A SURROGATE CONCEPT FOR ENGINEERING DATABASES

Andreas Meier and Raymond A. Lorie

IBM Research Laboratory
San Jose, CA 95193

ABSTRACT: Relational database systems are attracting interest from users outside the commercial areas for which such systems were initially designed. This paper describes a surrogate model for engineering data (CAD, VLSI, geographical data, etc.) which allows the user to define structural relationships among semantically related data. A binding mechanism between system-controlled surrogates and user-defined keys avoids the introduction of two independent identifier concepts. Finally, an implicit join with two built-in functions can be used to query and manipulate structured data in an efficient and flexible way.

1. USER KEYS VERSUS INTERNAL IDENTIFIERS

The need for some unique and permanent identifiers of database entities is clear. By definition, a primary key in the relational model uniquely identifies the contents of a tuple as well as the tuple in a relation. When employing such user-defined primary keys in an engineering environment, problems arise in three areas:

- Structure: With most relational database systems mapping of highly interrelated data into tuples in one or more relations has to be done entirely by the user. For instance, when a tuple of a hierarchical structure is deleted, it is the application program which has to delete all its children.

- Semantics: The user has to choose a special attribute (or attribute combination) as the primary key. This choice may either be artificial or may bear some semantic meaning (e.g., part serial number). The actual values of these keys are determined by the user and therefore may change. For instance, if two inventory databases are merged, some or all of the serial numbers may have to be changed.

- Performance: To access clustered tuples efficiently via their primary keys, a clustered index on the key fields may be defined. However, most relational systems do not maintain structural integrity constraints or cannot fetch a complex object with a single call to the database management system. As a result, engineering data has traditionally been organized as sets of files with the inherent disadvantages of high redundancy and poor data independence in order to gain better performance for the operations.

In the following, we define a surrogate concept as the basis for an engineering database system which alleviates the above drawbacks.

2. A SURROGATE MODEL

2.1. A Binding Mechanism with Two Built-in Functions

System-generated internal identifiers or surrogates /HAL 76/ are introduced for columns having the new type IDENTIFIER. Each internal IDENTIFIER value is system-wide unique (e.g., concatenation of processor number, database identification, and clock time) in order to allow for the merging of databases from different sites. Furthermore, the values of an IDENTIFIER attribute cannot be changed. The user has no control or access to the IDENTIFIER values although they may or may not be made available to him (e.g., it seems appropriate to return identifier values to the application program).

When a user creates a tuple containing an IDENTIFIER column, the system automatically generates the IDENTIFIER value and returns it to the user in the variable associated with the IDENTIFIER column. This identifier value may then be used to establish links between tuples of different relations (see 2.2 and 2.4). Obviously, it is never null.

An internal IDENTIFIER acts as an invariant value for each tuple, and no special attribute needs to

be chosen as the primary key. On the other hand, the user often likes to use a primary key that has some semantic meaning for him (e.g., social security number). The question arises of how to deal properly with system-generated and user-defined identifiers. The professional user (designer, database administrator, application programmer) might wish to know some system-generated identifiers, whereas the casual user should not have to worry about them.

To avoid introduction of two independent identifier concepts, we propose a binding mechanism by introducing an index. This index is restricted to a single column, i.e. unique key, and implies its corresponding IDENTIFIER column. With the new index, the user can retrieve data by user key rather than internal identifier. Furthermore, two built-in functions are defined to map the system-generated identifier onto the user-defined key and vice-versa:

KEY (identifier)	retrieves the user key corresponding to an internal identifier if one exists.
ID (user key)	retrieves the internal identifier of a specific user key value.

A one-to-one mapping between internal identifier and user key is guaranteed if the attribute of the indexed column was specified with the NOT NULL option. In this case, both functions KEY and ID yield a unique value which is never null (a non-existing operand produces an error message).

2.2. Hierarchical Relationships between Relations

In order to define and implement complex objects, the system-generated internal identifiers (type IDENTIFIER) are used. Besides technical reasons (clustering, non-composite keys, performance aspects, etc.) internal identifiers have a semantic meaning and may reference relations: component relations are distinguished by a COMPONENT_OF column that contains identifiers pointing to tuples in a corresponding parent relation.

We consider the definition of a simplified electronic module as an example. A MODULE is composed of several PARTs where each PART may contain some FUNCTIONs:

```
CREATE TABLE MODULE
       (MID      IDENTIFIER,
        NUMBER   INTEGER,
        PRIZE    DECIMAL(2),
        ...
CREATE TABLE PART
       (PID      IDENTIFIER,
        MID      COMPONENT_OF(MODULE),
        QUALITY  INTEGER,
        ...
CREATE TABLE FUNCTION
       (FID      IDENTIFIER,
        PID      COMPONENT_OF(PART),
        CODE     CHARACTER(10),
```

The new column types IDENTIFIER and COMPONENT_OF are extensions to System R /BLA 81/ and allow for the declaration of complex objects as molecular entities.

2.3. Implicit Hierarchical Joins

To retrieve structured data, a user would have to define several join predicates along particular branches involving IDENTIFIER and COMPONENT_OF columns. This drawback may be eliminated by bringing the whole implicit structure of a complex object up to the user interface as in the following sample query:

Q1: List all FUNCTION codes of MODULE 100.

```
SELECT DISTINCT FUNCTION.CODE
FROM   MODULE-FUNCTION
WHERE  MODULE.NUMBER=100;
```

The linear implicit join from MODULE to FUNCTION is an equi-join between the columns of type IDENTIFIER in the parent relation MODULE (resp. PART) and the columns of type COMPONENT_OF in the direct child relation PART (resp. FUNCTION). Whenever an implicit join touches a relation which is not explicitly declared in the FROM clause (e.g., PART in MODULE-FUNCTION), it is involved in the multiple join by definition and the user may formulate restrictions on it in the WHERE clause.

The notation for implicit join is quite general and allows specification of every possible choice of hierarchical or linear subschemas. Precise definitions and several illustrative examples are given in /MEI 83/.

With the introduction of implicit join, queries on complex objects become simpler and more intuitive. Instead of having one or more join predicates along particular branches of a complex object specified explicitly, the system recognizes the operator and automatically joins component tuples with their corresponding ancestor tuples. Moreover, the system knows both the structure and the internal representation of these special joins (1) from the system catalogs and can therefore optimize the operations accordingly.

2.4. Non-hierarchical References

Internal identifiers are very convenient for expressing hierarchical structure. However, surrogates may also be used to build non-hierarchical references: a tuple can refer to another tuple using a REFERENCE column. To refer to a root or a

(1) Our implementation of complex objects illustrates an interesting example of the class of γ-acyclic hypergraphs; see /FAG 82/, p.43.

component tuple of the same or different complex object, the IDENTIFIER value of the referenced tuple is put into the REFERENCE column of the referring tuple.

As soon as several relations refer to each other (e.g., by the COMPONENT_OF or REFERENCE values), we are again concerned about joining along these links. To avoid writing down all joins for this particular case, the built-in function KEY can be used. The argument of the KEY function is the column name of type COMPONENT_OF or REFERENCE which refers to an internal identifier column.

It may be noted that, even if the key value is null, the 'join' (actually, we only evaluate the built-in function KEY) can still be performed since only identifiers are needed. Also some of the selected REFERENCE values may be null depending on the semantics chosen.

The proposed KEY concept is minimal and it helps to avoid writing additional joins to retrieve a user key. It does not help when more than the user key is desired from the referenced relation. To retrieve other columns via the binding mechanism would destroy the First Normal Form property.

2.5. Data Manipulation Considerations

First, we turn our attention towards insertion of a tuple into a complex object. If the tuple to be inserted contains a COMPONENT_OF or REFERENCE type value, one needs to know the identifier of the component or referenced tuple. Very often this value will have been created in the same program and is therefore known. If not, a query is needed to retrieve this identifier from the database. To simplify the task of the programmer or the casual user, we have introduced a built-in function ID: It returns as value the internal identifier corresponding to the supplied key.

The deletion of tuples which belong to a complex object has special semantics. For instance, when a tuple is deleted, all tuples which are the children of the tuple must be deleted together with all their descendants (cascaded deletion). This is the only way tuples from different relations can be deleted by a single statement. Allowing another form of a single delete statement involving more than one relation could destroy the structure of the complex object, for example, by introducing orphans. However, as soon as we restrict a deletion to a single relation, an implicit join may be appropriate. Deleting through an implicit join is possible by specifying a subschema (given by a join operator) and naming the relation from which deletions should occur.

Implicit joins can be used advantageously in update statements. Here, because of the clean semantics of the complex object and implicit join, there is no ambiguity.

3. IMPLEMENTATION ASPECTS

System R has been extended to generate and support uniform IDENTIFIER values which are used for both external links (type COMPONENT_OF and REFERENCE) and internal links (with compression for first child, previous and next siblings). Therefore, one may quickly fetch all tuples under a parent without scanning the rest of the database.

System catalogs have been modified to capture the structure of a complex object. This structure information allows the system to analyze the implicit join operator and to find all necessary links in order to materialize the query. Also, a special system table is maintained for each hierarchy of relations in order to implement a fast intra-object access path. This path is used to enforce parent-child integrity constraints and provides better performance for clustered access and manipulation of tuples which belong to the same complex object.

The query optimizer (the subsystem which generates an optimal plan for evaluating a given query) has been modified to take advantage of that special access path. We have also evaluated the performance of the enhanced system.

4. CONCLUSION

In this paper, we described a surrogate model for relational databases to better support engineering and design applications. A binding mechanism provides a powerful tool to both the professional programmer and the casual user. An implicit join operator and two built-in functions simplify the query language and allow the user to retrieve and to manipulate structured data or part of it as a whole rather than assembling different relations and thinking about all known interrelationships. By defining the structure of objects to the system, structural integrity, ease of application programming, and improvements in performance can be achieved.

References.

/BLA 81/ Blasgen M.W., et al.: System R: An Architectural Overview. IBM Systems Journal 20, No. 1, 1981, pp. 41-62.

/FAG 82/ Fagin R.: Types of Acyclicity for Hypergraphs and Relational Database Schemes. IBM Research Report: RJ3330(39949), November 1981, San Jose, California (to appear: J. ACM).

/HAL 76/ Hall P., Owlett J., Todd S.: Relations and Entities. Nijssen G.M.(ed.): Modelling in Data Base Management Systems. North-Holland, Amsterdam 1976, pp. 201-220.

/MEI 83/ Meier A., Lorie R.A.: Implicit Hierarchical Joins for Complex Objects. IBM Research Report RJ 3775, January 1983.