record. Once positioned, a field can be read, updated, and in the case of a repeating field, new elements can be added and old elements deleted.

Each Dept object is identified with a Dept record in Figure 5.1. Applying the production Dname:Dept→CHAR(10) to a Dept object yields the name of the object's department. In terms of trace operations, the Dname production takes a trace that points to the entire Dept record and repositions it to the Dname field. Similarly, applying Emp:Dept→*Empl to a Dept object yields the stream of employees associated with that department. In terms of traces, this production takes a trace over the entire Dept record and repositions it to each element of the Empl repeating field, one element at a time. There is, in general, a straightforward implementation of stream translators that use traces and trace operations to navigate through fields of GENESIS records. It is this implementation that we evaluate in the next section.

### 5.2 Predicate Evaluation Experiments

As mentioned earlier, the runtime overhead incurred by a translator template packaging of algorithms is negligible for all but the simplest of computations. The computations that are most influenced are those involving the evaluation of predicates, where the contents of individual fields are examined. With this in mind, we built data-driven and demand-driven stream translators to evaluate GDL predicates over 1NF and ¬1NF GENESIS records. As a benchmark for these experiments, we also implemented parse trees which are used in some DBMSs to evaluate predicates on 1NF records.

We generated a set of predicates over the Department-Employee-Child database to examine the behaviors of the parse tree (PT), data-driven (DD), and demand-driven (DM) methods. Table 5.1 lists a representative sampling of these predicates. As targets of the experiment, 1NF Empl records and ¬1NF Dept records were used. (The schemes for these records were listed in Figure 2.5).

| Pred. # | Predicates | PT method | DM method | DD method | # of translators |
|---|---|---|---|---|---|
| A1 | Sex='x' (a character) | 0.10 | 0.10 | 0.10 | 1 |
| A2 | Salary=10.0 (a double prec. ft. pt. number) | 0.11 | 0.11 | 0.12 | 1 |
| A3 | D#=0 (an integer) | 0.10 | 0.10 | 0.10 | 1 |
| A4 | Ename="xxxxxxxxx" (a string of 9 characters) | 0.22 | 0.22 | 0.23 | 1 |
| B1 | Sex='x' AND Ename="x" | 0.32 | 0.33 | 0.41 | 3 |
| B2 | Sex='x' OR Ename="x" | 0.30 | 0.32 | 0.38 | 3 |
| B3 | Ename="x" AND Ename="x" AND Ename="x" | 0.54 | 0.55 | 0.63 | 4 |
| B4 | Ename="x" OR Ename="x" OR Ename="x" | 0.51 | 0.53 | 0.60 | 4 |
| B5 | Ename="x" AND (Ename="x" OR Ename="x") | 0.54 | 0.58 | 0.72 | 5 |
| B6 | Ename="xxxxxxxxa" OR (Ename="xxxxxxxx" AND Ename="xxxxxxxx") | 0.76 | 0.79 | 0.92 | 5 |
| C1 | Parts=10 | | 2.20 | 2.30 | 3 |
| C2 | Emp.Ename="xx" | | 2.88 | 3.20 | 4 |
| C3 | Emp.Child.Cname="xx" | | 5.67 | 6.03 | 5 |
| D1 | Emp.Eno>10 OR Emp.Ename="xx" | | 5.70 | 5.53 | 6DD 7DM |
| D2 | Emp.Child.Age>10 OR Emp.Child.Cname="xx" | | 11.41 | 9.10 | 7DD 9DM |
| D3 | Parts=10 OR Emp.Ename="xx" | | 5.23 | 5.78 | 6 |

Note: Times measured in milliseconds.

**Table 5.1 Predicates and Experimental Results**

Predicates in Table 5.1 are organized into four groups. Groups A and B are predicates over 1NF Empl records. Group A deals with primitive data types (integer, float, character, string) and group B deals with conjunctions and disjunctions of Group A predicates. Groups C - D are predicates over ¬1NF Dept records.

Group C tests for element membership in repeating groups and nested repeating groups, and group D involves conjunctions and disjunctions of Group A and Group C predicates. The number of translators used to express each predicate is listed in the left-most column.

The experiments were conducted on a dedicated VAX 750. Table 5.1 lists for each predicate the measured evaluation times. For Groups C and D, no times are listed for the PT method, as PTs apply only to 1NF queries. All measured times are given in milliseconds per evaluation.

Consider first the results of 1NF predicates on Empl records (Groups A and B). As expected, the DM and DD methods exhibit a comparable performance. DD is slightly slower only for the reason that we implemented a broadcasting capability to handle subexpression optimizations, whereas the DM implementation was not burdened by this extra overhead. (This required the introduction of a list of primary output stream mailboxes, one set of mailboxes for each translator to receive the broadcast). As can be seen from Table 5.1, the differences are rather small even under this biased case. More importantly, note that the performance of the DD/DM methods is virtually identical to the PT method. This equality was achieved by a simple optimization: translators that always produce a constant output are removed from the network. In our experiments, translators that computed constant offsets to fields were eliminated.

The predicates in Groups A and B certainly do not form a 'comprehensive set' of experiments for 1NF queries. As it is debatable whether any set is in fact comprehensive, we developed and validated an analytic model in [Leu86] to show that there is virtually no difference in execution times in PT and DM methods for a large class of 1NF predicates, which include those in Groups A and B.

Now consider the ¬1NF predicates on Dept records (Groups C-D). A cursory comparison of the 1NF and ¬1NF query run-times in Table 4.1 makes it tempting to conclude that ¬1NF implementations are too slow to compete with 1NF implementations. This would be wrong. On close inspection, two factors significantly inflate ¬1NF predicate timings. First, timing measurements on ¬1NF queries are record dependent. [8] For example, the more elements a repeating field has, the longer it will take for a predicate on that field to be evaluated.

Second, and most important, our implementation of stream translators relied on a prototype and unoptimized implementation of trace operations. This inflated the runtimes substantially. As both DM and DD methods call exactly the same trace operations (albeit in different orders), it is the difference in their run-times that is important. And these differences are marginal. The only known exception, as mentioned in Section 4.4, is when DD networks are rearranged to eliminate redundant computations of common subexpressions. The predicates D1 and D2 are such examples; "Emp" is the common subexpression for D1 and "Emp.Child" is the subexpression for D2. Note that D2 required 7 DD translators or 9 DM translators, and D1 used 6 DD or 7 DM translators.

The experiments confirm that both demand-driven and data-driven implementations of translators are suitable for database processing. We have chosen to use demand-driven translators as the basis for further research in GENESIS.

### 5.3 Interpretation v.s. Compilation

Executing a translator network is akin to an interpretive execution of an expression. If the functions of an expression involve little computation, it is well-known that a compiled version of the expression will run faster. (The reason, again, is that the overhead for crossing function boundaries is large compared to light-weight computations; compilation eliminates these boundaries and their overheads). If heavy computations are involved, the differences are negligible. The absolute run-times reported in our experiments indicate that interpretation is indeed expensive at the predicate evaluation level; a method of compiling a GDL expression/predicate into a single translator is needed. This is in line with standard implementations of high-performance DBMSs where predicates are compiled into machine code to gain an advantage in execution speed [Cha81].

---

[8] The records used in these experiments had a static form: the Parts repeating field had six elements; the Empl repeating field had three elements: one with no children, the second with one child, and the third with three children.

It is beyond the scope of our research to investigate the difficulties and merits of compiling a GDL expression (or subexpression) into a single stream translator. However, we note that a considerable body of relevant research exists on compiling functional programs [Hen80], and advanced compiler techniques are being studied which interweave the executions of composed functions to significantly decrease execution times [Fre86a-b]. Although we leave this problem open, whether an interpretive or compiled execution of translation networks is used does not diminish the usefulness of productions as the basis for a database computation model.

**6. Extensibility Issues**

As mentioned in the introduction, adding new data types and operators will be an essential feature of future DBMSs. GDM and GDL are well-suited for this task because they are 'open-ended'; i.e., there is no fixed set of data types and productions to be supported, and all data types and all productions are treated uniformily.

We envision that a library of stream translators will be maintained. New translators are added to the library as they are written. When a customized DBMS is to be assembled, selected translators are copied from the library and are INCLUDEd directly into the target system software. Our approach is to hardwire new types and operators into GENESIS, thereby gaining a measure of efficiency over more dynamic means of type/operator registration [Sto83]. This implies that the addition or removal of data types and operators will require the recompilation of some GENESIS modules. As the introduction of new types or operators should be infrequent given a careful specification of the target DBMS and applications, recompilation should be cost-effective.

To illustrate the power and extensibility of GDM and GDL, consider the following example from computer graphics. Suppose we have a Graphics database, where Graphics objects are described by line images. (For example, a box is described by twelve lines that outline its shape). Lines can be rotated, scaled, and translated by matrix multiplication, which can be handled by a stream translator Mm which transforms line objects into other line objects. Clipping lines to a viewing screen can be accomplished by a stream translator Clip, and displaying visible lines on a video output device can be done by a Display translator. Thus, the GDL expression:

Graphics.Lines.Mm.Clip.Display

could be used to display the contents of a graphical database. More conveniently, a DRAW macro could be defined which appends .Mm.Clip.Display onto an expression, thus customizing the 'display-database' operator as:

DRAW    Graphics.Lines

It is through the introduction of nontraditional database operators such as Mm, Clip, and Display that the attractiveness and extensibility of a functional approach is evident. By introducing macros such as DRAW, it is possible to customize the GDL interface so that it need not resemble a traditional DBMS interface.

Taking this one step further, it is possible to add new productions (e.g., IF_THEN) that conditionally evaluate GDL expressions. Rule systems could then be expressed. With few extra additions, a LISP-like language could be supported and general-purpose data processing algorithms (e.g., sorting) could be expressed in terms of GDL productions. Although this leap from a DML to a general-purpose programming language is possible, there are some very difficult engineering and efficiency problems that must be solved. We explain our point with an example.

Consider the database of Figure 6.1 which contains graphs where nodes are cities and edges are labeled with the distance separating cities. To compute the minimum distance between two cities would require an invocation of Dijkstra's algorithm [Dij59, Aho74]. As mentioned above, it is possible to express Dijkstra's algorithm in terms of IF_THEN productions and substituting main-memory references with DML calls. However, the resulting algorithm would be horribly inefficient. Like many algorithms, Dijkstra's algorithm was designed for main-memory databases; for it to work efficiently requires all of its data to be main-memory resident. Instead of retrieving a node at a time from secondary storage, a better solution to this query would be

to retrieve the entire graph (i.e., a stream of [city1, city2, distance] tuples) and use that as input to a stream translator D_A which stores these tuples in main-memory structures prior to invoking Dijkstra's algorithm. Thus, the query:

SELECT Edge.WHERE( Gname=g ).[Start'.Cname, End'.Cname, Dist].D_A(c1,c2)

would output the minimum distance between cities c1 and c2 on graph g.

Although it is an admirable goal to be able to express algorithms without knowledge of whether data is main-memory or secondary-storage resident, it is likely that performance will force a distinction as illustrated above. (We note that numerical analysts, for example, addressed this topic long ago. Numerical algorithms, in general, are quite different if data (e.g., matrices) are main-memory or secondary-storage resident [For77]). How the distinction is to be made is an important research issue in extensible DBMSs. [9]

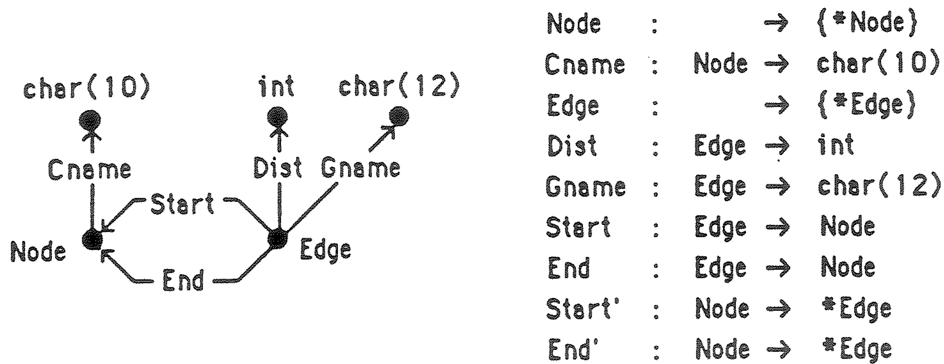



**Figure 6.1 A City-Distance Graph Database**

[9] We note that there are two approaches that are now being investigated, although no definitive results are yet available. One way is to provide hints to lower-level software to prefetch data that is likely to be referenced in the future [Car86, Row86]. Another way (perhaps related to the first) is to package and retrieve data in semantically meaningful units [Has82, Bat85, Zdo87].

## 6. Conclusions

Among the more interesting research topics in databases today are the support of ¬1NF relations, new data types and operators, and extensible database systems. Seemingly disparate, these topics are strongly related and functional data models and functional data languages bring them to a focal point. Functional models and languages are candidates for the semantic interfaces of extensible database systems. Not only can these models and languages accommodate new data types and operators, they provide the means by which operations on 1NF and ¬1NF databases can be expressed.

We have outlined a functional data model (GDM) and data language (GDL) that are targeted for the semantic interface to GENESIS, an extensible DBMS. GDL is an outgrowth of FQL and DAPLEX, and exhibits some of their best features. Specifically these are the support of nested aggregations in FQL and the absense of extension operators in DAPLEX. The combination of these features is achieved by treating functions and operations as stream rewrite rules called productions. Computations are represented by streams of tokens, where a token is either a database object or a delimiter which signals the end of one computation and the start of another. The novelty of our approach is making delimiters explicit in streams; not only does it lead to a conceptually simple model, but also one that is easy to implement.

We have explored various ways of realizing productions, and have noted that both data-driven and demand-driven implementations based on lazy evaluation are suitable for database computations. Our representation of streams requires the packaging of DBMS algorithms within standardized templates. There is virtually no run-time overhead for this packaging if the algorithms involve a considerable amount of computation. For lightweight computations such as predicate evaluations, experimental results show that predicates could be evaluated as fast as parse-tree methods. Further increases in processing efficiency will be gained if expressions are compiled.

In a wider context, the utility of functional/production languages is not limited to posing queries to a DBMS. We have briefly noted that such languages can also be used to express the composition of algorithms that define the internals of DBMSs [Bat87]. These productions (and their implementations) are identical to those that are presented in this paper, with the exception that streams of records, not objects, are mapped, and reentrancy must be supported. Thus, the GDL computation model is a cornerstone of the GENESIS implementation effort.

Future research in production data models includes: compilation techniques, query optimization, multiprocessor/multithreaded implementations of translator networks, translation of GDL expressions to operations on files and links, and solutions to the common subexpression problem for demand-driven translators and the multiple stream problem for data-driven translators.

**Acknowledgements.** We appreciate the helpful suggestions of the referees. We also thank Jim Barnett, Mike Mannino, and Brian Twichell for their comments on earlier drafts of this paper.

## 7. References

[Alb85] A. Albano, L. Cardelli, and R. Orsini, 'Galileo: A Strongly Typed, Interactrive Conceptual Language', **ACM Trans. Database Syst.** 10,2 (June 1985), 230-260.

[And86] T.L. Anderson, E.F. Ecklund, and D. Maier, 'PROTEUS: Objectifying the DBMS User Interface', **1986 Int. Workshop on Object-Oriented Database Syst.**, 133-147.

[Bat82] D.S. Batory and C.C. Gotlieb, 'A Unifying Model of Physical Databases', **ACM Trans. Database Syst.** 7,4 (Dec. 1982), 509-539.

[Bat85a] D.S. Batory and W. Kim, 'Modeling Concepts for VLSI CAD Objects', **ACM Trans. Database Syst.**, 10,3 (Sept. 1985), 322-346.

[Bat85b] D.S. Batory, 'Modeling the Storage Architectures of Commercial Database Systems', **ACM Trans. Database Syst.** 10,4 (Dec. 1985), 463-528.

[Bat86] D.S. Batory, J.R. Barnett, J.F. Garza, K.P. Smith, K. Tsukuda, B.C. Twichell, T.E. Wise, 'GENESIS: An Extensible Database Management System', to appear in **IEEE Trans. Software Engr.**. Also, Dept. Computer Sciences, University of Texas at Austin, TR-86-07, March 1986.

[Bat87] D.S. Batory, 'Building Blocks of Database Management Systems', submitted for journal publication. Also, Dept. Computer Sciences, University of Texas at Austin, TR-87-23, June 1987.

[Bat88] D.S. Batory, M. Mannino, et al. 'The GENESIS Data Model and Data Language', to appear.

[Bit83] D. Bitton, D.J. DeWitt, and C. Turbyfill, 'Benchmarking Database Systems: A Systematic Approach', **VLDB 1983**, 8-19.

[Bun79] P. Buneman and R.E. Frankel, 'FQL - A Functional Query Language', **SIGMOD 1979**, 52-57.

[Bun82] P. Buneman, R.E. Frankel, and R. Nikhil, 'An Implementation Technique for Database Query Languages', **ACM Trans. Database Syst.** 7,2 (June 1982), 164-186.

[Car85] M.J. Carey and D.J. DeWitt, 'Extensible Database Systems', **Islamorda Workshop on Large Scale Knowledge Base and Reasoning Systems**, Feb. 1985, 335-352.

[Car86] M.J. Carey, D.J. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J.E. Richardson, and E.J. Shekita, 'The Architecture of the EXODUS Extensible DBMS', **1986 Int. Workshop on Object-Oriented Database Syst.**, 52-65.

[Cha81] D.D. Chamberlin, et al., 'Support for Repetitive Transactions and Ad Hoc Queries in System R', **ACM Trans. Database Syst.** 6 #1 (March 1981), 70-94.

[Dad86] P. Dadam, et al., 'A DBMS Prototype to Support Extended $NF^2$ Relations: An Integrated View on Flat Tables and Hierarchies', **SIGMOD 1986**, 356-367.

[Day85a] U. Dayal and J. Smith, 'PROBE: A Knowledge-Oriented Database Management System', **Islamorda Workshop on Large Scale Knowledge Base and Reasoning Systems**, Feb. 1985, 103-138.

[Day85b] U. Dayal, et al., 'PROBE -- A Research Project in Knowledge-Oriented Database Systems: Preliminary Analysis', Computer Corporation of America, CCA-85-03, 1985.

[Dij59] E.W. Dijkstra, 'A Note on Two Problems in Connexion with Graphs', **Numerische Mathematik** 1, 269-271.

[Fis87] D.H. Fishman, et al., 'IRIS: An Object-Oriented Database Management System', **ACM Trans. Office Info. Syst.** 5,1 (1987).

[For77] G. Forsythe, M.A. Malcom, and C.B. Moler, **Computer Methods for Mathematical Computations,** Prentice-Hall, 1977.

[Fre86a] J.C. Freytag and N. Goodman, 'Rule-Based Translation of Relational Queries into Iterative Programs', **ACM SIGMOD 1986,** 206-214.

[Fre86b] J.C. Freytag and N. Goodman, 'Translating Aggregate Queries into Iterative Programs', **VLDB 1986,** 138-146.

[Gra84] P. Gray, **Logic, Algebra, and Databases,** Halsted Press, 1984.

[Has82] R.L. Haskin and R.A. Lorie, 'On Extending the Functions of a Relational Database System', **ACM SIGMOD 1982,** 207-212.

[Hen80] P. Henderson, **Functional Programming: Application and Implementation,** Prentice-Hall, 1980.

[Hei85] S. Heiler and A. Rosenthal, 'G-WHIZ: A Visual Interface for the Functional Model with Recursion', **VLDB 1985,** 209-218.

[Hud86] S.E. Hudson and R. King, 'CACTIS: A Database System for Specifying Functionally-Defined Data', **1986 Int. Workshop on Object-Oriented Database Syst.,** 26-37.

[Ker75] L. Kerschberg and J.E.S. Pacheco, 'A Functional Data Base Model', unpublished tech. rep., 1975.

[Leu86] T.Y. Leung, 'Implementation Techniques for an Extensible Database Data Language', M.Sc. Thesis, Dept. Computer Sciences, University of Texas at Austin, 1986.

[Lyn86] P. Lyngbaek and W. Kent, 'A Data Modeling Methodology for the Design and Implementation of Information Systems', **1986 Int. Workshop on Object-Oriented Database Syst.,** 6-17.

[Man86] F. Manola and U. Dayal, 'PDM: An Object-Oriented Data Model', **1986 Int. Workshop on Object-Oriented Database Syst.,** 18-25.

[Ozs85] Z.M. Ozsoyoglu and L-Y. Yuan , 'A Normal Form for Nested Relations', **ACM PODS 1985,** 251-260.

[Pis86] P. Pistor and F. Anderson, 'Designing a Generalized $NF^2$ Model with an SQL-Type Language Interface', **VLDB 1986,** 278-288.

[Rit74] D.M. Ritchie and F. Thompson, 'The UNIX Time-Sharing System', **Comm. ACM,** 17,7 (July 1974), 365-375.

[Row86] L.R. Rowe, 'A Shared Object Hierarchy', **1986 Int. Workshop on Object-Oriented Database Syst.,** 160-170.

[Rot87] M.A. Roth, H.F. Korth, D.S. Batory, 'SQL/NF: A Query Language for ¬1NF Relational Databases', **Information Systems,** 12, 1 (1987), 99-114.

[Rot86] M.A. Roth, 'Theory of Non-First Normal Form Relational Databases', Ph.D. Diss., Dept. Computer Sciences, University of Texas at Austin, 1986.

[Sch85] H.J. Schek, 'Towards a Basic Relational $NF^2$ Algebra Processor', **International Conference on Foundations of Data Organization,** Kyoto, Japan, May 1985, 173-182.

[Sha86] L.D. Shapiro, 'Join Processing in Database Systems with Large Main Memories', **ACM Trans. Database Syst.,** (Sept. 1986), 265-293.

[Shi81] D. Shipman, 'The Functional Data Model and the Data Language DAPLEX', **ACM Trans. Database Syst.,** 6 #1 (March 1981), 140-173.

[Sib78] E.H. Sibley and L. Kershberg, 'Data Architecture and Data Model Considerations', **AFIPS NCC 1977,** 85-96.

[Smi85] K.P. Smith, 'Design and Implementation of the GENESIS Record Manager', M.Sc. Thesis, Dept. Computer Sciences, University of Texas at Austin, 1985.

[Sto83] M. Stonebraker, B. Rubenstein, and A. Guttman, 'Application of Abstract Data Types and Abstract Indices to CAD Databases,' **ACM Database Week: Engineering Design Applications 1983,** 107-113.

[Sto85] M. Stonebraker, 'Inclusion of New Types in Relational Database Systems', **Conf. Data Engineering 1986,** 262-269.

[Sto85] M. Stonebraker, 'Inclusion of New Types in Relational Data Base Systems', Report UCB/ERL M85/67, Electronics Research Laboratory, University of California, Berkeley, 1985.

[Sto86] M. Stonebraker and L. Rowe, 'The Design of POSTGRES', **ACM SIGMOD 1986,** 340-355.

[Tur79] D.A. Turner, 'A New Implementation Technique for Applicative Languages', **Software Practice and Experience** 9 (1979), 31-49.

[Wad71] C.P. Wadsworth, 'Semantics and Pragmatics of the λ-Calculus', Chapter 4, Oxford University, Ph.D. Thesis, 1971.

[Yed86] L. Yedwab, private correspondence, 1986.

[Zan83] C. Zaniolo, 'The Database Language GEM', **ACM SIGMOD 1983,** 207-218.

[Zdo87] S. Zdonik and M. Hornick, 'Issues in the Implementation of an Object-Oriented Database', to appear in **Trans. Office Information Syst.,** 1987.