What Goes Around Comes Around

By

Michael Stonebraker
Joey Hellerstein

Abstract

This paper provides a summary of 35 years of data model proposals, grouped into 9 different eras. We discuss the proposals of each era, and show that there are only a few basic data modeling ideas, and most have been around a long time. Later proposals inevitably bear a strong resemblance to certain earlier proposals. Hence, it is a worthwhile exercise to study previous proposals.

In addition, we present the lessons learned from the exploration of the proposals in each era.   Most current researchers were not around for many of the previous eras, and have limited (if any) understanding of what was previously learned. There is an old adage that he who does not understand history is condemned to repeat it. By presenting "ancient history", we hope to allow future researchers to avoid replaying history.

Unfortunately, the main proposal in the current XML era bears a striking resemblance to the CODASYL proposal from the early 1970's, which failed because of its complexity. Hence, the current era is replaying history, and "what goes around comes around". Hopefully the next era will be smarter.

I Introduction

Data model proposals have been around since the late 1960's, when the first author "came on the scene". Proposals have continued with surprising regularity for the intervening 35 years. Moreover, many of the current day proposals have come from researchers too young to have learned from the discussion of earlier ones. Hence, the purpose of this paper is to summarize 35 years worth of "progress" and point out what should be learned from this lengthy exercise.

We present data model proposals in nine historical epochs:

Hierarchical (IMS): late 1960's and 1970's
Directed graph (CODASYL): 1970's
Relational: 1970's and early 1980's
Entity-Relationship: 1970's
Extended Relational: 1980's
Semantic: late 1970's and 1980's

Object-oriented: late 1980's and early 1990's
Object-relational: late 1980's and early 1990's
Semi-structured (XML): late 1990's to the present

In each case, we discuss the data model and associated query language, using a neutral notation. Hence, we will spare the reader the idiosyncratic details of the various proposals. We will also attempt to use a uniform collection of terms, again in an attempt to limit the confusion that might otherwise occur.

Throughout much of the paper, we will use the standard example of suppliers and parts, from [CODD70], which we write for now in relational form in Figure 1.

Supplier (sno, sname, scity, sstate)
Part (pno, pname, psize, pcolor)
Supply (sno, pno, qty, price)

A Relational Schema
Figure 1

Here we have Supplier information, Part information and the Supply relationship to indicate the terms under which a supplier can supply a part.

Figure 2 shows a few instances of sample data.

Suppplier   Part

16 General Supply Boston Ma   27 Power saw   7   silver
24 Special Supply Detroit Mi   42 bolts   12   gray

Supply

16 27   100 $20.00
16 42   1000 $.10
24 42   5000 $.08
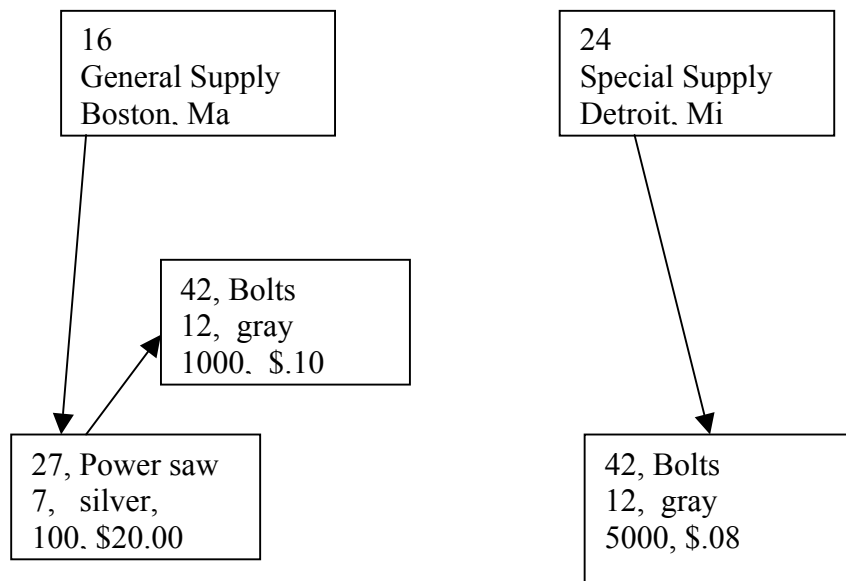
Some Sample Data
Figure 2

II IMS Era

IMS was released around 1968, and initially had a hierarchical data model. It understood the notion of a **record type,** which is a collection of named fields with their associated data types. Each **instance** of a record type is forced to obey the data description indicated in the definition of the record type. Furthermore, some subset of the named fields must uniquely specify a record instance, i.e. they are required to be a **key**. Lastly, the record types must be arranged in a **tree**, such that each record type (other than the

root) has a unique **parent** record type. An IMS data base is a collection of instances of record types, such that each instance, other than root instances, has a single parent of the correct record type.

This requirement of tree-structured data presents a challenge for our sample data, because we are forced to structure it in one of the two ways indicated in Figure 3. For the first of the two schemas, we also indicate our sample data in Figure 4.
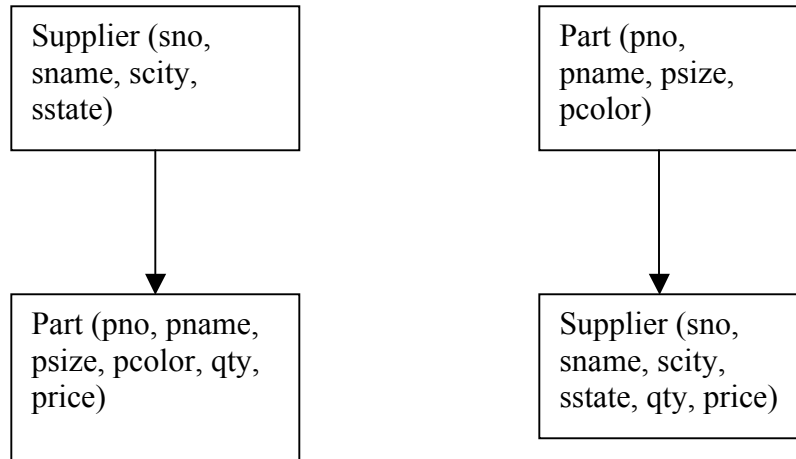
Two Hierarchical Organizations
Figure 3

Some Example Data
Figure 4

These representations share two common undesirable properties:

1) **Information is repeated**. In the first schema, Part information is repeated for each Supplier who supplies the part. In the second schema, Supplier information is repeated for each part he supplies. Repeated information is undesirable, because it offers the possibility for inconsistent data. For example, a repeated data element could be changed in some, but not all, of the places it appears, leading to an inconsistent data base.

2) **Existence depends on parents**. In the first schema it is impossible for there to be a part that is not currently supplied by anybody. In the second schema, it is impossible to have a supplier which does not currently supply anything. There is no support for these "corner cases" in a strict hierarchy.

IMS chose a hierarchical data base because it facilitates a simple data manipulation language, DL/1. Every record in an IMS data base has a **hierarchical sequence key** (HSK). Basically, an HSK is derived by concatenating the keys of ancestor records, and then adding the key of the current record. HSK defines a natural order of all records in an IMS data base, basically depth-first, left-to-right. DL/1 intimately used HSK order for the semantics of commands. For example, the "get next" command returns the next record in HSK order. Another use of HSK order is the "get next within parent" command, which explores the subtree underneath a given record in HSK order.

Using the first schema, one can find all the red parts supplied by Supplier 16 as:

```
Get unique Supplier (sno = 16)
Until no-more {
        Get next within parent (color = red)
}
```

The first command finds Supplier 16. Then we iterate through the subtree underneath this record in HSK order, looking for red parts. When the subtree is exhausted, an error is returned.

Notice that DL/1 is a "record-at-a-time" language, whereby the programmer constructs an algorithm for solving his query, and then IMS executes this algorithm. Often there are multiple ways to solve a query. Here is another way to solve the above specification:

```
Until no-more {
        Get next Part (color = red)
}
```

Although one might think that the second solution is clearly inferior to the first one; in fact if there is only one supplier in the data base (number 16), the second solution will outperform the first. The DL/1 programmer must make such optimization tradeoffs.

In addition, the IMS programmer must keep track of the two currency indicators (current record and current parent). This is made especially complicated because of the "corner cases". For example, where does the parent pointer go, if the parent is deleted by an update operation? Managing the currency indicators requires substantial error logic that is complex and error prone. Jim Gray reported that an IMS user required an average of 17 test runs per DL/1 statement to successfully do Q/A on IMS programs. Obviously, DL/1 programming was a complex, and IMS programmers make more money than other kinds of programmers!

IMS supported four different storage formats for hierarchical data. Basically root records can either be:

Stored sequentially
Indexed in a B-tree using the key of the record
Hashed using the key of the record

Dependent records are found from the root using either

Physical sequentially
Various forms of pointers.

Some of the storage organizations impose restrictions on DL/1 commands. For example the purely sequential organization will not support record inserts. Hence, it is appropriate only for batch processing environments in which a change list is sorted in HSK order and then a single pass of the data base is made, the changes inserted in the correct place, and a new data base written. This is usually referred to as "old-master-new-master" processing. In addition, the storage organization that hashes root records on a key cannot support "get next", because it has no easy way to return hashed records in HSK order.

These various "quirks" in IMS are designed to avoid operations that would have impossibly bad performance. However, this decision comes at a price: One cannot freely change IMS storage organizations to tune a data base application because there is no guarantee that the DL/1 programs will continue to run.

The ability of a data base application to continue to run, regardless of what tuning is performed at the physical level will be called **physical data independence**. Physical data independence is important because a DBMS application is not typically written all at once. As new programs are added to an application, the tuning demands may change, and better DBMS performance could be achieved by changing the storage organization. IMS has chosen to limit the amount of physical data independence that is possible.

In addition, the logical requirements of an application may change over time. New record types may be added, because of new business requirements or because of new government requirements. It may also be desirable to move certain data elements from one record type to another. IMS supports a certain level of **logical data independence**, because DL/1 is actually defined on a **logical data base**, not on the actual physical data base that is stored. Hence, a DL/1 program can be written initially by defining the logical data base to be exactly same as the physical data base. Later, record types can be added to the physical data base, and the logical data base redefined to exclude them. Hence, an IMS data base can grow with new record types, and the initial DL/1 program will continue to operate correctly. In general, an IMS logical data base can be a subtree of a physical data base.

It is an excellent idea to have the programmer interact with a logical abstraction of the data, because this allows the physical organization to change, without compromising the runability of DL/1 programs. Logical and physical data independence are important because DBMS application have a much longer lifetime (often a quarter century or more) than the data on which they operate. Data independence will allow the data to change without requiring costly program maintenance.

One last point should be made about IMS. Clearly, our sample data is not amenable to a tree structured representation as noted earlier. Hence, there was quickly pressure on IMS to represent our sample data without the redundancy or dependencies mentioned above. IMS responded by extending the notion of logical data bases beyond what was just described.

Supplier (sno, sname, scity, sstate)

Part (pno, pname, psize, pcolor)

Supply (pno, qty, price)
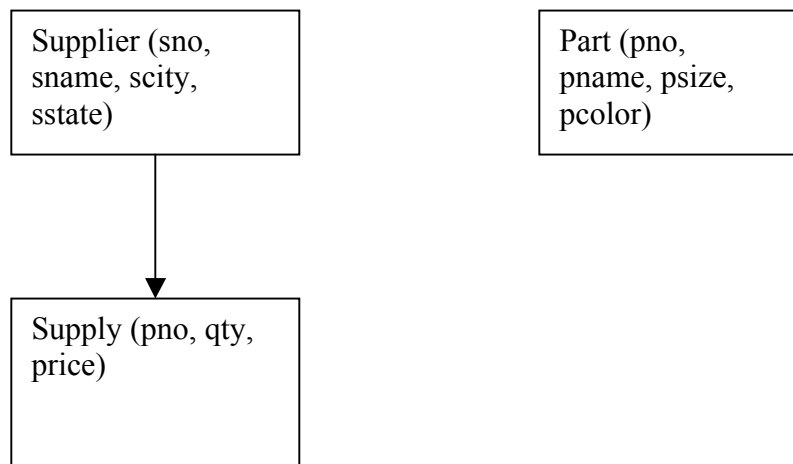
Two IMS Physical Data Bases
Figure 5

Suppose one constructs two physical data bases, one containing only Part information and the second containing Supplier and Supply information as shown in the diagram of Figure 5. Of course, DL/1 programs are defined on trees; hence they cannot be used directly on the structures of Figure 5. Instead, IMS allowed the definition of the logical data base shown in Figure 6. Here, the Supply and Part record types from two different data bases are "fused" (joined) on the common value of part number into the hierarchical structure shown.

Basically, the structure of Figure 5 is actually stored, and one can note that there is no redundancy and no bad existence dependencies in this structure. The programmer is presented with the hierarchical view shown in Figure 6, which supports standard DL/1 programs.
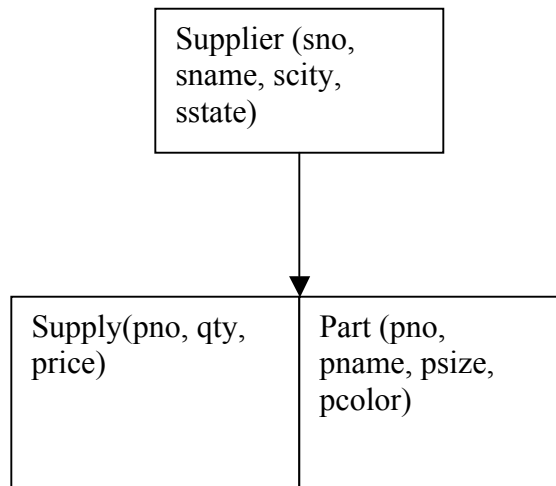
An IMS Logical Data Base
Figure 6

Speaking generally, IMS allow two different tree-structured physical data bases to be "grafted" together into a logical data base. There are many restrictions (for example in the use of the delete command) and considerable complexity to this use of logical data bases, but it is a way to represent non-tree structured data in IMS.

This complexity comes from two sources. First, the user is manipulating a "view" of the data, and updates must be mapped to a different structure. Support for views is hard even in relational data bases, and the added complexity of hierarchical structures makes it that much more complex. Second, the currency indicators are defined on the view, and must be mapped to other currency indicators on the real data bases. This mapping is especially complex.

The complexity of these logical data bases will be presently seen to be pivotal in determining how IBM decided to support relational data bases a decade later.

We will summarize the lessons learned so far, and then turn to the CODASYL proposal.

Lesson 1: Physical and logical data independence are highly desirable

Lesson 2: Tree structured data models are very restrictive

Lesson 3: It is a challenge to provide sophisticated logical reorganizations of tree structured data

Lesson 4: A record-at-a-time user interface forces the programmer to do manual query optimization, and this is often hard.

III CODASYL Era

In 1969 the CODASYL (Committee on Data Systems Languages) committee released their first report [CODA69], and then followed in 1971 [CODA71] and 1973 [CODA73] with language specifications. CODASYL was an ad-hoc committee that championed a directed graph data model along with a record-at-a-time data manipulation language.
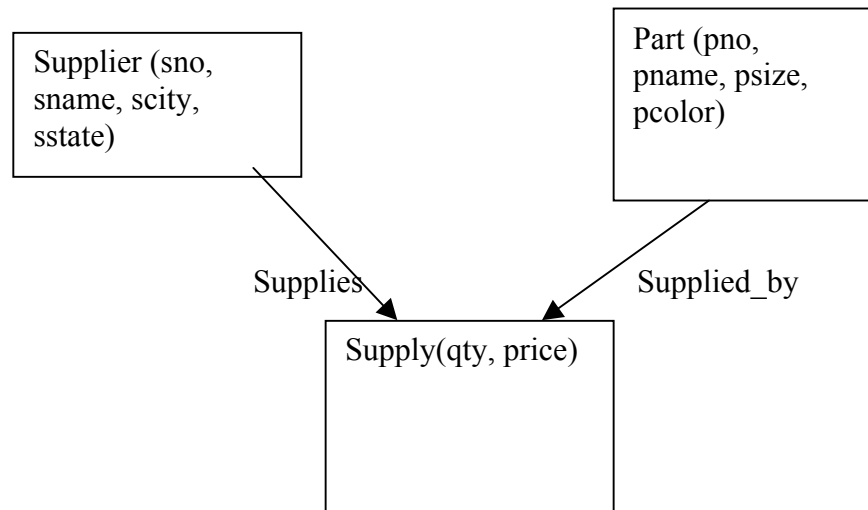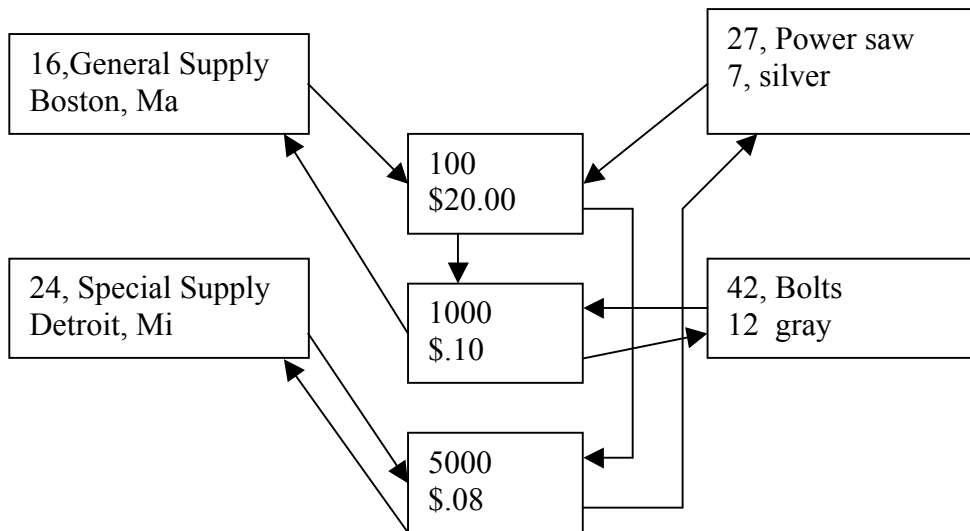

A CODASYL Directed Graph
Figure 7

This model organized a collection of record types, each with keys, into a directed graph, rather than a tree. Hence, a given record instance can have multiple parents, rather than a

single one, as in IMS. As a result, our Supplier-Parts-Supply example could be represented by the CODASYL schema of Figure 7.

Here, we notice three record types arranged in a directed graph, connected by two named arcs, called Supplies and Supplied_by. A named arc is called a **set type** in CODASYL, though it does not technically describe a set at all. Rather it indicates that for each record instance of the **owner** record type (the tail of the arrow) there is a relationship with zero or more record instances of the **child** record type (the head of the arrow). As such, it is a 1-to-n relationship between owner record instances and child record instances. For each parent record, there is a **set instance** (or **set** for short) of the set type. In the set are the parent record and all the member records that relate to the parent.

Figure 8 shows some example data with set instances represented by linked lists.

Some Example Data
Figure 8

A CODASYL directed graph is a collection of named record types and named set types that form a connected directed graph. A CODASYL data base is a collection of record instances and set instances that obey this direct graph-structured description.

Notice that Figure 7 does not have the existence dependencies present in a hierarchical data model. For example, it is ok to have a part that is not supplied by anybody. This will merely be an empty instance of a Supplied_by set. Hence, the move to a directed graph data model solves many of the restrictions of a hierarchy. However, there are still

situations that are hard to model in CODASYL.   Consider, for example, data about a marriage ceremony, which is a 3-way relationship between a bride, a groom, and a minister. Because CODASYL sets are only two-way relationships, one is forced into the data model indicated in Figure 9.
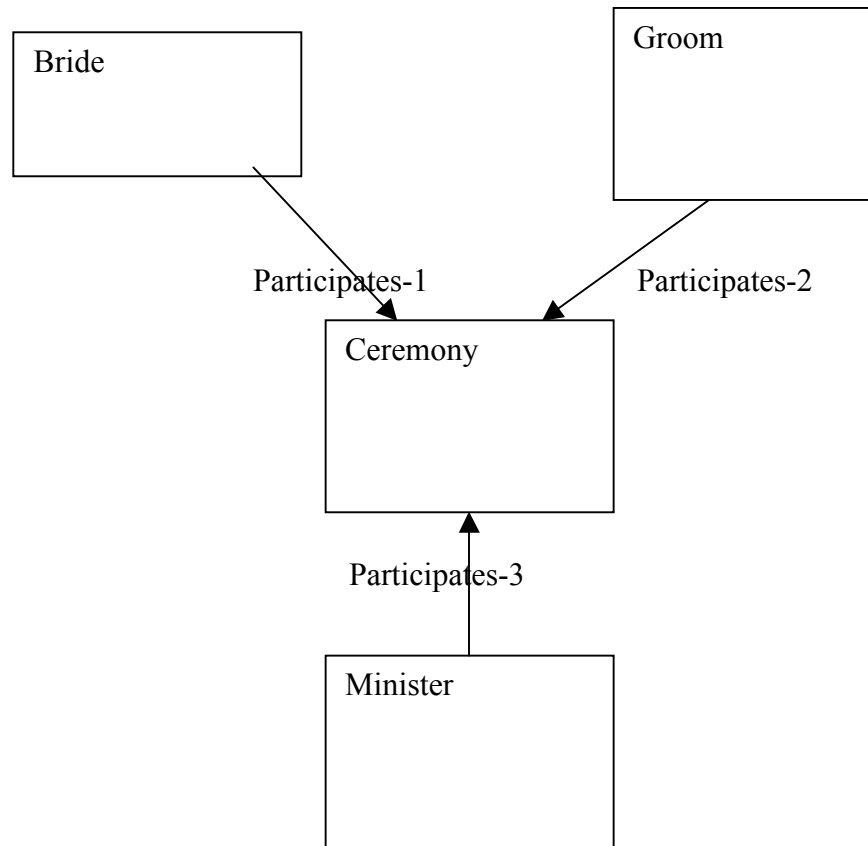
A CODASYL Solution
Figure 9

This solution requires three binary sets to express a three-way relationship, and is somewhat unnatural. Although much more flexible than IMS, the CODASYL data model still had limitations.

The CODASYL data manipulation language is a record-at-a-time language whereby one enters the data base at by selecting an instance of a record type and then navigates to desired data by following sets. To find the red parts supplied by Supplier 16 in CODASYL, one can use the following code:

```
Find Supplier (SNO = 16)
Until no-more {
        Find next Supply record in Supplies
```

```
            Find owner Part record in Supplied_by
            Get current record
            -check for red—
}
```

One enters the data base at supplier 16, and then iterates over the members of the Supplies set for this supplier. This will yield a collection of Supply records. For each one, the owner in the Supplied_by set is identified, and a check for redness performed. Ultimately, the Supplies set is exhausted, and the loop terminates.

The CODASYL proposal suggested that the records in each record type could either be hashed on the key in the record or stored clustered together with the owner record insome set. Several implementations of sets were proposed that entailed various combinations of pointers between the parent records and child records.

The CODASYL proposal provided essentially no physical data independence. For example, the above program fails if the key (and hence the hash storage) of the Supplier record is changed from sno to something else. With regard to logical data independence, CODASYL proposed the notion of a logical data base, which is constrained to be a subset of the record types and set types in a physical data base. Hence, record types and set types can be added to a CODASYL data base, and then the previous version of the schema defined as a logical data base. No attempt was made to support more complex transformations, such as those supported in IMS.

The move to a directed graph model has the advantage that no kludges are required to implement many-to-many relationships, such as in our example data. However, the CODASYL model is considerably more complex than the IMS data model. In IMS a programmer navigates in a hierarchical space, while a CODASYL programmer navigates in a multi-dimensional hyperspace. In IMS the programmer must only worry about his current position in the data base, and the position of a single ancestor (if he is doing a "get next within parent").

In contrast, a CODASYL programmer must keep track of the:

The last record touched by the application
The last record of each record type touched
The last record of each set type touched

The various CODASYL DML commands update these currency indicators. Hence, one can think of CODASYL programming as moving these currency indicators around a CODASYL data base until a record of interest is located. Then, it can be fetched. In addition, the CODASYL programmer can suppress currency movement if he desires. Hence, one way to think of a CODASYL programmer is that he should program looking at a wall map of the CODASYL directed graph that is decorated with various colored pins indicating currency. In his 1973 Turing Award lecture, Charlie Bachmann called this "navigating in hyperspace" [BACH73].

Hence, the CODASYL proposal trades increased complexity for the possibility of easily representing non-hierarchical data. CODASYL offers poorer logical and physical data independence than IMS.

There are also some more subtle issues with CODASYL. For example, in IMS each data base could be independently bulk-loaded from an external data source. However, in CODASYL, all the data was typically in one large network. This much larger object had to be bulk-loaded all at once, leading to very long load times. Also, if a CODASYL data base became corrupted, it was necessary to reload all of it from a dump. Hence, crash recovery tended to be more involved than if the data was divided into a collection of independent data bases.

In addition, a CODASYL load program tended to be complex because large numbers of records had to be assembled into sets, and this usually entailed many disk seeks. As such, it was usually important to think carefully about the load algorithm to optimize performance. Hence, there was no general purpose CODASYL load utility, and each installation had to write its own. This complexity was much less important in IMS.

Hence, the lessons learned in CODASYL were:

Lesson 5: Directed graphs are more flexible than hierarchies but more complex

Lesson 6: Loading and recovering directed graphs is more complex than hierarchies

IV Relational Era

Against this backdrop, Ted Codd proposed his relational model in 1970 [CODD70]. In a conversation with him years later, he indicated that the driver for his research was the fact that IMS programmers were spending large amounts of time doing maintenance on IMS applications, when logical or physical changes occurred. Hence, he was focused on providing better data independence.

His proposal was threefold:

Store the data in a simple data structure (tables)
Access it through a high level set-at-a-time DML
No need for a physical storage proposal

With a simple data structure, one has a better change of providing logical data independence. With a high level language, one can provide a high degree of physical data independence. Hence, there is no need to specify a storage proposal, as was required in both IMS and CODASYL.

A relational schema and example data for the Supplier-Parts-Supply example was shown in Figures 1 and 2. Our example query concerning the suppliers of red parts can be coded in SQL as follows:

Select S.sno
From Supplier S, Part P, Supply SS
Where S.sno = SS.sno and SS.pno = P.pno and P.pcolor = "red"

Moreover, the relational model has the added advantage that it is flexible enough to represent almost anything. Hence, the existence dependencies that plagued IMS can be easily handled by the relational schema shown earlier in Figure 1. In addition, the three-way marriage ceremony that was difficult in CODASYL is easily represented in the relational model as:

Ceremony (bride-id, groom-id, minister-id, other-data)

Codd made several (increasingly sophisticated) relational model proposals over the years, e.g: [CODD79]. Moreover, his early DML proposals were the relational calculus (data language/alpha) [CODD71a] and the relational algebra [CODD72a]. Since Codd was originally a mathematician (and previously worked on cellular automata), his DML proposals were rigorous and formal, but not necessarily easy for mere mortals to understand.

Codd's proposal immediately touched off "the great debate", which lasted for a good part of the 1970's. This debate raged at SIGMOD conferences (and it predecessor SIGFIDET). On the one side, there was Ted Codd and his "followers" (mostly researchers and academics) who argued the following points:

a) Nothing as complex as CODASYL can possibly be a good idea
b) CODASYL does not provide acceptable data independence
c) Record-at-a-time programming is too hard to optimize
d) CODASYL and IMS are not flexible enough to easily represent common situations (such as marriage ceremonies)

On the other side, there was Charlie Bachman and his "followers" (mostly DBMS practitioners) who argued the following:

a) COBOL programmers cannot possibly understand the new-fangled relational languages
b) It is impossible to implement the relational model efficiently
c) CODASYL can represent tables, so what's the big deal?

The highlight (or lowlight) of this discussion was an actual debate at SIGFIDET '74 between Codd and Bachman and their respective "seconds" [RUST74]. One of us was in the audience, and it was obvious that neither side articulated their position clearly. As a result, neither side was able to hear what the other side had to say.

In the next couple of years, the two camps modified their positions (more or less) as follows:

Relational advocates

a) Codd is a mathematician, and his languages are not the right ones. SQL [CHAM74] and QUEL [STON76] are much more user friendly.

b) System R [ASTR76] and INGRES [STON76] prove that efficient implementations of Codd's ideas are possible. Moreover, query optimizers can be built that are competitive with all but the best programmers at constructing query plans.

c) These systems prove that physical data independence is achievable. Moreover, relational views [STON75] offer vastly enhanced logical data independence, relative to CODASYL.

d) Set-at-a-time languages offer substantial programmer productivity improvements, relative to record-at-a-time languages.

CODASYL advocates

a) It is possible to specify set-at-a-time network languages, such as LSL [TSIC76], that provide complete physical data independence and the possibility of better logical data independence.

b) It is possible to clean up the network model [CODA78], so it is not so arcane.

Hence, both camps responded to the criticisms of the other camp. The debate then died down, and attention focused on the commercial marketplace to see what would happen.

Fortuitously for the relational camp, the minicomputer revolution was occurring, and VAXes were proliferating. We should stress how important this class of machines was. Previously, one could compute interactively on 16 bit machines (e.g. PDP 11's) or in batch on mainframes. It was a challenge to program around the address limitations of 16 bit machines, as noted in [STON76]. Batch processing has the obvious disadvantages. The VAX put 32 bit computing at an affordable price tag, it was immediately incredibly popular.

VAXes were an obvious target for the early commercial relational systems, such as Oracle and INGRES. Happily for the relational camp, the major CODASYL systems, such as IDMS from Culinaine Corp. were written in IBM assembler, and were not portable. Hence, the early relational systems had the VAX market to themselves. This gave them time to improve the performance of their products, and the success of the VAX market went hand-in-hand with the success of relational systems.

On mainframes, a very different story was unfolding. IBM sold a derivative of System R on VM/370 and a second derivative on VSE, their low end operating system. However, neither platform was used by serious business data processing users. All the action was on MVS, the high-end operating system. Here, IBM continued to sell IMS, Cullinaine successfully sold IDMS, and relational systems were nowhere to be seen.

Hence, VAXes were a relational market and mainframes were a non-relational market. At the time all serious data management was done on mainframes.

This state of affairs changed abruptly in 1984, when IBM announced the upcoming release of DB/2 on MVS. In effect, IBM moved from saying that IMS was their serious DBMS to a dual data base strategy, in which both IMS and DB/2 were declared strategic. Since DB/2 was the new technology and was much easier to use, it was crystal clear to everybody who the long-term winner was going to be.

IBM's signal that it was deadly serious about relational systems was a watershed moment. First, it ended once-and-for-all "the great debate". Since IBM held vast marketplace power at the time, it effectively announced that relational systems had won and CODASYL and hierarchical systems had lost. Soon after, Cullinaine and IDMS went into a marketplace swoon. Second, they effectively declared that SQL was the de facto standard relational language. Other (possibly better) query languages, such as QUEL, were immediately dead. For a scathing critique of the semantics of SQL, consult [DATE84].

Old programs   new programs

┌─────────────────┐
│   Relational   │
│   interface   │
└─────────────────┘

┌─────────────────┐
│   │
│   IMS   │
│   │
└─────────────────┘
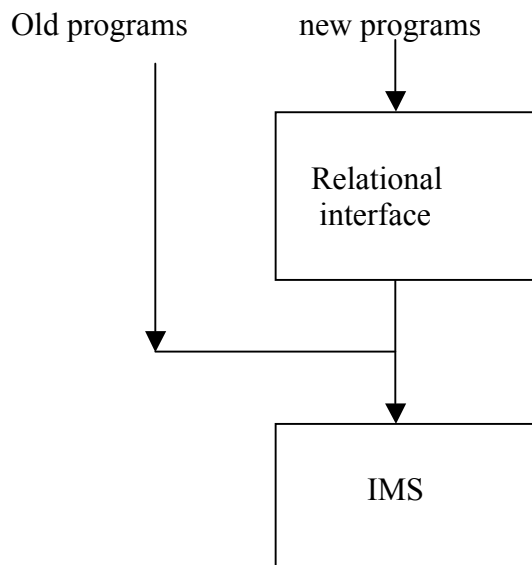
The Architecture of Project Eagle
Figure 10

A little known fact must be discussed at this point. It would have been natural for IBM to put a relational front end on top of IMS, as shown in Figure 10. This architecture would have allowed IMS customers to continue to run IMS. New application could be written to the relational interface, providing an elegant migration path to the new technology. Hence, over time a gradual shift from DL/1 to SQL would have occurred, all the while preserving the high-performance IMS underpinnings.

In fact, IBM attempted to execute exactly this strategy, with a project code-named Eagle. Unfortunately, it proved too hard to implement SQL on top of the IMS notion of logical data bases, because of semantic issues. Hence, the complexity of logical data bases in IMS came back to haunt IBM many years later. As a result, IBM was forced to move to the dual data base strategy, and to declare a winner of the great debate.

In summary, the CODASL versus relational argument was ultimately settled by three events:

a) the success of the VAX
b) the non-portability of CODASYL engines
c) the complexity of IMS logical data bases

The lessons that were learned from this epoch are:

Lesson 7: Set-a-time languages are good, regardless of the data model, since they offer much improved physical data independence.

Lesson 8: Logical data independence is easier with a simple data model than with a complex one.

Lesson 9: Technical debates are usually settled by the elephants of the marketplace, and often for reasons that have little to do with the technology.

Lesson 10: Query optimizers can beat all but the best record-at-a-time DBMS application programmers.

V The Entity-Relationship Era

In the mid 1970's Peter Chen proposed the entity-relationship (E-R) data model as an alternative to the relational, CODASYL and hierarchical data models [CHEN76]. Basically, he proposed that a data base be thought of a collection of instances of **entities**. Loosely speaking these are objects that have an existence, independent of any other entities in the data base. In our example, Supplier and Parts would be such entities.

In addition, entities have **attributes**, which are the data elements that characterize the entity. In our example, the attributes of Part would be pno, pname, psize, and pcolor. One or more of these attributes would be designated to be unique, i.e. to be a key. Lastly, there could be **relationships** between entities. In our example, Supply is a relationship

between the entities Part and Supplier. Relationships could be 1-to-1, 1-to-n, n-to-1 or m-to-n, depending on how the entities participate in the relationship. In our example, Suppliers can supply multiple parts, and parts can be supplied by multiple suppliers. Hence, the Supply relationship is m-to-n. Relationships can also have attributes that describe the relationship. In our example, qty and price are attributes of the relationship Supply.

A popular representation for E-R models was a "boxes and arrows" notation as shown in Figure 11. The E-R model never gained acceptance as the underlying data model that is implemented by a DBMS. Perhaps the reason was that in the early days there was no query language proposed for it. Perhaps it was simply overwhelmed by the interest in the relational model in the 1970's. Perhaps it looked too much like a "cleaned up" version of the CODASYL model. Whatever the reason, the E-R model languished in the 1970's.

An E-R Diagram
Figure 11

There is one area where the E-R model has been wildly successful, namely in data base (schema) design. The standard wisdom from the relational advocates was to perform data base design by constructing an initial collection of tables. Then, one applied normalization theory to this initial design. Throughout the decade of the 1970's there were a collection of normal forms proposed, including second normal form (2NF) [CODD71b], third normal form [CODD71b], Boyce-Codd normal form (BCNF) [CODD72b], fourth normal form (4NF) [FAGI77a], and project-join normal form [FAGI77b].

There were two problems with normalization theory when applied to real world data base design problems. First, real DBAs immediately asked "How do I get an initial set of tables?" Normalization theory had no answer to this important question. Second, and perhaps more serious, normalization theory was based on the concept of functional dependencies, and real world DBAs could not understand this construct. Hence, data base design using normalization was "dead in the water".

In contrast, the E-R model became very popular as a data base design tool. Chen's papers contained a methodology for constructing an initial E-R diagram. In addition, it was straightforward to convert an E-R diagram into a collection of tables in third normal

form [WONG79]. Hence, a DBA tool could perform this conversion automatically. As such, a DBA could construct an E-R model of his data, typically using a boxes and arrows drawing tool, and then be assured that he would automatically get a good relational schema. Essentially all data base design tools, such as Silverrun from Magna Solutions, ERwin from Computer Associates, and ER/Studio from Embarcadero work in this fashion.

Lesson 11: Functional dependencies are too difficult for mere mortals to understand. Another reason for KISS (Keep It Simple Stupid).

VI R++ Era

Beginning in the early 1980's a (sizeable) collection of papers appeared which can be described by the following template:

Consider an application, call it X
Try to implement X on a relational DBMS
Show why the queries are difficult or why poor performance is observed
Add a new "feature" to the relational model to correct the problem

Many X's were investigated including mechanical CAD [KATZ86], VLSI CAD [BATO85], text management [STON83], time [SNOD85] and computer graphics [SPON84]. This collection of papers formed "the R++ era", as they all proposed additions to the relational model. In our opinion, probably the best of the lot was Gem [ZANI83]. Zaniolo proposed adding the following constructs to the relational model, together with corresponding query language extensions:

1) **set-valued attributes**. In a Parts table, it is often the case that there is an attribute, such as available_colors, which can take on a set of values. It would be nice to add a data type to the relational model to deal with sets of values.

2) aggregation (tuple-reference as a data type). In the Supply relation noted above, there are two **foreign keys**, sno and pno, that effectively point to tuples in other tables. It is arguably cleaner to have the Supply table have the following structure:
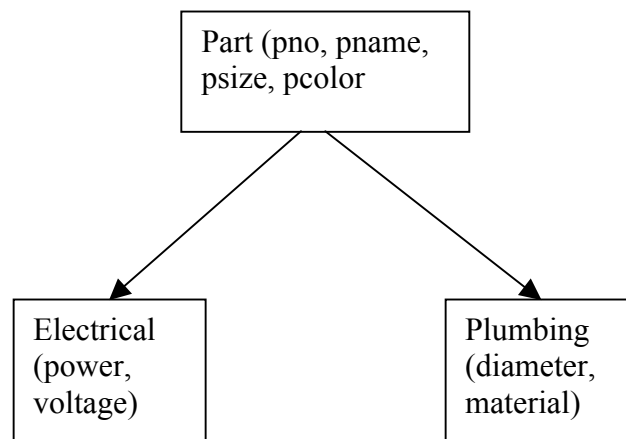
Supply (PT, SR, qty, price)

Here the data type of PT is "tuple in the Part table" and the data type of SR is "tuple in the Supplier table". Of course, the expected implementation of these data types is via some sort of pointer. With these constructs however, we can find the suppliers who supply red parts as:

Select Supply.SR.sno
From Supply
Where Supply.PT.pcolor = "red"

This "cascaded dot" notation allowed one to query the Supply table and then effectively reference tuples in other tables. This cascaded dot notation is similar to the path expressions seen in high level network languages such as LSL. It allowed one to traverse between tables without having to specify an explicit join.

3) generalization. Suppose there are two kinds of parts in our example, say electrical parts and plumbing parts. For electrical parts, we record the power consumption and the voltage. For plumbing parts we record the diameter and the material used to make the part. This is shown pictorially in Figure 12, where we see a root part with two specializations. Each specialization **inherits** all of the data attributes in its ancestors.

Inheritance hierarchies were put in early programming languages such as Simula [DAHL66], Planner [HEWI69] and Conniver [MCDO73]. The same concept has been included in more recent programming languages, such as C++. Gem was an early example of the application of this concept to data bases.

```
┌─────────────────┐
│ Part (pno, pname,│
│ psize, pcolor    │
└─────────────────┘
        │
   ┌────┴────┐
   ▼         ▼
┌──────────┐  ┌──────────┐
│Electrical│  │Plumbing  │
│(power,   │  │(diameter,│
│voltage)  │  │material) │
└──────────┘  └──────────┘
```

An Inheritance Hierarchy
Figure 12

In Gem, one could reference an inheritance hierarchy in the query language. For example to find the names of Red electrical parts, one would use:

Select E.pname
From Electrical E
Where E.pcolor = "red"

In addition, Gem had a very elegant treatment of null values.

The problem with extensions of this sort is that while they allowed easier query formulation than was available in the conventional relational model, they offered very little performance improvement. For example, primary-key-foreign-key relationships in the relational model easily simulate tuple as a data type. Moreover, since foreign keys are essentially logical pointers, the performance of this construct is similar to that available from some other kind of pointer scheme. Hence, an implementation of Gem would not be noticeably faster than an implementation of the relational model

In the early 1980's, the relational vendors were singularly focused on improving transaction performance and scalability of their systems, so that they could be used for large scale business data processing applications. This was a very big market that had major revenue potential. In contrast, R++ ideas would have minor impact. Hence, there was little technology transfer of R++ ideas into the commercial world, and this research focus had very little long-term impact.

Lesson 12: Unless there is a big performance or functionality advantage, new constructs will go nowhere.

VII The Semantic Data Model Era

At around the same time, there was another school of thought with similar ideas, but a different marketing strategy. They suggested that the relational data model is "semantically impoverished", i.e. it is incapable of easily expressing a class of data of interest. Hence, there is a need for a "post relational" data model.

Post relational data models were typically called semantic data models. Examples included the work by Smith and Smith [SMIT77] and Hammer and McLeod [HAMM81]. SDM from Hammer and McLeod is arguably the more elaborate semantic data model, and we focus on its concepts in this section.

SDM focuses on the notion of **classes**, which are a collection of records obeying the same schema. Like Gem, SDM exploited the concepts of aggregation and generalization and included a notion of sets. Aggregation is supported by allowing classes to have attributes that are records in other classes. However, SDM generalizes the aggregation construct in Gem by allowing an attribute in one class to be a **set** of instances of records in some class. For example, there might be two classes, Ships and Countries. The Countries class could have an attribute called Ships_registered_here, having as its value a collection of ships. The **inverse** attribute, country_of_registration can also be defined in SDM.

In addition, classes can generalize other classes. Unlike Gem, generalization is extended to be a graph rather than just a tree. For example, Figure 13 shows a generalization graph where American_oil_tankers inherits attributes from both Oil_tankers and American_ships. This construct is often called **multiple inheritance**. Classes can also be the union, intersection or difference between other classes. They can also be a subclass of another class, specified by a predicate to determine membership. For example,

Heavy_ships might be a subclass of Ships with weight greater than 500 tons. Lastly, a class can also be a collection of records that are grouped together for some other reason. For example Atlantic_convoy might be a collection of ships that are sailing together across the Atlantic Ocean.

Lastly, classes can have class variables, for example the Ships class can have a class variable which is the number of members of the class.

Most semantic data models were very complex, and were generally paper proposals. Several years after SDM was defined, Univac explored an implementation of Hammer and McLeod's ideas. However, they quickly discovered that SQL was an intergalactic standard, and their incompatible system was not very successful in the marketplace.
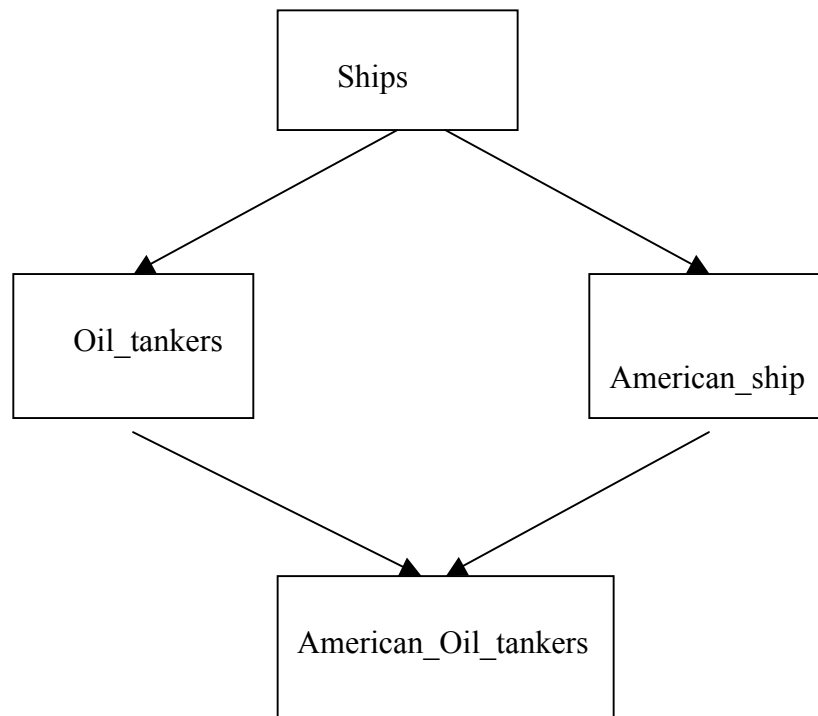
A Example of Multiple Inheritance
Figure 13

In our opinion, SDMs had the same two problems that faced the R++ advocates. Like the R++ proposals, they were a lot of machinery that was easy to simulate on relational systems. Hence, there was very little leverage in the constructs being proposed. The SDM camp also faced the second issue of R++ proposals, namely that the established

vendors were distracted with transaction processing performance. Hence, semantic data models had little long term influence.

VIII OO Era

Beginning in the mid 1980's there was a "tidal wave" of interest in Object-oriented DBMSs (OODB). Basically, this community pointed to an "impedance mismatch" between relational data bases and languages like C++.

In practice, relational data bases had their own naming systems, their own data type systems, and their own conventions for returning data as a result of a query. Whatever programming language was used alongside a relational data base also had its own version of all of these facilities. Hence, to bind an application to the data base required a conversion from "programming language speak" to "data base speak" and back. This was like "gluing an apple onto a pancake", and was the reason for the so-called impedance mismatch.

For example, consider the following C++ snippet which defines a Part Structure and then allocates an Example_part.

```
Struct Part {
        Int number;
        Char* name;
        Char* bigness;
        Char* color;
} Example_part;
```

All SQL run-time systems included mechanisms to load variables in the above Struct from values in the data base. For example to retrieve part 16 into the above Struct required the following stylized program:

```
Define cursor P as
        Select *
        From Part
        Where pno = 16;

Open P into Example_part
Until no-more{
        Fetch P (Example_part.number = pno,
                Example_name = pname
                Example_part.bigness = psize
                Example_part.color = pcolor)
}
```

First one defined a cursor to range over the answer to the SQL query. Then, one opened the cursor, and finally fetched a record from the cursor and bound it to programming

language variables, which did not need to be the same name or type as the corresponding data base objects. If necessary, data type conversion was performed by the run-time interface.

The programmer could now manipulate the Struct in the native programming language. When more than one record could result from the query, the programmer had to iterate the cursor as in the above example.

It would seem to be much cleaner to integrate DBMS functionality more closely into a programming language. Specifically, one would like a **persistent programming language**, i.e. one where the variables in the language could represent disk-based data as well as main memory data and where data base search criteria were also language constructs. Several prototype persistent languages were developed in the late 1970's, including Pascal-R [SCHM77], Rigel [ROWE79], and a language embedding for PL/1 [DATE76]. For example, Rigel allowed the above query to be expressed as:

```
For P in Part where P.pno = 16{
        Code_to_manipulate_part
}
```

In Rigel, as in other persistent languages, variables (in this case pno) could be declared. However, they only needed to be declared once to Rigel, and not once to the language and a second time to the DBMS. In addition, the predicate p.no = 16 was part of the Rigel programming language. Lastly, one used the standard programming language iterators (in this case a For loop) to iterate over qualifying records.

A persistent programming language is obviously much cleaner than a SQL embedding. However, it requires the compiler for the programming language to be extended with DBMS-oriented functionality. Since there is no programming language Esperanto, this extension must be done once per complier. Moreover, each extension will likely be unique, since C++ is quite different from, for example, APL.

Unfortunately, programming language experts have consistently refused to focus on I/O in general and DBMS functionality in particular. Hence, all programming languages that we are aware of have no built-in functionality in this area. Not only does this make embedding data sublanguages tedious, but also the result is usually difficult to program and error prone. Lastly, language expertise does not get applied to important special purpose data-oriented languages, such as report writers and so-called fourth generation languages.

Hence, there was no technology transfer from the persistent programming language research efforts of the 1970's into the commercial marketplace, and ugly data-sublanguage embeddings prevailed.

In the mid 1980's there was a resurgence of interest in persistent programming languages, motivated by the popularity of C++. This research thrust was called Object-Oriented

Data Bases (OODB), and focused mainly on persistent C++. Although the early work came from the research community with systems like Garden [SKAR86] and Exodus [RICH87], the primary push on OODBs came from a collection of start-ups, including Ontologic, Object Design and Versant. All built commercial systems that supported persistent C++.

The general form of these systems was to support C++ as a data model. Hence, any C++ structure could be persisted. It was popular to extend C++ with the notion of relationships, a concept borrowed directly from the Entity-Relationship data model a decade earlier. Hence, several systems extended the C++ run-time with support for this concept.

Most of the OODB community decided to address engineering data bases as their target market. One typical example of this area is engineering CAD. In a CAD application, an engineer opens an engineering drawing, say for an electronic circuit, and then modifies the engineering object, tests it, or runs a power simulator on the circuit. When he is done he closes the object. The general form of these applications is to open a large engineering object and then process it extensively before closing it.

Historically, such objects were read into virtual memory by a load program. This program would "swizzle" a disk-based representation of the object into a virtual memory C++ object. The word "swizzle" came from the necessity of modifying any pointers in the object when loading. On disk, pointers are typically some sort of logical reference such as a foreign key, though they can also be disk pointers, for example (block-number, offset). In virtual memory, they should be virtual memory pointers. Hence, the loader had to swizzle the disk representation to a virtual memory representation. Then, the code would operate on the object, usually for a long time. When finished, an unloader would linearize the C++ data structure back into one that could persist on disk.

To address the engineering market, an implementation of persistent C++ had the following requirements:

1) no need for a declarative query language. All one needed was a way to reference large disk-based engineering objects in C++.
2) no need for fancy transaction management. This market is largely one-user-at-a-time processing large engineering objects. Rather, some sort of versioning system would be nice.
3) The run-time system had to be competitive with conventional C++ when operating on the object. In this market, the performance of an algorithm using persistent C++ had to be competitive with that available from a custom load program and conventional C++

Naturally, the OODB vendors focused on meeting these requirements. Hence, there was weak support for transactions and queries. Instead, the vendors focused on good performance for manipulating persistent C++ structures. For example, consider the following declaration:

Persistent int I;

And then the code snippet:

I =: I+1;

In conventional C++, this is a single instruction. To be competitive, incrementing a persistent variable cannot require a process switch to process a persistent object. Hence, the DBMS must run in the same address space as the application. Likewise, engineering objects must be aggressively cached in main memory, and then "lazily" written back to disk.

Hence, the commercial OODBs, for example Object Design [LAMB91], had innovative architectures that achieved these objectives.

Unfortunately, the market for such engineering applications never got very large, and there were too many vendors competing for a "niche" market. At the present time, most of the OODB vendors have failed, or have repositioned their companies to offer something other than and OODB. For example, Object Design has renamed themselves Excelon, and is selling XML services

In our opinion, there are a number of reasons for this market failure.

1) absence of leverage. The OODB vendors presented the customer with the opportunity to avoid writing a load program and an unload program. This is not a major service, and customers were not willing to pay big money for this feature.
2) No standards. All of the OODB vendor offerings were incompatible.
3) Relink the world. In anything changed, for example a C++ method that operated on persistent data, then all programs which used this method had to be relinked. This was a noticeable management problem.
4) No programming language Esperanto. If your enterprise had a single application not written in C++ that needed to access persistent data, then you could not use one of the OODB products.

Of course, the OODB products were not designed to work on business data processing applications. Not only did they lack strong transaction and query systems but also they ran in the same address space as the application. This meant that the application could freely manipulate all disk-based data, and no data protection was possible. Protection and authorization is important in the business data processing market. In addition, OODBs were clearly a throw back to the CODASYL days, i.e. a low-level record at a time language with the programmer coding the query optimization algorithm. As a result, these products had essentially no penetration in the (very large) business data processing market.

Lesson 13: Packages will not sell to users unless they are in "major pain"

Lesson 14: Persistent languages will go nowhere without the support of the programming language community.

IX The Object-Relational Era

The Object-Relational (OR) era was motivated by a very simple problem. In the early days of INGRES, the team had been interested in geographic information systems (GIS) and had suggested mechanisms for their support [GO75]. Around 1982, the following simple GIS issue was haunting the INGRES research team. Suppose one wants to store geographic positions in a data base. For example, one might want to store the location of a collection of intersections as:

Intersections (I-id, long, lat, other-data)

Here, we require storing geographic points (long, lat) in a data base. Then, if we want to find all the intersections within a bounding rectangle, (X0, Y0, X1, Y1), then the SQL query is:

Select I-id
From Intersections
Where X0 < long < X1 and Y0 < lat < Y1

Unfortunately, this is a two dimensional search, and the B-trees in INGRES are a one-dimensional access method. One-dimensional access methods do not do two-dimensional searches efficiently, so there is no way in a relational system for this query to run fast.

More troubling was the "notify parcel owners" problem. Whenever there is request for a variance to the zoning laws for a parcel of land in California, there must be a public hearing, and all property owners within a certain distance must be notified.

Suppose one assumes that all parcels are rectangles, and they are stored in the following table.

Parcel (P-id, Xmin, Xmax, Ymin, Ymax)

Then, one must enlarge the parcel in question by the correct number of feet, creating a "super rectangle" with co-ordinates X0, X1, Y0, Y1. All property owners whose parcels intersect this super rectangle must be notified, and the most efficient query to do this task is:

Select P-id
From Parcel
Where Xmax > X0 and Ymax > Y0 and Xmin < X1 and Ymax < Y1

Again, there is no way to execute this query efficiency with a B-tree access method. Moreover, it takes a moment to convince oneself that this query is correct, and there are several other less efficient representations. In summary, simple GIS queries are difficult to express in SQL, and they execute on standard B-trees with unreasonably bad performance.

The following observation motivates the OR proposal. Early relational systems supported integers, floats, and character strings, along with the obvious operators, primarily because these were the data types of IMS, which was the early competition. IMS chose these data types because that was what the business data processing market wanted, and that was their market focus. Relational systems also chose B-trees because these facilitate the searches that are common in business data processing. Later relational systems expanded the collection of business data processing data types to include date, time and money. More recently, packed decimal and blobs have been added.

In other markets, such as GIS, these are not the correct types, and B-trees are not the correct access method. Hence, to address any given market, one needs data types and access methods appropriate to the market. Since there may be many other markets one would want to address, it is inappropriate to "hard wire" a specific collection of data types and indexing strategies. Rather a sophisticated user should be able to add his own; i.e. to **customize** a DBMS to his particular needs. A general purpose extension mechanism is also helpful in business data processing, since one or more new data types appear to be needed every decade.

As a result, the OR proposal added

user-defined data types,
user-defined operators,
user-defined functions, and
user-defined access methods

to a SQL engine. The major OR research prototype was Postgres [STON86].

Applying the OR methodology to GIS, one merely adds geographic points and geographic boxes as data types. With these data types, the above tables above can be expressed as:

Intersections (I-id, point, other-data)
Parcel (P-id, P-box)

Of course, one must also have SQL operators appropriate to each data type. For our simple application, these are !! (point in rectangle) and ## (box intersects box). The two queries now become

Select I-id
From Intersections

Where point !! "X0, X1, Y0, Y1"

and

Select P-id
From Parcel
Where P-box ## "X0, X1, Y0, Y1"

To support the definition of user-defined operators, one must be able to specify a user-defined function (UDF), which can process the operator. Hence, for the above examples, we require functions

Point-in-rect (point, box)

and

Box-int-box (box, box)

which return Booleans. These functions must be called whenever the corresponding operator must be evaluated, passing the two arguments in the call, and then acting appropriately on the result.

To address the GIS market one needs a multi-dimensional indexing system, such as Quad trees [SAME84] or R-trees [GUTM84]. In summary, a high performance GIS DBMS can be constructed with appropriate user-defined data types, user-defined operators, user-defined functions, and user-defined access methods.

The main contribution of Postgres was to figure out the engine mechanisms required to support this kind of extensibility. In effect, previous relational engines had hard coded support for a specific set of data types, operators and access methods. All this hard-coded logic must be ripped out and replaced with a much more flexible architecture. Many of the details of the Postgres scheme are covered in [STON90].

There is another interpretation to UDFs which we now present. In the mid 1980's Sybase pioneered the inclusion of **stored procedures** in a DBMS. The basic idea was to offer high performance on TPC-B, which consisted of the following commands that simulate cashing a check:

Begin transaction

Update account set balance = balance – X
Where account_number = Y

Update Teller set cash_drawer = cash_drawer – X
Where Teller_number = Z

Update bank set cash – cash – Y

Insert into log (account_number = Y, check = X, Teller= Z)

Commit

This transaction requires 5 or 6 round trip messages between the DBMS and the application. Since these context switches are expensive relative to the very simple processing which is being done, application performance is limited by the context switching time.

A clever way to reduce this time is to define a stored procedure:

Define cash_check (X, Y, Z)
        Begin transaction

        Update account set balance = balance – X
        Where account_number = Y

        Update Teller set cash_drawer = cash_drawer – X
        Where Teller_number = Z

        Update bank set cash – cash – Y

        Insert into log (account_number = Y, check = X, Teller= Z)

        Commit

        End cash_check

Then, the application merely executes the stored procedure, with its parameters, e.g:

Execute cash_check ($100, 79246, 15)

This requires only one round trip between the DBMS and the application rather than 5 or 6, and speeds up TPC-B immensely. To go fast on standard benchmarks such as TPC-B, all vendors implemented stored procedures. Of course, this required them to define proprietary (small) programming languages to handle error messages and perform required control flow. This is necessary for the stored procedure to deal correctly with conditions such as "insufficient funds" in an account.

Effectively a stored procedure is a UDF that is written in a proprietary language and is "brain dead", in the sense that it can only be executed with constants for its parameters. The Postgres UDTs and UDFs generalized this notion to allow code to be written in a conventional programming language and to be called in the middle of processing conventional SQL queries.

Postgres implemented a sophisticated mechanism for UDTs, UDFs and user-defined access methods. In addition, Postgres also implemented less sophisticated notions of inheritance, and type constructors for pointers (references), sets, and arrays. This latter set of features allowed Postgres to become "object-oriented" at the height of the OO craze.

Later benchmarking efforts such as Bucky [CARE97] proved that the major win in Postgres was UDTs and UDFs; the OO constructs were fairly easy and fairly efficient to simulate on conventional relational systems. This work demonstrated once more what the R++ and SDM crowd had already seen several years earlier; namely built-in support for aggregation and generalization offer little performance benefit. Put differently, the major contribution of the OR efforts turned out to be a better mechanism for stored procedures and user-defined access methods.

The OR model has enjoyed some commercial success. Postgres was commercialized by Illustra. After struggling to find a market for the first couple of years, Illustra caught "the internet wave" and became "the data base for cyberspace". If one wanted to store text and images in a data base and mix them with conventional data types, then Illustra was the engine which could do that. Near the height of the internet craze, Illustra was acquired by Informix. From the point of view of Illustra, there were two reasons to join forces with Informix:

a) inside every OR application, there is a transaction processing sub-application. In order to be successful in OR, one must have a high performance OLTP engine. Postgres had never focused on OLTP performance, and the cost of adding it to Illustra would be very high. It made more sense to combine Illustra features into an existing high performance engine.

b) To be successful, Illustra had to convince third party vendors to convert pieces of their application suites into UDTs and UDFs. This was a non-trivial undertaking, and most external vendors balked at doing so, at least until Illustra could demonstrate that OR presented a large market opportunity. Hence, Illustra had a "chicken and egg" problem. To get market share they needed UDTs and UDFs; to get UDTs and UDFs they needed market share.

Informix provided a solution to both problems, and the combined company proceeded over time to sell OR technology fairly successfully into the GIS market and into the market for large content repositories (such as those envisoned by CNN and the British Broadcasting Corporation). However, widescale adoption of OR in the business data processing market remained elusive. Of course, the (unrelated) financial difficulties at Informix made selling new technology such as OR extremely difficult. This certainly hindered wider adoption.

OR technology is gradually finding market acceptance. For example, it is more effective to implement data mining algorithms as UDFs, a concept pioneered by Red Brick and

recently adopted by Oracle. Instead of moving a terabyte sized warehouse up to mining code in middleware, it is more efficient to move the code into the DBMS and avoid all the message overhead. OR technology is also being used to support XML processing, as we will see presently.

One of the barriers to acceptance of OR technology in the broader business market is the absence of standards. Every vendor has his own way of defining and calling UDFs, In addition, most vendors support Java UDFs, but Microsoft does not. It is plausible that OR technology will not take off unless (and until) the major vendors can agree on standard definitions and calling conventions.

Lesson 14: The major benefits of OR is two-fold: putting code in the data base (and thereby bluring the distinction between code and data) and a general purpose extension mechanism that allows OR DBMSs to quickly respond to market requirements.

Lesson 15: Widespread adoption of new technology requires either standards and/or an elephant pushing hard.

X Semi Structured Data

There has been an avalanche of work on "semi-structured" data in the last five years. An early example of this class of proposals was Lore [MCHU97]. More recently, the various XML-based proposals have the same flavor. At the present time, XMLSchema and XQuery are the standards for XML-based data.

There are two basic points that this class of work exemplifies.

1) schema later
2) complex graph-oriented data model

We talk about each point separately in this section.

10.1 Schema Later

There are two interpretations to this notion. In the first one, a schema is not required in advance. In a "**schema first**" system the schema is specified, and instances of data records that conform to this schema can be subsequently loaded**.** Hence, the data base is always consistent with the pre-existing schema, because the DBMS rejects any records that are not consistent with the schema. All previous data models required a DBA to specify the schema in advance.

The second interpretation of "schema later" has come more recently. It suggests that schemas should be fluid, and easy to change. A schema exists in advance, but it should be trivial to evolve the schema as the meaning of the data changes. We will call this interpretation "**easy schema evolution**" to distinguish it from the first interpretation,

which we continue to call "**schema later**". We now discuss these two interpretations in turn.

10.1.1 Schema Later

In this interpretation the schema does not need to be specified in advance. It can be specified later, or even not at all. In a "**schema later**" system, data instances must be self-describing, because there is not necessarily a schema to give meaning to incoming records.   Without a self-describing format, a record is merely "a bucket of bits".

To make a record self-describing, one must tag each attribute with metadata that defines the meaning of the attribute. Here are a couple of examples of such records, using an artificial tagging system:

Person:
        Name: Joe Jones
        Wages: 14.75
        Employer: My_accounting
        Hobbies: skiing, bicycling
        Works for: ref (Fred Smith)
        Favorite joke: Why did the chicken cross the road? To get to the other side
        Office number: 247
        Major skill: accountant
End Person

Person:
        Name: Smith, Vanessa
        Wages: 2000
        Favorite coffee: Arabian
        Pastimes: sewing, swimming
        Works_for: Between jobs
        Favorite restaurant: Panera
        Number of children: 3
End Person:

As can be seen, these two records each describe a person. Moreover, each attribute has one of three characteristics:

1) it appears in only one of the two records, and there is no attribute in the other record with the same meaning.
2) it appears in only one of the two records, but there is an attribute in the other record with the same meaning (e.g. pastimes and hobbies).
3) it appears in both records, but the format or meaning is different (e.g. Works_for, Wages)

Clearly, comparing these two persons is a challenge. This is an example of **semantic heterogeneity**, where information on a common object (in this case a person) does not conform to a common representation. Semantic heterogeneity makes query processing a big challenge, because there is no structure on which to base indexing decisions and query execution strategies.

The advocates of "schema later" typically have in mind applications where it is natural for users to enter their data as free text, perhaps through a word processor (which may annotate the text with some simple metadata about document structure). In this case, it is an imposition to require a schema to exist before a user can add data. The "schema later" advocates then have in mind automatically or semi-automatically tagging incoming data to construct the above semi-structured records.

In contrast, if a business form is used for data entry, (which would probably be natural for the above Person data), then a "schema first" methodology is being employed, because the person who designed the form is, in effect, also defining the schema by what he allows in the form. As a result, schema later is appropriate mainly for applications where free text is the mechanism for data entry.

To explore the utility of schema later, we present the following scheme that classifies applications into four buckets.

Ones with rigidly structured data
Ones with rigidly structured data with some text fields
Ones with semi-structured data
Ones with text

Rigidly structured data encompasses data that must conform to a schema. In general, this includes essentially all data on which business processes must operate. For example, consider the payroll data base for a typical company. This data must be rigidly structured, or the check-printing program might produce erroneous results. One simply cannot tolerate missing or badly formatted data that business processes depends on. For rigidly structured data, one should insist on schema-first.

The personnel records of a large company are typical of the second class of data base applications that we consider. There is a considerable amount of rigidly structured data, such as the health plan each employee is enrolled in, and the fringe benefits they are entitled to. In addition, there are free text fields, such as the comments of the manager at the last employee review. The employee review form is typically rigidly structured; hence the only free text input is into specific comment fields. Again schema first appears the right way to go, and this kind of application is easily addressed by an Object-Relational DBMS with an added text data type.

The third class of data is termed semi-structured. The best examples we can think of are want ads and resumes. In each of these cases, there is some structure to the data, but data instances can vary in the fields that are present and how they are represented. Moreover,

there is no schema to which instances necessarily conform. Semi-structured instances are often entered as a text document, and then parsed to find information of interest, which is in turn "shredded" into appropriate fields inside the storage engine. In this case, schema later is a good idea.

The fourth class of data is pure text, i.e. documents with no particular structure. In this bucket, there is no obvious structure to exploit. Information Retrieval (IR) systems have focused on this class of data for several decades. Few IR researchers have any interest in semi-structured data; rather they are interested in document retrieval based on the textual content of the document. Hence, there is no schema to deduce in this bucket, and this corresponds to "schema not at all".

As a result, schema-later proposals deal only with the third class of data in our classification system. It is difficult to think up very many examples of this class, other than resumes and advertisements. The proponents (many of whom are academics) often suggest that college course descriptions fit this category. However, every university we know has a rigid format for course descriptions, which includes one or more text fields. Most have a standard form for entering the data, and a system (manual or automatic) to reject course descriptions that do not fit this format. Hence, course descriptions are an example of the second class of data, not the third. In our opinion, a careful examination of the claimed instances of class 3 applications will yield many fewer actual instances of the class. Moreover, the largest web site specializing in resumes (Monster.com) has recently adopted a business form through which data entry occurs. Hence, they have switched from class 3 to class 2, presumably to enforce more uniformity on their data base (and thereby easier comparability).

Semantic heterogeneity has been with enterprises for a very long time. They spend vast sums on warehouse projects to design standard schemas and then convert operational data to this standard. Moreover, in most organizations semantic heterogeneity is dealt with on a data set basis; i.e. data sets with different schemas must be homogenized. Typical warehouse projects are over budget, because schema homogenization is so hard. Any schema-later application will have to confront semantic heterogeneity on a record-by-record basis, where it will be even more costly to solve. This is a good reason to avoid "schema later" if at all possible.

In summary, schema later is appropriate only for the third class of applications in our classification scheme. Moreover, it is difficult to come up with very many convincing examples in this class. If anything, the trend is to move class three applications into class 2, presumably to make semantic heterogeneity issues easier to deal with. Lastly, class three applications appear to have modest amounts of data. For these reasons, we view schema later data bases as a niche market.

10.1.2 Schema Evolution

Current relational data bases have fairly primitive and rigid facilities for schema evolution. There is an Alter Table command whereby a table definition can be changed.

The old definition can then be defined as a view on top of the new table definition. In a similar way, a table can be split apart and then the old table defined as a view including a join. In this case the new tables must be constructed by projections from the existing table. There are at least two ways by which this construct could be made vastly more useful.

First, it would be nice if a table could be marked as "exploratory". In this way, a user could simply enter data that did not conform to the schema, and the system would automatically introduce an Alter Table command on the fly. Moreover, instead of converting all the data instances to the new definition immediately (in case the new schema splits apart the old record), it would be nice to perform this bulk copy operation in a "lazy" fashion as a background operation or even not at all. In this situation, the system would have to deal with the data, partly in the old format and partly in the new format. Such facilities would make schema evolution much more graceful and easier to use.

The second extension is a further generalization of these ideas. In the scientific data base community, it is often important to maintain what has come to be call "data lineage". Hence, not only should there be a schema for the data in a data set, but also, the system should keep track of the operations that have been previously applied to the data. This could include the algorithm for cleaning the data, any transformations that filtered the data, etc. Data lineage is especially relevant to satellite imagery. Here, the raw data is often a collection of images of the target area, one for each pass of the satellite, with some portions of each image obscured by cloud cover. There are many algorithms to choose the best portions of each image to include in a single composite image of the total area.

The scientist using such a derived data set needs to know the algorithm used to construct the composite image in order to determine whether it is useful for his purposes. As such, he needs to know the lineage of the data set. Current DBMSs have no capabilities for support of data lineage, and there is at least one community which would like much better capabilities.

We believe that schema evolution is an area where existing data base products are especially weak. Much better capabilities are possible, and we would hope that commercial products would move in this direction. A similar comment can be made about version control capabilities.

10.2 XML Data Model

We now turn to the XML data model.   In the past, the mechanism for describing a schema was Document Type Definitions (DTDs), and in the future the data model will be specified in XMLSchema. DTDs and XMLSchema were intended to deal with the structure of formatted documents (and hence the word "document" in DTDs). As a result, they look like a document markup language, in particular a subset of SGML. Because the structure of a document can be very complex, these document specification

standards are necessarily very complex. As a document specification system, we have no quarrel with these standards.

After DTDs and XMLSchema were "cast into cement", members of the DBMS research community decided to try and use them to describe structured data. As a data model for structured data, we believe both standards are seriously flawed. To a first approximation, these standards have everything that was ever specified in any previous data model proposal. In addition, they contain additional features that are complex enough, that nobody in the DBMS community has ever seriously proposed them in a data model.

For example, the data model presented in XMLSchema has the following characteristics:

1) XML records can be hierarchical, as in IMS
2) XML records can have "links" (references to) other records, as in CODASYL, Gem and SDM
3) XML records can have set-based attributes, as in SDM
4) XML records can inherit from other records in several ways, as in SDM

In addition, XMLSchema also has several features, which are well known in the DBMS community but never attempted in previous data models because of complexity. One example is **union types**, that is, an attribute in a record can be of one of a set of possible types. For example, in a personnel data base, the field "works-for" could either be a department number in the enterprise, or the name of an outside firm to whom the employee is on loan. In this case works-for can either be a string or an integer, with different meanings.

Note that B-tree indexes on union types are complex. In effect, there must be an index for each data type in the union. Moreover, there must be a different query plan for each query that touches a union type. If two union types containing N and M base types respectively, are to be joined, then there will be at least Max (M, N) plans to co-ordinate. For these reasons, union types have never been seriously considered for inclusion in a DBMS.

Obviously, XMLSchema is far and away the most complex data model ever proposed. It is clearly at the other extreme from the relational model on the "Keep It Simple Stupid" (KISS) scale. It is hard to imaging something this complex being used as a model for structured data. We can see three scenarios off into the future.

Scenario 1: XMLSchema will fail because of excessive complexity

Scenario 2: A "data-oriented" subset of XMLSchema will be proposed that is vastly simpler.

Scenario 3: XMLSchema will become popular. Within a decade all of the problems with IMS and CODASYL that motivated Codd to invent the relational model will resurface. At that time some enterprising researcher, call him Y, will "dust off" Codd's original

paper, and there will be a replay of the "Great Debate". Presumably it will end the same way as the last one. Moreover, Codd won the Turing award in 1981 [CODD82] for his contribution. In this scenario, Y will win the Turing award circa 2015.

In fairness to the proponents of "X stuff", they have learned something from history. They are proposing a set-at-a-time query language, Xquery, which will provide a certain level of data independence. As was discovered in the CODASYL era, providing views for a graph data model will be a challenge (and will be much harder than for the relational model).

10.3 Summary

Summarizing XML/XML-Schema/Xquery is a challenge, because it has many facets. Clearly, XML will be a popular "on-the-wire" format for data movement across a network. The reason is simple: XML goes through firewalls, and other formats do not. Since there is always a firewall between the machines of any two enterprises, it follows that cross-enterprise data movement will use XML. Because a typical enterprise wishes to move data within the enterprise the same way as outside the enterprise, there is every reason to believe that XML will become an intergalactic data movement standard.

As a result, all flavors of system and application software must be prepared to send and receive XML. It is straightforward to convert the tuple sets that are produced by relational data bases into XML. If one has an OR engine, this is merely a user-defined function. Similarly, one can accept input in XML and convert it to tuples to store in a data base with a second user-defined function. Hence OR technology facilitates the necessary format conversions. Other system software will likewise require a conversion facility.

Moreover, higher level data movement facilities built on top of XML, such as SOAP, will be equally popular. Clearly, remote procedure calls that go through firewalls are much more useful than ones that don't. Hence, SOAP will dominate other RPC proposals.

It is possible that native XML DBMSs will become popular, but we doubt it. It will take a decade for XML DBMSs to become high performance engines that can compete with the current elephants. Moreover, schema-later should only be attractive in limited markets, and the overly complex graph-structured data model is the antithesis of KISS. XMLSchema cries out for subsetting.. A clean subset of XML-schema would have the characteristic that it maps easily to current relational DBMSs. In which case, what is the point of implementing a new engine? Hence, we expect native XML DBMSs to be a niche market.

Consider now Xquery. A (sane) subset is readily mappable to the OR SQL systems of several of the vendors. For example, Informix implemented the Xquery operator "//" as a user-defined function. Hence, it is fairly straightforward to implement a subset of Xquery on top of most existing engines. As a result, it is not unlikely that the elephants

will support both SQL and a subset of XMLSchema and XQuery. The latter interface will be translated into SQL.

XML is sometimes marketed as the solution to the semantic heterogeneity problem, mentioned earlier. Nothing could be further from the truth. Just because two people tag a data element as a salary does not mean that the two data elements are comparable. One could be salary after taxes in French Francs including a lunch allowance, while the other could be salary before taxes in US dollar. Furthermore, if you call them "rubber gloves" and I call them "latex hand protectors", then XML will be useless in deciding that they are the same concept. Hence, the role of XML will be limited to providing the vocabulary in which common schemas can be constructed.

In addition, we believe that cross-enterprise data sharing using common schemas will be slow in coming, because semantic heterogeneity issues are so difficult to resolve. Although W3C has a project in this area, the so-called semantic web, we are not optimistic about its future impact. After all, the AI community has been working on knowledge representation systems for a couple of decades with limited results. The semantic web bears a striking resemblance to these past efforts. Since web services depend on passing information between disparate systems, don't bet on the early success this concept.

More precisely, we believe that cross-enterprise information sharing will be limited to:

Enterprises that have high economic value in co-operating. After all, the airlines have been sharing data across disparate reservation systems for years.

Applications that are semantically simple (such as e-mail) where the main data type is text and there are no complex semantic mappings involved.

Applications where there is an "elephant" that controls the market. Enterprises like WalMart and Dell have little difficulty in sharing data with their suppliers. They simply say "if you want to sell to me; here is how you will interact with my information systems". When there is an elephant powerful enough to dictate standards, then cross enterprise information sharing can be readily accomplished.

We close with one final cynical note. A couple of years ago OLE-DB was being pushed hard by Microsoft; now it is "X stuff". OLE-DB was pushed by Microsoft, in large part, because it did not control ODBC and perceived a competitive advantage in OLE-DB. Now Microsoft perceives a big threat from Java and its various cross platform extensions, such as J2EE. Hence, it is pushing hard on the XML and Soap front to try to blunt the success of Java.

There is every reason to believe that in a couple of years Microsoft will see competitive advantage in some other DBMS-oriented standard. In the same way that OLE-DB was sent to an early death, we expect Microsoft to send "X stuff" to a similar fate, the minute marketing considerations dictate a change.

Less cynically, we claim that technological advances keep changing the rules. For example, it is clear that the micro-sensor technology coming to the market in the next few years will have a huge impact on system software, and we expect DBMSs and their interfaces to be affected in some (yet to be figured out) way.

Hence, we expect a succession of new DBMS standards off into the future. In such an ever changing world, it is crucial that a DBMS be very adaptable, so it can deal with whatever the next "big thing" is. OR DBMSs have that characteristic; native XML DBMSs do not.

Lesson 16: Schema-later is a probably a niche market

Lesson 17: XQuery is pretty much OR SQL with a different syntax

Lesson 18: XML will not solve the semantic heterogeneity either inside or outside the enterprise.

XI Full Circle

This paper has surveyed three decades of data model thinking. It is clear that we have come "full circle". We started off with a complex data model, which was followed by a great debate between a complex model and a much simpler one. The simpler one was shown to be advantageous in terms of understandability and its ability to support data independence.

Then, a substantial collection of additions were proposed, none of which gained substantial market traction, largely because they failed to offer substantial leverage in exchange for the increased complexity. The only ideas that got market traction were the extendability ones in OR DBMSs, and these were performance constructs not data model constructs. The current proposal is now a superset of the union of all previous proposals. I.e. we have navigated a full circle.

The debate between the XML advocates and the relational crowd bears a suspicious resemblance to the first "Great Debate" from a quarter of a century ago. A simple data model is being compared to a complex one. Relational is being compared to "CODASYL II". The only difference is that "CODASYL II" has a high level query language. Logical data independence will be harder in CODASYL II than in its predecessor, because CODASYL II is even more complex than its predecessor.

We can see history repeating itself. If native XML DBMSs gain traction, then customers will have problems with logical data independence and complexity.

To avoid repeating history, it is always wise to stand on the shoulders of those who went before, rather than on their feet. As a field, if we don't start learning something from history, we will be condemned to repeat it yet again.

More abstractly, we see few new data model ideas. Most everything put forward in the last 20 years is a reinvention of something from a quarter century ago. The only concepts noticeably new appear to be:

Code in the data base (from the OR camp)
Schema last (from the semi-structured data camp)

Schema last appears to be a niche market, and we don't see it as any sort of watershed idea. Code in the data base appears to be a really good idea. Moreover, it seems to us that designing a DBMS which made code and data equal class citizens would be a very helpful. If so, then add-ons to DBMSs such as stored procedures, triggers, and alerters would become first class citizens. The OR model got part way there; maybe it is now time to finish that effort.

References

[ASTR76] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim Gray, Patricia P. Griffiths, W. Frank King, Raymond A. Lorie, Paul R. McJones, James W. Mehl, Gianfranco R. Putzolu, Irving L. Traiger, Bradford W. Wade, Vera Watson: System R: Relational Approach to Database Management. TODS 1(2): 97-137 (1976)

[BACH73] Charles W. Bachman: The Programmer as Navigator. CACM 16(11): 635-658 (1973)

[BATO85] Don S. Batory, Won Kim: Modeling Concepts for VLSI CAD Objects. TODS 10(3): 322-346 (1985)

[CARE97] Michael J. Carey, David J. DeWitt, Jeffrey F. Naughton, Mohammad Asgarian, Paul Brown, Johannes Gehrke, Dhaval Shah: The BUCKY Object-Relational Benchmark (Experience Paper). SIGMOD Conference 1997: 135-146

[CHAM74] Donald D. Chamberlin, Raymond F. Boyce: SEQUEL: A Structured English Query Language. SIGMOD Workshop, Vol. 1 1974: 249-264

[CHEN76] Peter P. Chen: The Entity-Relationship Model - Toward a Unified View of Data. TODS 1(1): 9-36 (1976)

[CODA69] CODASYL: Data Base Task Group Report. ACM, New York, N.Y., October 1969

[CODA71] CODASYL: Feature Analysis of Generalized Data Base Management Systems. ACM, New York, N.Y., May 1971

[CODA73] CODASYL: Data Description Language, Journal of Development. National Bureau of Standards, NBS Handbook 113, June 1973

[CODA78] CODASYL: Data Description Language, Journal of Development. Information Systems, January 1978

[CODD70] E. F. Codd: A Relational Model of Data for Large Shared Data Banks. CACM 13(6): 377-387 (1970)

[CODD71a] E. F. Codd: A Database Sublanguage Founded on the Relational Calculus. SIGFIDET Workshop 1971: 35-68

[CODD71b] E. F. Codd: Normalized Data Structure: A Brief Tutorial. SIGFIDET Workshop 1971: 1-17

[CODD72a] E. F. Codd: Relational Completeness of Data Base Sublanguages. IBM Research Report RJ 987, San Jose, California: (1972)

CODD72b] E.F. Codd: Further Normalization of the Data Base Relational Model. In Data Base Systems ed. Randall Rustin, Prentice-Hall 1972

[CODD79] E. F. Codd: Extending the Database Relational Model to Capture More Meaning. TODS 4(4): 397-434 (1979)

[CODD82] E. F. Codd: Relational Database: A Practical Foundation for Productivity. CACM 25(2): 109-117 (1982)

[DAHL66] Dahl, O. and Nygard, K: 'SIMULA; An ALGOL-based Simulation Language: CACM 9(9) 671-678 (1966).

[DATE76] C. J. Date: An Architecture for High-Level Language Database Extensions. SIGMOD Conference 1976: 101-122

[DATE84] C. J. Date: A Critique of the SQL Database Language. SIGMOD Record 14(3): 8-54 (1984)

[FAGI77a] Ronald Fagin: Multivalued Dependencies and a New Normal Form for Relational Databases. TODS 2(3): 262-278 (1977)

[FAGI77b] Ronald Fagin: Normal Forms and Relational Database Operators. SIGMOD Conference 1977: 153-160

[GO75] Angela Go, Michael Stonebraker, Carol Williams: An Approach to Implementing a Geo-Data System. Data Bases for Interactive Design 1975: 67-77

[GUTM84] Antonin Guttman: R-Trees: A Dynamic Index Structure for Spatial Searching. SIGMOD Conference 1984: 47-57

[HAMM81] Michael Hammer, Dennis McLeod: Database Description with SDM: A Semantic Database Model. TODS 6(3): 351-386 (1981)

[HEWI69] Carl Hewit: PLANNER: A Language for Proving Theorems in Robots. Proceedings of' IJCAI-69, IJCAI, Washington D.C.: May, 1969.

[KATZ86] Randy H. Katz, Ellis E. Chang, Rajiv Bhateja: Version Modeling Concepts for Computer-Aided Design Databases. SIGMOD Conference 1986: 379-386

[LAMB91] Charles Lamb, Gordon Landis, Jack A. Orenstein, Danel Weinreb: The ObjectStore System. CACM 34(10): 50-63 (1991)

[MCDO73] D. McDermott & GJ Sussman: The CONNIVER Reference Manual. AI Memo 259, MIT AI Lab, 1973.

[MCHU97] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, Jennifer Widom: Lore: A Database Management System for Semistructured Data. SIGMOD Record 26(3): 54-66 (1997)

[RICH87] Joel E. Richardson, Michael J. Carey: Programming Constructs for Database System Implementation in EXODUS. SIGMOD Conference 1987: 208-219

[ROWE79] Lawrence A. Rowe, Kurt A. Shoens: Data Abstractions, Views and Updates in RIGEL. SIGMOD Conference 1979: 71-81

[RUST74] Randall Rustin (ed): Data Models: Data-Structure-Set versus Relational. ACM SIGFIDET 1974

[SAME84] Hanan Samet: The Quadtree and related Hierarchical Data Structures. Computing Surveys 16(2): 187-260 (1984)

[SCHM77] Joachim W. Schmidt: Some High Level Language Constructs for Data of Type Relation. TODS 2(3): 247-261 (1977)

[SKAR86] Andrea H. Skarra, Stanley B. Zdonik, Stephen P. Reiss: An Object Server for an Object-Oriented Database System. OODBS 1986: 196-204

[SMIT77] John Miles Smith, Diane C. P. Smith: Database Abstractions: Aggregation and Generalization. TODS 2(2): 105-133 (1977)

[SNOD85] Richard T. Snodgrass, Ilsoo Ahn: A Taxonomy of Time in Databases. SIGMOD Conference 1985: 236-246

[SPON84] David L. Spooner: Database Support for Interactive Computer Graphics. SIGMOD Conference 1984: 90-99

[STON75] Michael Stonebraker: Implementation of Integrity Constraints and Views by Query Modification. SIGMOD Conference 1975: 65-78

[STON76] Michael Stonebraker, Eugene Wong, Peter Kreps, Gerald Held: The Design and Implementation of INGRES. TODS 1(3): 189-222 (1976)

[STON83] Michael Stonebraker, Heidi Stettner, Nadene Lynn, Joseph Kalash, Antonin Guttman: Document Processing in a Relational Database System. TOIS 1(2): 143-158 (1983)

[STON86] Michael Stonebraker, Lawrence A. Rowe: The Design of Postgres. SIGMOD Conference 1986: 340-355

[STON90] Michael Stonebraker, Lawrence A. Rowe, Michael Hirohama: The Implementation of Postgres. TKDE 2(1): 125-142 (1990)

[TSIC76] Dennis Tsichritzis: LSL: A Link and Selector Language. SIGMOD Conference 1976: 123-133

[WONG79] Eugene Wong, R. H. Katz: Logical Design and Schema Conversion for Relational and DBTG Databases. ER 1979: 311-322

[ZANI83] Carlo Zaniolo: The Database Language GEM. SIGMOD Conference 1983: 207-218