Daytona
And The Fourth-Generation Language Cymbal

Rick Greer

AT&T Laboratories – Research
Florham Park, NJ 07932
rxga@research.att.com

Abstract

The Daytona™ data management system is used by AT&T to solve a wide spectrum of data management problems. For example, Daytona is managing a 4 terabyte data warehouse whose largest table contains over 10 billion rows. Daytona's architecture is based on translating its high-level query language Cymbal™ (which includes SQL as a subset) completely into C and then compiling that C into object code. The system resulting from this architecture is fast, powerful, easy to use and administer, reliable and open to UNIX™ tools. In particular, two forms of data compression plus robust horizontal partitioning enable Daytona to handle terabytes with ease.

1. Daytona

The Daytona™ data management system is used by AT&T to solve a wide spectrum of data management problems. On the tiny end, Daytona provided the data manager for the DACS VI switch which only had 64MB of memory at the time. Since DACS VI used a real-time UNIX operating system, virtual memory could not be paged to swap disk. Consequently, the entire application, including the 15% that was allocated to the database, had to fit into the rather small amount of physical memory at all times. As another example, all of AT&T's (phone) call detail data (which represents most of the company's revenues) streams off the big 4E switches into a store-and-forward system called Billdats: Daytona provides the data management for Billdats II. At the high end, SCAMP, the Security Call Analysis And Monitoring Platform, uses the same Daytona as DACS VI to manage sequential and direct access to 9 weeks of all of AT&T's call detail data, comprising more than 10 billion records in a single table (plus four other large collections of call detail and summary data). SCAMP is used to analyze and detect fraud perpetrated against the company and to fulfill (often emergency) information requests from law enforcement. SCAMP handles more than 70,000 queries a month.

Daytona offers all the essentials of data management including a high-level query language, B-tree indexing, locking, transactions, logging, and recovery. Users are pleased with Daytona's speed, its powerful query language, its ability to easily manage large amounts of data in minimal space, its simplicity, its ease of administration, and its openness to other tools.

As will become apparent, in contrast to Daytona, other DBMS are much larger and tend to be closed systems (relatively speaking): they have chosen to implement their own (server-based) operating system, their own networking, user/login administration, performance monitoring, source code control and stored procedure management, and so on, and in some cases, even mail and cron job handling. Instead, Daytona reuses and leverages the software in its working environment. This makes it much smaller, simpler, and more open; in particular, Daytona users can work with Daytona using many of the tools they already have and know. Let's see how Daytona's low-overhead architecture and its query language lead to these benefits.

2. Daytona Architecture

Daytona revolves around Cymbal, its multiparadigm query language, which includes ANSI 89 SQL as a subset. Cymbal is processed by translating it completely into C. This translation process is more properly a compilation process of a 4GL language into a 3GL; to handle the last step, the system relies on a novel (and general-purpose) text generator called *backtalk*™ which walks a tree specification to generate a C program for the query, complete with a makefile. The resulting object modules are linked into an executable along with Daytona's own libraries, together with previously *backtalk*-generated object modules that provide the interface to user tables.

2.1 Four Modes Of Use

It is Daytona's unusual code-generation-based architecture that enables one and the same Daytona to handle with ease problems ranging from doing embedded data management on small real-time systems to managing the 4 terabyte SCAMP call detail warehouse (with its 16 gigabytes of memory and 32 250MHz processors). This architecture supports four modes of use:

Ad Hoc Queries — The simplest mode occurs when the Daytona user asks the system to translate, C-compile and run a Cymbal query.

Pre-compiled — Applications also have the option of pre-compiling parameterized queries. The application's GUI collects the parameters needed to invoke the previously compiled executables, whose output is returned to the GUI. This is the analog of SQL-based stored procedures.

Code Synthesis — The application writer can also use Daytona as a silicon programmer to generate C code according to high-level (meaning Cymbal) specifications. The corresponding object modules are then linked with the application's own object modules (and Daytona's libraries) into a single executable. In this synthesis of user and Daytona-generated code, user routines may call Daytona-generated routines which, in turn, may call user-coded C routines. This is an extremely efficient way to include data management in an application since Daytona is only a C function call away from the application code. DACS VI used Daytona in this way.

Generated Ad-Hoc — In this case, the application GUI actually generates ad-hoc Cymbal to express user requests, which are then processed as in the first mode. For example, SCAMP has a web interface wherein the CGI scripts invoke *backtalk* to process a collection of attribute-value pairs into Cymbal text which is then sent to the database machine for compilation and execution. The advantage here is that, on the spur of the moment, much more sophisticated Cymbal can be generated on the spot than could have been anticipated in a reasonably sized collection of pre-compiled parameterized queries. Modern computers are fast enough now that the generation, translation, and C-compilation times of these queries are perceived by the users to be acceptable; in fact, these set-up times are at least partially (if not many times over) offset by the speed of the resulting executable machine-code (as contrasted with the SQL interpretation process used by other DBMS).

An architectural footnote: there is a lot of tedious programmatic handshaking going on in typical C-embedded SQL situations where error codes are being checked, memory is being allocated to support converting from C-types to database types and so on. Daytona does not have C-embedded Cymbal; it avoids handshaking complications by offering instead two more attractive alternatives. On the one hand, Cymbal itself is intended to be powerful enough to express sophisticated queries at a simple and high level, where if ultimately necessary, C can be very easily called from that Cymbal. Secondly, code synthesis provides a simple and convenient way to access Daytona database functionality from application C code.

2.2 One Operating System Is Enough

Another implication of this architecture is that Daytona has no database server processes! In fact, it has no daemon processes of any kind. Every query executable is on its own to run and produce its answers. Most other DBMS have invested quite a bit of effort into creating database server processes, which provide many services including proprietary file systems, scheduling, caching, locking, parallelization, security, networking, and of course, query optimization and execution. Notice that with the exception of the last two, all of these services are provided by modern day operating systems to one degree/flavor or another. Instead of implementing another operating system, Daytona cuts out the middleman and in effect, uses the UNIX operating system itself as Daytona's server process. (Interestingly, Oracle™ has recently announced a product direction based on the opposite approach, which is, essentially, to get rid of the platform's operating system itself!)

Daytona's approach has several advantages. First, the same services are not being implemented twice and furthermore, one does not encounter the impedance/interference risks of "too many cooks spoiling the broth". Thus, Daytona is a far smaller DBMS than most. Consequently, it can fit on smaller machines and there is much less code to maintain (and correspondingly, much less opportunity for things to go wrong). As a result, Daytona has much easier OA&M (Operations, Administration, And Maintenance) requirements than most. As just one indicator, instead of the dozens of processes some other DBMS need to invoke on startup and keep healthy, Daytona has none at all; if your computer is up, Daytona is up.

Of course, even though Daytona doesn't utilize a database server process of its own, the user is more than welcome to write application-level servers using Daytona specifications. For example, SCAMP's data loading process is handled by a Cymbal program cloned into 5 concurrently running daemon processes. Application-level daemon processes are also necessary to handle application networking.

2.3 What The Environment Has To Offer

Daytona not only uses UNIX filesystems to store its data but the user even has the option of storing their data in the awk/Perl-compatible ASCII format of delimiter-separated fields, new-line terminated records. The use of this open format is reassuring to many users because they can actually see their data in their favorite text editor and because they can use standard UNIX tools on their data in the same form that is used by Daytona. Contrast this instead with storing data in a binary, proprietary format in 2K blocks, each containing a directory of pointers to slots within the block. (By the way, record deletion in Daytona is handled by overwriting the first byte with a delete byte and by using a free-list B-tree that enables best-fit recycled space to be used by subsequent inserts.)

Also, in terms of using what is available, modern day filesystems, such as that provided by Veritas™, are easy to use (no notion of primary allocation plus extents), either never need reorganization in practice or have sophisticated tools to do it, and offer many other features such as direct I/O, striping, mirroring, RAID-5, and logging (at the filesystem level).

Daytona performance monitoring is also straightforward: Daytona queries become processes which can be easily monitored with the performance tools that come with operating system. Shared text and shared libraries minimize the impact of multiple running processes on system resources. In fact, shared text is the Daytona analog of other DBMS' Shared SQL Areas.

2.4 Evolving To Meet AT&T's Needs

Over the past 15 years, Daytona has steadily evolved to meet the needs of AT&T projects. For example, in order to efficiently store the billions of records contained in the SCAMP data warehouse, Daytona's data format was extended to optionally include field- and record-level compression. At the field level, various tricks are used such as eliding default values and using special ASCII code bytes to represent pairs of digits. At the record level, a static dictionary of strings is computed for the table in question and the table is compressed record-by-record by replacing dictionary strings with 8 bit codes. (The advantage of compressing each record individually is that B-trees can still point to the first bytes of (compressed) records and consequently, there is no need to decompress an entire file in order to read out a particular record of interest.) Each of these compression levels has proven capable of 50% reduction. When

needing to store a terabyte, it is better to store 250 gigabytes.

The huge scale of SCAMP also made use of Daytona's robust horizontal partitioning feature which enabled a single table of call detail to be stored in 13104 files. Another example of Daytona evolution for SCAMP arose from the demands of SCAMP users to query the data as soon as possible. Since data is being appended continually, the indexing mechanisms were modified to support reliable but dirty (i.e., no-lock) reads at the same instants new records are being added. Many of Daytona's evolutionary steps can be seen in the development of its query language, Cymbal.

3. The Cymbal Query Language

Cymbal is a multiparadigm, fourth generation language that seamlessly integrates a procedural dialect with a first-order logic subset, ANSI 89 SQL, a sublanguage having to do with (declarative) set/list-formers and another one for describing database records. The procedural dialect includes assignments, conditionals, loops, function definitions, and compilation units called tasks (that are similar in function to Oracle PL/SQL packages). The first-order logic component is a domain calculus that employs the full assortment of connectives and quantifiers in unconstrained first-order combinations and is treated in a model-theoretic manner. The process of finding all values for the free variables in an assertion is handled in a backtracking manner reminiscent of Prolog but without using Horn clauses or being unduly sensitive to the order of the conjuncts. Cymbal's SQL dialect is implemented by translating it into the first-order logic component. A fluent Cymbal programmer freely intermixes all of the dialects in their query programs according to which is the most convenient, powerful, or concise at the time. They all get translated completely into C. Cymbal also supports a wide variety of types, function overloading, and basic type inference.

3.1 for_each_time Loops Bridge The Procedural And Declarative Realms

The bridge between the usual state-based semantics of procedural Cymbal and the stateless incremental satisfaction via backtracking semantics of logic Cymbal is the `for_each_time` loop. In the following example, the SUPPLIER and ORDER tables are joined on the Supp_Nbr attribute to produce the names of all Suppliers supplying more than 150 items.

```
set .qty_bound = 150;   /* <- just here for pedagogical purposes */
for_each_time [ .supplier, .qty ]
is_such_that(
    there_isa SUPPLIER where( Name = .supplier and Number = .supp_nbr )
    and there_isa ORDER where( Supp_Nbr = .supp_nbr
            and Quantity = .qty which_is > .qty_bound )
) do {
    do Write_Words( .supplier, .qty );
}
```

The first statement here, a `set` statement, assigns 150 to the variable `qty_bound`: in contrast to C, Cymbal distinguishes syntactically between a variable and its value. `.qty_bound` is the value of the variable `qty_bound`. The assignment statement reads: "set the value of `qty_bound` equal to 150".

The `for_each_time` statement itself consists of three parts: the first is a TUPLE that declares those free variables that will be getting values by virtue of satisfying the `is_such_that` assertion. Whenever a TUPLE of such values is found, the `do` group is executed using those values. (Class names like TUPLE are capitalized in Cymbal.)

Cymbal is intended to read fairly closely to (mathematical) English: the above `for_each_time` loop says in effect, "For each time the system can find values for the `supplier` and `qty` variables such that (there is a value for the `supp_nbr` variable such that) there is a SUPPLIER record whose Name attribute has a value equal to the `.supplier` and whose Number attribute has a value equal to `.supp_nbr` and there is an ORDER record whose Supp_Nbr value is equal to `.supp_nbr` and whose Quantity value is equal to `.qty` which is greater than `.qty_bound`, do write out a line consisting of `.supplier` and `.qty`

separated by spaces".

The there_isa constructs above are elements of a database record description sublanguage. Each Cymbal description gathers together in one place assertions about an object's class and other attributes. This is a more flexible representation system than fixed-arity logic predicates, which whenever used, would have to somehow mention all possible attribute-value pairs of an object, whether they were of immediate interest or not. [Buneman 94] reached a similar solution, although Cymbal does not distinguish syntactically between generating and test occurrences of a variable. This description sublanguage is far richer than it appears here but as is the case with all the examples in this section, space limitations preclude extensive discussion. In general, since Cymbal makes extensive use of the keyword-argument paradigm (with optional keywords and defaults), frequently there are many more options available than are actually chosen for explicit use in any given query.

Note that the supp_nbr variable is quietly and implicitly scoped to include both conjuncts. It is the means whereby the SUPPLIER table is being joined to the ORDER table on .supp_nbr. The first occurrence of a declarative variable in the parse tree for an assertion is taken to be its generating occurrence; subsequent occurrences are *ground* and constitute uses of the variable's value. So, the first occurrence of supp_nbr above serves to define its (finite) range of values and the second occurrence uses those values in a test on ORDER records. Disjunctions and if-then-elses are treated somewhat differently in that if a variable has a generating occurrence in one branch, then it must also have one in each of the other branches. The Cymbal optimizer feels free to move conjuncts around with the exception that it will not change the relative order of conjuncts that have generating occurrences for variables.

Note also the ease with which (outside) procedural variables like qty_bound are used within declarative assertions. Arguments to the PROCEDURE Write and its variant Write_Words are converted to strings automatically as needed. Cymbal *for_each_times* eschew any notion of database record cursor.

3.2 Boxes: In-Memory Tables

Recall set-formers from mathematics like { x^2 | x > 20 and x % 2 = 1 }. The Cymbal box, which is a generalization of the set-former concept, provides a kind of in-memory table (with indices) feature which, while of general use, was of particular interest to DACS VI with its in-memory-all-the-time application. Here is a query that caches in a box a 10% sample of maximum size 100 from the SUPPLIER table. The box consists of a list of supplier-city TUPLES that is sorted (and thus implicitly indexed) by the first and second components separately and independently. As one use of this cache, the user displays the suppliers and their cities taken from this box for cities that are nearby.

```
local: INT .max_samp_size

set [ .max_samp_size ] = read( from _cmd_line_  but_if_absent[ 100 ] );
set .tbl_sample_box =
    [ [ .supplier, .city ] :
        there_isa SUPPLIER from_a_bin_sample_of_frac .10
            where( Name = .supplier and City = .city )
        : with_sort_specs[ [1], [2] ]
          with_selection_index_vbl elt_cnt
          stopping_if( .elt_cnt > .max_samp_size ) ] ;

with_format _table_
do Display each [ .supplier, .near_city ]
each_time(
    .near_city Is_In [ "Camden", "Cherry Hill", "Philadelphia" ]
    and
    [ .supplier, .near_city ] Is_In .tbl_sample_box
)
```

There is a lot going on in this query. Firstly, the declarative assertion that characterizes the two-tuples of the box is the `there_isa` between the two colons. Notice the use of the `from_a_bin_sample_of_frac` keyword modifier to `there_isa` to specify the fraction of records to be included in the sample. The three keyword-argument pairs that serve to modify the definition of the box follow the second colon and end with the terminating square bracket. The box consists of all two-tuples that satisfy the assertion and the keyword argument conditions. The `with_selection_index_vbl` keyword indicates that the user-supplied variable `elt_cnt` will be set to the ordinal index of each box element as it is added to the box. Once that ordinal index exceeds `.max_samp_size`, box creation will stop. `.max_samp_size` is read from the command line by the TUPLE-valued function `read` when the executable is invoked, with a default of 100.

The next statement is a call to the keyword-argument-based procedure `Display` which instructs the system to display as a table the desired `supplier-near_city` pairs. (By the way, Daytona implements `Display` by using a `for_each_time`.) Since the second component of the subject tuple in:

```
[ .supplier, .near_city ] Is_In .tbl_sample_box
```

is ground and since the user requested a skip-list index on the second (or City) component of box-element TUPLES, the system will quietly use this index to look up the requested suppliers in *log n* time.

Consequently, we see that boxes are generalizations of what SETL [Schwartz 1986] calls set- and tuple-formers. While Cymbal pattern-matching is not yet as powerful as that described for *comprehensions* in [Buneman 1994], boxes do support indexing and sorting as well as other keyword-argument functionality which comprehensions do not. Of course, comprehensions are defined in a functional programming context whereas first-order logic is used to define boxes.

From a type standpoint, boxes are instances of a kind of omnibus collection type which includes lists, bags, and sets and where each box is effectively sui generis. In contrast to a type like a list in LISP, there is no simple, convenient type description that fits any and all boxes and can be compiled in advance in a library somewhere. Boxes can have duplicates or not, they can have a default ordering imposed on their elements or not, they can have TUPLES of varying sizes and types as elements, and they can be (multiply) indexed/sorted on any subsequence of components of those TUPLES. This is where the power of text generation comes in. When a Cymbal query uses a box, the type information for just that kind of box is generated at translation time for that particular box. On the basis of this typing, boxes can, for example, be passed safely as arguments to functions.

Boxes differ from record classes (Daytona's tables) in that boxes exist only in memory and are implemented using algorithms (including skip-lists) optimized for in-memory use exclusively. They are truly a programming language construct: they can be assigned to local or global variables or passed as arguments to functions. They are used primarily for in-memory sorting, duplicate elimination, and for supporting multiple *log n* access paths to (non-persistent) data.

3.3 Generalized Transitive Closure

Around 1986, a Daytona customer at AT&T Corporate Headquarters needed to freely navigate around on product and geographical hierarchies in order to do business modelling. A generalized transitive closure feature based on boxes was added to Daytona to support this. The transitive closure of a binary relation is essentially the union of all the results of joining the relation with itself arbitrarily many times, that process yielding null results after a finite number of iterations. Thus in order to specify a transitive closure, it suffices to specify just that base relation. Daytona generalizes this notion by allowing an arbitrary first-order assertion to characterize the defining binary relation and by supporting the use of additional modifying keyword-arguments.

Here is a complete Cymbal query that prints out in lexicographic order all NJ cities within 75 miles of a start point that have population > 10000.

```
define transitive PRED[
            TUPLE[ STR .to_city, INT .cum_dist ],
            TUPLE[ STR .from_city, INT .prev_cum_dist ] ] :
    Is_A_Big_Close_NJ_City_Reachable_From
with_specs(
    stepping_with(
        there_isa ROAD_SEGMENT where(
            From = .from_city
            and To = .to_city and Distance = .dist )
        and there_isa CITY where( Name = .to_city and State = "NJ" )
        and .cum_dist = .prev_cum_dist + .dist
    )
    selecting_if(
        there_isa CITY where( Name = .to_city and Population > 10000 )
    )
    backtracking_if( .cum_dist > 75 or Candidate_Selected_Before )
    sorted_by_spec[ 2, 1 ]
)

do Write( "Enter root city:  " );
set [ .root_city ] = read_line( );
do Display each[ .city, .distance, .path ]
each_time(
    [ .city, .distance ] Is_A_Big_Close_NJ_City_Reachable_From [.root_city, 0]
    with_path_vbl path
);
```

This is a fairly complicated transitive closure query to discuss right off the bat but hopefully, the essence of it, if not a full understanding, can be conveyed here. First, all Cymbal transitive closure is captured in a transitive PREDICATE definition. The `stepping_with` assertion here is the assertion with four free variables that serves as the binary TUPLE relation basis for defining the transitive predicate Is_A_Big_Close_NJ_City_Reachable_From. Each of the two TUPLE arguments that satisfy this PREDICATE is taken to be a node in an implicitly defined graph. The `stepping_with` assertion defines the base edge relationships among the nodes. So, the idea is to express how to generate from the current TUPLE node the next TUPLE nodes in a depth-first search of the graph. Notice how the root node `[.root_city, 0]` is specified in the `Display` portion of this query, after `.root_city` is read from *stdin*.

Examining the `stepping_with` assertion more closely, ROAD_SEGMENT is a table whose rows express the existence of a road between two points along with the length in miles of that segment. The CITY `there_isa` requires that the depth-first search stay in New Jersey, assuming that the atomic nature of ROAD_SEGMENTS is enough to assure that themselves. The third conjunct in the `stepping_with` assertion defines the distance in miles for the current path from the root, which is then used in the `backtracking_if` assertion to cause backtracking from the current path, if it gets too long. Also, if the current node has been selected before, then backtracking occurs in order to avoid infinite loops. The `selecting_if` assertion characterizes which of the visited TUPLES is to be selected for inclusion in the PREDICATE. These selected TUPLES are stored in a box, which in this case, is sorted on increasing distance from the root and then alphabetically by city name. The `with_path_vbl` keyword causes Daytona to keep track of a printable representation of the current path from the root, which can then be Displayed by means of the user-supplied variable `path`.

3.4 Arrays

While the skip-list algorithms employed by boxes for indexing provide welcome sorting and duplicate elimination capabilities, they are just not as fast as hashing. For that reason and also for notational

convenience, Cymbal provides both conventional and associative arrays to implement discrete functions in memory. Conventional arrays map TUPLES of INTEGERS to scalars or TUPLES. Each of the array dimensions is a finite lattice expressed by Cymbal interval syntax, which is illustrated by the right-hand-side argument of the following `Is_In` satisfaction claim:

```
.x Is_In [ 1 -> .y+.z by 4 ]
```

This is the lattice of INTEGERS beginning at 1 and increasing by 4 until exceeding `.y+.z` . The default value for the `by` keyword is `1`. When appropriate, conventional ARRAYS are very efficient in terms of space and speed of access.

However, there are many occasions when it is convenient to map TUPLES of any kind to scalars or TUPLES of any kind. Such maps are, of course, associative arrays. In contrast to many associative array implementations, the values of a Cymbal associative array may be TUPLES as well as being scalars. This provides considerable economy in storing and finding multiple pieces of information about given objects. In particular, it provides an efficient and flexible way to implement group-by queries, although in a low-level, procedural way, i.e., for each record scanned, compute one or more values which identify the group membership of that record and then update appropriately one or more cumulative aggregate statistics associated with that determined group.

Indeed, the following query defines the `qty_stats` associative array that maps supplier_number-date pairs to a pair consisting of the total number of orders associated with that domain pair and the total quantity ordered with those orders.

```
local: TUPLE[ INT .cnt, INT .qty ] .qty_stats[ INT, DATE ]
    = { ? => [ 0, 0 ] }

for_each_time [ .supp_nbr, .date, .qty ]
is_such_that(
    there_isa ORDER where( Supp_Nbr = .supp_nbr
        and Date_Recd = .date and Quantity = .qty )
) do {
    set [ .u, .v ] = .qty_stats[ .supp_nbr, .date ];
    set .qty_stats[ .supp_nbr, .date ] = [ .u+1, .v+.qty ];

/* This vvvvv is the equivalent of the preceding two steps.
    set .qty_stats[ .supp_nbr, .date ] = [ $#1+1, $#2+.qty ];
 */
}
```

The `qty_stats` definition qualifier

```
= { ? => [ 0, 0 ] }
```

causes references to domain elements that don't exist to be created with the initial values of `[ 0, 0 ]`. If this qualifier were not provided, attempts to access non-existent array elements would result in fatal runtime errors. In the shorthand equivalent syntax given above for updating the array elements, the $ stands for the left-hand-side of the assignment and the # is the component-selection operator which selects out the appropriately numbered component from the tuple: expand according to these rules and the intent of the assignment will be clear:

```
set .qty_stats[ .supp_nbr, .date ] =
    [ .qty_stats[ .supp_nbr, .date ]#1+1,
        .qty_stats[ .supp_nbr, .date ]#2+.qty ];
```

The challenge here is to implement these array element update operations efficiently. For example, it is desirable to hash-search for an element of the array exactly once in order to update it, instead of naively doing the search every time the array element expression is encountered in the query. The above code is translated into equivalent Cymbal that makes use of what amount to be pointers or references to array

element storage locations to achieve this end. Here is what it looks like:

```
set .y = qty_stats[ .supp_nbr, .date ];
set ..y = [ ..y#1+1, ..y#2+.qty ];
```

In Cymbal terminology, `y` is a VBL VBL, i.e., a variable whose value is a variable. Note the absence of a dot before `qty_stats` in the first assignment. This means that the expression `qty_stats[.supp_nbr, .date]` is considered to be a variable whose value is denoted by `.qty_stats[.supp_nbr, .date]`. What this amounts to here is that `y` is pointing to the storage location associated with the value of `qty_stats` at `[.supp_nbr, .date]`. This constitutes the sole use of the hash table in this example. All subsequent computations are done referring directly to that hashed-to storage location. The key is to see that `..y` is read as *the value of the value of y*, which is a TUPLE. Clearly, users themselves are free to use these pointers as well in their Cymbal. Note that Cymbal achieves with `.` what C achieves with `*` and `&`. VBL VBLS are also behind Cymbal's implementation of call-by-reference.

Oracle™ PL/SQL has associative arrays but the domains are uni-dimensional and consist solely of integers.

3.5 Aggregate Functions

Aggregate functions in Cymbal are defined declaratively by operating on the values produced by satisfying assertions in all possible ways. The first assignment below employs the scalar aggregate function `sum`. The second one computes three aggregates in parallel as it scans the ORDER table once; note that the `each_time` assertion for the three aggregate functions has been factored out into one place.

```
set .tot_qty = sum( over .qty each_time(
    there_isa ORDER where( Quantity = .qty )));

set [ .tot_orders, .min_date_recd, .max_qty_ordered ]
    = aggregates( of [ count(), min( over .dr ), max( over .qty ) ]
        each_time( there_isa ORDER where(
                Date_Recd = .dr and Quantity = .qty ) ));
```

3.6 Group By

Here is an example of a group-by query expressed in Cymbal:

```
    in_lexico_order
do Display each[ .product, .month, .year, .tot_sales, .pct_of_yearly_sales ]
each_time(
    [ .product, .month, .year, .tot_sales, .pct_of_yearly_sales ] Is_In
        { [ .p, .m, .y, sum( over .sales ),
            100 * sum(over .sales)/ sum(over .sales grouped_by[.p,.y])
          ] :
              there_isa SALE where(
                  Product = .p and Date = .d and  Amount = .sales )
              and .y = year_of(.d) and .m = month_of(.d)
            : having( sum( over .sales grouped_by[.p,.y]) > 0 )
        }
);
```

Here box syntax is being extended: if (partial) aggregate function calls such as `sum(over .sales)` appear in the expression list for the box, then the other non-aggregate-based quantities such as `.p` are taken to be the group-by quantities (note that they do not have to be table column values: they can be computed). If the aggregate is intended to group by a subset of these group-by quantities, then that subset can be identified by the explicit use of the `grouped_by` keyword; else the full set of group-by quantities is assumed to identify the groups of records being aggregated over. The contents of the box of

course are TUPLES consisting of the quantities identifying the group along with the aggregate values computed for that group. Being part of a box, these TUPLES can be sorted and indexed and so on.

These examples give some idea of the flavor and power of Cymbal but they necessarily provide only an incomplete picture.

4. Conclusion

Daytona is a general-purpose data management system revolving around a multiparadigm 4GL called Cymbal. It has evolved over the years to play critical roles in AT&T production systems, most recently managing a 4 terabyte call detail warehouse. For more technical information on Daytona, see http://www.research.att.com/projects/daytona . Daytona can be obtained through Global Technologies, Ltd., an AT&T VAR. See http://www.gtlinc.com .

References

[1] GREER, R. L. (1999) *All About Daytona*. AT&T Labs Technical Memorandum .

[2] GREER, R. L., and D. G. BELANGER (1989). *backtalk: A Deparser Generator And Macro Processor*. AT&T Bell Labs Technical Memorandum .

[3] BUNEMAN P., L. LIBKIN, D. SUCIU, V. TANNEN, L. WONG (1994). *Comprehension Syntax* SIGMOD Record 23(1): 87-96

[4] SCHWARTZ, J. T., R. B. K. DEWAR, E. DUBINSKY, E. SCHONBERG (1986). *Programming With Sets - An Introduction To SETL* Springer-Verlag, New York, New York.