Comments on the "Third-Generation Data Base System Manifesto"

David Maier

Oregon Graduate Institute
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 91-012

April, 1991

Comments on the "Third-Generation Data Base System Manifesto"

David Maier
Oregon Graduate Institute

April 18, 1991

Abstract

These notes are my reflections on the "Third-Generation Data Base System Manifesto" by the Committee for Advanced DBMS Function, the version of 22 April 1990, hereafter called "3GM". While this is a personal view, I benefitted from a discussion of the 3GM with Malcolm Atkinson, Francois Bancilhon, Dave DeWitt and Klaus Dittrich. I recommend that one read the 3GM before reading this document, as I assume familiarlity with its contents. [It has been printed in SIGMOD Record 19:3, September 1990.] These notes may be freely copied for personal use as long as they are duplicated in their entirety.

1 Introduction and Organization

It is unclear whether the 3GM is intended as a definition of a new class of DBMSs, as a prognostication, as a research and development agenda or as a marketing piece. It may be something of each, judging from the arguments in it. Some are on semantic and engineering grounds, but other are based on perceived customer demands or market forces. Most disturbing is an undercurrent of implication that requirements for next generation database systems should be tempered by what are compatible extensions to current relational models and technology. The message I read is that relational systems, or their slight extensions, are the "end of history" as far as database systems go. The bottom line is that there should be one flavor of next generation database system and that flavor should be extended relational.

There is a tone in the 3GM of "if it can't be added easily to current relational systems, it must be wrong" and that data models should only evolve if the current implementations can evolve along with them. We shouldn't abandon the successes of the relational model lightly, but they shouldn't bind us from exploring new territory. Columbus had to sail out of sight of land to find the New World. Even if third-generation systems end up looking a lot like their parents, I doubt database technology is best advanced by research that limits itself to relational extensions or is dictated by current practice. Research should be unfettered by the current state of affairs, in order to foster the most diversity in new models and implementation technology. Certainly research should be driven by real applications, but later benchmarking and analysis can winnow the applicable ideas from the impracticable, for incorporation in the next round of commercial systems. I question whether inheritance, method attachment and recursive complex objects would be showing up in relational extensions if not for the experimentation with object-oriented databases.

Most of these notes are structured as direct responses to individual tenets and propositions of the 3GM. However, the next section takes up three general topics that I will come back to throughout. The first is *type extensibility* and what database application developers really want from a type definition system. The second topic is trying to clarify what the term "rule" means in a database setting. The third is the distinction between types, collections and names.

I warn the reader that these notes were written in a rush and a sometimes agitated frame of mind. Expect lapses of grammar and rationality throughout.

2 Preliminaries

2.1 Type Extensibility

Many of us in the object-oriented database field have struggled to distill out the essence of "object-orientedness" for a database system. Several papers already propose definitions, and I've heard of four groups working on standards of some form. My own thinking about what the most important features of OODBs are has changed over time. At first I thought it was inheritance and the message model. Later I came to think that object identity, support for complex state, and encapsulation of behavior were more important. Recently, after starting to hear from users of OODBMSs about what they most value about those systems, I think that *type extensibility* is the key. Identity, complex state an encapsulation are still important, but insomuch as they support the creation of new datatypes.

Type extensibility means being able to augment the set of types that the database system supports, beyond just the records and sets or trees of records provided by most current systems. But "type extensibility" can stand for different capabilities—adding new base types, allowing nested record structures, adding an array constructor—so let me say what kind of type extensibility is most important for database programmers. It is support for what I will call ***manifest types***. Manifest types have three properties; they are ***first-class***, ***immediate***, and ***abstract***. I define these terms in turn.

- By *first-class* I mean that instances of user-defined types should be on the same footing as instance of system-supplied types. Those instances should be able to participate in collections or exist independently. They should be able to take instances of other types as arguments to their operations, such as putting an `Employee` object in a `Queue` or asking a `Gate` object if it is connected to a particular `Wire` object. They should enjoy complete data management support: transparent persistence, recovery, concurrency control, authorization, querying and so forth. A user-defined type should be an acceptable argument any place that a system-supplied type is allowed. User-defined types should be usable to define yet other types. (The first-class requirement is related to the notion of datatype completeness in PLs, and to the *definitional types* of Buneman and Ohori.)

- *Immediate* means that the type extension facilities are directly accessible to any database programmer. This availability, to me, encompasses two requirements. One is that new types can be defined at schema definition time, rather than only at database implementation or generation time, as some type extensibility mechanisms require. Second is that new types can be defined using only the DDL and DML of the database system. It is not necessary to use the database system implementation language, a separate application programming language, or a special database extension language.

- *Abstract* means that all or some of the implementation of a type can be hidden from clients of the type, be they other types, application programs or end users. Abstraction helps in data modeling, as it lets the database user and programmer view individual entities in the world as single data items in the database. It is not necessary to manipulate multiple data items when dealing with what is conceptually a unit, even if that unit is represented by several components. Abstraction also provides logical data independence, localizing the effects of changes in type implementations, thereby protecting other types an existing applications from needing modification, in many cases. When parameterization is supported as well, abstraction lets users create new type constructors.

Most OODBs and Persistent PLs support manifest types. I have not yet seen any relational extensions that do.

2.2 Rules

The term "rules" is heavily overloaded in its use in database literature and in the 3GM. *Rule* can have the broad or everyday sense of "an established principle or standard" in connection with running an enterprise or performing a task. "Rule" in this everyday sense encompasses many possibilities: a policy, a guideline, a heuristic, a convention, an administrative regulation, a good business practice, a definition. In computer science, "rule" is given a narrower meaning, as a statement in some formal system, such as a Prolog clause or a pattern-action pair in a production system. Let me use the term *human rules* to refer to rules in the everyday sense, and *machine rules* for the narrower meaning. Machine rules can capture some kinds of human rules; human rules must be formulated as machine rules for a computer system to process with them.

Within the domain of machine rules, I want to call out two subcategories, *declarative* and *procedural* rules. Declarative rules have the form "`if A then B`", meaning if statement `A` is true, you can conclude `B`. For example, "If the monthly salary of E is M, then the yearly salary of E is 12 * M." Procedural rules have the form "`if A do B`", meaning if the condition or event `A` occurs, perform the *action* `B`. For example, "If the salary of employee E increases by I, add I to the total salary of the department D of E." In the database setting, people have proposed quite powerful things for what the action `B` can be, including arbitrary DML commands or programming language procedures with embedded DML calls. I point out that "if-then" and "if-do" rules are quite different beasts. For many classes of "if-then" rules, such as the Horn clauses used in logic programming, it is possible to define the global semantics of a database with respect to a set of rules, using minimal model semantics. The database represents the minimal set of conclusions drawn by repeated application of the rules to derive new facts from the stored base facts. The important point is that this closure of the database under the rules is unique and independent of the order of the application of the rule (for most classes of rules used for deductive databases). Of course, the goal of query processing in deductive database systems is to avoid computing the entire closure of the database to answer a query about derived information. Another point is that "if-then" rules don't modify the database.

Procedural "if-do" rules, on the other hand, have no global semantics independent of their application order. Firing the rules in one order can take the database to a different state than firing them in a different order. A set of these rules seems like a parallel **goto** statement to me (or maybe navigational programming without a compass). I am wary of "if-do" rules in the database setting: huge cascades of triggered actions; not being able to determine if a set of rules terminates; debugging rule sets; not being able to figure out after the fact what caused a certain update to the database.

In future discussions of DB system features, it is important to avoid ambiguity with the term "rule" and to distinguish human rules from "if-then" rules from "if-do" rules, probably by using distinct terms for the different concepts. The 3GM confuses the human and machine senses of "rule" in justifying the requirement for rules. Obviously, if you walk into a business or talk to a designer and ask "Do you have rules you use in your work?", he or she will answer "Of course," having human rules in mind. If you question further, you are likely to find out that some of the human rules can be captured as machine rules, such as "The expected value of a loan is the principle plus interest adjusted by inflation" or "Whenever you add a chip to an IC board design, be sure you add connections for power and ground." But there will be other human rules that resist capture as machine rules, such as "Be sure that all documents for the Chartwell account use simple English" or "Keep run lengths of power and ground busses to a minimum." Thus, claims that customers from this or that enterprise or application require support for rules need to be scrutinized more closely to see what kind of rules are in use. Some human rules might be implemented as machine rules in the database, while an expert system or application program might be necessary to deal with others, and some will remain forever beyond the ken of database assistance: "Always put the stopper back on the ink bottle when you leave your desk."

It is an interesting exercise to think about casting the human form of the Macy's-Nordstrom rule (from Tenet 1) into a machine rule. (One cannot put an advertisement for Macy's on the same page as an advertisement for Nordstrom's.) One glitch is that the rule involves inequality. If `ad1` and `ad2` run on the same date and `ad1` is for Macy's and `ad2` for Nordstrom, then `ad1.page != ad2.page`. It's not clear that you could capture this knowledge as an "if-then" rule that would be useful for inferencing. Few inferencing systems can reason effectively with inequality. What about an "if-do" rule? What would be the action part of the rule? Will there be a `set ad1.page not equal to ad2.page` command? Finally, what qualifies exactly as an ad for Macy's? A stock offering? What about an ad seeking employees? How about if Macy's placed an ad promoting a charitable event? What if Nike places an ad that says "available at Macy's"?

2.3 Types, Collections and Names

The 3GM make scant distinction among types, collections and named values. In modern programming languages, defining a type, creating a collection of instances of that type, and declaring a variable to hold such a collection are separate activities. The 3GM assumes the relational status quo, where all three are lumped together. Adding a relation to a database scheme defines the tuple type, the set type over that tuple type (a relation type), creates an instance of the set type and assings that instance to a variable (the relation name). The tuple type, the set type and the variable are all lumped together with a single name (e.g., `Employee`). Artifacts of this view are further assumptions that functions are associated with collections (rather than instances or types), that each record type has a unique collection type derived from it, that there is one instance of that collection type (which contains all instances of the type), that collections are homogeneous, that a record cannot belong to multiple collections, that only collections can be named, and on and on. It also tends to interfere with having multiple implementations of a type and in providing a structured name space, which are handicaps in building large systems.

Contrast this situation with the OODB and Persistent PL view in which many different collection types can be defined over a given element type (`set of T, bag of T, list of T`), collections can be heterogeneous, there can be many instances of a collection type, an object can belong to multiple collections, and any object can be given a persistent name. (Some OODBs do provide classes that act like both a type definition and a collection of instances, but it is still possible for those instances to be named and belong to several collections.) This restricted view of types and collections, in my eyes, drastically limits the way the new features might be integrated in the next generation of database systems, and hamstrings their data models.

3 Tenets and Propositions

With the preliminaries taken care of, I now turn to the specific tenets and propositions of the 3GM. I take them in an order slightly different from that in the paper, preferring to treat propositions right after the tenets from which they follow.

3.1 Tenet 1: Besides traditional data management services, third-generation DBMSs will provide support for richer object structures and rules.

I have no quibble with "richer object structures." (However, I don't see a reason that "business data processing elements" should be regarded as fundamentally different from other data.) However, to me, listing "rules" confounds requirements with solutions. "Having rules" to me advocates a particular mechanism to support certain database capabilities, but misses stating directly what those capabilities are. From the paper, I see three such capabilities that rules are expected to support: inferencing, integrity constraints and event sequencing.

Inferencing says that the database contains knowledge that allows it to infer or derive information beyond the base facts it stores. Views are an example of a simple inferencing capability.

Integrity Constraints means the database can enforce restrictions on the database state beyond those given by the structural parts of the schema, such as keys, referential integrity and cardinality restrictions.

Event Sequencing means that the DBMS has information on the order of events that take place in an enterprise or design process and can automatically initiate some of those events.

Most of the later examples of rules in the 3GM are of the procedural "if-do" variety. I am unconvinced that they are the best approach for providing any of these capabilities. Databases should provide all three capabilities, but they don't necessarily need to be provided via rules. I will discuss the suitability of rules for each of these capabilities in turn, and also mention some alternative solutions to providing them. (It's kind of like telling a programming language designer you need iteration, conditional and case control structures, and he or she concluding that a **goto** statement is your requirement.)

3.1.1 Inferencing

The Horn-clause-style rules of deductive and logic database are a powerful mechanism for capturing a wide variety of knowledge and for processing with it. However, there are many other kinds of knowledge and strategies of knowledge processing that can be used for inferencing: classification hierarchies, statistical inference, approximate reasoning and domain transformations (such as FFT) to name a few. Some knowledge is best represented as algorithms. Even when certain kinds of knowledge can be captured as rules, rule processing might not be the best way of reasoning with the knowledge. Knowledge about arithmetic can be expressed with Horn-style rules, but reasoning about arithmetic relationships is probably better done using symbolic manipulation techniques, relaxation, linear programming or term-rewriting methods. Which brings me to an important point. Storing knowledge, be it in the form of rules or otherwise, and processing data using that knowledge are not the same thing. As knowledge is a shared, persistent resource, it is hard to argue that it shouldn't be stored in the database. Assuming that the database will do all the knowledge processing is more questionable. Control is still a big part in a deductive system, and deciding the best control strategy for answering a particular question is hard. Most such systems still need some human guidance on control strategies to give reasonable performance.

I don't want to assume that the database will be the only place (or the best place) for inferencing. There should be basic support for knowledge management and some kinds of deduction, but there also must be good linkages to external inference engines. Consider an analogy. Graphical user interfaces will be even more prevalent than inferencing capabilities in next-generation database applications. Forms and display definitions will be stored in the database (this already happens), but the actual activation and running of interfaces will be handled mostly by windowing and presentation systems.

3.1.2 Integrity Constraints

Are "if-do" rules the best means to denote integrity constraints? I think not. The semantics of integrity constraints are best expressed by a declarative statement of the relationship involved, rather than with a procedural rule that says how the relationship might be preserved. Further, integrity constraints often involve non-directed relationship between entities, and "if-do" rules are one-way. For example, to support a referential integrity constraint between collection `A` and collection `B`, one has to write two such rules, one to cover insertions to `A` and another to handle deletions from `B`.

Expressing constraints declaratively and automatically deriving rules to enforce them is of course one approach to supporting integrity constraints, but not the only one. Proving that transactions preserve constraints or automatic modification of methods to enforce constraints may be more efficient and controllable approaches. Another possibility is an exception and handler mechanism in the DML. Yet another approach are constraint programming techniques, such as as the satisfaction methods of ThingLab.

3.1.3 Event Sequencing

Again, while "if-do" rules might be an implementation possibility for automatic initiation of database events (sometimes called *active databases*), I think the description of event dependencies should be at a higher level: state machines, task decompositions, temporal constraints. Single rules are just too low a level at which to understand the semantics of event ordering. As before, "if-do" rules might be useful implementation technology, but I am unconvinced they should be visible at the data model level. (An interesting historical note is that "if-do" rules were in the 1969 CODASYL DBTG document, but were gone by the 1974 version.)

3.2 Propositions of Tenet 1

3.2.1 Proposition 1.1: A third-generation DBMS must have a rich type system

Following the discussion in Section 2.1 on type extensibility, I would rephrase this proposition as "A third-generation DBMS must support manifest types." The list of desirable features for a type definition system given in the 3GM are all probably ones you'd want to support manifest types, particularly a rich and freely composible set of constructors for structuring the representations of types. But the list falls short of providing manifest types. In particular, abstraction is required only for base types and not for all types. Why isn't abstraction a type former in its own right? There is also nothing in the list that would guarantee first-class or immediate types. The bottom line is whether databases are going to remain with structural semantics forever, or go with more sophisticated type definition systems.

Other comments

- I'd like to see an "object reference" type former in the list.
- **Line** is given as an example of a base type that one might want to add. But many applications using lines would not want to treat a line as an indecomposable scalar value. Consider a MacDraw-style application where one might represent the situation that two lines have been linked by having them share a common endpoint object.

- One reason that manifest types are nice is that you can use them to produce new type constructors. Thus, if you want a `queue`-constructor, a programmer can define a `Queue[T]` type, rather than having to wait around for the vendor to supply a `queue` constructor.
- The 3GM cites several references to show that some of the desirable features have already been proposed as extensions to or implemented as extensions of relational systems. A couple comments. To my recollection, the method given in [STON83] for adding sequences of records to a relational system works for one level of sequence constructor, but wouldn't work well for deeper compositions, such as a sequence of sequences, a set of sequences or a sequence of sets. The references on including an ADT system only concern new base types, not new manifest types.

3.2.2 Proposition 1.2: Inheritance is a good idea.

Depends on what's being inherited. I prefer to think in terms of hierarchies that a database system might support. There are several possibilities, such as a type hierarchy, where interface is inherited; an implementation hierarchy, where representations and methods are inherited; and a subset hierarchy, where membership of elements is inherited. These different hierarchies might have different requirements as regards multiple inheritance or strictness.

The discussion for this proposition continues to confuse types and collections, thus mixing up type and subset hierarchies. I might indeed want to enforce subset relationships between two collections that each contain instances of many types. It is also not obvious that I always want to regard *all* instances of a subtype as instances of a supertype.

In the example of Figure 1, would it be enough if an object could belong to both the `Employee` and the `Student` collections? With multiple collection memberships, is multiple inheritance necessary in the subset hierarchy?

3.2.3 Proposition 1.3: Functions, including database procedures and methods, and encapsulation are a good idea.

That should be "are good ideas." Here we see the type-collection confusion again, with functions being associated with a collection, rather than directly with instances of a type. If the `raise-sal` method is associated with `Employee` instances, then an `Employee` object can participate in several collections while retaining uniform behavior. (Of course, methods can be associated with collections as well.) There is also an assumption made that the envelope for encapsulation is the collection, rather than the single instance.

I disagree that functions or methods should only be written in a separate high-level language (HLL) with embedded DBMS calls. Why throw the impedance mismatch in the programmer's face? Why not give the DML enough power to that it suffices for writing methods. (It can still contain an associative access sublanguage.) With that approach, the method compiler can do much more in the way of type checking them. In addition, type implementors do not become dependent on having a compiler beyond what's in the database, for a particular HLL. Thus type implementations are portable to other instantiations of the database in other computing environments. Note also that if methods are written in a separate HLL, the database query optimizer understands little about what those methods do and will be very limited in the optimizations it can do.

If functions or methods are written in the DML, then "navigational" access is not such a problem, as the DML compiler can expand methods inline in other methods, to give large expressions with more opportunities for optimization.

The discussion that follows on *opaque* and *transparent* types is misguided in a couple respects, insisting that the structure of data elements must be visible to have efficient execution. First off, the particular problem in the example comes from assuming that `EMPLOYEES` are encapsulated at the level of collections rather than individual instances. Second, nothing says that what is encapsulated to clients of a type (such as an application program or another type definition) need be hidden from a trusted system component, such as the query optimizer. Third, transparent types let clients of the type create data dependencies on the internal representation of instances, making modifications to type implementations hard to insulate. In the example, if the `EMPLOYEE` type were transparent, the application programs could become dependent on the fact that the salary field is explicitly stored. This dependency creates a mess when one decides to change the representation to store the monthly salary and calculate the yearly salary from that.

Other comments:

- Reading towards the end of the section, the point is raised that some current relational systems can store procedures written in a HLL with embedded database calls, so modulo a little fussing with inheritance, they already satisfy this proposition. This discussion gives me the feeling that the goals of the 3GM are being set on the basis of what relational extensions might achieve rather than what is actually desired by users.

- This section indicates that it is useful for an application to be able to call a function on a data item. That implies to me that a function call is a good unit of communication between an application and the database, which contradicts later claims that such communication should always be via a query language statement.

3.2.4 Proposition 1.4: Unique Identifiers (UIDs) for records should be assigned by the DBMS only if a user-defined primary key is not available.

I have huge problems with this proposition. A key is a property of a collection, stating that—within the collection—elements are distinguishable by some part of their states. A UID, on the other hand, uniquely identifies an object in any context. Here again I see the "each instance belongs to exactly one collection" thinking. An entity in the real world can be a member of many collections. What might distinguish it in one collection might not distinguish it in another. For example, model numbers might serve to differentiate the products of one company, in its sales catalog. They would not necessarily distinguish products uniquely on an invoice issued by a wholesaler who carried products from many companies. The identity of an entity shouldn't depend on what collections it currently participates in.

The first paragraph of discussion is almost its own counterargument, talking of a primary key "that is known to *never* change." Never is a long time. It is hard to predict how collections of interest will evolve in an application. Room number might seem like a unique identifier for offices, until your company builds the second building. Companies merge—employee numbers may cease to be unique. Social security numbers may no longer be unique when your company goes multinational.

Actually, immutability of a property is not enough for it to be a key. There are also existence and one-to-one constraints. People sometimes end up with two social security numbers. What do you do when trying to enter an instance and the key value is as yet unknown or unassigned? Consider you have students keyed on student number, and you want to extend the database to keep track of people starting when they first apply (for first send for application material). Do I now have to assign student numbers to all applicants, even if they aren't admitted or don't accept?

Other comments:

- There is no reason that elements of views cannot have UIDs. If the view is a collection of existing objects, then they have UIDs already. If the view defines new objects, several schemes have been devised for generating UIDs, by people working on "object logics."

- Keys are useful, but in addition to UIDs, as an alternative for identifying an element in the context of a collection, or enforcing an integrity constraint. But having UIDs for global identification in addition gives more flexibility. An immutability requirement on keys might be too strict; uniqueness at a given point in time is sufficient. For example, an employee's number might change when he or she changes sites within a company. I've been told that Swiss social security numbers are unique, but a woman's number can change when she marries. You might want to insist that such a number is a key, even if it is not immutable. In other words, why enforce immutability in order to have a unique, human-readable handle on elements of a particular collection? A collection of customers might want two keys, an old account number and a new account number. Why should a key be restricted to local state? We might want items in a collection distinguishable on the basis of some other class of entities to which they are related. For example, an awards committee might have one member from each division, so we would want the division of each employee's department to be a key in that context.

3.2.5 Proposition 1.5: Rules (triggers, constraints) will become a major feature in future systems. They should not be associated with a specific function or collection.

The first problem here is that the proposition seems to equate rules, triggers and constraints, which are different beasts to me. (Also, if you just say "rules," you might not get the kind you wanted.) I've already discussed rules at length, so here I just address a couple points in the discussion of this proposition. First, there is the assertion that if rules were associated with methods or functions, then the *code* for a the rule has to be duplicated in many many methods. This assertion is nonsense, as the code can be put in one place and invoked by name. Vbase had method combination features that allowed such code sharing. This approach also addresses the later remark about being able to query rules or constraints. The bodies or declarations of rules could all be kept in one collection for querying. An alternative, of course, would be to have appropriate rule or trigger code automatically inserted by a compiler. Note that *enforcing* rules or constraints in methods is not incompatible with *declaring* them independently.

In regard to support for event sequencing, it might at first seem a good model feature if triggered actions are all defined centrally, and not associated with methods. "I want this action to happen whenever an update of this form happens, no matter how the update came to be made." I grant that this semantics is what you want in some cases, but in thinking through some examples, I thought of many cases where the action triggered (or whether it is triggered) should depend on the intent of an update. For example, a newspaper might have a "change fee" if an advertiser wants to modify the copy in ad after it has been accepted. However, if an advertising editor orders a change (say during copy editing, to change the layout slightly), there is no charge. Two updates, one of each kind, might have exactly the same effect on the text of an ad object, but only one should trigger the action that adds a charge to a customer account. This case can be handled by having two methods, one for customer changes and one for editorial changes, with only one triggering another action.

Another reason for sometimes associating event triggering with methods is that some events might need to be triggered by actions that cause no updates. Suppose that you want certain bank account statements to be numbered consecutively for each customer. Then, after the operation of generating a monthly statement, which only reads database state, you want to initiate an event that increments the "last statement number" information with the account. Such a linkage is easy to accomplish with a trigger associated with a method, but hard to do with a trigger on changes to object state.

Which brings us to the Joe and Sam example. This example scares me, because it illustrates how unwisely a programmer might use a rule mechanism, and points to the wisdom of declarative notations for integrity constraints and event sequencing. What is the real intent of "Whenever Joe gets a salary adjustment, propagate the change to Sam"? It seems to me that the intent is to keep Joe and Sam's salary the same. If that is the intent, then there should have been another rule to propagate changes in Sam's salary to Joe. (And probably another to deal with the case that the Joe item is created anew after the constraint has been defined.) Better to have a single declarative constraint that say the two salaries are equal. But take a step back. Why would someone want to keep two salaries the same in real life? Probably because both employees are in the same job category and tied to the same salary scale. This rule looks like a hack trying to fix up a botched schema design; it's treating the symptom and not the disease. The Sam and Joe data items ought to be referring to a common "salary category" item. I worry that if-do rules will get database users in a lot of trouble, especially if they are used with insufficient forethought. Someone might think naively that Sam and Joe are in the same salary category, so their salaries should be tied together (along with everyone else in that category). But what about merit increases? Should Sam get one because of Joe's good work?

Other comments:

- I don't see how rules can be independent of a collection in the types = collections model of the 3GM. How is one going to identify Joe or Sam if not by a key relative to some relation?
- The 3GM claims that some relational systems have rules or triggers already. I haven't seen much reported yet about how and how successfully these capabilities are being used. (But perhaps I'm not looking in the right places.)
- The discussion throughout seems to assume that rules or constraints are indicated at the schema level. In design applications, there are many uses for instance-level constraints. For example, "The layout of this functional unit in this VLSI design must fit within that bounding rectangle."
- I wonder if this obsession with rules might come from creative listening to customer desires. Customers want support for inferencing, or integrity constraints, or event sequencing; what the relational developer hears is "Oh, they want rules!" I repeat my opinion that if-do rules are one possible implementation mechanism, but that the requirement should be expressed at a higher level.

3.3 Tenet 2: Third generation DBMS's must subsume second generation DBMSs.

I agree, but with the caveat that "subsume" means "include the features of" rather than "be directly upward compatible." Looking back, second generation DBMSs certainly didn't subsume first generation systems in the latter sense. The main points here concern query languages and data independence.

The discussion of query languages starts by knocking down a strawman that nobody even tried to put in a cornfield, the strawman being the position that some applications never wish to run queries. CAD applications are used to push over the scarecrow. I don't know anyone in the OODB community who holds this opinion. The point is not whether an application will never require a certain kind of access, but that other kinds of access are more important. There is no point in providing one kind of access if a more important kind is not present. Where is the highest efficiency needed in CAD? Probably in the display of graphics, which involve traversals of networks of heterogeneous objects, rather than in getting back a table of the cost of components in a design. Yes, certainly CAD designers will want to query data associatively sometimes. The question in the short term is which is more tolerable: fast display with somewhat slower queries, or slower display rendering in exchange for fast associative access. (Not to say that I don't think it possible someday soon to give them both.)

True, OODB developers did not spend lots of time on their associative access initially, but probably rightly so. Good associative access wouldn't be worth much to their target markets if they couldn't deliver the speed in navigational access. Adding a query language didn't seem like the most pressing problem facing OODBs initially, although now companies are adding that functionality.

Next, I'd like to inject a little reality into the discussion. I think there is a "Myth of the Query Language," which is that relational query languages allowed end users to formulate ad hoc queries to answer their own questions. The problem is that answering a question is a matter of extracting the proper information and displaying it in an intelligible manner. Query languages gave little help with the latter. (For example, query languages return tuples from one relation, and the desired answer might need display of related tuples from several relations.) It wasn't until relational systems started adding report generators, graphing utilities and windowing systems that end users had much success with answering their own questions. And even with those tools, getting from a question to a query that involves three or more relations is still pretty daunting.

To me the essence of querying is to be able to describe regularity in data, and provide a language to exploit those regularities. SQL and its brethren really only exploit a very limited kind of regularity—sets of structurally homogeneous records. What if the regularity is in the operations, and not the structure? What if the top level structure is a matrix or a sequence, rather than a set. Current query languages return sets of homogeneous records. What if the client needs a group of related objects of different types? (It might not be acceptable to join the tuples together if they must be individually identifiable for update purposes.) Current query languages are also not very good when the query is against information encoded as structure rather than data values, such as looking for a consecutive pattern in a sequence.

Other comments:

- The authors of the 3GM say they have talked to many CAD application designers, and that all specify a query language as a necessity. Again, I wonder about "creative listening." Is perhaps the requirement actually that associative access be supported, independent of whether that access is through SQL or some other mechanism?
- I see no reason why OODBs can't have views. I've already seen a half-dozen reasonable proposals on how to add them.
- The reason query languages can be made efficient is that one can predict I/O accesses to bulk data from them, hence optimize and schedule those accesses. You don't need sets to be predictive. Knowing that one is going to traverse a graph structure or multiply two matrices gives foreknowledge of I/Os, and ought to be equally amenable to optimization and planning. Current query evaluation technology doesn't deal with those cases (although some vectorizing compilers do optimize I/O on array operations).
- There is no reason to believe that current calculus-based query languages are the last word. To me the essentials of a query language are that it be high-level (more "what" than "how"), efficient and generic. (The last term means that it works with data of new types as soon as those types are defined, without the need to create special display interfaces on them.) I can imagine systems besides a calculus query language that satisfy those requirements.
- OODBs seem to me to provide a high degree of data independence, because changes in *either* the logical or physical structure of the data can be masked by the message interface to an object.

3.4 Propositions of Tenet 2

3.4.1 Proposition 2.1: Essentially all programmatic access to a database should be through a non-procedural, high-level access language.

A lot of misperceptions in the discussion of this proposition. One is the failure to distinguish methods from application programs. Having an application program make a separate call to the database to traverse each reference between records is one thing. Having methods that make such accesses is another. The database has knowledge of methods and can combine them into bigger chunks for optimization and evaluation. I agree that having an application program issue a lot of low-level accesses to the database is not a good idea, but this is not what OODBers advocate. Application programs can communicate with the database using high-level message expressions, each which expresses many individual accesses against database objects. Sending messages can be high level and declarative.

Another point is that navigational notation doesn't mean naive evaluation. Making that assertion is like saying relational query languages must be inefficient because their semantics involves cross products. (And I'm sure that claim was probably made in the early days of relational systems.) Writing navigational expressions in a method is not a terrible thing. If the method gets called directly from an application, then a query language probably couldn't do any better on a simple access. If the method is called by another that applies it over a collection, then the access can be optimized and take advantage of indexes and other auxiliary access paths. The order that data is accessed need not be the one that results from naive invocation of the methods involved. Navigational access is a notation—one that SQL isn't very good at expressing (although I expect later versions of SQL will have path notation).

When access is navigational and *not* predictable—say traversing a structure based on some complicated search function—it is better to handle it by having methods call each other withing the database process rather than by an application program issuing droves of small queries to the database.

Other comments:

- Having the user always specify the data he or she is interested in via SQL says he or she is only allowed to be interested in homogeneous sets.

- It has been advocated that queries as a data type is a better way to express linking than reference. But consider that a reference is guaranteed to lead to exactly one item, while a query can lead to 0 or more.

- I detect some underlying assumptions. One is that object reference must be implemented by pointers. With encapsulated state, a reference could be captured as a pointer, or it could be represented via access to some relationship. I agree that two-way relationships are wanted sometimes, but that doesn't mean that a reference need only be traversed in the direction it is defined. Some OODBs support automatic maintenance of inverses of one-way relationships. Another assumption seems to be that pointers can only be processed by immediate pointer following. Having pointers doesn't preclude a join that collects all pointers from many objects and reorders them before accessing them in secondary storage.

3.4.2 Proposition 2.2: There should be at least two ways to specify collections, one by using enumeration of members and one using the query language to specify membership.

This discussion muddles together a number of concepts, two of which are specification and implementation. Whether or not a collection is independent or is derived from another collection is a modeling issue. It depends on the semantics of the application. In any application, some sets will have to be explicit enumerations. The base relations in a relational database, for example, are going to be explicit enumerations. In the example offered, **ALUMNI** will have to be represented as an enumeration. "Old guard" is a derived set, but the alumni travel club will be another enumerated set.

How to implement independent and derived sets is another matter. A good DBMS should provide choices. One way to implement a derived set is by storing an expression to compute its extension (and OODBs can certainly cope with such a representation). Another is to store an explicit list of elements with support for propagating updates between it and the base collection. The latter has the advantage that it allows either the base set or the derived set to be updated. The extensional representation can be fast for certain classes of queries, such those involving intersection of two collections. One can also use expressions or explicit lists for independent collections, although expressions get clunky for large sets, as the size of the expression will grow with the size of the set. There is also a possibility that a derived set need not be a separate named entity. Its existence might be captured by a method attached to another collection. For example, the **ALUMNI** collection might understand the `oldGuard` message.

Another concept that is being mixed in here is that of constraints between collections. The discussion ignores the fact that two collections can be related without one being derivable from another. The collection of teaching assistants might be constrained to be a subset of the set of fulltime graduate students. That doesn't mean that the teaching assistant collection can be derived from the graduate student collection. Both can change independently, as long as the subset constraint is preserved.

Other comments:

- It makes sense to me that a DBMS might support collections that are a mixture of independent and derived components. Consider the collections of books in different branch libraries on campus. Probably the division of books between branches can mostly be derived for the master collection by selection on call number, but there might be some books that need to be explicitly listed, such as reference volumes available in all branches, and some books that are in the rare books collection rather than in the topical libraries.

- There needs to be a better vehicle than SQL for derived collections of heterogeneous elements.

- Independent vs. derived sets is not exactly the same thing as manual vs. automatic set membership in CODASYL. But in an OODB it is certainly possible to arrange that new instances of a type are automatically added to some enumerated collection on creation.
- There are several possibilities for how to represent an explicit enumeration. It might be as a list of values, or as a list of UIDs, or as a list of logical keys, each of which would be most efficient under certain conditions. For example, one might want a list of values when the collection is actually part of the private state of some object. If I represent a polygon by a set of points, it makes sense to store that set as a list of coordinate pairs directly, rather than to make them first-class objects and refer to them by UID or key value.
- The example is given that if explicit enumerations are used for collections derived from `ALUMNI`, then every time an application programmer inserts an element into `ALUMNI`, he or she must manually update all the derived collections. This scenario is nonsense. Methods on the `ALUMNI` object can propagate these updates automatically.
- Footnote 2 at the beginning of the paper gives a very limited (and relational) vie of collections by saying they are named sets of homogeneous records (i.e., they are relations). I take a broader view that collections can be any bulk type with member elements, possibly of heterogeneous types and which might or might not be explicitly named.

3.4.3 Proposition 2.3: Updatable views are essential.

OODBs give you some nice mechanisms to support updatable views. They can provide views that select a subset of objects from a collection, or that provide a different protocol to instances of a type. By keeping track of the identity of the original elements, it helps in implementing view updates. Such views can be used to preserve an application interface when the implementation of elements of a collection changes.

Views can also be defined with virtual objects as members. Just because the members have no physical existence, it doesn't mean they can't be provided with UIDs. I've seen several proposals for deriving UIDs for derived objects based on the UIDs of the object or objects from which the virtual object was created.

3.4.4 Proposition 2.4: Performance indicators have almost nothing to do with data models and must not appear in them.

True, but complex object structure and logical groupings of objects can be part of the semantics of the data and should be expressible in the model. Such information affects the meaning of operations such as archiving, concurrency control, copying, encapsulation, authorization, recovery, propagation of updates and versioning. A database that isn't given such semantic information in its model can't enforce it and can't use it to optimize queries and physical placement. It's very useful to know that a given record is a part of the private state of an object, and not reachable except through the object, compared to knowing that the record is just a tuple in a relation that might or might not be referenced from several places. Complex object structure also tells you something about expected access patterns to data. It's useful to visualize a database as a parking garage. We can imagine a parking garage where cars are disassembled when they arrive, and all tires put on one floor, all engines blocks on another, and so forth. This organization is great if I want to know what percentage of people drive on Michelins or I want to wash all the windshields quickly. But if customers mostly want to manipulate individual cars in their entirety, it is useful to reflect that view in my enterprise.

3.5 Tenet 3: Third generation DBMSs must be open to other subsystems.

Certainly, but what's wrong with the database having a single language to write methods, with a well-integrated declarative component, and invoking those methods from multiple HLLs?

3.6 Propositions of Tenet 3

3.6.1 Proposition 3.1: Third generations DBMSs must be accessible from multiple HLLs.

Here I see the assumption again that the application language must be the same as the method language. Certainly when the two are the same (giving a persistent programming language), a very nice programming system results. But one can also have a system where database types are implemented using methods written in one language, and instances of those types are manipulated via messages sent from an application written in another language. Nobody seems to complain that relational databases typically have only one language for data manipulation. I don't see the problem then in having one language for methods, as long as there is a mechanism for applications in other languages to invoke those methods. I'm not convinced that an embedded query language is necessarily superior to, say, a message-passing syntax. What seems important to me is that a large unit of work be transmitted to the database in one interaction. A single message send can invoke just as much processing as a query.

Other comments:

- The fact that an application, such as LOTUS 1-2-3, is coded in a particular language doesn't mean that methods have to be written in the same language. My Prolog programs do just fine talking to a file system written in C.

- Are particular programming languages really the issue here anyway? Twenty years from now almost no one will write HLL code directly. It will be written by other programs that generate if from specifications, application generation tools, or compilers of more abstract representations of program actions. If that is the case, what difference does it make what language those programs spit out?

3.6.2 Proposition 3.2: Persistent X for a variety of X's is a good idea. They will all be supported on top of a single DBMS by compiler extensions and a (more or less) complex run time system.

Maybe. But even if there is a single underlying data subsystem, it is not clear to me that the best implementation of persistent X is to keep a cache of objects in the application's address space. The object cache and its run time has a lot of responsibility for matching up X types and database types, and for managing the movement of objects. Should this cache be something outside the database, or should it be part of the database buffer space, but that resides at the application site? (See Proposition 3.4 for more on the point that the workstation-server boundary need not be the application-database boundary.) There is a lot to be said for trying to make as much as possible of the runtime for the persistent language an integral part of the DBMS. First off, all the program caches for different X's are going to have much of their functionality in common. Why duplicate this functionality in N runtimes for N different languages (particulary when some of it might already be implemented in the database itself)? There is a big maintenance problem here—changes or extensions to the database interface require modifications to N runties and object caches. There is also the claim that the run time must implement the types from language X that don't map directly to database types. It seems much better to provide these types via manifest-type-definition facilities within the database. That way the database can know something about those types for query optimization. The example of a data element being updated 100 times is offered as evidence that there must be an object cache in the application process space, otherwise there will be 100 calls to the database. It seems likely to me that those 100 updates were part of a small number of higher-level actions. Those higher-level actions can be expressed as methods, and each invoked with a single call to the database.

3.6.3 Proposition 3.3: For better or worse, SQL is intergalactic dataspeak.

Are we talking a standard for machines or people here? More and more, SQL is being generated by programs, not people, via graphical and form interfaces, 4GLs and application generators. So though it may end up being a standard for system interoperability, that doesn't mean it will remain a standard for programmers or end users. Furthermore, SQL is a changing standard. SQL3 might be an object-oriented language.

Other comments:

- If SQL eventually does become something manipulated only by programs, then it should probably be replaced by a more structured `Query` datatype, with operations for constructing and modifying queries, rather than a string representation.
- The 3GM notes that to support persistent X, the database will have to support persistent variables for X. Note that this requirement does not mesh well with the current relational practice of only relations having persistent names.
- There is a claim made that lack of an SQL interface was the downfall of some early OODB products. I know of only one OODB product that was taken off the market, Vbase. There seems to be a fair amount of agreement that the main problem was a non-standard object extension to C. The same company is still in business with a follow-on product, Ontos, that uses C++.

3.6.4 Proposition 3.4: Queries and their resulting answers should be the lowest of communication between a client and a server.

This might be a moot point. The authors advocate having stored procedures or functions in the database. If new version s of SQL can call those functions, then a call to a single function could be a legal SQL query. On the other side, the interface to many OODBs from an application includes the ability to send an arbitrary expression over the database types to the database for evaluation. Such an expression could easily express an associative query. So the two paradigms might not end up being that different.

Be that as it may, the real problem with the discussion here is that it confuses the application-database interface with the workstation-server boundary. The two need not be the same. In particular, part of the DBMS can be executing on the workstation, to manage local buffers and optimize queries, for example. In a local network of personal workstations and database server machines, even if the servers have more powerful processors, the bulk of the available cycles will be on the workstations. It makes sense to do as much database processing on them as possible, leaving the servers free to work on just the tasks that can best be done centrally.

This confusion is carried over into mixing the logical result of a database request with physically how the data in conveyed across a network. Say a request returns a selection from a relation. The server could extract the tuples from that relation, package them into messages and send them across the net. But if the workstation maintains database buffers, why not leave the data in its disk format on the server? Send across the relevant pages intact and let the workstation extract the right information from those pages. Deppisch and Obermeit ["Tight database cooperation in a Server-Workstation Environment," 7th Intl. Conference on Distributed Computing Systems, IEEE, 1987] describe an implementation that moves page images sometimes, even though requests are formulated as high-level queries.

Finally, the 3GM offers the Hagmann and Ferrari paper as evidence that any internal interface at a level lower than SQL will be inefficient. I point out that their work is aimed at determining the best partitioning between a shared central processor and a dedicated backend database machine (though they do mention workstation-server architectures briefly). So, for example, where combined CPU time might be a good metric for a pair of central machines, its utility is not obvious in a network with many inexpensive workstations. Secondly, their benchmark was chosen to reflect an data processing workload, not access to, say, design databases. Finally, those authors did not experiment with the sensitivity of their results to physical aspects of the data (such as clustering) or to relative buffer sizes between the two machines.

Other comments:

- One problem with SQL as it is now constituted is that it only returns sets. That is not always the most appropriate semantic unit of transfer for an application. A CASE system might want a syntax tree as the result of a query. Scientific applications often want matrices or sequences as results.
- *If* Persistent X is going to use an application cache, are SQL queries the best way to populate it? Perhaps transitive closures of single objects are a better idea.
- It may be that for inter-DBMS communication, requests and interchange of data must be value-based and of restricted form. But that requirement shouldn't limit the internal interfaces of a single DBMS.
- If database functions are a useful mechanism for application-database communication, then maybe query languages should be functional rather than based on predicate logic.
- Under most implementations, embedded SQL ends up being a procedure-call mechanism anyway, after preprocessing.
- Perhaps the point being argued against here is a get-object/get-field interface from the application to the database. I agree that such an interface is not a good idea. But I see the natural interface to an OODB as methods requests, which are a higher level than single-object and get-field access.

3.7 Tenet 4: Third generation DBMSs should be simple, formally defined and clean.

This tenet doesn't actually appear in the 3GM. I have my doubts that it could be satisfied by DBMSs that are relational systems with bags and buckets wired on. I don't think that a DBMS that attempts to provide all the features in the 3GM without taking into account advances in type theory and programming language design can possibly hope to satisfy this tenet.

4 Parting Shot and a Warning

In the summary, the authors of the 3GM say that 20 years of history show that query language access to databases is the only way to go. But for 20 years relational systems haven't supported CAD or multimedia applications. Object-oriented programming is here to stay—why make application programmers deal with multiple paradigms? Applications want to deal with objects, not structural access to particular representations.

"Object-oriented" is a current synonym for "good" in sales literature. Relational vendors are starting to advertise their products as object-oriented or as supporting object-oriented programming. In most cases the conversion to object-orientation was carried out more by marketing departments than by engineering groups. (It's faster and cheaper that way.) Don't let them fool you—BLOBS do not an OODB make.

5 Acknowledgements

Some of this was written while I was visiting at GIP Altaïr.