An Extension of the Database Language SQL to Capture More Relational Concepts

Gottfried Vossen* and Jim Yacabucci
Department of Computer Science and Engineering C-014A
University of California, San Diego
La Jolla, CA 92093

August 1987

Abstract

An extension of the database language SQL is described which introduces several new concepts into the language that are standard in the relational model, but surprisingly not present in available SQL-based systems such as SQL/DS and DB2.

1 Introduction

The language SQL for relational database management was originally developed by IBM for use in their System/R prototype and later in their DBMS products such as SQL/Data System [5] and DB2 [3]. Subsequently, the language has received great attention also by other vendors since it was chosen by ANSI as the basis for a standard for relational languages [1]; for example, SQL is now available also under Ingres, Oracle, and Unify, and more implementations even on top of network systems are supposed to come. However, when comparing SQL to standard features of the relational model for databases, several discrepancies between theory and practice can be discovered, e.g., several concepts present in the model are not supported by this language and its (IBM) implementation.

In this paper, we describe an extension of SQL which takes care of several such gaps. In section 2, we collect some of the criticisms that have been raised against the SQL/Data System and its language SQL itself, and we point out those which we consider of major importance. In section 3, we describe the syntax and the semantics of SQL extensions which add considerable modeling power to the language; these extensions primarily have to do with the data definition capabilities and, consequently, with the organization of the SQL data dictionary. In section 4, we briefly report on a prototype implementation of these extensions, that was carried out within a student's project at UCSD; in particular, several (new or extended) commands have to be executed in a modified way. Section 5 summarizes our description.

*The work of this author was supported by the NATO Science Committee under a Postdoctoral Fellowship, Grant No. DAAD 300/402/508/7.

We assume the reader to be familiar with the SQL language on the level of [3] or even [5], so that several commands will be mentioned in the sequel without further explanation.

2 Some Critics of IBM's SQL/Data System and its Language

Since the arrival of the SQL language in the market place, several authors have raised criticisms against it. The most complete collection certainly is the summary presented by Date in [4], which has also had an impact on the SQL standard proposed by ANSI [1]. In this section, we review some objections regarding aspects of the relational model not supported by the language and add several other ones, which were also mentioned in [7].

Following textbooks for classroom use that cover the relational model [6, 7, 9], a relational database is conceptually described in a *database schema.* Intuitively, this term is used to completely describe a well-defined "section of the real world", about which data is to be stored, on an abstract, conceptual level, and in particular to distinguish one application from another. Formally, a database schema consists of a set of relation schemas and a (possibly empty) set of ("interrelational") constraints that describe how relation schemas are dependent on each other. A relation schema in turn consists of a set of attributes and a set of ("intrarelational") constraints which state how attributes are semantically related.

Our first observation is that the notion of a database schema is not explicitely supported by the SQL language; the only notion which could be used as a substitute is that of a "dbspace", by which distinct collection of tables can be separated physically. Instead of having this notion available, the SQL/DS system just maintains one large collection of relation schemas, in which different applications can be intermixed and are conceptually distinguishable only by creator identifications.

Following [2, 4], at least two classes of constraints should be present to restrict the possible state-space of a given database schema in a reasonable way. These are generally refered to as *entity integrity* and *referential integrity.* Both types of constraints can conveniently be expressed in the relational model using *key dependencies* for the former and *inclusion dependencies* for the latter class. While inclusion dependencies, which generalize foreign key constraints and describe semantically meaningful links between different relation schemas, are not supported at all by SQL, key dependencies can be expressed implicitely: If for a relation schema R with attribute set X a subset K of X is supposed to be the (primary) key, a unique index on K has to be created immediately upon creation of the R table. It is obvious that this poor support of integrity constraints is far from being realistic with respect to real-world database applications, in which keys arise very naturally, and in which already the establishment of a relationship between two entity-sets or their types gives rise to inclusion dependencies in a straightforward manner.

From a "purely relational" point of view, further critics can be raised against the semantics of queries expressable through the SELECT statement: A simple projection does not remove duplicate tuples from its result unless explicitely forced to do so; any natural join of several relations always requires a careful, explicit statement of all relevant "(equi-) join-conditions" within a corresponding theta-join.

3 Syntax and Semantics of the Proposed Extensions

In this section, we describe the extensions we propose to the SQL language and to the functionality of an underlying DBMS. We consider data definition commands first and then discuss necessary augmentations to the data dictionary. We follow common conventions when describing the syntax of commands, e.g., keywords are written in capital letters, while user-definable names are written in small letters. Any part of a statement which is included in brackets is optional, while braces delimit alternatives, one of which has to be chosen. Alternatives itself are separated by vertical bars.

The explicit support for the notion of a database schema is achieved by several means: Basically, whenever a user works with the system, (s)he works with *one* particular database. Before this database can be manipulated, it has to be *opened* by

```
USE dbschema-name
```

Once this command has been issued, all commands that follow will automatically refer to dbschema-name. By issuing another USE-command, the user can "move" from one database to another. The only exception to this rule is the use of the dictionary, which can always be accessed without a previous USE.

Before a database schema can be used, it has to be created. To this end, the command

```
CREATE DBSCHEMA dbschema-name
```

has to be used; a new database schema with the specified name will be opened (and is thus in USE immediately) if this name does not yet exists for a schema and if the issuing user has the privilege to create database schemas.

Conversely, an existing database schema can be dropped using

```
DROP DBSCHEMA dbschema-name
```

This command causes the system to erase *all* information which is related to this schema, e.g., all corresponding relation schemas *and* their contents together with the database schema itself.

With respect to the creation of new database schemas, we assume that among all users of a particular database, at least one has "DBA authority", which is recorded in the data dictionary as described later. This authority also implies that he has the right to add relation schemas to the database schema in question, and that he can grant this right to or revoke it from other users (see also below). We consider it reasonable to restrict the possibilities to declare constraints to one key per relation schema and to acyclic inclusion dependencies between them. This makes it possible to leave users the same flexibility with respect to the declaration of new relation schemas as in SQL/DS, e.g., an existing database schema can be augmented at any time.

The command to create a new relation schema differs significantly from what is available in SQL up to now, and is described next. Its general format is

```
CREATE TABLE table-name
        (column-name-1 type-1 [{NOT NULL | KEY}]
                [REF [creator.] table-name-1.column-name-a]
        [, column-name-2 type-2 [{NOT NULL | KEY}]
                [REF [creator.] table-name-2.column-name-b] ] ... )
```

The execution of this command will result in an error message if no database schema has been opened earlier (by USE or CREATE DBSCHEMA). Otherwise, a new relation schema named table-name will be created as usual, but with the following additions: Any attribute can be included in the key for this schema simply be adding the option KEY to its declaration. Since key-attributes should not get null values, this option implies the NOT NULL option (thus, attributes may still be declared as NOT NULL without participating in the key). In addition, inclusion references to other relation schemas, which have to be declared already, can be specified using the optional REF-clause. If, say, attribute A of relation schema R is declared as "REF $S.B$", this corresponds to the inclusion dependency $R[A] \subseteq S[B]$. Clearly, references to several other relation schemas can be declared at the same time.

The following example illustrates the use of the CREATE TABLE-command: Suppose that a library wants to store three tables BOOKS (for all the books in the library), READERS (for its readers), and OUT (for relating readers to the books they have checked out). Let BOOKS have the attributes *BID, AUTH*, and *TITLE*, READERS have *RID, NAME*, and *ADDR*, and OUT have *BID, RID*, and *DUEDATE*. We further assume that a BID-value [RID-value] uniquely identifies a book [reader], respectivily, that books must exist in the library before they can be checked out, and that readers must be recorded before they can check out a book. Thus, the inclusion dependencies

$$\mathrm{OUT[BID]} \subseteq \mathrm{BOOKS[BID]}$$
$$\mathrm{OUT[RID]} \subseteq \mathrm{READERS[RID]}$$

should hold in any valid database state. Since the BOOKS-READERS-relationship recorded in OUT is of type many-one, it is sufficient to use BID as the only key-attribute for relation schema OUT as well. Thus, we arrive at the following declaration (domains are omitted):

```
CREATE DBSCHEMA LIBRARY
CREATE TABLE BOOKS
        (BID ...KEY, AUTH ...NOT NULL, TITLE ...NOT NULL)
CREATE TABLE READERS
        (RID ...KEY, NAME ...NOT NULL, ADDR ...)
```

```
CREATE TABLE OUT
    (BID ...KEY REF BOOKS.BID,
     RID ...NOT NULL REF READERS.RID,
     DUEDATE ...NOT NULL)
```

When a relation schema is dropped from a database schema, its references to other relation schemas are lost. However, if it is referenced by other schemas, e.g., it appears oh the right-hand side of an inclusion dependency, an attempt to drop it will result in an error-message.

The creator of a database schema can also give permission to add or drop relation schemas to other users, as mentioned earlier. To this end, the GRANT and REVOKE commands of SQL have been extended correspondingly. The syntax of the new GRANT command is as follows:

```
GRANT { {list-of-privileges | ALL} ON [creator.] table-name
        | MODIFY ON dbschema-name }
        TO {list-of-users | PUBLIC}
```

In our current implementation, the list-of-privileges is restricted to "READ" (for querying a table) and "WRITE" (for insert, delete, and modify), where the latter implies the former; also, views are not considered at the moment. The MODIFY-privilege on a database schema implies that the grantee may create and drop new relation schemas with respect to this schema.

Conversely, privileges can be withdrawn using the REVOKE command, which is identical in syntax to the GRANT Command, but with the keyword GRANT [TO] replaced by REVOKE [FROM], respectively.

We next describe the implications which these modifications and extensions have on the SQL dictionary. Basically, the data dictionary is organized as a collection of tables under SQL/DS, e.g., in the same way ordinary relations are organized, which implies that a user can conveniently use the same language for querying a database and for querying the dictionary. A complete description of the dictionary can, for instance, be found in [5]. However, a number of changes have become necessary in order to reflect the introduction of the notion of a database schema and the corresponding reorganization of the possibilities to grant access rights properly.

Up to now, for each relation managed by SQL/DS there has been one entry in the dictionary table SYSCATALOG[1]; it appears appropriate to use this table now for recording database schemas. Correspondingly, this table now has the attributes *DBNAME* (the name of a database schema), *DCREATOR* (the user-id of the creator of that schema), and *NTABLES* (the number of relation schemas currently defined for that database schema). More attributes such as *REMARKS* or *DBSPACENAME*, which are present

[1]Here and in the following, we omit the SQL/DS-internal prefixes "SYSTEM." or "SQLDBA." for dictionary table-names.

in the original SYSCATALOG, can easily be added if this is desired.

The former dictionary table SYSCATALOG has been replaced by a new table called SYSTABLES, which describes the relation schemas associated with each database schema in more detail. To this end, it contains the attributes *DBNAME* (the name of a database schema, which also appears in SYSCATALOG), *TNAME* (the name of a relation schema appearing in the corresponding database schema), *TCREATOR* (the user-id of the creator of this relation schema), and *NCOLS* (the number of attributes of this relation schema). Again, other attributes such as *TABLETYPE* (to distinguish base relations from views), *REMARKS* or *DBSPACENAME* can be added for convenience.

The dictionary table SYSCOLUMNS, which describes every attribute of each relation schema in detail, has undergone the changes necessary to reflect key- and inclusion-constraints. Once more, this table now also contains the attribute *DBNAME*; the attributes which have been left unchanged are *TNAME, TCREATOR, CNAME* (the name of an attribute), *COLNO* (the sequence number of this attribute as it appeared in the CREATE TABLE command), *COLTYPE*, and *LENGTH*. Other new attributes are *NULLS/KEY*, which contains the value "K" if the corresponding attribute participates in a key for the relation in question, "N" if only nulls are not allowed, and " " otherwise, and *REFERENCE*, which records the reference to an attribute of another table (by containing an entry of the form "table-name.column-name") if the current attribute participates in an inclusion dependency. Finally, the attribute *REMARKS* can again be added if needed.

We discuss the dictionary tables which keep track of access rights next. A new dictionary table SYSDBSAUTH has been introduced to record which user is allowed to modify database schemas (e.g., add or drop relation schemas). The entries of this table, which has the attributes *DBNAME, USERNAME*, and *TABLEAUTH*, may be provided by the DBA or the system administrator; in addition, they are generated by the system if a database schema-creator grants a MODIFY-privilege on his schema to another user (as described earlier). Note that the attribute *TABLEAUTH*, which contains "Y" if the user in question is allowed to modify the corresponding database schema and "N" otherwise, originally appeared in the SQL dictionary table SYSUSERAUTH (see below); it seems appropriate to record this information in a separate table now.

The catalog relation SYSTABAUTH, which summarizes the manipulation privileges (query and update) that individual users have on tables in any database, has been diminished for our purposes; its new attributes are *DBNAME, TCREATOR, TNAME, GRANTEE* and *PRIVILEGE*, the latter of which contains "R" if the grantee is only allowed to query the corresponding relation, and "W" otherwise. Thus, we no longer make a distinction between the rights to insert, delete, or modify, and we currently do not privide the possibility to grant access rights to third parties.

The table SYSUSERAUTH is another record of access rights, this time regarding definition privileges. The attributes of this table, which is also used by SQL/DS, have been modified as follows: The column *NAME* contains the user-ids of people who are allowed to work with the DBMS at all. For each such user, *DBSCHEMAAUTH* contains "Y" if this user has the right to create database schemas, and "N" otherwise. The attribute

TABLEAUTH has been dropped from this table (see explanation above). Note that a user may be allowed to create tables without being allowed to establish new databases; the corresponding access rights are recorded in SYSDBSAUTH and SYSUSERAUTH, respectively. Column *DBAAUTH* of table SYSUSERAUTH contains "Y" is this user is the DBA, and "N" otherwise. Finally, the password of each user is recorded in column *PASSWORD*.

The important feature of SYSUSERAUTH is that only the DBA is allowed to write *and* even to read it, since the password information has to be secret. In order to provide users with information on who is allowed to do what, a second table SYSUSERLIST is contained in the dictionary (as under SQL/DS), which is a projection of SYSUSERAUTH onto all attributes except *PASSWORD*.

It should be mentioned that other dictionary tables (such as the original SYSVIEWS, SYSDROP or SYSSYNONYMS) have not been added to the catalog of our system so far; an extension, however, is straightforward. On the other hand, the table SYSINDEXES is no longer needed in our implementation as far as its use for supporting indexes that "simulate" keys is concerned, since keys are now explicitely represented in a relation schema.

4 Remarks on the Implementation

In this section, we briefly describe some features of the implementation of the SQL extensions summarized above; further details can be found in [8]. Our experimental system was implemented in C++ and is currently running on a VAX 11/750 under 4.3 BSD UNIX.

Generally speaking, database schemas are mapped to the UNIX file system so that there is a one-to-one correspondence between a schema name and a directory name. All schemes are realized as subdirectories or the root (the directory containing the database system).

When a database schema is created, a system call to "mkdir" is executed. Once a schema has been opened, all references to tables within this schema are internally mapped to the file system by attaching the name of the opened schema as a prefix to the table name. Thus, all relations corresponding to relation schemas in the same database schema are contained in a unique directory. Similarly, system calls to "rmdir", and "rm filename" are utilized for the DROP SCHEMA and the DROP TABLE operations.

While these features of the system are quite straightforward when the UNIX operating system is utilized, the issues mentioned next regard the new capabilities of the system with respect to the maintainance of integrity constraints and are therefore unique to our implementation: As was mentioned in the previous section, any key defined via a CREATE TABLE command is recorded in the corresponding dictionary entry for the table in question along with all other pertinent information. Each attribute in the attribute list of the table internally contains a binary field which is set if that attribute is contained in the key.

During an INSERT operation, all tuples which are to be inserted must first be checked to insure that the key property will not be violated. This is done by performing a SELECT operation which searches the operand relation for tuples whose key values agree with those of a tuple appearing in the current INSERT command. The SELECT statement is generated by scanning the attribute list of the relation, and including a condition of the form "$A = a$", where a is a value for A from a new tuple, in the WHERE clause of the statement for each attribute A whose key flag is set; if the key consists of several attributes, the corresponding conditions are connected by logical AND. If the result of this SELECT operation is non-empty, the insertion will obviously violate the key and is thus rejected. The same check is performed for each tuple to be inserted.

Similarly, if a user tries to change the value of a key attribute via a MODIFY operation, the same type of check must be executed to assure that key values are not duplicated.

As with key information, reference information about inclusion dependencies to an attribute of another relation is stored in the SYSCOLUMNS table of the dictionary. In addition, the name of the referencing relation it is internally kept in a dependent list of the referenced relation, e.g., in a list of relations which contain a reference to it. This information is needed for performing update operations on the referenced relation.

In all tests for referential integrity of the form, say, $R[A] \subseteq S[B]$, it has to be verified whether or not $\pi_A(R) \subseteq \pi_B(S)$ holds; clearly, no test is required for insertions into S or deletions from R. For an INSERT operation on R, the basic strategy is to traverse the attribute list of R and to generate a SELECT command for each reference to a distinct relation whose FROM clause mentions the referenced relation, say, S and whose WHERE clause consists of a conjunction of conditions, one for each attribute of R which has a reference to S. If these SELECT operations yield non-empty results, the insertion will be executed as intended.

When a user performs a delete or modify operation on relation R which is referenced by one or more other relations, similar tests must be performed in the reverse direction. In this case, the dependent list of R will be traversed. For each relation mentioned in this list, the SELECT operation described above must be executed to insure that after the deletion or modification, correct references will still exist. Finally, it should be mentioned that a MODIFY operation on any relation mentioned in an inclusion dependency yields a test on the other.

In the current implementation, also the usual relational projection and (natural) join operations are available. Thus, a projection will by default remove duplicate tuples appearing in the result, and a natural join of, for example, two relations named R and S can now conveniently be expressed as

SELECT * FROM R, S

Finally, it should be mentioned that people who are interested in our implementation and have noncommercial purposes can obtain a demonstration copy of the system from the authors.

5 Conclusions

In this paper we have described the motivation behind an extension of the database language SQL to capture more concepts usually associated with the relational model. We have proposed several language constructs to express new SQL modeling capabilities such as entity and referential integrity, we have described the appropriate changes to the data dictionary, and we have outlined the main concepts of an experimental implementation of these ideas.

It turns that — as far as database definition is concerned — users of our extended language find it much more convenient to use than the original, although the full range of SQL capabilities is not yet available in our system. We hope to gain more experience in the future, and that proposals such as the current one can contribute to future enhancements of standardization efforts for relational languages.

References

[1] American National Standards Institute. *Draft Proposed American Standard Database Language SQL.* Draft ISO/TC 97/SC 21/WG 5-15/N 90, New York 1985.

[2] E.F. Codd. *An Evaluation Scheme for Database Management Systems that are claimed to be Relational.* Proc. 2nd IEEE International Conference on Data Engineering 1986, 720-729.

[3] C.J. Date. *A Guide to DB2.* Addison-Wesley Publ. Co. 1984.

[4] C.J. Date. *A Critique of the SQL Database Language.* ACM SIGMOD Record 14 (3) 1984, 8-54.

[5] IBM Corporation. *SQL/Data System Terminal User's Reference.* Publication No. SH24-5017-2, Release 2, Endicott, NY, 1983

[6] D. Maier. *The Theory of Relational Databases.* Computer Science Press 1983.

[7] G. Vossen. *Data Models, Database Languages, and Database Management Systems.* Addison-Wesley Publ. Co. 1987 (in German).

[8] J. Yacabucci. *OurBase 5000 — A Relational Database Management System.* Internal Report, CSE Department, UCSD, August 1987.

[9] C.C. Yang. *Relational Databases.* Prentice-Hall 1986.