Journal of Intelligent Information Systems, , 1–18 ()

P2: A Lightweight DBMS Generator

DON BATORY AND JEFF THOMAS {batory,jthomas}@cs.utexas.edu
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

Editor:

Abstract. A *lightweight* database system (LWDB) is a high-performance, application-specific DBMS. It differs from a general-purpose (*heavyweight*) DBMS in that it omits one or more features *and* specializes the implementation of its features to maximize performance. Although heavyweight monolithic and extensible DBMSs might be able to emulate LWDB capabilities, they cannot match LWDB performance.

In this paper, we describe P2, a generator of lightweight DBMSs, and explain how it was used to reengineer a hand-coded, highly-tuned LWDB used in a production system compiler (LEAPS). We present results that show P2-generated LWDBs reduced the development time and code size of LEAPS by a factor of three and that the generated LWDBs executed substantially faster than versions built by hand or that use an extensible heavyweight DBMS.

Keywords: lightweight DBMS, extensible DBMS, GenVoca, P2.

1. Introduction

General-purpose DBMSs are *heavyweight*; they are feature-laden systems that are designed to support the data management needs of a broad class of applications. Among the common features of heavyweight DBMSs are support for databases larger than main memory, client-server architectures, and checkpoints and recovery. A central theme in the history of DBMS development has been to add more features to enlarge the class of applications that can be addressed. As the number of supported features increased, there was sometimes a concomitant (and possibly substantial) reduction in performance. A hand-written application that does not use a DBMS might access data in main memory in tens of machine cycles; a comparable data access through a DBMS may take tens of thousands of machine cycles. It is well-known that there are many applications that, in principle, could use a database system, but are precluded from doing so by performance constraints (Miranker, 90-91; Brant, 91-93).

Extensible or *open* database systems (Batory, 88; Carey, 90; Haas, 90; Stonebraker, 91-93; Wells, 92) promoted DBMS customization by enabling individual features or groups of features to be added or removed. Unfortunately, extensible DBMSs were basically customizable heavyweight DBMSs; their architecture and implementations (e.g., layered designs, interpretive executions of queries) imposed the onerous overheads of heavyweight DBMSs. While extensibility *can* improve per-

formance, it has been our experience that the gains are rarely sufficient to satisfy the requirements of performance-critical applications.

A *lightweight* database system (LWDB), in contrast, is an application-specific, high-performance DBMS that omits one or more features of a heavyweight DBMS *and* specializes the implementations of its features to maximize performance. Examples include main memory DBMSs (e.g., Smallbase (Heytens, 94)), persistent stores (e.g., Texas (Singhal, 92)), and primitive code libraries (e.g., Booch Components (Booch, 87)). Each of these examples strip features from a general-purpose DBMS (e.g., Smallbase removes the disk-resident database feature, Texas removes client-server architectures, and the Booch Components further strip checkpoints and recovery) and demonstrate the performance advantages gained by doing so. In principle, an application achieves its best performance when it uses a "lean and mean" LWDB that exactly matches its needs.

Because there are no formalizations, tools, or architectural support, LWDBs are hand-crafted monolithic systems that are expensive to build and tune. The challenge in building LWDBs stems from scalability: a "lean and mean" LWDB by definition supports m features out of a set of n features, where m is application-dependent and n is constantly growing (Biggerstaff, 94). Clearly, the number of unique combinations of features is exponential, and thus, building a *library* of LWDBs that implement unique combinations is both impractical and unscalable. We believe that the only way of economically producing LWDBs that exactly match application needs is via *generation*.

In this paper, we describe P2, a generator of LWDBs. P2 provides the architectural support to assemble high-performance LWDBs from component libraries. P2 users code their applications in a database programming language that is a superset of C. P2 automatically builds (generates) a custom LWDB by analyzing application code and by following user-specified directives that define the database features that are to be supported. P2 performs many optimizations at generation-time: it compiles queries, inlines code to manipulate indices, and partially evaluates code statically. These optimizations enable the performance of P2-generated LWDBs to be comparable or to exceed that of hand-written LWDBs.

A classical lightweight database application is the LEAPS production system compiler (Miranker, 90-91; Brant, 91-93). LEAPS produces the fastest sequential executables of OPS5 rule sets by relying on highly-tuned, complex, and unusual data management features. No existing DBMS provides the features and performance necessary for LEAPS: non-extensible heavyweight DBMSs lack certain features and performance, and extensible heavyweight DBMSs lack performance. Thus, prior to the introduction of P2, LEAPS relied on hand-coded LWDBs. In this paper, we present results that show P2-generated LWDBs reduced the development time and code size of LEAPS by a factor of three and that the generated LWDBs executed substantially faster than versions built by hand or using an extensible heavyweight DBMS.

2. Generating Lightweight Database Systems

The conventional approach to lightweight database system construction is fraught with problems. With a partial understanding of the work loads that a LWDB is to support, LWDB designers invent data/storage structures and algorithms that match the perceived need. Implementing the design is tedious, expensive, and time-consuming, as it often involves adapting, coding, and debugging well-known algorithms. Once completed, the LWDB is integrated with the target application to see how well it performs. Without exception, the anticipated work load is different than the actual work load, and thus some of the design decisions/features of the hand-coded LWDB are recognized to be sub-optimal. At this point, designers face two unpleasant options: either leave the LWDB as is, knowing that its performance could be improved, or redesign and recode the LWDB for yet another round of testing. Redesigning has the additional unpleasant side-effect that the interface to the LWDB may change, which in turn, would cause parts of the application that use the LWDB to be recoded.

There are two fundamental problems with this approach. First, LWDBs should not have ad hoc interfaces. A LWDB should provide a stable, well-designed interface that would permit applications to be insulated from changes in LWDB implementations. Second, there needs to be a way of reusing well-known algorithms, so that the rote tasks of adapting, coding, etc., can be largely avoided. These are the motivating objectives of P2. To accomplish them, P2 users follow a two-phase approach to the development of LWDBs and their applications.

The first phase is *application development*. P2 extends the C language with special data types (e.g., cursors and containers). LWDB applications are coded in terms of these data types *without regard to how these types are implemented*. This approach radically simplifies programming: application development using high-level database abstractions is substantially easier than using low-level, ad hoc interfaces of hand-crafted LWDB modules. In Section 2.1, we present the data model and embedded language of P2. In Section 4, we document the productivity gains by programming with P2 types.

The second phase of development is LWDB *feature specification*, i.e., how the features of a LWDB are declared and how implementations of the P2 data types are to be generated. *For P2, a lightweight database system for an application is the implementation of the P2 data types that it references.* We will see in Section 2.2 that both feature specification and data type implementations are accomplished by composing components from the P2 library. An important advantage of this approach is that it is possible to radically alter the implementations of cursors, containers, etc., of an application (via a recombination of components) to improve application efficiency *without* modifying application code. Thus, tuning P2 LWDBs is considerably simplified. We demonstrate the power of this capability in Section 4.

2.1. Phase 1: Application Development using P2 Data Types

The P2 data modeling concepts are rather conventional: a P2 *database* consists of one or more containers, where a *container* is a sequence of elements that are instances of a single data type. Container elements can be retrieved, referenced, and updated via *cursors*. Our choice of these abstractions was deliberate: we wanted the P2 API to be as familiar and easy to learn as possible to database programmers.

The P2 data language is a superset of C; cursors and containers are added as built-in parameterized types.[1] Containers are parameterized by the type of element that is to be stored; cursors are parameterized by the container to be traversed and optionally by a selection predicate and/or sort criterion. An abbreviated declaration of a container of `EMP_TYPE` instances and a cursor that references selected instances are given below. In general, P2 cursor and container types are first-class; they can be used like any C type. (Note that predicates in P2 are strings; attribute `A` of the element referenced by a cursor is denoted `$.A`. The `$` denotes to P2 the name of the cursor):

```
typedef struct { ... } EMP;         // C struct declaration
container <EMP> emp_container;      // abbreviated container declaration
typedef cursor <emp_container>      // cursor typedef declaration
  where "$.dept == 7 && $.age < n" // n is a user-defined variable
EMP_CURSOR;

EMP_CURSOR emp_cursor, *p, a[5];    // cursor declarations

int f(EMP_CURSOR *p) { ... }        // function with cursor parameter
```

P2 offers an (extensible) set of operations on cursors and containers. The code fragment below illustrates the P2 `foreach` construct, which is used to iterate over elements of a container. Once a cursor is positioned, the referenced element can be examined, updated, and/or deleted.

```
foreach(emp_cursor) {               // for each selected employee
  printf("%s",emp_cursor.name);     // print employee name
  if (emp_cursor.job == 7)          // if employee id is 7
     delete(emp_cursor);            // delete employee
  else                              // else
     emp_cursor.dept = 12;          // update employee
}
```

Composite cursors are used to retrieve tuples of elements produced by multicontainer retrievals. A composite cursor `k` is an n-tuple of cursors, one cursor per container to be joined. A particular n-tuple of elements $(e_1, e_2, \ldots, e_n)$ is represented by having the i-th cursor of `k` positioned on element e_i. By advancing `k`, successive tuples of a multicontainer join are retrieved. A composite cursor (`compcurs`)

declaration is given below that joins the `department` and `employee` containers. `d` and `e` are aliases for the cursors over the `department` and `employee` containers, respectively:

```
compcurs <d department, e employee> // composite cursor declaration
  where "$d.dept == $e.dept" k;

foreach(k) {                            // for each pair
  printf("(%s,%s)",k.d.name,k.e.name); // print department, employee
  delete(k.d);                          // delete department
}
```

Perhaps the most novel aspect of composite cursors is that P2 permits the elements referenced by a composite cursor to be updated. Unlike view updates (where changes are restricted (Keller, 82)), updates are unrestricted, but they may effect the tuples that are subsequently retrieved. For instance, once an element of a tuple is deleted, that element should not belong to any subsequently retrieved tuple. In the code fragment above, if tuples of `k` were computed eagerly (i.e., set-at-a-time), `k` might return tuples with deleted elements. Figure 1a shows an eager join returning `department-employee` tuples `(d1,e2)` and `(d1,d3)` after department `d1` has been deleted. (Clearly, modifying or deleting previously deleted elements is meaningless). Figure 1b, in contrast, shows a *valid* join which notes database updates performed since the last advance of `k`, and skips tuples containing deleted elements. P2 generates the code that supports valid retrieval semantics.

(a) tuples returned by eager join

(b) tuples returned by valid join

Figure 1. Eager and valid joins.

Tuple validation is specified through a `valid` clause predicate, which disqualifies tuples for retrieval. The following declaration and code fragment eliminates the problems of Figure 1a by only returning tuples with undeleted `department` elements. `deleted()` is a P2 operation that returns `TRUE` iff the specified element has been deleted.

```
compcurs <d department, e employee> // valid composite cursor declaration
  where "$d.id == $e.id"
  valid "!deleted($d)" v;
```

```
foreach(v) {                                // for each valid pair
  printf("(%s,%s)", v.d.name, v.e.name); // print department, employee
  delete(v.d);                              // delete department
}
```

Tuple validation is a general-purpose feature that is useful in graph traversal and garbage collection algorithms, where previously positioned cursors may suddenly find themselves referencing deleted elements, and automatic repositioning of cursors upon advancement is critical for algorithm correctness.

2.2. Phase 2: Feature Specification Using Component Compositions

Coding LWDB applications in terms of P2 data types is straightforward. The second phase of P2 application development is to define the features that the application's LWDB is to support and to declare how implementations of the P2 data types are to be generated. The key to any generative approach is to create a *domain model* of families of P2 data type implementations (i.e., families of LWDBs), where individual members of this family have a precise and unique specification in the model (Prieto-Diaz, 91). We used the GenVoca model to express our domain model of LWDBs (Batory, 92).

As a brief overview, the GenVoca model of software system generation was distilled from the experiences of building generators for the disparate domains of database management systems, communication protocols, avionics, file systems, and data structures (Batory, 88; Hutchinson, 91; Batory, 93; Coglianese, 93; Heideman, 94). The motivation for these generators was the scalability problem outlined in Section 1: customized software systems implement m features out of a possible n features. Rather than building an exponential number of monolithic systems that offer unique sets of features, one should build systems by composing primitive components that encapsulate individual features. Thus, by making feature combinatorics explicit, it is possible to describe vast families of systems with a relatively small number of components. In P2, a target LWDB is specified as a composition of P2 components.

The set of components that implement the same interface is called a *realm*. A realm is, in effect, a library of plug-compatible and interchangeable components. Among the realms of P2 are `ds` and `mem`. `ds` components export a standardized container-cursor interface. Among the components of `ds` are those that implement common storage structures (e.g., binary trees, doubly-linked ordered and unordered lists) and cursor-container mappings (e.g., free lists of previously deleted elements, sequential and random storage). `mem` components export standardized memory allocation and deallocation operations. Among its members are components that manage space in persistent and transient memory. A partial listing of `ds` and `mem` components are given below.

```
ds = {
  odlist[key, ds],                // key-ordered doubly-linked list
  bintree[key, ds],               // binary tree
  dlist[ds],                      // unordered doubly-linked list
  avail[ds],                      // free list of deleted elements
  mlist[key, ds, ds],             // multilist indexing
  predindx[pred, ts, ds],         // predicate index
  hpredindx[pred, ts, key, ds],   // hashed predicate index
  tlist[ts, ds],                  // timestamp ordered lists
  htlist[ts, ds],                 // hashed timestamp ordered lists
  malloc[mem],                    // heap storage
  array[mem],                     // sequential storage
  delete_flag[ds],                // logical element deletion
  ...
}

mem = {
  transient,                      // transient memory allocation
  persistent[file],               // memory mapped persistence
  ...
}
```

Note that components have two kinds of parameters: *realm parameters* such as `ds` and `mem` (i.e., parameters that are instantiated by components) and *nonrealm* or *configuration parameters* such `key`, `pred`, `ts`, and `file` (i.e., parameters that are instantiated by key field names, predicates, timestamp field names, file names, etc.). To illustrate their distinction, consider the component `odlist`, which encapsulates the concept of key-ordered linked lists. `odlist` has two parameters: a nonrealm parameter `key` and a realm parameter `ds`. The `key` parameter declares the key field of the list. The `ds` parameter indicates that `odlist` imports the `ds` interface. Other components are interpreted in a similar way.[2] Currently there are over fifty P2 components. LWDBs are defined by compositions of components, called *type equations*, that typically reference up to twenty components. We will illustrate a P2 type equation shortly.

A unique feature of P2 components (and GenVoca components, in general) is that they are *program transformations* that encapsulate consistent large-scale data and operation refinements. It is beyond the scope of this paper to explain the GenVoca methodology or to present an in-depth discussion of these concepts and their relationships; we will, however, illustrate the essential ideas with elementary examples.

A *large-scale transformation* is a program transformation that refines multiple data types simultaneously. All P2 components refine element, cursor, and container data types in a consistent manner. As an example, the `odlist` component transforms a container of elements into a container whose elements are linked together onto an ordered doubly-linked list.

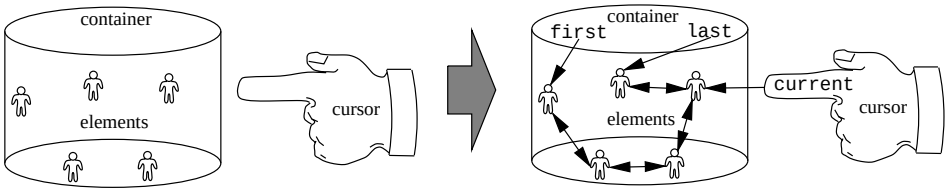

Figure 2. The odlist transformation.

`odlist` encapsulates the following data refinements:

- *element* types are augmented with `next` and `prev` pointer fields (for double-linking).
- *container* types are augmented with `first` and `last` pointer fields (for head and tail list accessing).
- *cursor* types are augmented with a `current` pointer field (to indicate the current element of the list).

`odlist` is a large-scale transformation because it automatically refines element, container, and cursor data types by adding new data members, algorithms, and optimizations (e.g., query optimization, code inlining, partial evaluation) for the ordered-list feature. So, the way to understand the `odlist` transformation is that it takes a P2 program (with cursors, containers, elements) as input, and produces a refined P2 program (with refined cursors, containers, elements) as output. By cascading transformations (i.e., composing components in type equations), implementation details of the target LWDB are progressively revealed.

Some simple P2 type equations are:

```
typex {
  T1 = odlist["age", delete_flag[malloc[transient]]];
  T2 = bintree["name", odlist["city", array[persistent["x"]]];
};
```

`T1` means that the elements of a container will be linked together onto a doubly-linked list ordered on field `"age"` (by `odlist`). List nodes will be marked deleted without reclaiming their space (by `delete_flag`) and will be allocated from a heap (by `malloc`) that resides in transient memory (by `transient`).

On the other hand, `T2` defines a very different storage structure. `T2` declares that the elements of a container will be linked together onto a binary tree whose key field is `"name"` (by `bintree`); binary tree nodes will be linked together onto a linked list ordered by field `"city"` (by `odlist`). List nodes will be stored sequentially (by `array`) in persistent file `"x"` (by `persistent`).

Type equations are declared via the P2 `typex` declaration; complete container declarations specify the type of elements to store *and* the type equation that defines the container/cursor/element implementations:

```
container <EMP> stored_as T1 // full container declaration
  emp_container;
```

As these examples suggest, P2 programmers are armed with a small handful of P2 components that can be composed in vast numbers of ways to produce large families of distinct LWDB implementations. This powerful feature allows P2 users to explore different LWDB's implementations easily by altering just a container's type equation and recompiling; no other source code modifications are needed. Further details about type equations and P2 components are discussed in (Batory, 93; Batory 94b-c).

3. The LEAPS Lightweight Database Application

The LEAPS production system compiler is a classical lightweight database application. LEAPS (*L*azy *E*valuation *A*lgorithm for *P*roduction *S*ystems) produces the fastest sequential executables of OPS5 rule sets (Miranker, 90-91). A LEAPS executable is a lightweight database application, because it represents its database of assertions as a set of containers, and because it uses unusual search algorithms and novel container implementations to enhance rule processing efficiency; no heavyweight DBMS offers the performance or features needed by LEAPS.

As a brief overview, OPS5 is a forward-chaining rule programming language (Cooper, 88). An OPS5 program is a set of rules; an OPS5 rule named `done` is shown below. It consists of a left-hand side of three condition elements, an arrow (`-->`), and a right-hand side with two actions.

```
(p done
  (context ^value done)
  (last_seat ^seat1 <seat>)
  (seating ^seat2 <seat>)
-->
  (write Yes we are done)
  (modify 1 ^value print_results))
```

Each *condition element* (CE) specifies a container and one or more selection predicates. Names in angle brackets (`<>`) denote variables whose values are to be instantiated during a search. The first CE of `done` defines the selection predicate over the `context` container:

```
context.value == 'done'
```

The next two CEs join the `last_seat` and `seating` containers by the equijoin predicate:

```
last_seat.seat1 == seating.seat2
```

The selection predicate of an OPS5 rule is the conjunction of the predicates of its CEs, which in this case is:

```
context.value == 'done' && last_seat.seat1 == seating.seat2
```

OPS5 execution follows a *match-select-act cycle*: rules whose predicates can be satisfied are identified (*match*), a satisfiable rule is chosen (*select*), and the actions of the chosen rule are evaluated by a tuple that satisfies the rule predicate (*act*). (The actions of the `done` rule print the string `"Yes we are done"` and update the `value` field of the selected `context` element to be `'print_results'`). This cycle continues until no rule can be satisfied.

LEAPS translates OPS5 rule sets into C programs (Figure 3). To implement LEAPS using P2, we wrote a translator RL (*R*eengineered *L*eaps) that converts an OPS5 rule set into a P2 program that embeds the LEAPS algorithms. The RL-generated P2 program is converted into a C program by the P2 generator, thus accomplishing in two translation steps what the LEAPS compiler does in one.

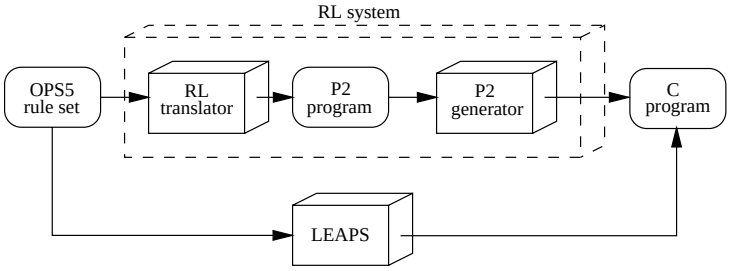

Figure 3. Relationship between LEAPS and RL.

3.1. Phase 1: Application Development

The LWDB applications produced by RL are very complicated. Every OPS5 rule set is translated into a set of containers, a composite cursor type for each rule, and the LEAPS algorithms that manipulate elements, cursors, and containers. The rule set containing the `done` rule would have at least the following P2 containers, where `RLx` is a LEAPS container type equation to be defined in Section 3.2.

```
container <CONTEXT>   stored_as RLx context;
container <LAST_SEAT> stored_as RLx last_seat;
container <SEATING>   stored_as RLx seating
```

The `done` rule itself would be translated into the following composite cursor data type:

```
#define done_query    "$a.value == 'done' && $b.seat1 == $c.seat2"
#define done_temporal "$a.ts <= gts && $b.ts <= gts && $c.ts <= gts"
#define done_valid    "!deleted($a) && !deleted($b) && !deleted($c)"
typedef compcurs <a context, b last_seat, c seating>
  where done_query " && " done_temporal
  valid done_valid   DONE_CURSOR_TYPE;
```

The containers to be joined by `done` are parameters to the `compcurs` declaration. The `done` selection predicate is expressed by `done_query`. This part of the translation is simple.

During rule set execution, it is possible that multiple cursors of `DONE_CURSOR_TYPE` may be active. Timestamps are used by LEAPS to achieve fairness—i.e., to preclude rules from being fired more than once by the same tuple of elements. LEAPS augments every element with a timestamp field `ts`, whose value indicates when the element was inserted. `done_temporal` is the temporal predicate used by LEAPS to accomplish fairness; `ts` is the timestamp field of an element and `gts` is a global timestamp whose value is determined by LEAPS.

When a rule is fired, a composite cursor is pushed on a stack, thereby suspending the execution of its joins. Only when the cursor is popped off and advanced are its joins resumed. During the time the cursor is on the stack, any or all of the elements that it referenced may have been deleted. Consequently, the cursor must be validated upon advancement. The `done_valid` predicate defines the valid conditions.

There are many other sources of complexity in LEAPS. For example, an OPS5 rule can have any number of negated condition elements. A *negated* CE is a predicate that disqualifies tuples of elements that satisfy the (positive) CEs of a rule. An unusual aspect of negated CEs is that their predicate is temporal; additional containers (called *shadows*) must be created to contain the elements deleted from non-shadow containers in order to evaluate negated CEs. As another example, LEAPS reduces string matching time by maintaining a symbol table (i.e., a container) so that element address comparisons can be used in place of expensive character-by-character string comparisons. Other details are explained in (Batory 94a).

3.2. Phase 2: Feature Specification

LEAPS algorithms require containers to be searched in timestamp order. Thus, all container storage structures used by LEAPS maintain timestamp ordering. The P2 `tlist` component maintains elements of a container on a time-stamp ordered list. (The P2 component `odlist` might also be used, where the key-field would be the element's timestamp field. `tlist` is preferred, however, as it has special optimizations that give it superior performance.)

Because rule sets are static, LEAPS takes advantage of the fact that it knows the complete set of predicates that will be evaluated during rule set execution. A

special storage structure, called a predicate index, is used to enhance rule processing efficiency. A *predicate index* is a timestamp-ordered list of elements that satisfy a given predicate; the predicate itself is over a single container and has no variables. The P2 component `predindx` implements predicate indices.

The reclamation of deleted elements in LEAPS is delayed until execution reaches a fix-point; the reason is that composite cursors (whose executions have been suspended) may reference deleted elements. For this reason, elements are logically (but not physically) deleted using the `delete_flag` component.

Finally, as the number of elements to be stored is unbounded, allocation of storage space must be done through a heap (using the P2 component `malloc`). Memory allocation in transient memory is accomplished using the P2 component `transient`; persistent memory allocation via memory-mapped I/O is accomplished using `persistent`.

An unusual, but critical, aspect of the LEAPS algorithms is its dependency on nested loop join algorithms. All other join algorithms (hash-join, merge-sort, etc.) create intermediate relations during join processing. The LEAPS implementors discovered the creation of intermediate relations to be a primary obstacle to fast rule processing.[3] Given this dependency, we discovered that we could improve the efficiency of nested loop joins by emulating hash joins. This was accomplished by replacing timestamp-ordered lists with hashed-timestamp ordered lists (component `hlist`) and predicate indices with hashed-predicate indices (component `hpredindx`). That is, rather than searching a time-stamp ordered list for elements with a given join key, we hashed on join keys to search a (small) bucket of timestamp-ordered elements. As we'll see in Section 4, using these structures yields a substantial improvement in LEAPS performance.

The general forms of the P2 type equations that we used to store LEAPS databases are `RL1`-`RL4`:

```
typex {
  RL1 = predindx[...tlist[delete_flag[malloc[transient]]]...];
  RL2 = predindx[...tlist[delete_flag[malloc[persistent["x"]]]...];
  RL3 = hpredindx[...htlist[delete_flag[malloc[transient]]]...];
  RL4 = hpredindx[...htlist[delete_flag[malloc[persistent["x"]]]...];
}
```

For each equation, there are zero or more predicate indices (or hashed predicate indices) maintained per container. `RL1` differs from `RL2` only in the transient or persistent storage of containers. `RL3` differs from `RL1` by replacing structures that don't use hashing with ones that do. `RL4` is the persistent storage counterpart of `RL3`.

4. Results

The LEAPS algorithms are notoriously difficult to understand. In interviews with the LEAPS development team, they felt that their expertise would enable them to

rewrite LEAPS in 2-3 months, whereas novices (us) would take at least twice that long to code (e.g., 6 months). It did take us several months to comprehend the algorithms, but only took us two months to code RL.[4] As supporting evidence, RL is less than 5K lines of C, `lex`, and `yacc`. LEAPS is four times larger—almost 20K lines: 10K for the basic compiler and another 10K for the run-time system included in all LEAPS-produced executables. Thus *for the LEAPS application and LWDBs, using P2 reduced the development time and code size by a factor of three.*

We discovered two reasons for this. First, P2 offers substantial leverage in developing LWDBs and their applications. P2 is currently 50K lines of code; it performs general optimizations that LEAPS experts had to hand-code into their compilers. Second, by far the most substantial productivity gain was using P2 data types to express the LEAPS algorithms. Although complicated, the LEAPS algorithms are elegant when expressed in P2. The P2 separation of LWDB implementation details from its client applications significantly reduced the complexity RL's development and the understanding, coding, and debugging of the LEAPS algorithms.

To help us evaluate the performance of RL/P2-generated programs, the LEAPS development team provided us with OPS5 rule set benchmarks: `tripl` (3 rules that output 3-tuples of numbers ranked in descending order), `manners` (8 rules that find seating arrangements with constraints), `waltz` (33 rules that define a 2-D line labeling program), and `waltzdb` (38 rules that define a more complex version of `waltz`). Each of these rule sets processed scalable input data sets; programs that generated these data sets were included with each rule set. The LEAPS development team also provided us with two versions of LEAPS: `OPS5.c` (a version that generates programs whose databases are main-memory resident (Miranker, 90-91)) and `DATEX` (a version that generates programs whose databases are disk-resident (Brant, 93)). `DATEX` databases are stored by Jupiter, the (heavyweight) Genesis file management system (Batory, 88). Thus, `OPS5.c` and `DATEX` provided us with an ideal opportunity to evaluate the scalability of P2: we could compare P2-generated LWDBs with both hand-coded main-memory LWDBs and a heavyweight extensible disk-resident DBMS. We accomplished this using the same P2 programs generated by RL, but swapping the type equations `RL1` and `RL2` (defined previously).

Prior to benchmarking, it was our goal to have RL/P2 generated programs have a performance within 10% of LEAPS. We expected to be slower because (1) P2 is a general-purpose tool, whereas LEAPS was hand-coded by experts, and (2) we converted OPS5 programs to C programs in two translation steps, whereas LEAPS accomplished this in one step (Figure 3).

Results of our benchmarking are presented in Figure 4.[5] Let's consider first the performance of `RL1` and `RL2`. In all cases, their performance exceeded that of `OPS5.c` and `DATEX`. `RL1` was typically *two times faster* than `OPS5.c`, while `RL2` was typically *fifty times faster* than `DATEX`.

`RL1`'s improved performance over `OPS5.c` was due to several reasons. First, P2 generated code is more efficient than that of `OPS5.c`; P2 performs optimizations automatically that are difficult, if not impractical, to do by hand. Second, expressing the LEAPS algorithms in terms of P2 abstractions clearly revealed some simple

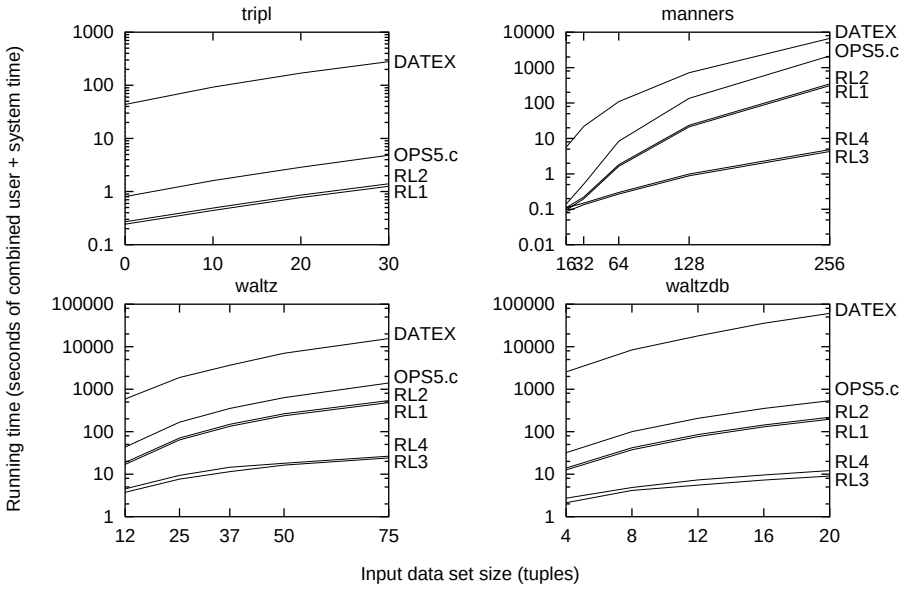

Figure 4. Results of LEAPS experiments.

optimizations that were otherwise obscured. Third, the design and implementation of `OPS5.c` was so complicated that it was necessary to replicate predicate indices for understandability; eliminating replicated indices was trivial in the RL/P2 version. Fourth, `OPS5.c` used a tagged type system and performed dynamic garbage collection unnecessarily. (This was an example of a LWDB being designed to meet perceived needs that never arose.) It was through our experiments with RL/P2 that the LEAPS implementors learned that garbage collection was unnecessary.

`RL2`'s improved performance over `DATEX` was due in part to the reasons cited over `OPS5.c`, but by far the most substantial gains came from eliminating the large overheads of heavyweight (extensible) DBMS construction. These included: layered software designs, interpretive execution of queries, buffer management, and general-purpose storage structures that were not as efficient as the LEAPS-optimized storage structures of `OPS5.c`. These overheads caused `DATEX` to be slower than `OPS5.c` by more than an order of magnitude, whereas swapping the `transient` component in `RL1` with `persistent` to produce `RL2` (i.e., replacing transient memory with memory-mapped I/O) reduced performance by only a few percent.

When we swapped `RL1` and `RL2` with `RL3` and `RL4` (i.e., when we used hashed structures instead of non-hashed), we observed an astounding performance improvement for three of the rule sets. For `manners`, `waltz`, and `waltzdb`, `RL3` executed over *an order of magnitude faster* than `OPS5.c` and `RL1`; `RL4` was *three orders of magnitude faster* than `DATEX`. The reason is simple: nested loops is an inefficient join algorithm; by emulating hash joins, we obtained big improvements in performance.

We did not achieve speedups for `tripl`, as `tripl` has only inequality-joins and thus hash-joins could not improve its performance.

We can make three important observations here. First, experimenting with very different LWDB implementations was effortless: all we needed to do was to alter type equations and recompile. Second, when needed components were absent from the P2 library (as was the case for `htlist` and `hpredindx`), it took us only three days to write them. We were able to reuse other components of the P2 library to minimize our coding efforts. In contrast, `DATEX` was a full rewrite of `OPS5.c` and took many months to complete. Third, P2 provides a technology by which customized LWDBs (i.e., customized type equations) can be generated *per rule set* to maximize performance; this capability is impossible with standard LWDB implementation techniques, including those used by LEAPS.

5. Observations, Insights, and Related Work

Database applications change over time. New or customized storage structures or query processing algorithms might be needed to enhance application performance. When new structures are needed, the DBMS must provide facilities for the orderly and efficient *migration* of data: i.e., the conversion of data stored in one structure to that of another.[6] Migration is a problem faced by all DBMSs, lightweight and heavyweight, but most lightweight systems don't support migration. When applications entirely "outgrow" the DBMS they currently use, a new DBMS must be selected, and applications must be ported to use the interface of that DBMS. This outgrowth problem is more common for LWDBs than heavyweight DBMSs. P2 offers important advantages for addressing these problems. First, P2's type equations greatly reduce the problem of selection, since they allow very high-level selection of DBMS features and implementation. Second, P2's realms greatly reduce the problem of porting, since they provide standardized interfaces which are largely independent of LWDB features and are entirely independent of LWDB implementation.

Building customized DBMSs from components has been an active area of research for over ten years. Genesis was the first "lego" technology for constructing heavyweight DBMSs (Batory, 88). It is our experiences with Genesis and the performance of heavyweight DBMSs that led us to develop P2; P2 relies on the same conceptual ideas but has a much more sophisticated and performance-driven implementation. The DMEA (*D*ata *M*anagement *E*xtension *A*rchitecture) of Starburst independently evolved many of the central features of Genesis: Starburst attachments are "layers" similar to Genesis and P2 components (Haas, 90). Each attachment encapsulates the data processing, query optimization, etc., algorithms associated with a (heavyweight) implementation of a DBMS "feature". It is this approach to encapsulation that offers the desired "plug-and-play" feature of attachments. So the central ideas on which P2 is based have had a long history of independent development and verification.

Extensible DBMSs have matured significantly beyond these early (late-1980s) efforts to become *universal servers*, i.e., heavyweight extensible systems that allow

extensions to the type system in order to support arbitrary data types, such as document, time series, image, and spatial data (Ubell, 94; Norman, 96). These extensions (called *DataBlades* by Montage, Illustra, and Informix) are often packaged as modules that users can plug-in to servers at run-time. To support arbitrary data types, universal servers allow the addition of base and composite types, functions, functional indices, access methods, and storage managers. These extensions allow very special-purpose, efficient algorithms, but universal servers themselves must be very general to support such extensions. For example, DataBlades require virtual dispatch of database functions, which adds overhead to all applications, even those that do not use the extensions (Norman, 96). The generative paradigm of P2 is general enough to permit similar extensions to the type system, but compiles away unnecessary overhead by making the extensions at compile-time, rather than run-time.

The generality offered by heavyweight DBMSs might be required for one-of-a-kind, legacy, and/or enterprise-critical applications. But for many mass-produced, embedded, and/or performance-critical applications, run-time efficiency is essential. LEAPS is a classical example where the few minutes P2 spends on optimization and generation is amortized by the vast performance improvement gained over the run-time generality of heavyweight DBMSs.

6. Conclusions

The data management needs of many applications are not met by conventional DBMSs: non-extensible heavyweight DBMSs lack certain features and performance, and extensible heavyweight DBMSs lack performance. What is needed are lightweight DBMSs, database systems that omit features of heavyweight DBMSs and that optimize the implementations of the supported features to maximize performance.

In this paper, we described P2, a lightweight DBMS generator that combines a relatively traditional data model and embedded data language with a powerful model of software system construction (GenVoca). This combination of technologies enables P2 to generate efficient LWDBs automatically from a simple set of specifications (e.g., GenVoca type equations). We reported results of several experiments on a very complex LWDB application (LEAPS) that showed P2 generates efficient code, offers a powerful form of LWDB customizability, and substantially simplifies the development of LWDB applications as well as the tuning of LWDBs by enabling different algorithms/features to be tried merely by plugging and unplugging components.

We are currently extending the capabilities of P2. New components will offer additional DBMS features (e.g., concurrency control, client/server architecture, set-oriented queries) as well as a greater variety of implementations of existing features (e.g., t-trees (Lehman, 86) and sort-merge joins). This will allow us to use P2 to generate LWDBs for a broader range of applications.

We believe lightweight DBMSs have a wide applicability and practical importance. We feel that our work with P2 demonstrates that generating lightweight DBMSs is feasible. In the hope that P2 will benefit other researchers, we provide the source code and documentation for P2 via anonymous ftp and the web:

`ftp.cs.utexas.edu:/pub/predator/`
`http://www.cs.utexas.edu/users/schwartz/`

Acknowledgments

We thank David Brant at The Applied Research Laboratories at the University of Texas for supporting our research. We thank Dan Miranker and Bernie Lafaso for their patience in explaining the LEAPS algorithms to us, and we thank Vivek Singhal for his insightful comments on earlier drafts of this paper.

Notes

1. We chose C, rather than C++, as the host language for P2 initially for convenience because our target applications were written in C. Also, it was not clear to us when we began the P2 project in 1991 how successful LWDB generation would be. C++ would, in retrospect, have been a better host language.
2. Components that export and import the same interface, such as `odlist`, are called *symmetric*; most P2 (and Gen Voca components, in general) are symmetric. Symmetric components can be composed in virtually arbitrary ways; this feature significantly enhances the scalability and composibility of GenVoca components.
3. Remember that the execution of many composite cursor retrievals can be suspended during rule execution in LEAPS. If cursors used temporary files for intermediate join results, the space and time complexity of LEAPS would greatly increase. Computing joins in a "lazy" manner gives LEAPS its name and execution efficiency.
4. The un-optimized algorithms of LEAPS were coded in a week. P2 was being written at the time of our RL work; the remainder of the two months included the time spent waiting for P2 to be debugged and the time needed to add the myriad optimizations to RL that LEAPS uses (Batory, 94a).
5. The timing results presented here were obtained on a SPARCstation 5 with 32 MB of RAM running `SunOS 4.1.3` using the `gcc2.5.8` compiler with the `-O2` option. Similar results have been obtained on other systems.
6. In P2, data migration is equivalent to type conversion: i.e., translating data one of type to that of another.

References

D. Batory, J. Barnett, J. Garza, K. Smith, K. Tsukuda, B. Twichell, and T. Wise, "Genesis: An Extensible Database Management System," IEEE Transactions on Software Engineering, November 1988, pp. 1711-1730.

D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components," ACM Transactions on Software Engineering and Methodology, Vol. 1, No. 4, October 1992, pp. 355-398.

D. Batory, V. Singhal, M. Sirkin, and J. Thomas, "Scalable Software Libraries," ACM SIGSOFT, December 1993.

D. Batory, "The LEAPS Algorithms," Department of Computer Sciences, University of Texas at Austin, Technical Report 94-28.

D. Batory, J. Thomas, and M. Sirkin, "Reengineering a Complex Application Using a Scalable Data Structure Compiler," ACM SIGSOFT, December 1994.

T. Biggerstaff, "The Library Scaling Problem and the Limits of Concrete Component Reuse," Proceedings of the Third International Conference on Reuse, November 1994.

G. Booch, Software Components with Ada, Benjamin/Cummings, 1987.

D. Brant, T. Grose, B. Lofaso, and D. Miranker, "Effects of Database Size on Rule System Performance: Five Case Studies," Proceedings of the 17th International Conference on Very Large Data Bases (VLDB), 1991.

D. Brant and D. Miranker, "Index Support for Rule Activation," ACM SIGMOD, May 1993.

M. J. Carey, D. J. DeWitt, G. Graefe, D. M. Haight, J. E. Richardson, D. T. Schuh, E. J. Shekita, and S. Vandenberg, "The Exodus Extensible DBMS Project: An Overview," in D. Maier and S. Zdonik (editors), Readings on Object-Oriented Database Systems, Morgan-Kaufmann, 1990.

L. Coglianese and R. Szymanski, "DSSA-ADAGE: An Environment for Architecture-based Avionics Development," Proceedings of AGARD 1993.

T. Cooper and Nancy Wogrin, Rule-based Programming with OPS5, Morgan-Kaufmann, 1988.

L. Haas, W. Chang, G. Lohman, J. McPherson, P. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita, "Starburst Mid-Flight: As the Dust Clears," IEEE Transactions on Knowledge and Data Engineering, March 1990, pp. 143-161.

M. Heytens, S. Listgarten, M. Neimat, K. Wilkinson, "Smallbase: A Main-Memory DBMS for High-Performance Applications," HP Labs Technical Report, December 1994.

J. S. Heideman and G. J. Popek, "File-System Development with Stackable Layers," ACM Transactions on Computer Systems, February 1994.

N. Hutchinson and L. Peterson, "The x-kernel: an Architecture for Implementing Network Protocols," IEEE Trans. Software Engineering, January 1991.

A. Keller, "Updates to Relational Database Through Views Involving Joins," in P. Scheuermann (editor), Improving Database Usability and Responsiveness, Academic Press, 1982.

T. Lehman and M. Carey, "Query Processing in Main Memory Database Management Systems," ACM SIGMOD, June 1986.

D. Miranker, D. Brant, B. Lofaso, and D. Gadbois, "On the Performance of Lazy Matching in Production Systems," Proc. National Conference on Artificial Intelligence, 1990.

D. Miranker and B. Lofaso, "The Organization and Performance of a TREAT Based Production System Compiler," IEEE Transactions on Knowledge and Data Engineering, March 1991.

M. Norman and R. Bloor, "To Universally Serve," Database Programming and Design, Vol. 9, No. 7, July 1996, pp. 26-35.

R. Prieto-Diaz and G. Arango, Domain Analysis and Software Systems Modeling, IEEE Computer Society Press, 1991.

V. Singhal, S. Kakkad, and P. Wilson, "Texas: An Efficient, Portable Persistent Store," Persistent Object Systems: Proc. Fifth International Workshop on Persistent Object Systems (San Miniato, Italy), September 1992, pp. 11-33.

M. Stonebraker and G. Kemnitz, "The Postgres Next-Generation Database Management System," Communications of the ACM, October 1991, pp. 78-92.

M. Stonebraker, "The Miro DBMS," ACM SIGMOD, 1993.

M. Ubell, "The Montage Extensible DataBlade Achitecture," ACM SIGMOD, 1994.

D. Wells, J. Blakeley, C. Thompson, "Architecture of an Open Object-Oriented Database Management System," IEEE Computer, October 1992, pp. 74-82.

Received Date
Accepted Date
Final Manuscript Date