A DBMS for Large Design Automation Databases

Mark N. Haynie

Amdahl Corp.
1250 E. Arques M/S 213
Sunnyvale, CA 94086

Abstract

Large capacity Design Automation (CAD/CAM) database management systems require special capabilities over and above what commercial DBMSs or small workstation-based CAD/CAM systems provide. This paper describes one such system, **Tacoma**, used at Amdahl Corporation for the storage and retrieval of LSI and VLSI mainframe computer designs. Tacoma is based on the relational model with additional object-oriented database features.

1 Introduction

1.1 Supported Environment

Tacoma is a DBMS oriented to large system Design Automation (DA) problems. The system is in use by Amdahl Corporation for the design and tracking of LSI, VLSI and system level logic designs of large general purpose mainframe computers. The central database repository approach is ideal for the sharing of designs among nearly 150 logic design, design verification and design implementation personnel. There are also approximately 50 software engineers dedicated to the development and support of the DA environment.

Amdahl DA activities are performed under Amdahl's UTS[1] operating system, a Unix[2] System V based system. The design system runs on Amdahl 5890-300 and 5880 complexes, each a dual multi-processor. The systems are configured with 256 MB and 196 MB of main memory, respectively. This configuration delivers approximately 60 MIPS. The design database consists of up to 40 gigabytes of data. Silicon Graphics and Sun workstations support design entry, manual routing and other duties. Some logic design is performed on synchronous 3278 terminals attached to UTS.

[1] UTS is a trademark of Amdahl Corporation.

[2] Unix is a registered trademark of AT&T.

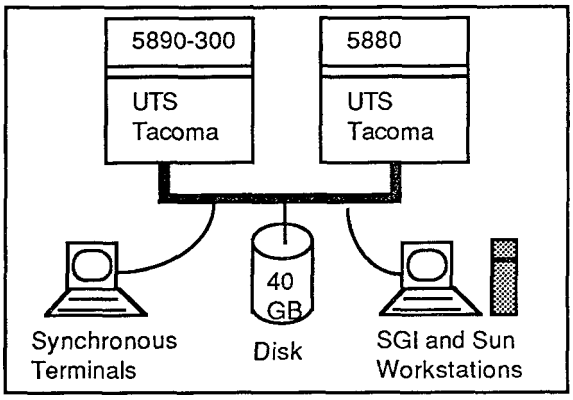


Fig. 1. Logic design and testing Environment.

1.2 DA Requirements

Design Automation at Amdahl is performed differently than CAD/CAM operations at chip design houses. Therefore, Amdahl DA database management needs are different from the classic custom-VLSI CAD shop. A number of factors make this so.

First, logic design is emphasized over physical design. ECL LSI gate array design allows logic designers to lay out their circuits using standard multi-gate macros, usually without regard to the physical restraints of the chip. Custom VLSI houses concentrate more on compacting the most onto chip real estate thereby requiring the designer to worry about the nitty-gritty layout details early in the design process. The result is that our DBMS must capture relatively simple interconnect information from the start. Physical information is added later by automatic partitioning and routing software. There can be several physical implementations of a single logic design.

Second, designers work on hundreds of chip, card, frame or system designs each with several revisions. The bill of materials management becomes critical. Each machine level is made up of hundreds of parts. The immediate availability of a version (not necessarily the most recent one) of each design is important to the system level integration simulations that are performed.

In some ways the above parallel the commercial DBMS requirements of data entry and inventory management.

However, an engineering DBMS must perform differently from its commercial counterparts [8].

First, instead of a large number of short transactions, DA databases usually see a few very long transactions. Some test generation, routing or speed analysis programs may take hours to complete. The user considers this as a single transaction. Further, there are even longer, *conversational transactions*[7] which may last several days. In this case, a designer begins working on a circuit and doesn't "check it in" until it is in a reasonable state of completeness (as defined by various consistency checking mechanisms). Schema definition and cooperative applications are important to making conversational transactions work in the Tacoma environment.

Second, since the transactions are necessarily long, massive amounts of data are read and written during a typical transaction. Some chip speed analysis models are on the order of tens of megabytes, card models in the hundreds.

Third, a complete revision history of the design hierarchy of a machine level must be maintained. During the life of a product, engineering changes require that logic cards returned from the field be reworked into the latest revision.

The DA decomposition paradigm is inherently object-oriented. Macros (elemental logic parts such as adders) are composed of transistors. Chips are composed of macros and so on. Within each decomposition layer the subcomponents are networked, between layers they are considered a hierarchy. If a designer wants to simulate his chip function, he need only simulate that chip and the macros it contains. If a designer wants to simulate his chip's interaction with another chip on the same card, he simulates at the card level. Running a simulation on chips between different cards requires complex card-stack simulations.

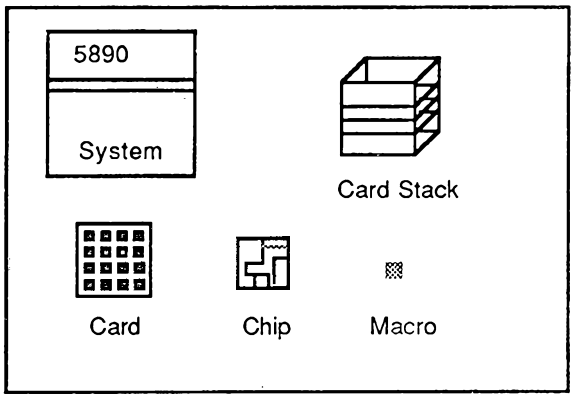


Fig. 2. Design Decomposition Hierarchy.

The decomposition hierarchy is orthogonal to the revision history mechanism. This, in turn, is orthogonal to the bill-of-materials hierarchy. Tacoma set out to meet the needs of representing these various views as close as possible to a flat file relational framework. What results is a relational data model with an object-oriented look.

2 Data Model

2.1 Dynamic Relations

The Tacoma data model used to represent these complex relationships is fully described in [2] and [3]. For purposes of discussion, we'll describe a few simple DA structures. Pins on a physical part are represented by pin number and position. Connection points (cps) are connections between logic components and pins or other components. Nets are "wires," a combination of connection points and/or pins.

Representing pin information of all parts in a machine within a single, physical table would be unmanageable. Instead, we break up what would be a very large pins table into a number of table instances. A table instance can be considered a complex object in the form of a relation. The table instance can be created, removed assigned or shared on an object basis.

A table instance can be thought of as being an attribute value in another table. This *relation as a data type* viewpoint is quite helpful. Since a relation data type can be part of any table, this provides arbitrary networking of relational instances. Because of this, Tacoma's complex objects can be made up of shared subobjects. This approach is more general and efficient either than simulating objects via encapsulated query procedures, as is proposed in [9] or linked records of a large relation as in [7].

Relation data type columns are strongly bound to the table they represent. In other words, if a column represents "nets" table instances, it can only hold "nets" table instances. Referencing a relation data type attribute value causes the table instance it represents to be referenced. These column data elements are called *relational object references*. The actual mechanics are illustrated in the next chapter on **SQL Extensions**.

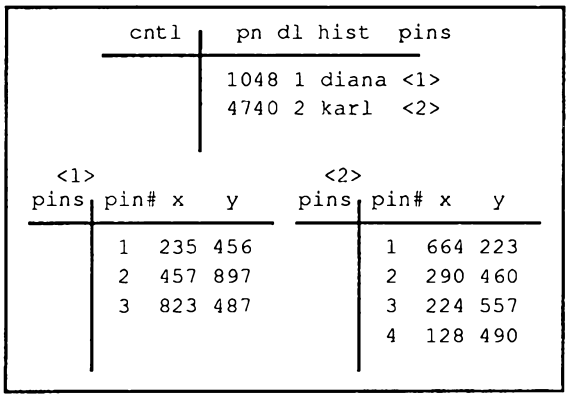


Fig. 3. Static control table and dynamic table instances.

Relational instances are called *dynamic tables* in Tacoma. Standard relations are also supported. *Static tables* are those with only one instance and require no other table to supply relational object references to them. A handle to a network of dynamic table types requires at least one static table to act as a directory, or a *control* table (see figure 3).

Treating relations as complex objects has several advantages. A known, standard interface deals with the objects as if they were simple data types. Further, the internals of the complex objects can be manipulated with the the same user interface used for simple objects. Tacoma is not a complete object-oriented database, however. It does not support user definable methods to deal with these objects as a whole, as others have proposed for CAD environments [1, 6]. It is possible to define operations as macros in the extended language if they can be wholly specified in terms of SQL statements.

Relation objects can be shared among a number of designs by inserting the references to them into columns declared in like ways. This allows true network structures to be created. Duplication of the contents of an object is not performed when references are inserted.

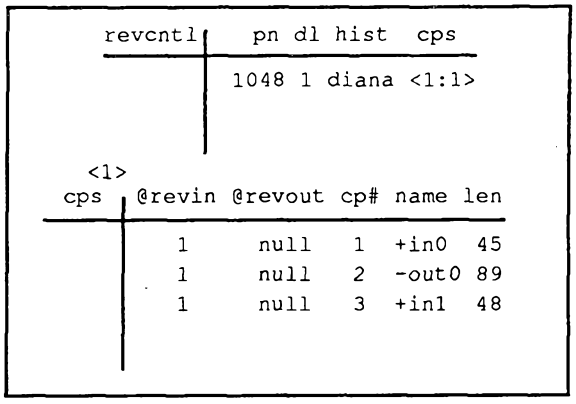
2.2 Revision Relations

The relational object concept allows an easier classification of complex objects represented in terms of other objects. However, it by itself does not solve the versioning problem inherent in computer aided design systems. Each "machine level" is made up of relation instances representing versions of several parts. In a practical sense, version 3 of any given part is very much like version 2 of that same design. To store them as separate instances would waste space as well as time by duplicating the data.

Revision relations[4] are dynamic relations which contain revision history information in the form of an original design and deltas to that design. Each new revision of a table is a newly defined view of that table.

All references to the revision history of a particular design are to the same physical table But, they include a revision number in the reference as well. The Tacoma query language interface adds predicates to select only those tuples representing the desired revision. Thus, to a typical application program (one wishing to view only a single revision at a time), it appears that it is viewing a separate table.

Data manipulation language operations such as insert, delete and update act upon revision relations differently from those of a non-revision nature. They preserve the prior versions of the relation by not altering the existing tuples in the table. Instead, the changed or deleted tuples are marked as being old and new tuples (the deltas) are inserted.



revcntl

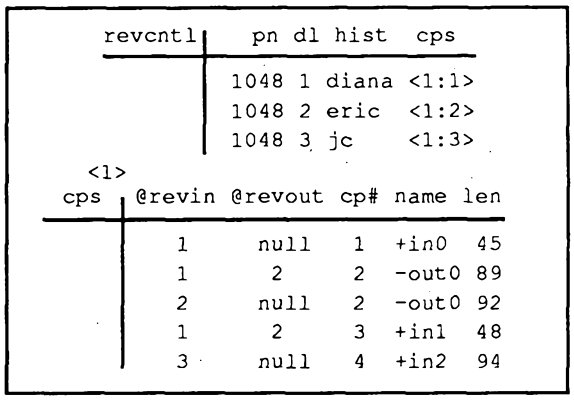
pn	dl	hist	cps
1048	1	diana	<1:1>

<1> cps

@revin	@revout	cp#	name	len
1	null	1	+in0	45
1	null	2	-out0	89
1	null	3	+in1	48

Fig. 4. Revision relation model after one revision.

Figure 4 shows the initial version of a table representing logic connection point information. This is represented in terms of a revision relation. The column `@revin` indicate at what revision a tuple was inserted. Column `@revout` indicates the revision a tuple was removed (if `null`, it is still current). These two columns are normally not printed by Tacoma "select all," but access to them may be gained by applications seeking changes between revisions.



revcntl

pn	dl	hist	cps
1048	1	diana	<1:1>
1048	2	eric	<1:2>
1048	3	jc	<1:3>

<1> cps

@revin	@revout	cp#	name	len
1	null	1	+in0	45
1	2	2	-out0	89
2	null	2	-out0	92
1	2	3	+in1	48
3	null	4	+in2	94

Fig. 5. Revision relation after updates.

In the second revision of the design, pin number 2 has been modified and pin 3 deleted. Pin 4 was added during revision three. Queries operating on the `pins` column of the `revcntl` table's second tuple would see a relation with two tuples, representing pin 1 and a modified pin 2. Note that multiple changes to a tuple during a single revision are not recorded.

Most design entry tools capture the revision information automatically: The logic gate editor displays the current design, and any changes to the design are tagged and entered into the database under a separate revision number for the design. Tools that need to distinguish between two different revisions can make use of the revision relation in a different way. They can look at the "all revision" view of the relation rather than at a single revision. The tools would use `@revin` and `@revout` explicitly to, say, rework old boards with new

designs by cutting and inserting discrete wires requires such a facility.

Revision relations provide a simple, single-leveled revision history model. To represent design alternatives using this facility, for instance, would require making copies of the single relation at the point the alternatives diverge. Separate revision relations would be controlled separately from that point for each alternative.

2.3 Dictionary Considerations

When using the data model provided by Tacoma, an extensive design control or bill of materials schema is required to capture the hierarchical decomposition as well as the history structure. A typical structure is broken into a two layer model shown in figure 6.

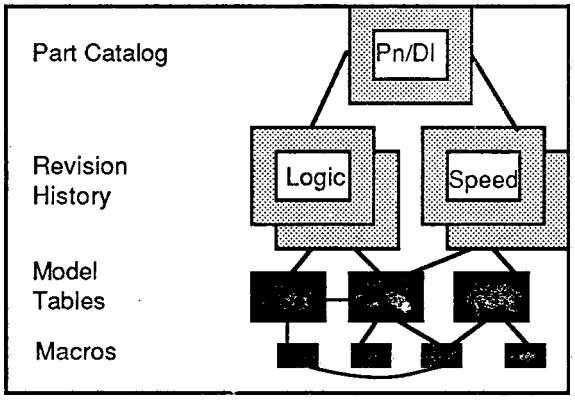


Fig. 6. Database Schema leveling using the Tacoma Data Model.

The broad-based part number and design level information is stored in a root static table. A series of revision history control tables make up the next level of control. Typically, one table type might exist for each subsystem or DA tool set supported by the database. Each tuple would contain relation object references for a number of *model tables*. These two layers make up the control information. Below that lies the "real data."

These model tables are defined in a format designed for a specific DA tool. They, in turn, are made up of references to low level relational objects at the macro level. These macros may be shared across many models.

The "control" tables are managed by Tacoma differently from data tables because they have different requirements. Operations on control tables are record oriented, whereas data tables are relation oriented. Thus, different locking, sharing and buffering criteria are performed between these table classifications. The details are discussed below.

Note that this directory structure unfortunately requires some amount of navigation of the database, as in a network data model.

For relational purists, the Tacoma data model provides for "flat file" representations of all relation instances and revision relations. A system table can be queried to obtain all instances of a table type, concatenating these tables together. Although such an operation would be unthinkable in a database of any size, this capability does allow the data model to pass certain relational DBMS benchmarks.

One final note on schemas. The system tables (recording column, index, table and privilege information) controlling a Tacoma database are themselves relational instances. This allows a great number of table types to be defined and many instances of those tables to be created without any one self defining table growing too large.

3 SQL Extensions

3.1 Standard SQL

Tacoma supports standard SQL in both an ad hoc query environment and application program host-language environment. The SQL language had to be extended in order to accomplish these relational data model extensions.

One area in which Tacoma is lacking is its fourth generation language and screen building areas. Tacoma is not designed for simple data entry and statistics gathering applications. DA applications contain sophisticated algorithms which cannot easily be modeled by such application building systems. Each DA tool provides its own user interface using common windowing and menu building code. This code is not part of the DBMS services.

Tacoma concentrates less on application building environments and more on data services: providing necessary primitives for extracting and updating massive amounts of DA data. This chapter gives a brief overview of those extensions which support the Tacoma data model outlined above.

3.2 Relation Instance Creation

Relation instance definition is accomplished by a `define` statement. It has the same syntax as the SQL `create` statement. The difference is that a statement such as,

```
define table pins(pin#(integer),
    x(integer), y(integer))
```

makes a data dictionary entry for the table without laying aside space for the physical table. This statement defines the dynamic instance seen in figure 3.

Relation instances cannot exist without having a reference to them contained in another table. These relation object references are contained in attributes defined with the `access` data type. These columns may be in static

(regular) or other dynamic (instance) relations. They are strongly typed to the relation instance type they will hold.

```
create table cntl(pn(integer),
    dl(integer), hist(char(5)),
    pins(access(pins))
```

In the above example, a `pins` column is a relation object of type `pins`.

A null value for a relation object reference means, as one would expect, that no relation object exists for that value. To create a new instance requires that the `create` keyword be assigned that column either with the `insert` or `update` SQL statement.

```
insert into cntl : <1048, 1, 'john',
    create>


update cntl set pins = create where
    pins = null
```

Relation instance creation requires a relation object reference to be created to that instance.

Two references to the same relation object can be made by assigning the same column value to different tuples.

```
insert into cntl : select 4742, dl,
    hist, pins from cntl where <pn,
    dl> = <4740, 1>
```

A relation object is deleted when all references to it are deleted, either by tuple removal or assigning the column the null value.

3.3 Relation References

The relation name alone is sufficient to reference static relations, just as in SQL. To reference a specific dynamic relation instance, it must be qualified with the relation object reference found in an `access` column. This is done with the help of the using clause. For instance, to retrieve the second instance of the pins table (figure 3) one would enter,

```
select * from pins using
    (select pins from cntl where <pn,
    dl> = <4740, 2>)
    where pin# >= 3
```

The inner query must select a table of the same type it qualifies. If an inner query selects no access attribute then no operation is performed on the table. The Tacoma data model specifies that when more than one relation object is retrieved by the inner query the objects are concatenated into a single table.

3.4 Revision Relation Creation

Revision relation instances are created in the same way as revision instances, with the `create` keyword. The initial version number 1 is assigned new revision relations. A set of built-in function calls operate on revision relation references to create further versions, or reveal their internal revision number. Usually, the internal revision number is unimportant: The schema would define an external revision number stored in a control table as the means of identifying the revision.

```
insert into revcntl : select pn,
    dl+1, hist, nextrev(pins) from
    revcntl where <pn, dl> = <1048,2>;
```

The statement will insert the same revision relational object reference, with a new revision number, into the `revcntl` table. Other functions are defined to find the current revision of any relational object reference, set the revision to a specific number or find the highest revision defined for that relational object.

Queries are modified by Tacoma to select only those tuples available at that level. To increase retrieval performance of queries involving no other keyed columns, indexes may be defined on the `@revin` and `@revout` columns.

Referring back to figure 5, suppose we wish to find all the 'in' connection points of part 1048 at design level (revision) 2. The predicates added by Tacoma are in italics. Note the use of regular expressions in Tacoma, as demanded by our Unix users.

```
select cp#, name, length    from cps
    using (select cps from revcntl
    where pn = 1048 and dl = 2)
    where name ~= '.in.*' and (@revin
    <= thisrev() and @revout >
    thisrev())
```

The `thisrev` function returns the revision currently associated with the relation object reference. Using an `allrevs` function on the inner `cps` reference selection would negate the effect of the added predicates. In this case, programs usually select the `@revin` and `@revout` columns to determine their own application-specific revision processing.

3.5 Environments

To ease the burden of navigation though the design and revision hierarchy, Tacoma has added a "current directory" concept. An application can build up an *environment* containing relational object references to any number of table types.

```
environment select cps from revcntl
    where <pn, dl> = <1048, 2>
```

In the above example, any further reference to the `cps` table (that isn't qualified with an overriding `using` clause) would use the relation objects selected during the environment statement. The relation object references are stored in a *private* table (a temporary table spanning transactions, but not sessions, which is user specific) `environs`.

3.6 Application Support

Most of the access to a Tacoma DA database is through C-language application programs. Additional support beyond what one would expect to find in SQL host language support includes special link list and array building primitives. Relation-at-a-time operations are stressed in this framework.

```
fill into ?(arr.(num%d, nam%s,
    len%d))[*] : select cp#, name,
    length from cps;
```

The `fill` statement will load the memory-resident array `arr` (with fields `num`, `nam` and `len`) with the contents of the current `cps` revision. The simple Tacoma precompiler requires that hints on field data types (e.g., `%d`) be given in the statement.

Remember that DA programs primarily transfer large amounts of data between memory resident data structures and disk resident tables. The `fill` statement is ideal for this type of access.

The `apply` statement transfers data in the opposite direction from the `fill`. An application specifies the key column(s) of the table and array or linked list of structures. The structure is assumed to contain an additional field `action` indicating the disposition (insert, update, delete or no action) of the structure element in the table.

```
apply ?(arr.(num%d, nam%d, len%d))[*]
    to cps(cp#, name, length) key cp#;
```

These primitives allow application designers to consider dynamic table instances as objects. They allow them to easily transfer the object from the database to memory and back. Acting upon the structure the object in the most convenient way (either with C statements or SQL).

4 Implementation

4.1 General

The Tacoma DBMS incorporates many of the concepts developed for relational systems over the past twenty years. B+ trees are used as Tacoma's prime indexing mechanism, queries are precompiled, etc.

This section discusses deviances from standard implementations of general purpose relational systems used in Tacoma.

4.2 Locking and Transactions

Availability of data is an important quality of systems which support long running transactions. Speed analysis or logic simulation programs may run hours and, during that time, need to see a consistent view of the database. Consistent to themselves, not necessarily globally consistent across all transactions.

Because a great number of table instances are read during the transaction, transactions cannot block on read locks of design tables they come in contact with. In addition, another user may wish to make a new revision of one of those tables. Even if it is currently being read. The writer cannot block until the read transaction completes.

If Tacoma employed the usual transaction locking control of most DBMSs it would be impossible to satisfy these conflicting requirements.

Instead, Tacoma uses a form of shadow buffering transaction management to allow multiple copies of a table to exist at the same time. Therefore, a reader and a writer of a single table can coexist without lock out. The view of a table by the reader is guaranteed to be consistent (non-changing) during the transaction. The writer of the same table instance sees a different view of the table, with its changes applied. Even if the writer commits before the reader, the reader will continue to see the view before the update of the table. A new reader, however, will see the newly committed version of the table.

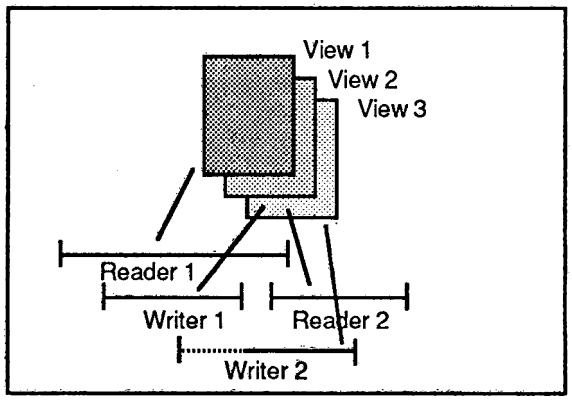


Fig. 7. Transaction time lines indicate how shadow buffering maintains different views of the same instance.

In figure 7, reader 1 will see view 1 of the table instance. The writer will see the same view initially, but any changes to the table are unseen by reader 1. When the writer commits, any new readers will see the new view 2 of the data. View 1 of the table is discarded once reader 1 commits. A new writer (2) would block until the first writer commits, then would begin working on new view of the table creating a third view. At one point in this example (near the completion of reader 1) there are three views of the table active.

Note that these views are hidden from the user and are independent of revision relations, which is an externalized, permanent versioning mechanism.

A disadvantage of this method of locking is that the granularity is limited to the table instance. The decreased overhead from the limited amount of locking and deadlock detection does increase performance.

Such a locking protocol would be useless for control or system (internal) tables, however. For example, if two distinct applications wished to create new instances of different tables and store their relational object references in a common control table, the second transaction would block until the first completed, even if they were working on different part numbers.

That is why Tacoma has a separate locking strategy for these types of tables. This protocol is a record-level locking more commonly found in other systems. It obeys the write-blocks-on-read philosophy found there. To define a table of this type, one would specify the `control` keyword on the `create` statement.

4.3 Server Layout

Because a shadow buffering mechanism is used instead of before and after image journaling, there is minimal data transmission between server and user space. Data table I/O can be performed directly from the user's server, eliminating I/O server bottlenecks. Control table I/O and journaling is handled by a central server. The result is that only the lock server has a significant amount of inter-process communication overhead.

4.4 Buffer Management

Tacoma buffer management is integrated with the UTS paging system. Buffers are allocated in virtual storage. Data buffers are not shared in a common pool, but are divided between users. However, the amount of virtual buffer space available is determined by the percentage of buffers allocated by UTS to real storage at any one time.

The buffering algorithm allows space to be adjusted dynamically, cognizant of other applications competing for real memory on the system. On the other hand, during periods of relative system inactivity, a transaction can acquire 200 MB of real memory buffer space, if needed.

This mechanism is sufficient for caching most complex objects referenced by an application during a transaction. Small objects are completely cached. Large objects are cached on a page basis.

4.5 Table Management

Dynamic table instances are managed as separate objects. The system catalogs for "table" and "index" information contain typical history and usage information. But physical object information is contained in a separate table instance. A simplified definition of two of these system tables is below. `Tables` is a static table, one `tinsts` table exists for each dynamic table (as referenced by the `tinsts` column within `tables`).

```
tables(tablename, definedate,
    defineuser, ..., tinsts)


tinsts(aa, references, credate,
    creuser, moddate, moduser, ...,
    phys-placement-info)
```

Relational object references are unique (within a table type) four byte instance (and possibly, revision) number. This is referred to as `aa` in the above table definition. The one level of indirection imposed by this scheme allows table instances to be moved by database utilities without requiring modification to relational object references.

A definition of a new table causes the insertion of a new `tables` tuple and an instance of `tinsts` to be created. Dropping a table causes the reverse to occur. The creation of a new instance will cause a tuple to be inserted in the corresponding `tinsts` instance. The `references` column contains the number of relational object references to that particular instance throughout the database. It is maintained every time a relational object reference is assigned or modified (by overwriting it with another or the null value). When the reference count drops to zero, the table instance is removed and the `tinsts` tuple removed.

5 Conclusion

Tacoma effectively manages Amdahl's diverse design automation design files. The current production databases hold approximately 200 relation object types. Instantiations of those types number in the tens of thousands. Eventually, the databases will grow to over 40 GB.

Tacoma has demonstrated that to build an engineering database management system one should start with sound fundamentals; namely, the relational data model and a standard interfacing language SQL. Extensions to the language and data model can then bridge the gap between the commercial environment (for which these systems were originally intended) and the engineering world.

References

[1] Bapa Rao, K.V. "An Object-Oriented Framework for Modelling Design Data." *Int'l Workshop on Object-Oriented Database Systems*, 1986 (ISBN 0-8186-0734-3).

[2] Haynie, M.N. "The Relational/Network Hybrid Data Model." *Proc. 18th Design Automation Conf.*, 1981, pp., 646-652.

[3] Haynie, M.N. "The Relational Data Model for Design Automation Databases." *Proc. 20th Design Automation Conf.*, 1983, pp. 559-603.

[4] Haynie, M.N. and K.W. Gohl. "Revision Relations." *IEEE Technical Committee on Database Engineering* Vol. 7, No. 2, June 1984.

[5] Katz, R.H, E. Change, R. Bhateja, "Version Modeling Concepts for Computer-Aided Design Databases." *ACM SIGMOD Conf. on Management of Data*, 1986 (SIGMOD Record Vol. 15, No. 2), pp. 379-386.

[6] Ketabchi, M.A. "Object-Oriented Data Models and Management of CAD Databases." *Int'l Workshop on Object-Oriented Database Systems*, 1986 (ISBN 0-8186-0734-3).

[7] Lorie, R.A., W. Plouffe. "Complex Objects and their use in Design Transactions." *ACM SIGMOD Conf. on Management of Data*, 1983 (SIGMOD Record Vol. 13, No. 4) pp. 115-122.

[8] Sidle, T.W. "Weaknesses of Commercial Data Base Management Systems in Engineering Applications." *Proc. 17th Design Automation Conf.*, 1980 pp. 57-61.

[9] Stonebraker, M., J. Anton, E. Hanson. "Extending A Database System with Procedures," *ACM Transactions on Database Systems*, Vol. 12, No. 3, Sept. 1987.