Incremental Database Systems: Databases from the Ground Up

Stanley B. Zdonik
Department of Computer Science
Brown University
Providence, RI 02912

Abstract - *This paper discusses a new approach to database management systems that is better suited to a wide class of new applications such as scientific, hypermedia, and financial applications. These applications are characterized by their need to store large amounts of raw, unstructured data. Our premise is that, in these situations, database systems need a way to store data without imposing a schema, and a way to provide a schema incrementally as we process the data. This requires that the raw data be mapped in complex ways to an evolving schema.*

1. Introduction

There has been much discussion lately about application-specific databases such as scientific, geographic, and multimedia databases. This paper argues that there are significant commonalties in the requirements of these applications, and that these commonalties can be addressed by a major evolutionary step in the design of database management systems.

Database systems have evolved significantly over the last several decades, but they reflect assumptions about the way data is modeled and stored that make them much too inflexible for this broad class of new applications. Thus, this paper examines some of these assumptions, and briefly suggests ways in which they could be relaxed. It is important to realize that we envision this as an evolutionary step that would become a part of existing technology.

Object-oriented databases [ZM90] have been successful largely because of their approach to type extensibility. Application-level types can be defined as data abstractions with arbitrary new interfaces. These interfaces are cleanly separated from their implementations. New polymorphic types can be related to each other through the subtype relationship as long as the interfaces of these types obey well-defined compatibility rules.

These features should be retained in database systems of the future. We need to ask, however, if the basic object-oriented paradigm, as we currently know it, is powerful enough to support the applications of the next decade. This paper explores application areas that stress our current technology, and, therefore, provide a seed for future thought and design.

This research was supported in part by the Advanced Research Projects Agency under ARPA order numbers 1133 and 8225 and administered by Office of Naval Research under contracts N0014-91-J-4085 and N00014-91-J-4052 respectively.

SIGMOD /5/93/Washington, DC,USA
© 1993 ACM 0-89791-592-5/93/0005/0408...$1.50

2. The Problem

One of the most fundamental assumptions of current database technology is that the first step in constructing a database is the design of a schema. The schema specifies the data structures that can be stored in the database and provides a vocabulary for all interactions with the database. Without the schema, the database is useless.

Once a schema has been defined, the database can then be populated with data. Thus, in the normal database view, the schema comes first and the data comes second. This paper challenges that assumption.

When a new object is created, the system allocates space in the persistent store to hold the data associated with that object. This allocation follows a few simple rules as described in the internal schema, another predefined view of how data is to be handled. The data is typically organized into one or more contiguous chunks each of which is subdivided into smaller regions or fields. Responsibility for allocating and managing this storage lies within the system.

There are many data handling problems that arise out of a need to gather raw data and process it incrementally over time. At the time of capture, this data has little or no known form, and there is no opportunity to transform it in any way. Often, the data is received from some other system or sensor at a very high rate, making it impossible to process at the time of capture (e.g., scientific data). In other circumstances, the data is captured in a simplified form because little is known about it (e.g., text or video). These applications will be discussed in detail in the next section.

Of course, this is not to say that this data does not contain structure. There may be a great deal of interesting substructure. Many of the applications that work with this data are responsible for determining this substructure. Discovering additional substructure presents an opportunity to generate metadata or schema. In other words, for this style of database, objects proceed from amorphous data streams to much more highly classified entities.

The challenge, then, is to permit raw data capture in whatever form is available and then to create mappings from this data to a more organized schema. These mappings will often be different from instance to instance with storage that may be non-contiguous and overlapping.

3. Example Applications

In this section, we will discuss several application areas that seem, on the surface, to be quite unrelated. In many respects, though, these applications share important characteristics. All of them require an ability to deal with very large volumes of information, but beyond this, they share a property that we call *incrementality*.

Incrementality refers to the ability to start with low-level data and map it to schema-level types a little at a time. The schema and the data will exist before the connections between them are created. Furthermore, these connections can be made more specific as more is known about the data.

An important example of this class of database arises out of **scientific applications** [FJ90, SO84]. Consider a satellite that is sending weather maps to a monitoring station. Image enhancement techniques might be applied to the data as it arrives, but it is stored as a large bit map without any real information concerning its contents.

Now, suppose that there is an application program that is able to perform feature extraction on this image. Suppose that it is able to identify and categorize various kinds of storms. While the weather map is an object, each storm might be considered to be an object as well. The storms are embedded inside the weather map. The fact that this weather map contains one or more storms would be stored as a part of the weather map object which might cause it to become reclassified as an *inclement weather map*. Each storm might also contain some substructure. For example, if the storm is a hurricane, the storm object might have an eye, a size, and a location.

It should be noted that there is no requirement that the objects that are discovered in the weather map be disjoint. Other feature extractors could identify weather fronts that intersect several of the storms. The embedded objects do not necessarily partition the weather map nor do they form a hierarchy.

The *weather map* type might contain some constraints regarding cloud motion that is based on the normal position of the jet stream. A new weather map or series of weather maps might cause the feature extraction application to determine that the jet steam has moved to a position that would not be allowed by the constraints in the current schema. The application should suggest a change in the schema based on this new information.

Financial applications also have many of the same characteristics. Consider a stock trading application. The data that is collected comes from the stock exchange in the form of a time series of prices. At the time the data is gathered, there is little more known about it; however, at some time in the future, analysts will run many statistics programs on this data to discover relevant features. Perhaps an analyst will look at graphs of statistical trends to identify patterns that have been seen in the past. These patterns might suggest additional categorizations for a stock. For example, it might become apparent that a particular stock is under-priced.

These applications are also inherently multidimensional. They share with scientific applications the need to support the discovery of interesting dimensions. The data begins as one-dimensional time series of price information, and later might need to be organized along a second dimension that captures the sales volume as well.

Multimedia applications [SZ87] are also a fertile ground for the need for incrementality. Multimedia systems of the future will allow users to build documents that have components that are drawn from many diverse media types. For example, text, graphics, audio, and video will be mixed freely in unpredictable ways. It must be possible to relate objects of these types to each other and to identify subobjects from larger objects.

In the latter case, we could have a portion of a musical composition captured from a digital recording. We might want to identify interesting musical phrases that are to be treated as objects in their own right. Moreover, for each of these phrases, we might want to run a signal processor over them to separate each of the instruments. The division of the musical composition into phrases is simply a new partition of the original data, while the separation into voices requires computing new signals from the old. This might loosely be considered a partitioning of the signal in the frequency domain.

This implies that, in this kind of database, derived objects need to be supported as naturally as they are in a spreadsheet. If we create new objects for each of the voices, we would want to keep track of the dependency on the original signal. If the original signal is edited in one of the voices, we would expect the corresponding voice object to change. The ability to process this information in arbitrary ways is very much like what was described in the previous paragraphs.

In a multimedia environment, we would also need to embed **graphics and video applications**. These applications are data intensive and share the need to incrementally define new object boundaries over their data. Suppose that the database stored a *movie* object as a sequence of *frame* objects. The application of the *play* method to a movie object causes the database to deliver the frames one at a time. This will work if the bandwidth of the network that connects the client workstation to the server is fast enough to handle this data rate.

Now suppose that the network load begins to increase to the point where there is no longer enough available bandwidth to transfer the movie. In this case, the system must decide between sending frames slower than the required rate or sending incomplete frames. If smoothness of motion is important to our users, it might decide to send every other scan line of each frame, thereby reducing the bandwidth requirement by half. The images would have less definition, but the smoothness of the motion would be preserved.

This would not be possible if the system could not impose a new template on the frame objects. The system would need to create a new movie object made up of half-frame objects, each of which is created by referencing every other line in a full-frame object. This requires that the half-frame object, then, not be a single contiguous chunk.

For **text**, there are many examples in which embedded objects arise. Suppose that we have a structured document that is composed of chapters each of which is composed of sections, and each section is, in turn, made up of paragraphs. A text formatting program might impose other object templates over this existing structure. For example, it might create *page* objects that extract the paragraphs and pieces of paragraphs that would be printed on a given page.

As a final example, **heterogeneous databases** [MH92] share certain characteristics with the other applications that we have described in this section. The integration of multiple heterogeneous databases requires mapping the schema of the component databases to a single global schema. Unlike the above scientific or financial applications, the component databases already come with a schema. However, in many cases it is easier and more efficient to "ignore" it from the point of view of the mapping.

Current approaches to heterogeneous database integration involves mapping the schema of the component databases to a common, global schema. By taking this approach, the resulting system must perform an extra level of mapping for each component database. It would be much more efficient to simply provide mappings from the underlying file structures to the global schema. If incremental database systems have a rich set of mapping tools, this would be feasible. In this way, the heterogeneous database problem is similar to scientific databases in that the data is captured without a schema. The mapping to a schema happens latter.

The above applications suggest that there is a wide class of application areas in which data is captured in raw form, and application programs explore and discover things about this data. Feature extraction and financial analysis programs are examples of these applications.

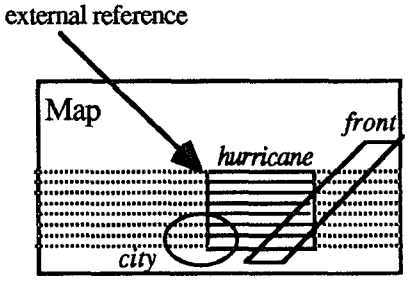

Figure 1: A Weather Map with Embedded Objects

4. What is needed?

Database systems have been very successful in practice. In standard applications, schemes can be created and storage patterns can be determined before the data is collected and the application programs are built. We will call this approach *top-down databases* since the initial activity involves creating high-level abstractions and then populating these abstractions with lower-level data.

We are entering an era in which data is being created and stored in large volumes long before we know what it contains or what we are going to do with it. It is, therefore, necessary to think about database systems that can handle this kind of data and make it possible (maybe even easy) to manipulate it flexibly after it has been collected.

There is a need to extend top-down databases with the ability to handle data from the *bottom-up*. In this way, we would start with the low-level data and implement mappings to higher-level abstractions or types. As a result, objects of a given logical type will not be stored uniformly since they were not laid out by a DBMS. There is no fixed set of implementation techniques to choose from as there are in conventional database systems. The next section discusses these ideas in more detail.

An incremental database system, then, must provide features that are not found in conventional databases. These features should be added to existing top-down, non-incremental database systems as opposed to having a system that only works bottom-up. In this way, the resulting system would allow users to work from either end toward a single common view.

The next few sections outline some of the features that must be added to make database systems more incremental.

4.1. Non-Contiguous Objects

Objects in any database system are implemented as if they were contiguous chunks of storage. This is in fact how they are typically stored. Sometimes it is possible to fragment a single object into multiple pieces each of which is stored in a separate file, but in these cases, all objects of the same type are fragmented in exactly the same ways. Moreover, this fragmentation is typically decided a priori by the DBA.

The applications that we have described have requirements that make this assumption difficult to work with. First, objects of the same type may be of very different sizes and may be fragmented into pieces that have nothing to do with the content of the object, but have more to do with where raw data landed on a mass storage device. Second, since we are allowing the discovery of objects within objects, there is no guarantee that the data that makes up a subobject is stored contiguously.

In the weather map example, a satellite image might be stored as a matrix of points. A hurricane might be identified as a rectangle within this matrix. This is depicted in Figure 1. The data that makes up the image of the hurricane includes the proper vertical fragments of a set of horizontal scan lines.

An incremental database requires a way to describe the mapping of such a collection of data blocks into a view that treats them as if they were a single, contiguous chunk. Contiguous chunks of memory are a simple and convenient abstraction with which to create higher-level structures. Therefore, typed objects would be built on top of a contiguous memory abstraction and oid's (i.e., object identifiers) would be assigned to these chunks whether they be real or virtual.

It would be useful to have a set of operators (an algebra) that would allow one to specify mappings of the kind that we encountered with the hurricane example. As an example of the character of these operators, consider the following expression that returns the hurricane object as a contiguous chunk. Note that this expression would be used to describe the mapping, not literally to construct the hurricane object.

Concat (
  Apply ($\{y_1, \ldots, y_2\}$,
    λ (i) **Project** (scanline (wm, i),x_1,x_2)))

This expression is a function of a weather map and a set of four coordinates delimiting the extent of a hurricane. **Project** is a substring operation that extracts the bytes between and including x_1 and x_2 from the sequence of bytes specified by the first argument *wm*, in this case, a weather map. **Apply** applies a function to a set and returns the set of the results. Here, the first argument is an ordered set of integers that identify the scan lines in which the hurricane exists. **Concat** takes a set of sequences and produces one long sequence. **Scanline** is a method of the type *WeatherMap* that returns a single scan line as specified by the second argument.

Expressions of this form need to be compiled into data structures called *mapping descriptors* that describe the embedded object. A reference to one of these objects would be translated through the appropriate descriptor. For example, if an application program tried to access the first 1000 bytes of the hurricane, the system would compute which scan lines were needed and how to transfer the result of any operation on those scan lines onto the physical storage.

Further applications may discover embedded objects in the hurricane such as the eye of the storm. This would result in a mapping descriptor for the eye in terms of the hurricane which is a virtual contiguous object. The system would then have procedures for combining mapping descriptors into a single mapping descriptor for the eye in terms of the weather map.

Notice that the operators **Project** and **Apply** that are used in the above example are operators for ordered types. In particular, they are defined over one-dimensional ordered types, in this case the address space of RAM. The weather map and the hurricane are both mappings from a one-dimensional address space to a two-dimensional abstract object. Ordered and multi-dimensional types occur frequently in these applications. If we are going to produce new mappings from ordered types to other ordered types, we will need to enrich our type models, query algebras, and optimization techniques to deal adequately with them.

4.2. Embedded objects

A standard assumption in database implementation requires objects to not overlap. This simplifies the implementation of many database features such as concurrency control, index maintenance, and object movement. However, in the applications that we are discussing here this is often not realistic.

Again, consider our weather map example. The weather map is an object with its own assigned oid. However, as feature extraction programs are run on it, new objects may be discovered (e.g., a hurricane). These objects are just that - objects with their own identity.

The standard way to handle this situation would be to copy the data out of the weather map to a separate hurricane object. There are several problems with this approach. First, the relationships between the hurricane and other features in the map become disconnected. If we later wanted to ask what cities the hurricane was near, the hurricane would have to be relocated within the map. Second, for big objects, this could consume unreasonable amounts of storage. Third, it opens up all the classic problems associated with copies: e.g., coordinated update and concurrency control.

Many systems store headers at the beginning of each object. The header contains information about the type, length, and version of each object. While a type implementation with headers should not be precluded, it cannot be required either. The reason for this is that the applications in question require that there be embedded objects. The hurricane is a fragment of the weather map. No header was present when the map was created and no header may be stored there after the fact.

These embedded objects must be first-class. Any other object must be able to refer to them as they would any other top-level object. In order to support this, pointers constructed out of oid's might be dereferenced through an object table that contained the mapping descriptors. These mapping descriptors would be interpreted relative to the beginning of the containing object.

There are potential problems with embedded objects if they are allowed to overlap. If they are updatable, then we must ensure that a change to a contained object is viewed as a change to the container from the point of view of the concurrency control system. Since there can be multiple containers, the system must find each of the containers to propagate the fact that these other objects changed as well.

4.3. Coordinate Systems

Much of the data that we have been describing can be conceptualized as existing within some coordinate system or frame of reference. For the map example, the coordinate system is the natural two-dimensional coordinates of a flat space. For the financial example, one possible coordinate system is the one-dimensional coordinates of time, while another is the two dimensional space of (time, company).

Often, an application requires more than one coordinate system to be active at the same time; moreover, it must be possible to add new coordinate systems to an existing data set. It must be possible to move back and forth freely from one coordinate system to another. In some sense, the mappings that we have been discussing represent exactly this behavior at a lower level. Storage is a one-dimensional coordinate system that is mapped to the two-dimensional coordinate system of the weather map.

A query might produce different results in different coordinate systems. As a simple example, consider the query that asks for the average salary of some department at year 7. Clearly, this answer would depend on the definition of the origin on the time line. If we are measuring time since the company began, we would get the value on the seventh year of operation, while if this is years AD, the query would attempt to get the answer in biblical times.

The database should provide support for translating between these spaces. If a query in coordinate system C_X contains a nested query in coordinate system C_Y, then the system should do the necessary translation and return the answer in the coordinate system of the outermost query, C_X.

While multidimensional coordinate systems are very much like compound keys, the notion of coordinate system translation is not normally an operation done on keys. Changing key values by applying coordinate transformations is useful in our target applications as long as an object keeps track of the coordinate system that it is currently being viewed in.

Another aspect of handling data within coordinate systems is the matching of coordinate values. Consider a clinical trials database that contained patient data over time to monitor the effects of drug testing. While each measurement of, say, a patients temperature will have an exact time coordinate associated with it, for comparison, we might want to compare patient A's temperature at 1:00 pm with patient B's temperature at 1:30, because they were both the second measurement of the day. The difference in their times is not significant for this comparison. Here the question becomes how do we define rules for doing this kind of imprecise matching of coordinates.

4.4. Breaking Abstraction

Data abstraction and type extensibility are fundamental contributions of the object-oriented paradigm. Incremental databases would certainly want to retain the ability to extend their types in this way. Standard object-oriented wisdom requires that the data abstraction never be violated. That is, once the programmer encapsulates this representation by the method definitions, no other user or code may have access to this data.

In an incremental database, it may be necessary to expose the representation in order to build new abstractions or views above it. It will certainly be the case that the builder of an abstraction cannot anticipate all of the functionality that may lie hidden in the data for a given object. Therefore, it must be possible to extend the abstractions from the ground up.

Of course this violation of the encapsulation principle would have to be conducted in limited and safe ways. Only schema-constructing programs or privileged users would need to do this. Encapsulation largely protects applications from changes to the implementation, thus, programs that were written previously would still use the old interface that must still be available.

4.5. Object Type Migration

Database systems typically assume that an object's type is known at creation time. The type of an object is not allowed to change over time. Correctness concerns motivate this approach to database type systems, but it has been pointed out in other contexts [Zd87] that this assumption is too strict for many cases involving long-lived data.

This is true of our incremental database environments as well. It must be possible to adjust the type of an object as we learn more about it. When this happens, other objects or parts of the system that have dependencies on the previously known type must be adjusted, if necessary, as well. This seems to suggest that a data model based on conformance [BH86] is more appropriate.

In an incremental database we must allow objects to change their types and let the schema evolve at the same time [Zd87]. If we adopt a conformance-based type model, as objects change their type they will automatically become instances of other types, as specified by their new interface.

In this type of database, we would also need to categorize these objects into a rich lattice of predicate-based sets (i.e., classes) much as is provided by the SDM [HM81]. That is, as more is known about an object, it might enter additional sets or

categories. The discovery of additional behavior (i.e., properties or methods) or new values for known properties could select the object for membership in other sets. For example, once a hurricane is discovered in a weather map, the size and shape of the swirling clouds could cause it to be categorized as a type 3 hurricane.

4.6. Lower quality objects

When an application program requests an object, the database system dutifully returns exactly that object. It may return it in pieces, but it nevertheless will make the entire object available to the application. In applications that have very high bandwidth requirements, it might be reasonable for the database system to decide that it is not possible to deliver the requested objects in their entirety, but, rather, to return some approximation or partial object instead.

Consider the example described above of a multimedia system that must play a video. In a heavily loaded network, the database server might decide to send half frames consisting of every other scan line. This requires additional processing at the client to provide full frames by doubling each scan line. Type-specific cooperation between distributed components is important to achieve acceptable performance.

5. Conclusions

There is a great need for database systems that are considerably more fluid. The data that they contain must be able to take whatever shape (i.e., schema) is required. The data must often be collected first and analyzed second. The data will likely contain subobjects that are discovered after it is collected. The database system must allow these subobjects to be identified *in vitro* and treated as first-class objects without forcing the data to be copied. As these objects are discovered, the database management system must be able to accommodate them into the existing type system.

There is a great deal of similarity between this approach and database views [AB91, HZ90, Ru92, SL91]. This paper argues for complex ways to map stored data to abstractions. In this sense, we are talking about views. In fact, the whole point of discovering more about objects is to extend their views. All objects (with the possible exception of the raw data) are views.

There are, however, some significant differences between incremental databases and, say, relational views. These include:

1. *No single model or language in which views are defined.* The language will need to match the characteristics of the application.
2. *No single level over which views are defined.* There can be no assumptions that all data is describable from a single simple viewpoint (e.g., relations).
3. *Mappings are over ordered and multidimensional data.* For example, since RAM addresses are ordered, creating mappings over bitmaps involves operators for single-dimensional structures.
4. *Mappings are constantly being remapped as knowledge evolves.*

There is also some parallel between incremental databases and knowledge mining in databases [AG92, HC92, MK92]. Incremental databases are, indeed, concerned with the discovery of additional knowledge about database objects; however, unlike the knowledge mining work, they cannot assume that there is a single, high-level schema that is to be investigated for patterns. instead, the problem here is how to remap low-level data to higher-level typed structures. The bulk of the work in the knowledge mining area is analogous to the feature extractors that run as applications on an incremental database system. These two areas could be quite synergistic.

Many of the problems described in this paper require solutions at the level of database implementation techniques; however, we have also pointed out some needed research at the level of the data model. Development of the kind of incremental, fluid system that this paper advocates will facilitate a number of important new application areas that need to collect and manipulate relatively unstructured data.

6. References

[AB91] Abiteboul, S. and A. Bonner, "Objects and Views", in *Proceedings of the 1991 ACM SIGMOD*, Denver, CO, May, 1991.

[BH86] Black, A., N. Hutchenson, E. Jul, H. Levy, "Object Structure in the Emerald System", in *Proceedings of 1st OOPSLA*, ACM, Portland, OR, September, 1986.

[FJ90] French, J., A. Jones, and J. Pfaltz, "Summary of the Final Report of the NSF Workshop on Scientific Database Management", in *SIGMOD Record*, Vol. 19, No. 4, pp. 32-40, 1990.

[HM81] Hammer, M. and D. McLeod, "Database Description with SDM: A Semantic Database Model", in *ACM Transactions on Database Systems*, Vol. 6, No. 3, pp. 351-386, September, 1981.

[HZ90] Heiler, S. and S. Zdonik, "Object Views: Extending the Vision", in *Proceedings of the Conference on Data Engineering*, Los Angeles, CA, February, 1990.

[MH92] Manola, F., S. Heiler, D. Georgakopoulos, M. Hornick, and M. Brodie, "Distributed Object Management" in *International Journal of Intelligent and Cooperative Information Systems*, Vol. 1, No. 1, March, 1992.

[MK92] Michalski, R, L. Kerschberg, K. Kaufman, and J. Ribeiro, "Mining for Knowledge in Databases: The INLEN Architecture, Initial Implementation and First Results", in *Journal of Intelligent Information Systems*, Vol. 1, No. 1, August, 1992.

[Ru92] Rundensteiner, E., "MultiView: A Methodology for Supporting Multiple Views in Object-Oriented Databases", in *Proceedings of the 18th VLDB* Vancouver, Canada, August, 1992.

[SL91] Scholl, M., C. Laasch, and M Tresch, "Updatable Views in Object-Oriented Databases", in *Proceedings of the Second Conference on Deductive and Object-Oriented Database Systems*, Germany, 1991.

[SO84] Shoshani, A., F. Olken, and H. Wong, "Characteristics of Scientific Databases", in *Proceedings of the 10th VLDB Conference*, Boston, MA, June, 1984.

[SZ87] Smith, K. and S. Zdonik, "Intermedia: A Case Study of the Differences Between Relational and Object-Oriented Databases", in *Proceedings of the Second OOPSLA*, Orlando, FL, October, 1987.

[Zd87] Zdonik, S., "Can Objects Change Type? Can Type Objects Change?", in *Proceedings of the First International Workshop on Database Programming Languages*, Roscoff, France, September, 1987.

[ZM90] Zdonik, S. and D. Maier eds., "Readings in Object-Oriented Database Systems", *Morgan-Kaufmann*, San Mateo, CA, 1990.