

Edgar H. Sibley  
Panel Editor

*Developers of a Wall Street financial application were able to exploit a relational DBMS to advantage for some data management tasks (the good). For others, the relational system was not helpful (the bad), or could be pressed into service only by means of major or minor contortions (the ugly). The authors identify database constructs that would have simplified developing the application (the ideal).*

# Using A Relational System On Wall Street: The Good, The Bad, The Ugly, And The Ideal

Steve Rozen and Dennis Shasha

Conventional wisdom has it that complex decision support is an extremely promising application of the relational model. Partly for this reason, one of the authors (Rozen) was hired to design the database for a decision support system used by Wall Street investment managers. The application, based on the Oracle [30] relational database management system, was to replace a previous one written on top of a file system. Problems with the previous application system included the inability to enhance its functionality to suit new analytical applications, the inability to exchange data with other company information, and an antiquated teletype user interface.

The developers were able to reimplement the functionality of the old system in seven months, and their success depended largely on the virtues of the relational model. However, problems still remained. To illustrate both the promise and the problems, in this article, we focus on several critical design requirements and the ability of the relational system to satisfy them. In all important respects, our analysis is independent of Oracle and reflects issues that the use of any relational database management system would raise.

Our view of the relational model is that defined by standard SQL [18] or QUEL [38]. Technically, these are languages equivalent in expressive power to the relational calculus augmented with aggregates (and grouping), a small set of functions on column domains (e.g., addition, substring), a data definition language, update operations, and sorting operators. We assume that relational systems support sharing (concurrency control) and transaction recovery.

## THE APPLICATION

The application, called BondDB, is designed to support buying and selling bonds<sup>1</sup> by institutional sales people, traders, and risk managers in an investment bank. Users of BondDB can

- (1) try to find investments for a client that are both less expensive than the client's current investments and more profitable as long as interest rates remain below some specific value,
- (2) estimate a likely decline in the value of an investment for a typical day, based on the recent variability of a bond's price, to try to keep a trader from holding too many risky bonds, or
- (3) evaluate the fair value of an investment today for different future interest rate scenarios.

To perform these analyses, BondDB requires information about the bonds and their historical behavior.

BondDB's financial calculations and user interfaces made it impractical to construct the entire system using only a combination of SQL and application generation tools provided with the database management system. Therefore the application programs are written almost entirely in a conventional programming language.<sup>2</sup>

## THE DATA

BondDB contains the following data:

<sup>1</sup>Technically, the database contains many types of fixed-income securities, only some of which are termed bonds. The distinction is unimportant for the purposes of this paper.

<sup>2</sup>BondDB's user interface and analysis code consists of about 100 programs running on a VAX cluster under the VMS operating system. The language actually used is C, a common choice on Wall Street. Our conclusions would be similar, however, for other languages, such as Fortran, COBOL, or Pascal and for other choices of hardware.

- (1) basic information on the characteristics of about 10,000 different bonds,
- (2) time series of daily quotes for about 3,000 of these bonds, totaling some 1.8 million points,
- (3) about a dozen different kinds of bonds, and
- (4) information on portfolios of bonds.

We draw our examples from the representation of two main entity types: bonds and portfolios. Figure 1 is a simplified entity-relationship diagram [13] of BondDB's data-representation requirements.

**Bonds**

A bond is basically a loan from the holder of the bond to the issuer. It specifies a sequence of payments of principal and interest according to some rules or schedule. Here is a simple example.

Issuer: U.S. Treasury  
 Face Value: \$1,000.00  
 Payments: 5/31/88—\$38.75  
           11/30/88—\$38.75  
           5/31/89—\$38.75  
           11/30/89—\$38.75  
           + \$1,000.00 face value  
 Type of Bond: Treasury Note  
 Coupon: 7 3/4%  
 Maturity date: 11/30/89

In this case there are four semi-annual interest payments of \$38.75, and the principal is paid back in a lump sum after two years.

**Portfolios**

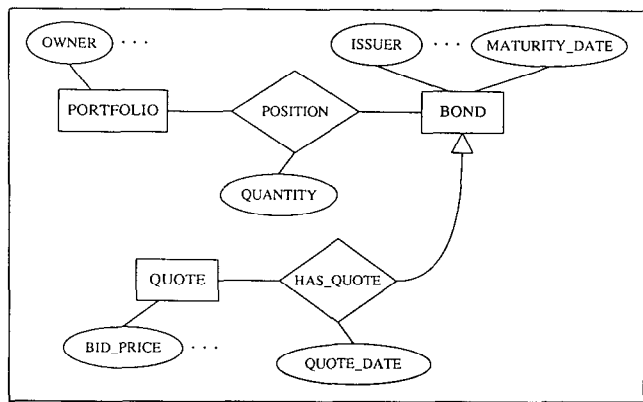
A portfolio is set of bonds that have been bought in various quantities. Each {bond, quantity} pair is called a *position*. For some analyses BondDB also needs to record when the bond was bought and for how much. BondDB also includes some information about the portfolio itself, such as the name of its owner, and the order in which to display the portfolio's positions.

**BondDB AND THE RELATIONAL MODEL—HOW GOOD A MATCH?**

In this section, we discuss central data management requirements of BondDB and how the developers tried to meet them using the relational database management system. We then describe features of an ideal (yet, we believe, realizable) system that could provide superior support.

**Management of Bulk Data Types**

One of the virtues of the relational model is that statements of its query language operate on entire relations at a time. One can formulate set-oriented expressions such as the following,



**FIGURE 1. Simplified Entity-Relationship Diagram of BondDB's Persistent Data**

```

SELECT BOND_ID FROM BOND
WHERE BOND_ID NOT IN
(SELECT BOND_ID FROM PRICE);
    
```

which prints the IDs of all bonds that have no price history (the good).

A *bulk data type* is a composite data type whose instances contain a dynamically varying, and potentially large, number of elements. Thus sets and relations are bulk data types, as are multisets, sequences (lists), trees, and graphs. BondDB required usable implementations of bulk data types other than relations and the ability to include instances of such types as attributes of bonds and portfolios.

For example, many bonds are characterized by a sequence of time intervals and prices indicating when and for how much the issuer can *call* the bond (i.e., prepay the loan to the bond holder). An example is the following:

Issuer: Brooklyn Union Gas  
 Type of Bond: Preferred stock  
 Last dividend: \$ 2.47  
 Face value: \$25.00  
 Moody's Rating: A2  
 SP rating: A  
 'Call' schedule: 8/31/81 to 8/30/86—\$27.92  
                   8/31/86 to 8/30/91—\$27.36  
                   8/31/91 to 8/30/96—\$26.81  
                   8/31/96 to 1/01/15—\$26.25

In this case, the issuer can call the bond at \$27.36 per share any time between 8/31/86 and 8/30/91. The most common query is to retrieve the call information when the other information about this security is fetched. A more sophisticated query might want to know the monetary value of the calls, which is a function of the bond's current price, price volatility, time to maturity, and remaining call schedule. Both queries require manipulation of the CALL SCHEDULE as a sequence.

Representing the call schedule with multiple attributes, that is,

```
BOND( . . . , CALL_DATE1, CALL_PRICE1,
      . . . , CALL_DATE1, CALL_PRICE1, . . . ),
```

would lead to well-known problems.

One of the problems is that maximum size of a call schedule may not be known when the schema is designed. If CALL<sub>n+1</sub> is added, all code using call schedules will need to be modified. Furthermore, variables in SQL cannot range over columns. For example, to print just the remaining call dates and prices, we would have to write:

```
SELECT CALL_DATE1, CALL_PRICE1 FROM BOND
      WHERE CALL_DATE1 > TODAY;
```

```
      :
```

```
SELECT CALL_DATEn, CALL_PRICEn FROM BOND
      WHERE CALL_DATEn > TODAY;
```

Therefore the developers include a separate table for this information for all bonds in the database:

```
CALL(BOND_ID, CALL_DATE, CALL_PRICE),
```

where BOND\_ID and CALL\_DATE constitute the key. The information for the Brooklyn Union Gas (BUG) example is then stored in two tables, BOND and CALL as seen in Figure 2. To fetch all information about the bond BUG-PS from a relational system into a host language one needs two queries (the ugly):

```
SELECT* FROM BOND WHERE
      BOND_ID = 'BUG-PS';
```

and

```
SELECT* FROM CALL WHERE
      BOND_ID = 'BUG-PS'
      ORDER BY CALL_DATE;
```

Ideally, one would like to express this relationship directly by defining an attribute CALL\_SCHEDULE of type SEQUENCE (of tuples) on the BOND table:

```
CREATE TABLE BOND (
      BOND_ID CHAR(10),
      ...
      CALL_SCHEDULE SEQUENCE (
        CALL_DATE DATE,
        CALL_PRICE FLOAT,
      )
      ...
);
```

Then to retrieve this information one would simply write:

```
SELECT* FROM BOND;
```

This would be followed by operations that can handle sequences as standard relational query languages handle relations.

BOND						
BOND_ID	Issuer	Type	Last Dividend	Face Value	Moody	S&P
...						
BUG-PS	BUG	Pref. Stock	\$2.47	\$25.00	A2	A
...						

CALL		
BOND_ID	CALL_DATE	CALL_PRICE
...		
BUG-PS	8/31/81	\$27.92
BUG-PS	8/31/86	\$27.36
BUG-PS	8/31/91	\$26.81
BUG-PS	8/31/96	\$26.25
...		

FIGURE 2. BOND and CALL Tuples for BUG-PS

### Procedures as Data

BondDB's developers took advantage of the capability of relational systems to store queries as views. For example, the following statements basically store a procedure to ensure that only the owner of a portfolio can modify it.

```
CREATE VIEW UPDATE_PORTFOLIO AS
SELECT* FROM ALL_PORTFOLIO WHERE
ALL_PORTFOLIO.OWNER = USER;
```

```
GRANT SELECT, UPDATE, INSERT, DELETE
ON UPDATE_PORTFOLIO TO PUBLIC;
```

In this example the stored query, which takes advantage of the set-at-time relational operators, provides a succinct, executable specification of the modifiability constraints on portfolios (the good).

Some relational systems offer triggers, i.e., statements to be executed whenever some change to the data, such as an insert, occurs [5]. Many database management system implementors consider triggers to be an exotic feature, but they could have been useful to BondDB's developers. For example, in BondDB, to make the most recent price quickly available, it is stored with the bond characteristic information in the BOND table. Thus an insert or update of the PRICE table may actually require modifications to two tables, PRICE and BOND. BondDB could have used triggers to ensure that the prices stored in the BOND table were always up-to-date as shown in Figure 3. For a DBMS with triggers, application programs would need to modify PRICE only. If the need to maintain the most recent price disappeared, or if it became necessary to maintain the most recent five prices, there would be no need to modify the application programs, because the update action on PRICE would be encapsulated in the database.

BondDB's need to store procedures went further than triggers and views. Some data can be stored succinctly in the form of a rule, but viewed by client software as the data generated by the rule (an extension of the

```

CREATE TRIGGER PRICE_INSERT ON PRICE
FOR INSERT AS UPDATE BOND
SET
    BOND.BID_PRICE = INSERTED.BID_PRICE,
    BOND.QUOTE_DATE
    = INSERTED.QUOTE_DATE
FROM BOND, INSERTED
WHERE INSERTED.QUOTE_DATE
    > BOND.QUOTE_DATE
AND INSERTED.BOND_ID = BOND.BOND_ID;
CREATE TRIGGER PRICE_UPDATE ON PRICE
FOR UPDATE AS UPDATE BOND
SET BOND.BID_PRICE = INSERTED.BID_PRICE
FROM BOND, INSERTED
WHERE INSERTED.QUOTE_DATE
    = BOND.QUOTE_DATE
AND INSERTED.BOND_ID = BOND.BOND_ID;

```

**FIGURE 3. Use of Triggers (Sybase Extended SQL Syntax) [39]**

view concept). For example, in some, but not all, call schedules, a sequence of rows can be expressed as a rule, e.g.,

BUG-PS becomes callable on 8/31/81 at \$27.92; thereafter the call price declines three times by 55 2/3 cents at five-year intervals; the call price is rounded to the nearest cent.

Other information can be expressed only procedurally, such as rules that determine the coupon of floating rate bonds based on the London interbank lending rate and the treasury bill rate. For example, the following formula determines one bond's coupon payments:

$$\text{Coupon} = \min \left\{ \begin{array}{l} 3M\text{-Libor} + 2.25\%, \\ \max\{3M\text{-Tbill} + 3\%, 3M\text{-Libor} + 1.75\%\} \end{array} \right\}.$$

A typical query is:

For a given future interest rate scenario, and during a given time period, what payments will the holder receive?

In cases such as these it would be best if the database could store the procedure but allow client software to view the data as a sequence (the ideal). What the developers have to do now is to represent BUG-PS as the sequence that host-language applications must manipulate (the ugly). They do not represent the lending-rate-dependent data at all, since it depends on values that are unknown in advance (the bad).

### Data Abstraction

Initially, BondDB's developers represented financial entities by a straightforward transcription of the normalized relational schema into C structs. The developers assumed it would be satisfactory to supply application programmers with routines which return query results

as arrays of structs. They also assumed that the programmers could be relied on to manipulate the struct representations of database tuples appropriately. There were two main flaws with the approach:

- (1) different application programmers tended to write routines with overlapping functionality, and
- (2) some kinds of bonds have important information in more than one table (e.g., the BOND and CALL tables in Figure 2) because of normalization, making it difficult for the programmer to know which tables contain information about a given bond.

Thus this approach caused reliability problems from the beginning.

Eventually, the developers realized that bonds and portfolios are modified and analyzed in certain stereotypical ways. For example, portfolios may be fetched from or written to the database, and manipulated or analyzed using mathematical tools. So, to an application, the portfolio is a collection of data with some set of operations that can be issued on that data. This is the definition of an abstract data type [21].

When the developers recognized the abstract data type character of entities represented in BondDB, they defined ADTs in C to represent BondDB's major financial constructs, (e.g., bonds and portfolios). The developers use the relational system for its virtues of ensuring recovery and concurrency control, associative access, the facility with which they can set up and modify relational schemas, and the ease with which they can generate query code for tables. In order to operate on an entity, the developers must map the data from relations to the C-language structures of the ADT, manipulate it using C-based abstract data type primitives, and then possibly return it to relations.

Ideally, the developers would define the abstract data types directly in the database system. Putting ADTs into the database dovetails well with the ideals of the previous sections, because

- (1) BondDB's ADTs have component instances of bulk data types,
- (2) one would like to store the operations of the ADT in the database as well as the data, and
- (3) sufficiently powerful ADT facilities could be used to define new bulk data types if necessary.

### Physical Data Independence

The relational model provides the application program with independence from internal data structures in three main ways:

- (1) Queries will have the same result regardless of the physical organization of the tables and the availability or unavailability of indexes.
- (2) Application programmers need not be concerned with the order of columns in tables or with columns that are not referenced since columns are referenced by name rather than by offset from the beginning of a record, for example.

- (3) Stored procedures in the form of views let the database management system present virtual tables, providing an optional additional layer of independence.

BondDB developers relied on all three kinds of physical data independence (the good).

The options for specifying details of physical implementation, however, did not always provide enough efficiency for BondDB (the bad). Performance considerations forced the developers to implement part of the database using operating system facilities. In particular, they installed a copy of the BOND table in shared memory, thus partly duplicating the function of the database management system's cache (the ugly). At that point they had to hand-code search structures since a linear search of the shared memory consumed more CPU resources than a lookup in the database using indexes.

The developers also were forced to rely on reserved fields to enable them to incorporate additional columns to the C struct without simultaneously re-compiling and re-linking all application programs. The management headaches introduced in maintaining the copied data were substantial. For example, whenever important data needed to be corrected, the shared memory copy had to be reinitialized making it unavailable for about 15 minutes. Furthermore, there was a period of several hours each day when the shared memory copy did not agree with the database copy.

To say that, ideally, the hardware should have run faster is not helpful. At the least, an ideal database management system should provide more options for specifying implementation aspects. In this example, being able to fix a table in memory would have been helpful.

### Recovery and Concurrency Control

There is one final issue we observed that ties the physical level to the abstract level. One of the important steps forward in database systems was the decoupling of concurrency control and recovery from the data model (relational, network, or hierarchical). This decoupling is not always desirable, at least for BondDB.

Users must be able to modify portfolios when they buy or sell bonds in them. Each modification may require several operations. For example, to add a bond, a user provides the characteristics of the bond (if it does not already exist in the database) and the quantity bought. To make this set of operations atomic (i.e., all-or-nothing) in a file system which only provides atomic operations on records, the developers would have to order the operations carefully and provide adequate redundancy to ensure recoverability. For example, when adding a position with a bond not yet in the database, one would first save the bond characteristic information, and then the position because without the bond characteristic information, the position is meaningless.

The transaction notion, supported by relational systems, is a big improvement over the facilities usually

provided by a file system (the good). Transaction support automatically provides recoverability at the level of a whole group of operations constituting, for example, the addition of a bond to a portfolio.

Implementing such transactions normally requires holding update locks until the end of the transaction (to prevent other transactions from reading uncommitted updates). In most systems the smallest item that can be locked is a page, and BondDB stored several portfolio positions per page. Unfortunately, there is a small set of actively modified portfolios to which most position records are added. This results in records from different portfolios interleaved on the same pages being updated concurrently. The developers therefore frequently received the complaint that a user was locked out of his portfolio when no one else was accessing it (the bad).

To overcome this problem, the developers defined a protocol in which processes only select for update (i.e., exclusively lock) the header record (the one containing the portfolio ID, the owner, and so on). All other portfolio information (hundreds of positions) is selected from the database without locking. BondDB then modifies the portfolio in user space. This works, but is inconvenient because to avoid unnecessary database updates, BondDB must keep track of what data in the portfolio has been modified and what has not. Code to keep track of how the portfolio has changed is hard to debug and test. Furthermore, there is the danger that an interactive SQL user, unaware of the protocol, might accidentally circumvent it. In an ideal system the unit of locking would be a portfolio, which is an abstract data type, so two users could modify different portfolios concurrently.

### RELATED WORK

Codd has faulted the SQL language for inadequate handling of null values and other design problems, and has faulted SQL implementations for permitting duplicate rows [14, 15]. Here we assess the utility of the relational model *per se*, rather than a particular query language. For example, some systems may permit duplicate rows partly out of real need for multi-relations. Associated difficulties are often due to inconsistent or obscure rules for handling the duplicate rows.

Much criticism of the relational model focuses not on its possible shortcomings for traditional business applications, but on either

- (1) its inadequate expressiveness for non-traditional database applications, such as design databases, cartographic databases, and multi-media databases, or
- (2) its inadequate performance for high transaction applications, notwithstanding recent advances in high-transaction systems such as NonStop SQL [40].

Sometimes both criticisms play a role, as in Peinl, Reuter, and Sammer's discussion of a bid-matching system for a stock exchange [31]. In this system problems arise because the relational system has inadequate knowledge of the semantics of the ADTs represented in it and

therefore cannot make optimizations necessary for the application's stringent performance requirements.

Nevertheless, some researchers have observed the shortcomings that BondDB encountered. For example, Bancilhon and Maier assert that current systems are not satisfactory for business applications because of the *impedance mismatch* between the relational system and the host language and then go on to discuss possible improvements [8]. The impedance mismatch is the discontinuity in type system and computational model between the database management system and the host language; the database management system utilizes set-at-a-time expressions with a non-procedural language, while the host language offers record- and field-level expressions in an imperative language. Indeed the impedance mismatch was one of the factors that forced BondDB's developers to manage two representations of each financial entity and supply translation code between the two.

Many workers are constructing systems that might be more satisfactory. The design space is large, but the work is promising.

Stonebraker, citing the success of the relational model, proposes evolutionary changes in POSTGRES, which supports user-defined data types and complex objects via the storage of procedures as attribute values [36, 37]. POSTGRES also allows lower-level access when performance requirements demand it. Another approach to extending the relational model, by allowing nested relations, is taken by AIM-II [17] and R<sup>2</sup>D<sup>2</sup> [22], which supports, in addition, user defined ADTs for complex objects.

Many projects (e.g., ENCORE [33, 35], GARDEN [34], Iris [20], O<sub>2</sub> [24], ORION [9, 23] PROTEUS [3], Trellis/Owl [28]) and even products (e.g., OPAL/Gemstone [16, 26], PROBE [19, 27], Vbase [4, 29]) have begun to build object-oriented database management systems whose basic persistent objects are ADT instances.

Some of the systems above, and others (e.g., Starburst [25, 32]) allow the application developer to include additional database access structures. Other systems (e.g., EXODUS [11, 12], GENESIS [10]) provide a set of building blocks from which to assemble a database management system tailored to a particular application. Finally, some researchers are attempting to erase the distinction between the programming language and the database management system altogether. Examples of this approach are Galileo [1, 2], and PS-Algol [6, 7].

## CONCLUSION

This article attempted an empirical assessment of the tools BondDB's developers had at their disposal—a relational database management system with a conventional programming language.

We saw that, while relational systems are good, their limitations require ugly workarounds, even though realizable improvements would result in a cleaner application implementation.

This was by no means a complete listing of all issues the developers faced. Some, such as politics, are perva-

sive but immune to a technological solution. Others, like the need for rapid development in a competitive environment depend on factors other than the data model such as the availability of people and machine resources. Application programmers do not ask for magic, however, just the best tools that can be efficiently implemented.

Necessary features of an ideal system begin to emerge. It would:

- (1) include a richer selection of built-in bulk data types than relations only, and support data abstraction to allow users to introduce instances of their own complex data types—our example was call schedule sequences and corresponding operations as database values,
- (2) allow procedures written in arbitrary programming languages to be stored in the database and viewed either as procedures or as the data that they produce at any given time—our example was floating rate coupons,
- (3) offer more tuning options, such as the ability to fix certain data in main memory, and
- (4) allow locking of logical objects, such as portfolio instances.

To summarize our challenge to designers of future database management systems: Go ahead, make our database.

**Acknowledgments.** This work was partially supported by the Office of Naval Research under grant N00014-85-K-0046. Special thanks to Norman M. Birkett, David M. Siegel, and Gudmundur Vigfusson, who helped define BondDB's information architecture, for many enlightening discussions on what would be better for BondDB, and to Yosi Ben-Dov for balancing the short-term exigencies of the users against the long-term exigencies of maintainable development. We also thank Norman M. Birkett, Gudmundur Vigfusson, and the anonymous reviewers for suggesting improvements.

## REFERENCES

1. Albano, A., Cardelli, L., and Orsini, R. Galileo: A strongly-typed, interactive conceptual language. *ACM Trans. on Database Syst.* 10, 2(June 1985), 230-260.
2. Albano, A., Ghelli, G., Occhiuto, M. E., and Orsini, R. A strongly-typed, interactive object-oriented database programming language. In *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems* (Pacific Grove, Calif., Sept. 23-26). ACM/SIGMOD and IEEE-CS TC, New York, 1986, pp. 94-103.
3. Anderson, T. L., Ecklund, E. F., Jr., and Maier, D. PROTEUS: Objectifying the DBMS user interface. In *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems* (Pacific Grove, Calif., Sept. 23-26). ACM/SIGMOD and IEEE-CS TC, New York, 1986, pp. 133-145.
4. Andrews, T., and Harris, C. Combining language and database advances in an object-oriented development environment. In *Proceedings of the 2d International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '87)* (Orlando, Fla., Oct. 4-8). ACM/SIGPLAN, New York, 1987, pp. 430-440.
5. Astrahan, M. M., et al. System R: A relational approach to database management. *ACM Trans. Database Syst.* 1, 2(June 1976), 97-137.
6. Atkinson, M. P., and Morrison, R. Procedures as persistent data objects. *ACM Trans. Prog. Lang. and Syst.* 7, 4(Oct. 1985), 539-559.
7. Atkinson, M. P., et al. An approach to persistent programming. *Comput. J.* 26, 4(1983), 360-365.

8. Bancilhon, F., and Maier, D. Multilanguage object-oriented systems: New answer to old database problems? Tech. Rep. 21-88, 1988. Altair, Domaine de Voluceau BP105, Rocquencourt 78153 Le Chesnay Cedex, France.
9. Banerjee, J., et al. Semantics and implementation of schema evolution in object-oriented databases. In *Proceedings of SIGMOD '87 International Conference on the Management of Data* (San Francisco, Calif., May 27-29). ACM/SIGMOD, New York, 1987, pp. 311-322.
10. Batory, D. S., Leung, T. Y., and Wise, T. E. Implementation concepts for an extensible data model and data language. *ACM Trans. Database Syst.* 13, 3(Sept. 1988), 231-262.
11. Carey, M. J., DeWitt, D. J., and Vandenberg, S. L. A data model and query language for EXODUS. In *Proceedings of SIGMOD '88 International Conference on the Management of Data* (Chicago, Ill., June 1-3). ACM/SIGMOD, New York, 1988, pp. 413-423.
12. Carey, M. J., et al. The architecture of the EXODUS extensible DBMS. In *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems* (Pacific Grove, Calif., Sept. 23-26). ACM/SIGMOD and IEEE CS-TC, New York, 1986, pp. 52-65.
13. Chen, P. P. The entity-relationship model: Toward a unified view of data. *ACM Trans. Database Syst.* 1, 1(Jan. 1976), 9-36.
14. Codd, E. F. Fatal flaws in SQL (part 2). *Datamation* 34, 17(Sept. 1988), 71-74.
15. Codd, E. F. Fatal flaws in SQL (part 1). *Datamation* 34, 16(Aug. 1988), 45-48.
16. Copeland, G., and Maier, D. Making Smalltalk a database system. In *Proceedings of SIGMOD '84 International Conference on the Management of Data* (Boston, Mass., June 18-21). ACM/SIGMOD, New York, 1984, pp. 316-325.
17. Dadam, P., et al. A DBMS prototype to support extended NF relations: An integrated view on flat tables and hierarchies. In *Proceedings of SIGMOD '86 International Conference on the Management of Data* (Washington, D. C., May 28-30). ACM/SIGMOD, New York, 1986, pp. 356-367.
18. Date, C. J. *Database: A Primer*. Addison-Wesley, Reading, Mass., 1983.
19. Dayal, U., and Smith, J. M. PROBE: A knowledge-oriented database management system. In *On Knowledge Base Management Systems: Integrating Artificial Intelligence and Database Technologies*, M. L. Brodie, Ed. Springer-Verlag, New York, 1986, pp. 227-257.
20. Fishman, D. H., et al. Iris: An object-oriented database management system. *ACM Trans. Office Inf. Syst.* 5, 1(Jan. 1987), 48-69.
21. Guttag, J. Abstract data types and the development of data structures. *Commun. ACM* 20, 6(June 1977), 396-404.
22. Kemper, A., Lockemann, P. C., and Wallrath, M. An object-oriented database system for engineering applications. In *Proceedings of SIGMOD '87 International Conference on the Management of Data* (San Francisco, Calif., May 27-29). ACM/SIGMOD, New York, 1987, pp. 299-310.
23. Kim, W., et al. Composite object support in an object-oriented database system. In *Proceedings of the 2d International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '87)* (Orlando, Fla., Oct. 4-8), ACM/SIGPLAN, New York, 1987, pp. 118-125.
24. Lecluse, C., Richard, P., and Velez F. O. An object-oriented data model. In *Proceedings of SIGMOD '88 International Conference on the Management of Data* (Chicago, Ill., June 1-3). ACM/SIGMOD, New York, 1988, pp. 424-433.
25. Lindsay, B., McPherson, J., and Pirahesh, H. A data management extension architecture. In *Proceedings of SIGMOD '87 International Conference on the Management of Data* (San Francisco, Calif., May 27-29). ACM/SIGMOD, New York, 1987, pp. 220-226.
26. Maier, D., Stein, J., Otis, A., and Purdy, A. Development of an object-oriented DBMS. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '86)* (Portland, Oreg., Sept. 29-Oct. 2). ACM, New York, 1986, pp. 472-482.
27. Manola, F., and Dayal, U. PDB: An object-oriented data model. In *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems* (Pacific Grove, Calif., Sept. 23-26). ACM/SIGMOD and IEEE-CS TC, New York, 1986, pp. 18-25.
28. O'Brien, P., Bullis, B., and Schaffert, C. Persistent and shared objects in Trellis/Owl. In *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems* (Pacific Grove, Calif., Sept. 23-26). ACM/SIGMOD and IEEE CS-TC, New York, 1986, pp. 113-123.
29. Ontologic, Inc. *Vbase Integrated Object System Technical Overview*. Billerica, Mass., 1986.
30. Oracle Corp. *Oracle Database Administrator's Guide Version 6.0*. Part No. 3601-V6.0. Belmont, Calif., 1988.
31. Peinl, P., Reuter, A., and Sammer, H. High contention in a stock trading database: A case study. In *Proceedings of SIGMOD '88 International Conference on the Management of Data* (Chicago, Ill., June 1-3). ACM/SIGMOD, New York, 1988, pp. 260-268.
32. Schwartz, P., et al. Extensibility in the Starburst database system. In *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems* (Pacific Grove, Calif., Sept. 23-26). ACM/SIGMOD and IEEE CS-TC, New York, 1986, pp. 85-92.
33. Skarra, A. H., and Zdonik, S. B. The management of changing types in an object-oriented database. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '86)* (Portland, Oreg., Sept. 29-Oct. 2). ACM, New York, 1986, pp. 483-495.
34. Skarra, A. H., Zdonik, S. B., and Reis, S. P. An object-server for an object-oriented database system. In *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems* (Pacific Grove, Calif., Sept. 23-26). ACM/SIGMOD and IEEE CS-TC, New York, 1986, pp. 196-204.
35. Smith, K. E., and Zdonik, S. B. Intermedia: A case study of the differences between relational and object-oriented database systems. In *Proceedings of the 2d International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '87)* (Orlando, Fla., Oct. 4-8), ACM/SIGPLAN, New York, 1987, pp. 452-465.
36. Stonebraker, M., Katz, R., and Patterson D., and Ousterhout, J. The design of XPRS. In *Proceedings of the 14th VLDB Conference* (Los Angeles, Calif., Aug. 29-Sept. 1). VLDB Endowment, Arlington, Vir., 1988, pp. 318-330.
37. Stonebraker, M., and Rowe, L. The design of POSTGRES. In *Proceedings of SIGMOD '86 International Conference on the Management of Data* (Washington, D.C., May 28-30). ACM/SIGMOD, New York, 1986, pp. 340-355.
38. Stonebraker, M., et al. The design and implementation of INGRES. *ACM Trans. Database Syst.* 1, 3(Sept. 1976), 189-122.
39. Sybase, Inc. *Commands Reference*. Document number 3241-2.1. Berkeley, Calif., 1987.
40. The Tandem Performance Group. A benchmark of NonStop SQL on the debit credit transaction. In *Proceedings of SIGMOD '88 International Conference on the Management of Data* (Chicago Ill., June 1-3). ACM/SIGMOD, New York, 1988, pp. 337-341.

**CR Categories and Subject Descriptors:** D.3.3 [Programming Languages]: Language Constructs—*abstract data types*; H.2.1 [Database Management]: Logical Design—*data models*; H.2.3 [Database Management]: Languages—*data description languages; data manipulation languages; query languages*; H.2.8 [Database Management]: Database Applications; H.4.2 [Information Systems Applications]: Types of Systems—*decision support*; J.1 [Computer Applications]: Administrative Data Processing—*financial*; K.6.3 [Management of Computing and Information Systems]: Software Management—*software development; software maintenance*

**General Terms:** Design

**Additional Key Words and Phrases:** Non-first-normal-form database management systems, object-oriented database management systems, persistent programming languages, relational database management systems

---

#### ABOUT THE AUTHORS:

**STEVE ROZEN** is currently a Ph.D. candidate at New York University's Courant Institute of Mathematical Sciences after spending two years on Wall Street. His thesis topic is the design and implementation of BULK, a very high level persistent language for prototyping and developing database applications.

**DENNIS SHASHA** is currently an assistant professor of computer science at the Courant Institute of Mathematical Sciences at New York University and a collaborator in a distributed database and an object-oriented database project at AT&T Bell Laboratories. His research interests are in parallel algorithms for data structures and in persistent set-based programming languages. He is also author of *The Puzzling Adventures of Dr. Ecco*.

Authors' Present Address: Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012. shasha@cs.nyu.edu.; rozen@csc2.nyu.edu.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.