OBJECT MANAGEMENT IN A RELATIONAL DATA BASE SYSTEM

Michael Stonebraker
Electronics Research Laboratory
University of California, Berkeley

Abstract

This paper first presents a collection of capabilities in the area of object management that are desired by "non business data processing" applications. Three approaches to providing this function, application specific systems, semantic data models and high leverage extensions to the relational model are examined. The advantages of the latter approach are described.

1. INTRODUCTION

Relational data base systems are considered to be a good fit for the needs of business data processing applications. Consequently, in the commercial marketplace they should displace older technology solutions over the next decade in all but the highest transaction rate environments. However, relational data base systems do not work well in other application areas such as those with spatial data bases (e.g. CAD), applications where text and/or images must be supported, engineering data bases which have a variety of special data types (e.g arrays, complex numbers, polar coordinates, etc.) and expert data base applications.

There are several capabilities, not present in existing systems, which are required by this collection of applications:

1) new data types

Current systems give the user the capability to define individual data items of type integer, floating point number or character string. In non-business data processing applications, one requires additional data types (e.g. point, line, polygon, line group, time, two-dimensional coordinates, polar coordinate), and the performance penalty inherent in simulating these objects on top of the basic types provided is prohibitive. Moreover, operators specific to the data type are required (e.g. intersection of two polygons, area of a polygon).

2) complex data types, i.e. data types which are composed of collections of other data items

For example, an object of type document might be composed of sections which are then composed of paragraphs, etc. Moreover, since a section may appear in many documents, data items must be shared among multiple complex objects.

3) union data types

Consider, for example, the salaries of employees. For some, the salary is a dollar amount. However, the salary of others might be determined procedurally (e.g. the salary of a manager is automatically 1.1 times

This research was sponsored by the U.S. Air Force Office of Scientific Research Grant 83-0254 and the National Science Foundation under Grant 85-04633

the salary of his highest paid direct report). The salary of foreign employees is expressed in pounds sterling. Hence, salary must be expressible in dollars, pounds or procedurally, and is thereby a union type.

4) varying models of complex objects.

Some users want "isa hierarchies" [SHIP80], a construct which cleanly supports generalization [SMIT77]. Other users want "part-of hierarchies" [KATZ83] which allow complex design objects to be built up out of simpler objects. Yet others want objects with no required structure at all. For example, a document can have a section composed of text, the definition of a graph to be constructed by a graphics package, a report to be constructed by a report writer, the definition of an image to be printed as a bit array, and various user procedures which return arbitrary data. The section of this example has no predetermined structure.

5) versions and snapshots [KATZ84, WOOD83]

Design data bases contain many partial designs (which may be based on different assumtions concerning speed, cost or size). These must be arranged into a tree of versions so that the dependencies of designs on earlier ones can be preserved. The same capability is required in source code control systems and other software engineering data bases.

6) alerters and triggers

Consider a program which shows a data item on the screen and then modifies the screen image whenever the data item changes. For example, "items in inventory" or "items sold" might be displayed by such a program. In current data managers the program must run the appropriate query, sleep for a predetermined length of time and then run the query again. Only alerters [BUNE79] provide a more efficient possibility. Triggers [ESWA75] are alerters which perform data base updates and are useful if one wants to percolate updates to dependent data items. For example, an application might want to give all employees who work for Mike a raise equal to one-fourth of the raise granted to Mike. Triggers and alerters are simply data base versions of the "demons" which have been widely implemented in AI programming languages which perform object management (e.g. LOOPS [BOBR84]).

7) rules and inference

Many applications have fields in their data bases that are more easily described by rules than stored extensionally as data values. For example, the teaching load of professors in the EECS department can be concisely described by the following rules:

1) The normal load is 8 contact hours per year
2) The scheduling officer gets a 25 percent reduction
3) The chairman does not have to teach
4) Faculty on research leave receive a reduction proportional to their leave fraction
5) Courses with less than 10 students generate credit at 0.1 contact hours per student
6) Courses with more than 50 students generate EXTRA contact hours at a rate of 0.01 per student in excess of 50
7) Faculty can have a credit balance or a deficit of up to 2 contact hours

These rules change frequently, as do the leave status, actual course assignments to faculty, and the identity of the scheduling officer. It is more natural to store the above rules in a DBMS and then infer the actual teaching load of individual faculty than to store the teaching load as ordinary data and attempt to enforce the above rules as extremely complex integrity constraints.

The above example illustrates rules in a business environment. However, expert data base applications typically wish to store "knowledge" as a collection of rules of the above form. In addition design rules in a CAD data base are also of the above composition. Many AI programming languages provide

such rules and inference (e.g. Planner [HEWI71] and Prolog [CLOC81]), and there is a widespread perceived need for such capabilities in a data manager.

This "laundry list" of problems is similar to others that have been recently constructed (e.g. [MAIE84]) and there are three approaches to correcting these deficiencies:

a) construct application specific systems which include only needed function. This approach is widely proposed for CAD applications.

b) construct a new comprehensive data model and query language which can solve most of the above problems

c) provide mechanisms within the relational model to correct the deficiencies.

In the next section we comment on the problems inherent in the first two approaches. Then, in Section 3 we present two very high leverage extensions for the relational model and suggest that it is the best building lock for object managers.

2. APPROACHES TO NEXT GENERATION DATA MANAGERS

2.1. Application Specific Object Managers

Constructing application-specific systems is widely advocated by architects of VLSI data base systems (e.g. [KATZ85, KIM85]). They propose to build CAD facilities (versions, configurations, equivalence relationships, layers, check-in-check-out, etc.) on top of a simple object manager which will support retrieving and storing arbitrary objects (i.e. an access method for complex objects). Any searching for specific objects or subobjects is left to the application program. They argue that VLSI design tools typically retrieve fairly large objects into virtual memory, perhaps converting them to a specialized representation, make extensive computations on the object, and finally put the object back in the data base. A design rule checker is given as the prototypical application which obeys this paradigm. Such "batch" applications cannot make sensible use of conventional data management capabilities.

This approach, while perhaps making sense in the short run, will be a long term disaster. First, current VLSI design tools may be primarily "batch" programs; however, there are some (e.g. KIC [KELL81]) which are VLSI design browsers. Such tools require the ability to move around in the design space and find all the cells or rectagons that are within a particular area of real estate on the chip. Such tools would benefit greatly from a query language to assist with the search for needed data.

In more general CAD environments, browsers become even more important. For example, if a building is put in a CAD data base, one of the major functions will be to show a user what is visible in the building from a specific angle and point of reference. Again, a query language will be a very useful capability.

Lastly, there is a perceived need to allow clients of CAD software to put their own data into a CAD data base. For example, in a data base containing a printed circuit board, a user might want to include arbitrary user data for each chip package (e.g. who he buys it from, how much it costs, the probability of failure, etc.). A user of such a data base naturally wants a query language, for example to ask what the total package cost of his circuit is.

The advocates of application-specific systems will probably find that they will be forced by user demands over time to construct a full function data manager with at least the capabilities of a current relational system. The history of computing is littered with special purpose data managers which grew in function over time to provide all the capabilities of a full-function general purpose system in an unmaintainable package that was no faster than general purpose commercial offerings.

In summary, I feel that CAD data base applications will demand solutions to most of the "laundry list" of problems in the introduction, and a comprehensive solution will ultimately be required. Moreover, I have a hard time thinking of other application areas which currently require special purpose function that will not ultimately follow the same path of requiring system extension. Even in high performance

transaction processing environments (the prototypical business application with specialized requirements), there is a definite trend toward toward more ad-hoc interactions [GAWL85] and the resulting requirement for a full function system.

2.2. Semantic Data Models

We now turn to creating a new data model as the solution to the "laundry list" of needed function. Such data models are usually called semantic data models, and include some subcollection of the following modelling constructs [CHEN76, HAMM81, CODD79, SHIP80, MYLO80, PLOU84]:

aggregation	generalization
cover aggregation	entities
attributes	relationships
functions	classes
long objects	semantics of time
null values	default values
extended data types	triggers
ordered relations	unique identifiers
checkin-checkout	outer joins
breakable locks	versions
unnormalized relations	data base procedures
support for "fat cursors"	alerters
transitive closure operator	nested transactions
system generated identifiers	expanded data dictionary
output formats for fields	input formats for fields
edit checks for fields	referential integrity
forced n-th normal form (for some n)	

Each of these constructs is useful in some application area. Unfortunately, including all in one software system within a reasonable amount of implementation effort is impossible. Moreover, the union of all these constructs will produce an impossibly complex reference manual. Lastly, if the research community (or any community of users) was asked to produce a list of the above constructs ordered by importance, there would likely be N lists from N people with little correlation between the lists.

Hence, the proposents of various semantic data models tend to specialize their data models to particular environments and leave out of their models constructs which are required in other environments. For example, there are few semantic data models which include "part-of hierarchies" or versions, thereby limiting their usability in CAD environments. Consequently, they open themselves to the same sort of pressure for extensions that will be faced by the advocates of application specific systems.

Moreover, a data manager for a semantic data model will have to co-exist with the relational systems which manage the business data processing data of a client. For example, the data on suppliers will presumably be in relational form. Hence, to answer the query, "how much does the cost of the packages on my circuit board increase if supplier X raises his price by Y?", one must be able to "join" data in the new data model to relational data. Support for this function will increase the complexity of a semantic data model system.

The last problem with semantic data models concerns the skill level of users. A semantic data model typically extends the relational model with several new constructs. Hence, the user must decide between representing his data relationally and using one or more of the new constructs. For example, a system that supports generalization would allow employees to be specialized to salaried employees and hourly employees. The wages of salaried people are a constant while those of the hourly people are computed as hours-worked times hourly-rate. A designer has the option of using the generalization hierarchy, or creating different tables for salaried and non-salaried employees and using the view mechanism to compose them (if the union operator is supported in view definitions). Moreover, he can use more than two tables and perform his own joins. There will be performance consequences to these choices which he must understand.

The people who design real data bases tend not to possess PhD's in computer science. In fact, if current trends continue, there will be a considerable shortage of trained data base designers for the forseeable future. Hence, it is likely that data bases will be designed by less and less skilled persons. Currently, data base design for the relational model is not easy for most application designers. Moreover, tuning up a relational data base system to run an application is considered a "black art" (and probably an ideal application for an expert system). If one proposes a data model of greater complexity than the relational model, the following "reality checks" are useful:

> The difficulty of using a data model increases much more than linearly with the number of data model constructs.
>
> The difficulty of tuning a data base system increases much more than linearly with the sum of number of data model constructs and the number of tuning parameters.

Hence, to be cost-effective any extension must have very great expressive power.

One possible conclusion from the above discussion is that complexity is bad. Certainly this was part of the message conveyed by the original relational enthusiast [CODD70]. Moreover, there is ample evidence that anything in excess of "spartan simplicity" is bad in other environments. PL/1 (and perhaps ADA) are considered languages of excessive complexity. They lead to complex and slow compilers and cause user difficulties with the constructs. Books such as [BROO75] are full of anecdotes indicating the pitfalls of attempting to get complex systems to work. Additional examples are presented in [LAMP83], and a statement of this principle in the data base arena appears in [DATE84].

An alternate conclusion is that data base system designers should look for extended concepts with maximum "leverage", i.e. maximum usefulness to the user with a minimum increase in system complexity. The basic problem with semantic data models is that each construct has only modest leverage. I feel that a better base for next generation applications would be a relational system with a small collection of strategic high-leverage extensions. The next section briefly explores two such constructs.

3. HIGH LEVERAGE EXTENSION TO RELATIONAL DATA MANAGERS

3.1. An Extendible Type System

It seems clear that an extendible type system is a good idea [REHF84, STON83]. Not only is such a facility useful for engineering applications, but also it appears desirable in certain business data processing situations also. For example, most data base systems implement a data type for "dates" and some implement subtraction as an operator for the date data type. As one would expect, the subtraction operator yields the following example answers:

April 15 - March 15 = 31 days
March 15 - February 15 = 28 days

However, this notion of subtraction for dates is inappropriate for a client who computes interest on financial bonds. In this application, one receives an equal amount of interest each month, and hence all months are assumed to have 30 days. In this environment, one wants a new definition of subtraction, e.g.:

April 15 - March 15 = 30 days
March 15 - February 15 = 30 days

The cost of performing this alternate definition of subtraction in an application program is dramatic. One must issue a data base query to get relevant data, then perform the computation in a user program, then do another data base update to put the appropriate value back. To a first approximation, the applications runs twice as slow as it would if the correct subtraction was available as a DBMS operator. Hence, I believe that an extendible type system presents a very high leverage extension. Moreover, it requires no changes to the relational model. To illustrate the leverage of this construct, we will compare it with an alternate extension, correct treatment of null values. Both concepts have similar intellectual and implementation complexity; however one is vastly more powerful and therefore, has more leverage than the other.

The implementation difficulties of adding extended types to a DBMS have been explored in [ONG83, STON86] and are:

1) The parser must be changed to allow table driven operator names
2) Query processing routines must be extended to optimize the new operators. In general the optimizer must be converted to be table driven.
3) Access methods must be extended to support collating sequences other than ascending ASCII.
4) The system level documentation must be changed to document interface routines for supporting new types and operators.
5) The user level documentation must be extended to indicate the possibility of new operators and types.

With such a proposal a wealth of new types can be implemented. Moreover, a library of types can be built up and shared among a community of users. For each new type, it is possible to implement a slightly different version which has null values with appropriate semantics.

On the other hand, one can hard-wire null values into a data manager. Such an action requires:

1) The parser must be extended to include the keyword "null".
2) Query processing routines must optimize queries which include nulls.
3) Special code must be added to aggregate evaluation to test for an empty set and return a null answer.
4) Access methods must correctly handle calls that access null values.
5) User level documentation must be changed to explain that queries will have two different answers depending on whether nulls are present in the relation.

It is probably true that extended data types have somewhat greater implementation complexity than null values; however, the difference is not large. Moreover, extended types can be used to model nulls as well as many other objects. Obviously, data base systems should implement extended types rather than null values. We now turn to a second high-leverage extension.

3.2. Data Base Procedures

An extendible type system provides the ability to define a column of a relation to be any sort of user defined object. Then, the client can implement any operators that he wishes for this data type by providing an appropriate procedure to evaluate the operator. This "object-operator" paradigm is similar to the Smalltalk or Loops paradigm of objects and methods. The problems with this paradigm are twofold.

1) It is difficult to provide search capabilities within an object

2) It is difficult to share subobjects

Consider an object of data type "bolt". A user does not usually want the whole bolt returned to his application; rather he might want only the thread pitch, the size of the head or the weight. Each of these requests would have to be supported by a different operator for the bolt data type, and one would have a vast collection of such operators providing search and extraction of subobjects. The result is sure to be confusing. Moreover, portions of the search capabilities of existing data managers will be constructed within the application programs which implement these operators.

The second problem is one of shared subobjects. The same head might be used for several different bolts. In this case, one would want the data for the head to appear only once and be shared by any higher level objects which use it. A data item in a column of a relation is considered as an atomic object by the data manager, and such sharing is not readily accomplished.

The goal is to rectify both problems as well as provide a mechanism which can be used to support ANY user desired semantics for structured objects. Query language procedures as a data type are one high leverage way to satisfy this goal. The basic concept is that a field in a relation can have a value consisting of a collection of query language commands.

Consider, for example, a conventional EMP relation with the requirement of storing data on the various hobbies of employees. The desired form of the EMP relation would be:

EMP (name, age, salary, hobbies)

Three relations containing hobby data might be:

SOFTBALL (emp-name, position, average)
SAILING (emp-name, rating, boat-type, marina)
JOGGING (emp-name, distance, best-time, shoe-type, number-of-races)

Each gives relevant data for a particular hobby. For example, Smith could be added as the catcher of the softball team by:

```
append to SOFTBALL (name = "Smith", position = "catcher", average = 0)
```

Then, Smith could be added as an employee by:

```
append to EMP (
        name = "Smith"
        age = 40
        salary = 10000,
        hobbies = "retrieve (SOFTBALL.all)
                where SOFTBALL.name = "Smith""
    )
```

In this case, the first three values are conventional fields while the fourth is a field of data type "collection of commands in QUEL". The value of this last field is obtained by executing the command (s) in the field.

It is clear that fields of type QUEL automatically provide shared subobjects. One need only construct multiple queries which obtain data from the same place. However, to use the data manager to provide search capabilities within an object, extensions to the query language must be provided. First, one must allow the components of an object to be directly referenced. For example, one could retrieve the batting average of Smith as follows:

```
retrieve (EMP.hobbies.average)
where EMP.name = "Smith"
```

This “multiple dot” notation has many points in common with the data manipulation language GEM [ZANI83], and allows one to conveniently access subsets of components of complex objects. In addition, one should provide the capability of executing data in the data base. For example, one can find all the hobby data for Smith by running the following command:

```
execute (EMP.hobbies) where EMP.name = "Smith"
```

Lastly, one should be able to reach into an object and update a component subobject. For example, to change the position of Smith from catcher to outfield, one could make a direct update to the SOFTBALL relation. However, it is cleaner to allow the update to be made through the EMP relation as follows:

```
replace EMP.hobbies (position = "outfield")
where EMP.name = "Smith"
```

A complete language which supports QUEL procedures has been described in [STON85] along with a suggested query processing plan. In the remainder of this section we make several comments on this concept.

First this construct can model "isa hierarchies", "part-of hierarchies", hierarchies with no forced structure, as well as non-hierarchies. Examples of all constructs appear in [STON84], and will not be repeated here. Hence, many of the complex object situations can be modelled with one construct. However, the leverage of this construct does not stop here. One obtains data base procedures for free. Moreover, all current data managers support query language commands anyway (as compiled query plans). Hence, all we are proposing is to leverage a construct that is already present by making procedures "first class objects" instead of "second class objects".

In addition, one can apply caching techniques to procedural objects. For example, a QUEL procedure has a textual representation. In addition, it may have a compiled query plan which will be executed by the run-time system. Besides caching the query plan one can also cache the answer to the command if it is

a retrieve. An efficient way to invalidate the cache when updates occur is described in [STON85].

Such caching provides additional examples of leverage. For example, one can use this capability to write a very simple preprocessor. At compile time the preprocessor inserts a command into a system relation, e.g.:

SAVED (id, QUEL-code)

A demon can compile the command asynchronously (or on demand) and cache the answer if the command is a retrieval. At run time the user program simply runs the following query:

execute (SAVED.QUEL-code) where SAVED.id = some-value

The plan will have been generated and perhaps the answer cached between compile time and run time. Hence, the performance of retrievals may be "blindingly fast", as they may have been pre-executed. Also, no special purpose code is required to invalidate precomputed query plans, as is present in [ASTR76].

This caching implementation also leverages applications which wish to keep computations as redundant data. For example, suppose one wants to keep the average salary of all the employees in each department. This can be supported by the following relation:

AVERAGES (dept, QUEL-code)

Any reference to the QUEL-code field will generate the average directly if it is cached. Hence, the user is spared the overhead and complexity of maintaining such computations himself.

This section has indicated how modest extensions to a relational data base system can generate a system with the capability of supporting many kinds of complex objects as well as substantial added capabilities of general interest. An implementation of a new data base system. POSTGRES, [STON85] with these (and other) features is underway.

4. CONCLUSIONS

This paper has explored three ways that next generation data base systems can support non-business data processing applications. The advocates of application specific object managers will face pressure for general purpose capabilities in all environments that I can think of. Incremental construction of a full function system in this fashion has not worked well in the past.

Advocates of semantic data models face a leverage problem. There is a large collection of modelling constructs which are potentially useful, and each one seems to provide only modest leverage. The union of all constructs is impossibly complex. Realistic subsets may generate application specific systems and pressure for extensions.

Two high leverage extensions to the relational model were then presented; extended data types and query language procedures. Most required function in the "laundry list" of capabilities from the introduction appear to be supportable by these two extensions. Hence, the simplicity of the relational model is retained while simultaneously allowing various semantic data models to be efficiently simulated by application level software.

REFERENCES

[ASTR76] Astrahan, M. et. al., "System R: A Relational Approach to Data," ACM-TODS, June 1976.

[BOBR84] Bobrow, D., and Stefik, M., "The LOOPS Reference Manual," XEROX-PARC Technical Report, 1984.

[BROO75] Brooks, F., "The Mythical Man Month," Addison Wesley, Reading Mass., 1975.

[BUNE79] Buneman, P. and Clemons, E., "Efficiently Monitoring Relational Databases," ACM-TODS, June 1979.

[CLOC81] Clocksin, W. and Mellish, C., "Programming in Prolog," Springer-Verlag, West Berlin, Germany, 1981.

[CHEN76] Chen, P., "The Entity-Relationship Model: Toward a Unified View of Data," ACM-TODS, March 1976.

[CODD70] Codd, E., "A Relational Model of Data for Large Shared Data Banks," ACM-CACM, June 1970.

[CODD79] Codd, E., "Extending the Database Relational Model to Capture More Meaning," ACM-TODS, Dec. 1979.

[DATE84] Date, C., "A Critique of the SQL Database Language," SIGMOD RECORD, November 1984.

[ESWA75] Eswaren, K., "A General Purpose Trigger Subsystem and Its Inclusion in a Relational Data Base System," IBM Research, San Jose, Ca., RJ 1833, July 1976.

[GAWL85] Gawlich, D., "Minutes of High Performance Transaction Systems Conference," Asilomar, Ca., September 1985.

[HAMM81] Hammer, M. and McLeod, D., "Database Description with SDM: A Semantic Data Model," ACM-TODS, Sept. 1981.

[HEWI71] Hewitt, C., "Planner: A Language for Proving Theorems in Robots," Proc. 1971 Internation Joint Conference on Artificial Intelligence, 1971.

[KATZ83] Katz, R., "Managing the Chip Design Data Base," IEEE Computer, Dec 1983.

[KATZ84] Katz, R. and Lehman, T., "Database Support for Versions and Alternatives of Large Design Files," IEEE-TSE, March 1984.

[KATZ85] Katz, R., et. al., "Version Modelling Concepts for Computer-aided Design Databases," submitted for publication.

[KELL81] Keller, K., "KIC: A Graphics Editor for Integrated Circuits," Masters Report, University of California, Berkeley, Ca., June 1981.

[KIM85] Kim, W., private communication.

[LAMP83] Lampson, B., "Thoughts on System Design," Proc. 9th Symposium on Operating System Principles, Bretton Woods, N.H., Sept. 1983.

LORI83] Lorie, R. and Plouffe, W., "Complex Objects and Their Use in Design Transactions," Database Week Conference on Engineering Applications, IEEE Computer Society Press, May 1983.

[MYLO80] Mylopoulis, J. et. al., "A Language Facility for Designing Interactive Database-intensive Systems," ACM-TODS, June 1980.

[ONG83] Ong, J. et. al., "An Implementation of An Abstract Data Type Facility," SIGMOD RECORD, March 1983.

[REHF84] Rehfuss, S. et. al., "Particularity in Engineering Data," Proc. 1st International Conference on Expert Data Base Systems, Kiowah, S. C., Oct. 1984.

[SHIP80] Shipman, D., "The Functional Data Model and the Data Language DAPLEX," ACM-TODS, June 1980.

[SMIT77] Smith, J and Smith, D., "Database Abstractions: Aggregation and Generalization," ACM-TODS, June 1977.

[STON83] Stonebraker, M. et. al., "Application of Abstract Data Types and Abstract Indices to CAD Data," Database Week Conference on Engineering Applications, IEEE Computer Society Press, May 1983.

[STON84] Stonebraker, M., et. al., "QUEL as a Data Type," Proc. 1984 ACM-SIGMOD Conference on Management of Data, Boston, Mass., June 1984.

[STON85] Stonebraker, M. and Rowe, L., "The Design of POSTGRES," Electronic Research Laboratory, University of California, Berkeley, Ca., November 1985.

[STON86] Stonebraker, M., "Inclusion of New Types in Relational Data Base Systems," Proc. 2nd International Conference on Data Engineering," Los Angeles, Ca., Feb 1986.

[WOOD83] Woodfill, J. and Stonebraker, M., "An Implementation of Hypothetical Relations," Proc 9th VLDB Conference, Florence, Italy, Dec 1983.

PostgreSQL is the worldâ€™s most advanced open source database, and per the PostgreSQL Wikipedia page it is an object-relational database management system (ORDBMS) with an emphasis on extensibility and standards compliance. In this article, we try to understand why would PostgreSQL be named an object-relational thing. What is Object Oriented Programming and how does that apply to a database system? Table of Contents. Object Orientation in Progamming Languages. Object Orientation in PostgreSQL. Object-Relational Database Management System. Conclusion. Object Orientation in Progamming Languages. Should we consider giving students an object oriented database (if they still exist)? This saves time on ORM and plumbing. Should we stick with RDBMS and tell students to roll their own ORM?Â You need to be able to walk before you run, and utilizing ORM eliminates a very important step in the learning process regarding using a relational database in an application. You should stick to a VERY lightweight data access framework--one that requires students to write their own SQL and (at best) doesn't allow this scope of the code to be tied to the UI or (at least) doesn't require it. I am admittedly unfamiliar with the Java world as it relates to actual enterprise development, but I realize (somewhat begrudgingly) that it's THE environment in the educational system. A relational database management system (RDBMS) is a program that allows you to create, update, and administer a relational database. Most relational database management systems use the SQL language to access the database. What is SQL? SQL (Structured Query Language) is a programming language used to communicate with data stored in a relational database management system. SQL syntax is similar to the English language, which makes it relatively easy to write, read, and interpret. Many RDBMSs use SQL (and variations of SQL) to access the data in tables. For example, SQLite is a relational databa...