UTCS Technical Report 95-04
February 1995

# P2: An Extensible Lightweight DBMS[1]

**Jeff Thomas and Don Batory**
**Department of Computer Sciences**
**The University of Texas at Austin**
**Austin, Texas 78712**
**{jthomas, batory}@cs.utexas.edu**

## 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 explore LWDB applications, systems, and implementation techniques. We describe P2, an extensible lightweight DBMS, 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 for LEAPS executes substantially faster than versions built by hand or that use an extensible heavyweight DBMS.

## 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 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 (e.g., LEAPS [Mir90-91]).

*Extensible* or *open* database systems were a major step toward DBMS customization (e.g., TI's Open OODB [Wel92], IBM's Starburst [Haa90], Berkeley's Postgres/Miro/Illustra [Sto91-93], Wisconsin's Exodus [Car90], and Texas's Genesis [Bat88]). Extensible DBMSs enabled individual features or groups of features to be added or removed from a general-purpose DBMS to produce a database system that more closely matched the needs of target applications. Unfortunately, extensible DBMSs were basically customizable heavyweight DBMSs; their architecture and implementations (e.g., layered designs, interpretive executions of queries) still imposed the onerous overheads of heavyweight DBMSs. While feature customization of DBMSs *can* indeed improve performance, it has been our experience that the gains are rarely sufficient to satisfy the requirements of performance-critical applications.

1.This research was supported in part by Applied Research Laboratories at the University of Texas and Schlumberger.

*Lightweight* database management systems (LWDBs) appear to be the next major evolutionary trend in DBMS design. A lightweight DBMS 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 of LWDBs include main memory DBMSs (e.g., Smallbase [Hey94]), persistent stores (e.g., Texas [Sin92]), and primitive code libraries (e.g., Booch Components [Boo88]). Each of these examples strips features from a general-purpose DBMS (e.g., Smallbase removes the databases larger than main memory feature, Texas removes client-server architectures, and the Booch Components further strip checkpoints and recovery) and demonstrates 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.

There are broad application classes that require lightweight, not heavyweight, DBMSs. Because there are no formalizations, tools, or architectural support, LWDBs are hand-crafted monolithic systems that are expensive to build and tune. Clearly, what is needed are lightweight DBMSs that are extensible. In this paper we describe P2, an extensible lightweight DBMS. P2 provides architectural support and implementation techniques to assemble high-performance lightweight DBMSs 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 the application code and by following user-specified directives that define the database features that are to be supported. P2 performs many optimizations at compile-time: it compiles queries, inlines code to manipulate indices, and partially evaluates code statically, thus enabling the performance of P2-generated LWDBs to be comparable or exceed that of hand-written LWDBs.

We explain how we used P2 to generate LWDBs for the LEAPS production system compiler. LEAPS produces the fastest sequential executables of OPS5 rule sets by relying on highly-tuned, complex, and unusual data management features [Mir90-91]. 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.

We explain how we used P2 to generate different LWDBs for LEAPS. Our experimental results show:

- *Quality.* When using the same algorithms, a main-memory LWDB generated by P2 executed at least as fast as the hand-coded version. This shows that P2 produces quality code.
- *Extensibility.* A persistent LWDB generated by P2 ran *orders* of magnitude faster than the hand-coded version built on top of an extensible heavyweight DBMS. Also, when more advanced database retrieval algorithms are used, a P2 generated main-memory LWDB ran *orders* of magnitude faster than a hand-coded version. Both experiments demonstrate the extensibility of P2 and its ability to produce high-performance, customized LWDBs.

## 2 The Problem of Extensible Lightweight DBMSs

Informally, a LWDB trades generality for increased performance. Figure 1 is a partial list of DBMSs features whose absence offers opportunities for performance optimization. None of the features (or optimizations) listed are particularly novel. Rather, the choice of whether these and many other features should be omitted in LWDB construction arises all the time. In particular, Figure 2 displays a spectrum of existing lightweight and heavyweight DBMS products and the features of Figure 1 that each supports. Observe that different products offer different combinations of features.

| Feature | Optimization Possible if Feature is Omitted |
|---|---|
| *Concurrency Control*.<br>Multiple, simultaneous threads of control. | Avoid overhead of concurrency control. |
| *Checkpoints and Recovery*. | Use file copying to achieve database recovery and checkpointing. |
| *OS Files*.<br>Use the file system and disk storage organizations provided by the operating system. | Read and write "raw" disks directly in order to optimize data placement on disk and buffer management. |
| *Persistence*.<br>Store a copy of the database on disk. | Store data in transient memory and thereby avoid the overhead of disk I/O. |
| *Large Databases*.<br>Databases larger than main memory. | Use regular memory addresses for tuples instead of general object identifiers (OIDs). Use main-memory storage structures instead of page-based disk structures. |
| *Dynamic Queries*. | Avoid query parsing, optimization, and interpretive execution at run-time. |
| *Client/Server*. | Run DBMS engine and client on the same processor and/or address space. Exchange data without extraneous copies, and implement operations via procedure calls (or inlining) rather than RPC. |
| *Joins*.<br>Multi-relation queries. | Avoid overhead of multi-relation query optimization and maintaining relation statistics. |
| *Data Distribution*.<br>Data resides on multiple computers connected by a communication network | Avoid overhead of network communication, fragmenting queries according to location, and maintenance of data location tables. |
| *Set-Oriented Queries*. | Lazily compute retrievals and avoid retrieving unnecessary tuples and storing large intermediate results. |

Figure 1: Features and the optimizations possible if they are omitted.

Now consider Figure 3, which lists applications that manage databases of tuples and the features of Figure 1 that are needed by these applications. Note that (with the exception of P2) none of the DBMS products listed in Figure 2 exactly match the needs of Figure 3 applications; these products always have one or more extraneous features. Consequently, because of this mismatch, the performance of an application will be unnecessarily restrained if these products are used. There are undoubtedly many other features that would further restrain application performance that could have been added to Figure 1.

Hand-crafting application-specific LWDBs is very expensive. It is not the case that a library of LWDBs will eventually exist for which applications needing LWDB support will find an exact match. Such a library would be exponential in size (i.e., the number of distinct LWDBs that have some subset of $n$ features is $2^n$), and would constantly be growing (i.e., $n$ would always increase) [Bat93, Big94]. The key to solving this mismatch problem is one of generation: to cover a wide spectrum of LWDB implementations, there needs to be a tool to construct any member of a large family of LWDBs automatically. In the next section, we explain how P2 is a generator of LWDBs.

## 3 The P2 Data Model and Language

Choosing the appropriate abstractions on which to base the P2 data model and data language were critical both to P2's implementation as well as its utility to LWDB applications. Our analysis of many LWDB applications revealed that common database abstractions, namely containers and cursors, were fundamental, but generalizations were needed in order to synthesize efficient code. These generalizations were a consequence of treating common data structures as implementations of database relations (classes). In this

| | | LWDB | Concurrency Control | Checkpoints | OS Files | Persistence | Large Databases | Dynamic Queries | Client/Server | Joins | Data Distribution | Set-Oriented Queries |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Monolithic | Main Memory DBMS | IMS/VS Fast Path | ● | | | ● | | ● | ● | ● | | ● |
| | | MMDBS/OBE | ● | | | ● | | ● | ● | ● | | ● |
| | | Smallbase | ❍ | | ● | ❍ | | ❍ | | ● | | |
| | Persistent Object Stores | Texas | | ● | ❍ | ● | | | | | | |
| | | QuickStore | ● | ● | ● | ● | | | ● | | | |
| | | ObjectStore | ● | ● | ● | ● | | ● | ● | | | |
| | Other | Mini SQL | ● | ● | ● | | ● | ● | ● | | ● | |
| Libraries | | dbm | | | ● | ● | ● | | | | | |
| | | tdbm | ● | ● | ● | ❍ | ● | | ● | | ● | |
| | | Booch Components | ❍ | | ● | ❍ | | | | | | |
| | | Code Farms | | | ● | ❍ | | | | | | |
| Extensible DBMS | | Exodus | ● | ● | ● | ● | ● | ● | ● | ● | | ● |
| | | Genesis | ● | ● | ● | ● | ● | ● | ● | ● | | ● |
| | | P2 (existing) | | | ● | ❍ | | | | ❍ | | |
| | | P2 (potential) | ❍ | ❍ | ❍ | ❍ | ❍ | ❍ | ❍ | ❍ | ❍ | ❍ |

Figure 2: LWDB features (● indicates that the system provides the feature, ❍ indicates that the system may optionally provide or omit the feature according to the needs of the target application).

| Application | Tables to Maintain | Concurrency Control | Checkpoints | OS Files | Persistence | Large Databases | Dynamic Queries | Client/Server | Joins | Data Distribution | Set-Oriented Queries |
|---|---|---|---|---|---|---|---|---|---|---|---|
| DBMS lock manager | lock tables | ● | | | ● | | | | ● | | |
| Operating systems | page & segment tables | ● | | | | | | | ● | | |
| Compilers | symbol tables | | | | | | | | ● | | |
| Name Servers | host name & address tables | | | | ● | | | | ● | ● | |
| Persistent stores | page tables | | | | ● | | | | ● | | |
| Spelling checkers | dictionaries | | | ● | ● | | | | ● | | |
| LEAPS | assertion tables | | | ● | ● | | | | ● | | |

Figure 3: Application features (● indicates that the application requires the feature).

section, we outline the P2 data model and its embedded data language. In Section 4 we explain how P2 programs are translated into efficient C programs.

## 3.1 Cursors and Containers

A *container* is a sequence of elements, where all the elements are instances of a single data type. Elements can be referenced and updated only by a run-time object called a *cursor* (see Figure 4).

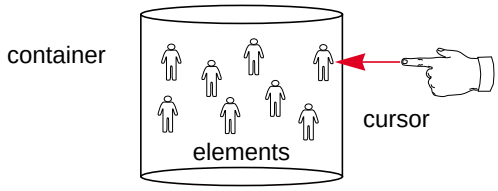



Figure 4: Basic P2 abstractions.

The P2 embedded data language is a superset of C; P2 adds cursors and containers (including their operations) as primitive data types. An abbreviated declaration of a container of **`EMPLOYEE_TYPE`** instances is shown below, along with declarations of a cursor (**`all_employees`**) that references all elements of the container, a second cursor (**`selected_employees`**) that references only elements that satisfy a predicate, and a third cursor (**`sorted_employees`**) that references selected elements in sorted order. Note that predicates in P2 are expressed by strings: attribute **`A`** of the element referenced by a cursor is denoted **`$.A`**. The **`$`** denotes to P2 the name of the cursor.

```
typedef struct { ... } EMPLOYEE_TYPE;   // C struct declaration

container <EMPLOYEE_TYPE> employee;     // P2 declaration of employee container

cursor <employee> all_employees;        // P2 cursor declaration

cursor <employee>
   where "$.deptno == 10 && $.age < 50"
   selected_employees;                  // cursor declaration with predicate

cursor <employee>
   where "$.age > 50" orderby ascending lastname
   sorted_employees                     // cursor declaration with ordering
```

In general, P2 containers and cursors are parameterized data types. 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 sort criterion. Cursor and container types are first-class; they can be used like any C data type:

```
typedef cursor <employee> where "$.age >= 20"
   AGE_CURSOR;                              // cursor typedef decl

AGE_CURSOR c, *c_ptr, c_array[10];          // assorted cursor variable decls

int foo( AGE_CURSOR *c ) { ... }            // function with cursor parameter
```

P2 offers an (extensible) set of operations on cursors and containers. The (admittedly nonsensical) 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.

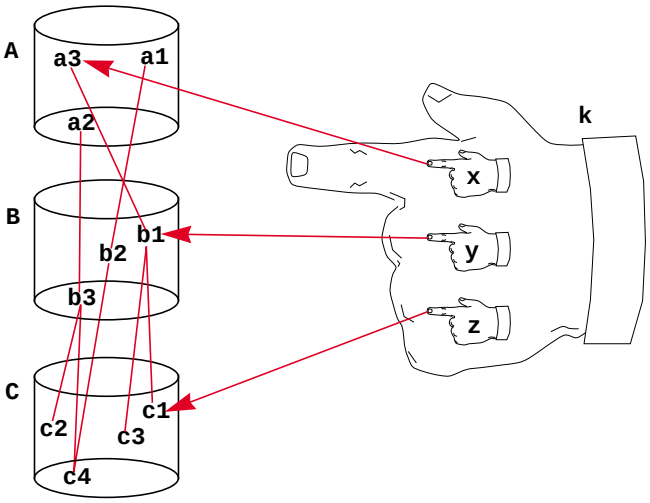
```
foreach( selected_employees )               // for each selected employee
{
  printf( "%s\n", selected_employees.name ); // print employee name

  if (selected_employees.jobcode == 7)      // examine employee jobcode
```

```
        delete( selected_employees );          // delete employee
    else
        selected_employees.salary *= 1.10;     // update employee
}
```

## 3.2 Composite Cursors

A *relationship* among containers $C_1$, $C_2$, ..., $C_n$ is a set of n-tuples <$e_1$, $e_2$, …, $e_n$> where element $e_i$ is a member of container $C_i$. Figure 5a depicts a relationship for containers **A, B**, and **C** whose 3-tuples are **{(a3,b1,c1), (a3,b1,c3), (a1,b2,c4), (a2,b3,c2), (a2,b3,c4)}**.




(a) Relationship

```
// composite cursor declaration

compcurs < x A, y B, z C >
where predicate($x,$y,$z) k;
```

```
// typical usage--print names of
// each element of all 3-tuples

foreach ( k ) {
   printf("( %s, %s, %s )\n",
   k.x.name, k.y.name, k.z.name);
}
```

(b) Composite Cursor (c) Cursor Declaration and Usage

Figure 5: A multicontainer relationship, a composite cursor, and a composite cursor declaration.

A *composite cursor* iterates over the tuples of a relationship. More specifically, a composite cursor **k** is an n-tuple of cursors, one cursor per container of a relationship. A particular n-tuple <$e_1$, $e_2$, ..., $e_n$> of a relationship is encoded by having the *i*th cursor of **k** positioned on element $e_i$. By advancing **k**, successive tuples of a relationship are retrieved. Figure 5b depicts a composite cursor **k**, with subcursors **k.x**, **k.y**, and **k.z** that reference containers **A**, **B**, and **C**, respectively. The 3-tuple that **k** references is **<a3,b1,c1>**.

Figure 5c shows a P2 declaration for **k**. The names of cursors are listed as container aliases (e.g., **x** is the cursor for container **A**, **y** for **B**, and **z** for **C**). Besides providing a useful shorthand, aliases are necessary to express the join of a container with itself. The predicate, which defines the 3-tuples that **k** will iterate over, is expressed in terms of cursor references (**$x**, **$y**, **$z**). The **foreach** loop of Figure 5c shows a typical use of composite cursors, where on every **foreach** loop iteration the names of the elements of a retrieved tuple are printed.

Like cursors and containers, composite cursors are parameterized first-class data types. The examples below show a **typedef** declaration for a composite cursor and a function that is parameterized by a variable of that type:

```
typedef compcurs < d department, e employee >
        where "$d.deptno == $e.deptno" CUSTOMIZED_CURS;

int bar ( CUSTOMIZED_CURS cc ) { ... };
```

Akin to modifying tuples through database views, P2 permits the elements of a composite cursor to be updated [Kel82]. Instead of restricting the updates that can be performed, our analysis of LWDB applications suggested a very different approach was needed: updates should not be restricted, but updates may effect the tuples that are subsequently retrieved. For example, once an element of a tuple is deleted, that element should not belong to any subsequently retrieved tuple. Composite cursor **c** of Figure 6a returns pairs of related **department** and **employee** elements. The **foreach** loop of Figure 6a prints each retrieved ordered pair and then deletes the **department** element of that pair.

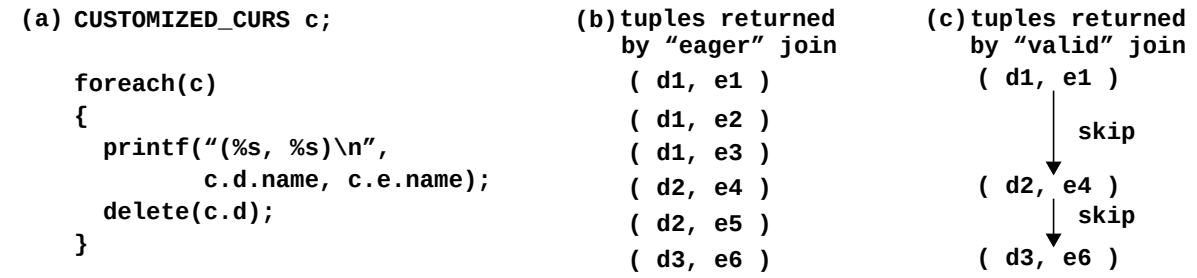



Figure 6: Updating elements within a **foreach** loop.

If tuples of **c** were computed eagerly (i.e., set-at-a-time), **c** might return tuples with deleted elements. That is, Figure 6b shows an eager join returning tuples (**d1,e2**) and (**d1,d3**) after element **d1** has been deleted. (Clearly, modifying or deleting previously deleted tuples is meaningless). In contrast, a *valid* retrieval would note database updates that were performed since the last advance of **c**, and would skip tuples containing deleted elements (Figure 6c). P2 generates the code that supports valid retrieval semantics.

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 6a by only returning tuples with undeleted **department** elements. **deleted()** is a P2 operation that returns **TRUE** if the specified element has been deleted.

```
compcurs < d department, e employee >
where "$d.deptno == $e.deptno"
valid "!deleted($d)" valid_composite_cursor;

foreach( valid_composite_cursor ) // skips pairs with deleted departments
{
   printf("(%s, %s)\n", c.d.name, c.e.name);
   delete( valid_composite_cursor.d );
}
```

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.

## 4 The P2 Generator

A LWDB for an application is the set of P2 cursor, composite cursor, and container data types that are referenced. It is the task of P2 to generate the code for the application's LWDB given a specification of the features to be supported.

P2 exploits physical data independence. LWDB applications are written in terms of P2 cursor, composite cursor, and container data types *without regard to how these types are implemented*. This approach radically simplifies LWDB application development: programming with high-level database abstractions is

substantially easier than using low-level, ad hoc interfaces of hand-crafted LWDB modules. Moreover, by standardizing the interfaces to P2 data types, it is possible to swap implementations (that can be radically different) to improve performance without modifying application code. Thus, tuning P2/LWDB applications is considerably simplified. We illustrate this capability in Section 5.

The primary challenge in the development of P2 was not standardizing the interfaces of P2 data types (this was comparatively easy), but rather generating vast families of implementations. Merely developing libraries of plug-compatible LWDBs is emphatically not sufficient. As we explained in Section 2, each LWDB encapsulates a distinct set of features (e.g., nested loop join algorithms, predicate indices, memory mapped persistent storage); multiple LWDBs differ on the features each encapsulates. The number of features that could be included in a LWDB is itself quite large, but the number of distinct *combinations* of these features (and hence the number of distinct LWDBs) is exponential. Consequently, P2 must rely on a powerful model of software generation.

P2 is based on GenVoca, a scalable model of software construction [Bat92, Bat94b]. GenVoca is a distillation and generalization of the concepts that were originally developed in Genesis (a database system generator), and later in Avoca/*x*-kernel (communications protocol generator [Hut91]), Ficus (file system generator [Hei94]), and ADAGE (avionics generator [Cog93]). GenVoca is scalable, because it is possible to build vast families of software systems from a small number of components. The key is that GenVoca components encapsulate *individual* features; combinations of features are expressed as compositions of components. Thus, GenVoca does not eliminate the problem of feature combinatorics, but simply makes it explicit. Combinations of components are called *type equations*, and P2 type equations occupy only a few lines of any P2 application.

P2 is a sophisticated preprocessor and component library comprising over 50,000 lines of C [Bat93, Bat94c]. It works by translating an abstract P2 program into a concrete C program. The P2 program is abstract and not executable, as it is a C program that references P2 data types and programming constructs that have no implementation. Using the supplied type equations, the P2 generator translates a P2 program into an executable C program by generating implementations for every P2 data type that was referenced. The implementation of the P2 data types constitutes the generated LWDB (see Figure 7).

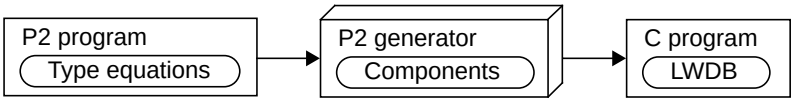



Figure 7: Transforming P2 programs into C programs.

Each P2 component is a forward refinement program transformation that encapsulates a consistent data, operation, and metadata operation refinement. Encapsulating data and operation refinements is a familiar concept: object-oriented classes and ADTs specifically deal with such encapsulation. However, the encapsulation of metadata is a new twist and lies at the heart of both heavyweight and lightweight DBMS optimizations. Metadata knowledge specifies the conditions under which certain optimizations can be applied and certain algorithms should be used to obtain optimal performance. One simple way that P2 encapsulates metadata is the distinction it makes between code that the P2 generator is to execute from the code that is to be generated. The following code fragment is taken from the P2 attribute indexing component; the **`upd(c,f,v)`** operation updates field **`f`** of the element referenced by cursor **`c`** with the value **`v`**:

```
upd( c, f, v ) // update field f of element c with value v
{
   if (strcmp( f, %a.indexed_field ) == 0) // if f is the indexed field
   %{
       remove_from_index (c);              // remove element from index
       upd( c, f, v );                     // pass update to lower levels
       add_to_index (c);                   // relink element into index
```

```
    %}
    else
    %{
       upd( c, f, v );                    // pass update to lower levels
    %}
}
```

All text enclosed within **%{** ... **%}** defines a code fragment that is to be generated; statements outside **%{** ... **%}** are executed by P2. The **%a** symbol refers to the parameters of a component. In the above example, code for unlinking and relinking an element to an index is generated only if the indexed field is updated.

Query optimization is another example of metadata knowledge. Optimization of single-container predicates in P2 is accomplished in a manner similar to that used in Open OODB, Starburst, and Genesis. When code for P2 cursors is to be generated, each component "examines" the cursor's predicate and returns an estimate of the cost to use that component's data/container structure to process the cursor's predicate. All components of the container's type equation are polled, and the component submitting the cheapest estimate becomes responsible for the generation of that cursor's retrieval code (e.g., **foreach()**).

A twist to P2 query optimization is that no statistics are known (or kept) for transient containers, hence a different form of cost-based query optimization is needed. P2 solves this problem by using common heuristics (e.g., given a key of an element, locating the element in binary tree is faster than locating it in an unordered list). P2 statically ranks the efficiencies of components on different query categorizations (e.g., primary key search, search of elements whose keys belong to a given range, scan). Each polled component determines the category in which the cursor's predicate belongs, and then returns the predetermined cost estimate/rank. As an example, the predicate **$.name == 'Batory' && $.age > 40** could be categorized by a binary tree component as a key retrieval (if **name** is the primary key of the tree), or a range retrieval (if **age** is the key), or a scan (if the primary key is neither **name** or **age**). In general, we have found these heuristics to work surprisingly well (see results in Section 6).

Another example of encapsulated metadata knowledge is the manipulation of predicates for optimal code generation. Again consider the predicate **$.name == 'Batory' && $.age > 40** and suppose **age** is the key of a binary tree. The binary tree component would factor this predicate, using the **$.age > 40** fragment to locate the first element that satisfied the predicate, and applying the residual **$.name == 'Batory'** to all elements sequentially thereafter (since the **$.age > 40** predicate is known to be satisfied a priori). Optimizations such as this are critical for generating efficient code and avoiding unnecessary predicate evaluations.

Another example of predicate manipulation is the handling of boolean functions that have one or more cursor arguments (e.g., **foo($x,$y,$z)**) in composite cursor predicates. Such functions are evaluated as soon as possible in P2-generated code (i.e., immediately after the joins where all cursor arguments have been found [Hel93]).

Further details about type equations and the design and implementation of P2 components are discussed in [Bat93, Bat94c].

## 5 LEAPS

The LEAPS (*L*azy *E*valuation *A*lgorithm for *P*roduction *S*ystems) production system compiler produces the fastest sequential executables of OPS5 rule sets [Mir90-91]. A LEAPS executable is a database application, because it represents its database of assertions as a set of containers. LEAPS is typical of a LWDB application, because it uses unusual search algorithms and novel container implementations to enhance

rule processing efficiency; no heavyweight DBMS could offer the performance or features needed by LEAPS.

OPS5 is a forward-chaining rule programming language [Coo88]. 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 check_done)
   (last_seat ^seat <l_seat>)
   (seating ^seat2 <l_seat>)
-->
   (write Yes we are done)
   (modify 1 ^value print_results))
```

Each condition element (CE) lists a container name followed by one or more selection predicates. Names in angle brackets **<>** denote variables whose values are to be instantiated during a search. For example, the first CE defines the selection predicate **`value == 'check_done'`** over the **`context`** container. The next two CEs join the elements of the **`last_seat`** container to elements of the **`seating`** container by the equijoin predicate **`last_seat.seat == seating.seat2`**. The predicate of an OPS5 rule is defined by the conjunction of the predicates of its CEs.

OPS5 execution follows a *match-select-act cycle*: *match*—identify rules whose predicates can be satisfied, *select*—choose one of the identified rules and a tuple of elements that satisfies that rule's predicate, and *act*—execute (or *fire)* the actions of the rule using the selected tuple. The actions of the **done** rule are to print the string "**`Yes we are done`**" and to modify the **`value`** field of the selected **`context`** element to be **`print_results`**.

LEAPS translates OPS5 rule sets into C programs (Figure 8). 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.

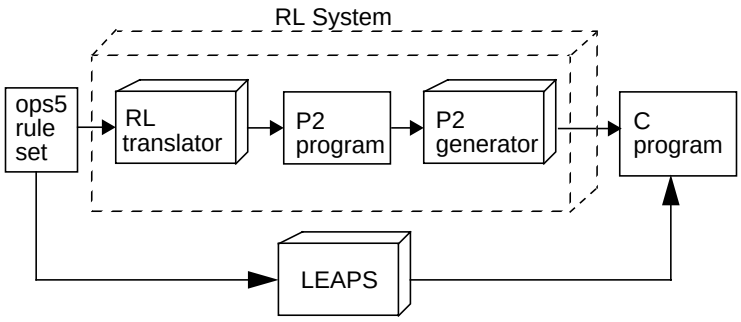



Figure 8: Relationship between LEAPS and RL.

The LWDBs produced by RL are quite complicated. Every OPS5 rule set has a set of containers and a composite cursor type for each rule. For example, the rule set from which the **done** rule was taken would have at least the following P2 container declarations:

```
container < CONTEXT >  context;   // CONTEXT, SEATING, and
container < LAST_SEAT> last_seat; // LAST_SEAT are C struct
container < SEATING >  seating;   // typedefs
```

The **done** rule is translated into the following composite cursor data type:

```
#define done_query    "$a.value == check_done && $c.seat2 == $b.seat"
#define done_temporal "$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 **done** selection predicate is expressed by the **done_query** macro. The containers to be joined (along with their aliases) are parameters to the **compcurs** declaration. LEAPS algorithms require that every element have a timestamp to indicate when it was inserted. During rule set execution, it is possible that multiple cursors of **done_cursor_type** may be active. LEAPS uses timestamps in **compcurs** declarations to achieve fairness—i.e., to preclude rules from being fired more than once by the same tuple of elements. The **done_temporal** macro expresses the LEAPS temporal predicate for the **done** rule where **ts** is the timestamp field of an element and **gts** is a global timestamp set by the LEAPS algorithm.

During rule set execution, LEAPS creates an instance of a composite cursor and advances the cursor to the first tuple of elements. This tuple is used to fire the actions of the corresponding rule. The cursor is then pushed on a stack, thereby suspending the execution of its joins. Only until 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.

LEAPS takes advantage of the fact that it knows the complete set of predicates that will be evaluated during a rule set execution. LEAPS uses a special structure, called a *predicate index*, to enhance rule processing efficiency. A predicate index is a list of elements that satisfy a given predicate; a predicate that is indexable references a single container and has no variables. As an example, the condition element **(context ^value check_done)** of the **done** rule results in a predicate index over the **context** container; the predicate to be indexed is (**context.value == 'check_done'**).

There are other sources of complexity in LEAPS LWDBs. 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. To reduce string matching time, symbol tables are created so that fast pointer comparisons can be used in place of string comparisons, and so on. Further details of the features of LEAPS LWDBs are discussed in [Bat94a].

## 6 Results

There are two different hand-coded versions of LEAPS: $LEAPS_{transient}$ (also called OPS5c) produces executables that store containers in transient memory [Mir90-91]; $LEAPS_{persistent}$ (also called DATEX) produces executables that store containers in persistent memory [Bra93]. $LEAPS_{persistent}$ used Jupiter, the heavyweight extensible file manager of Genesis [Bat88], for persistent storage.

We used RL to generate a single P2 program for each OPS5 rule set. By modifying only type equations (which account for a few lines in each P2 program and that define the composition of P2 components that implement the LWDB to be used), we were able to produce an executable (called $RL_{persistent}$) to compare with the corresponding $LEAPS_{persistent}$ executable, and two other executables (called $RL_{transient}$ and $RL_{tran-}$

$_{sient\text{-}hash}$) to compare with LEAPS$_{transient}$ executables. In all cases, modifying the type equations of a P2 program took minutes; *no other source code had to be modified.*

For our evaluation, we used a collection of OPS5 rule sets—**`tripl`**, **`manners`**, **`waltz`**, and **`waltzdb`**—that have become standard benchmarks for evaluating the performance of OPS5 execution engines [Bra91]. We ran each benchmark with a series of different input file sizes, where size is measured by the number of initial assertions. The timing results presented here were obtained on a SPARCstation 5 with 32 MB of RAM running SunOS 4.1.3 using the **`gcc`** 2.5.8 compiler with the **`-O2`** option. Similar results have been obtained on other architectures [Bat94c].

Our first experiment revealed the overheads imposed by heavyweight systems (Figure 9). We compared the performance of LEAPS$_{persistent}$ executables to RL$_{persistent}$ executables; persistence of RL$_{persistent}$ containers was achieved by merely swapping the transient storage component in the P2 type equations with a memory-mapped component. For all input file sizes, RL$_{persistent}$ executables ran at least an order of magnitude faster than LEAPS$_{persistent}$ executables. Figure 9 indicates the considerable overhead imposed by the buffer management of disk pages, B+ tree storage structures, interpretive query evaluation, and volume management of the heavyweight Genesis file manager. While it could be argued that these results are solely a consequence of the inefficiency of LEAPS$_{persistent}$/Jupiter rather than the efficiency of the RL/P2 generated code, our next experiment suggests otherwise.

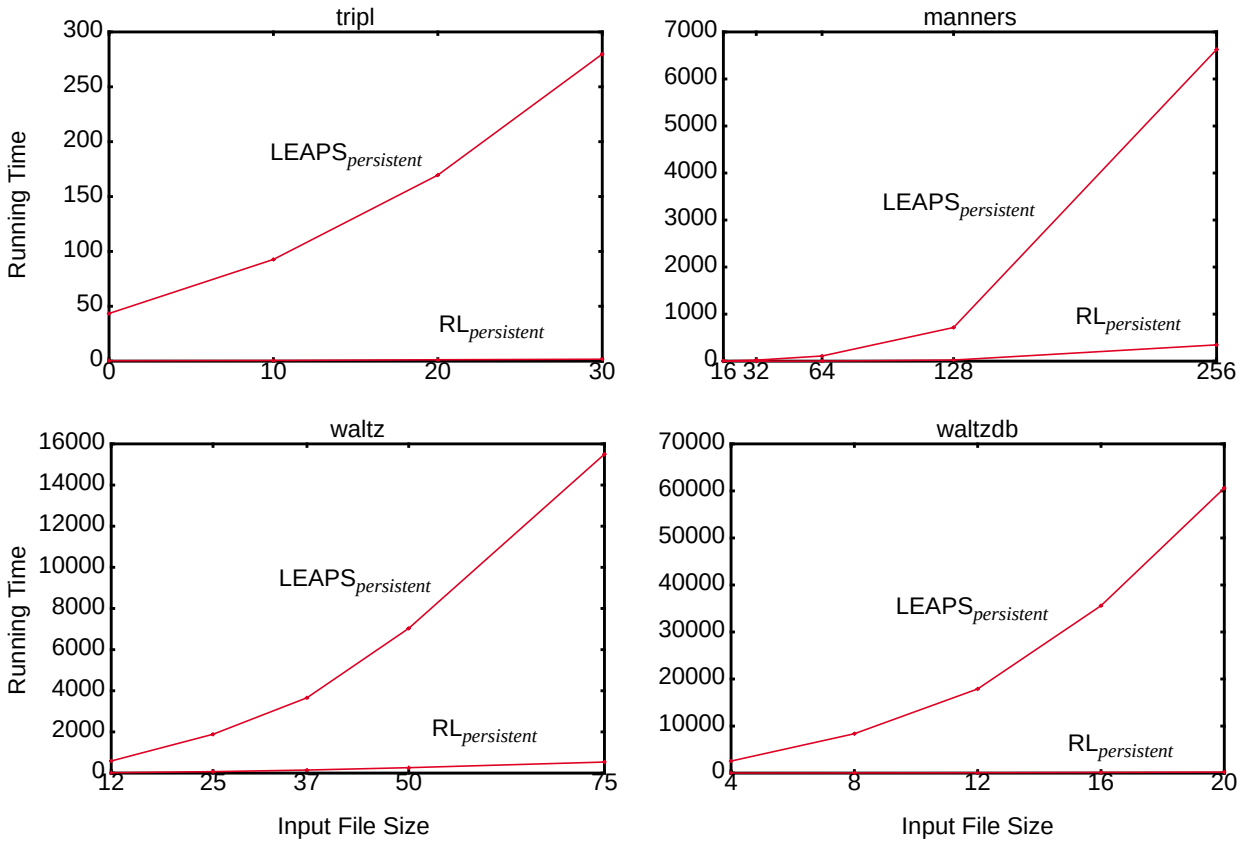



Figure 9: LEAPS$_{persistent}$ and RL$_{persistent}$ execution times.

Our second experiment showed that RL and P2 generated high-quality code (Figure 10). We compared the performance of LEAPS$_{transient}$ executables to RL$_{transient}$ executables. (RL directly produces RL$_{transient}$ source files, so no modification of their type equations was necessary). For all input file sizes, RL$_{transient}$ executa-

bles performed better (by about 50%) than LEAPS$_{transient.}$ The differences are attributable to our cleaner design of LEAPS algorithms (programming LEAPS algorithms using the high-level abstractions of P2 made it possible to see certain simplifications and optimizations that otherwise were obscured). The LEAPS algorithms have long been known to be difficult to understand; expressing them in P2 substantially clarified their explanation [Bat94a].

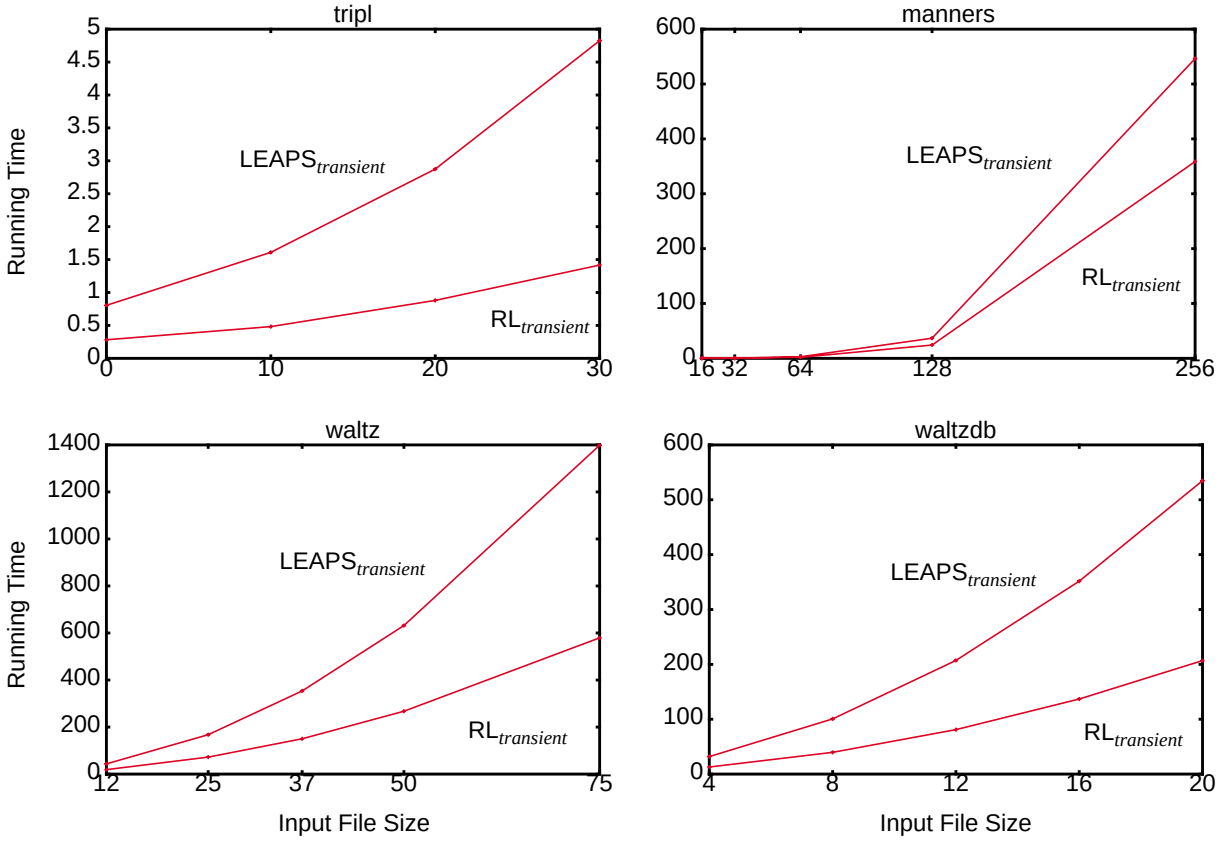



Figure 10: LEAPS$_{transient}$ and RL$_{transient}$ execution times.

Our last experiment demonstrated the value of being able to easily modify LWDB implementations (Figure 11). The vast majority of execution time in LEAPS is spent joining containers. One of the primary reasons why LEAPS$_{transient}$ produces executables whose performance far surpasses OPS5 interpreters is that LEAPS$_{transient}$ does not produce intermediate results during joins; it uses a nested loop algorithm. Unlike DBMS-preferred join algorithms like hash-joins and sort-merge-joins, nested loops does not produce intermediate files or require storage for temporary relations during join execution. (The importance of this observation is critical: during the execution of LEAPS programs, hundreds or thousands of multiway joins could be active (or suspended on a LEAPS stack) at any one time. The amount of memory required to store temporary relations in such cases would be gargantuan, thus nullifying any potential performance gains that might result in using other join algorithms). As the use of nested loops is required by LEAPS, this did not preclude us from storing elements of a container using a hashed structure. Combining nested loops with hashed structures effectively allowed us to emulate performance of hash-join algorithm [Sha86]. To achieve this modification required two extra P2 components to be written and added to the type equations of RL$_{transient}$ source files. (These components took a few days to write; the type equations took a few minutes to edit).

We knew the performance of RL **`tripl`** executables would not be affected by this change, as it only performs non-equijoins. We were stunned, however, by the performance improvements of RL$_{transient\text{-}hash}$ exe-

cutables over LEAPS$_{transient}$. Figure 11 shows the difference can be orders of magnitude, growing larger as the data set size increases. We did not alter the asymptotic complexity of the LEAPS algorithms, but rather the constant. That is, we still have to perform the same number of container searches, but instead of scanning a container, we simply scan a fraction of a container that is in a particular bucket. By increasing the number of hash buckets as the data set size increases, we are able to scale the performance of LEAPS algorithms gracefully. It is interesting to note that LEAPS$_{persistent}$ attempted to accomplish the same result (using different algorithms). Because of the overhead of Jupiter (and possibly the algorithms or implementation of LEAPS$_{persistent}$), the scalability of LEAPS$_{persistent}$ algorithms were not clearly evident in the original papers.

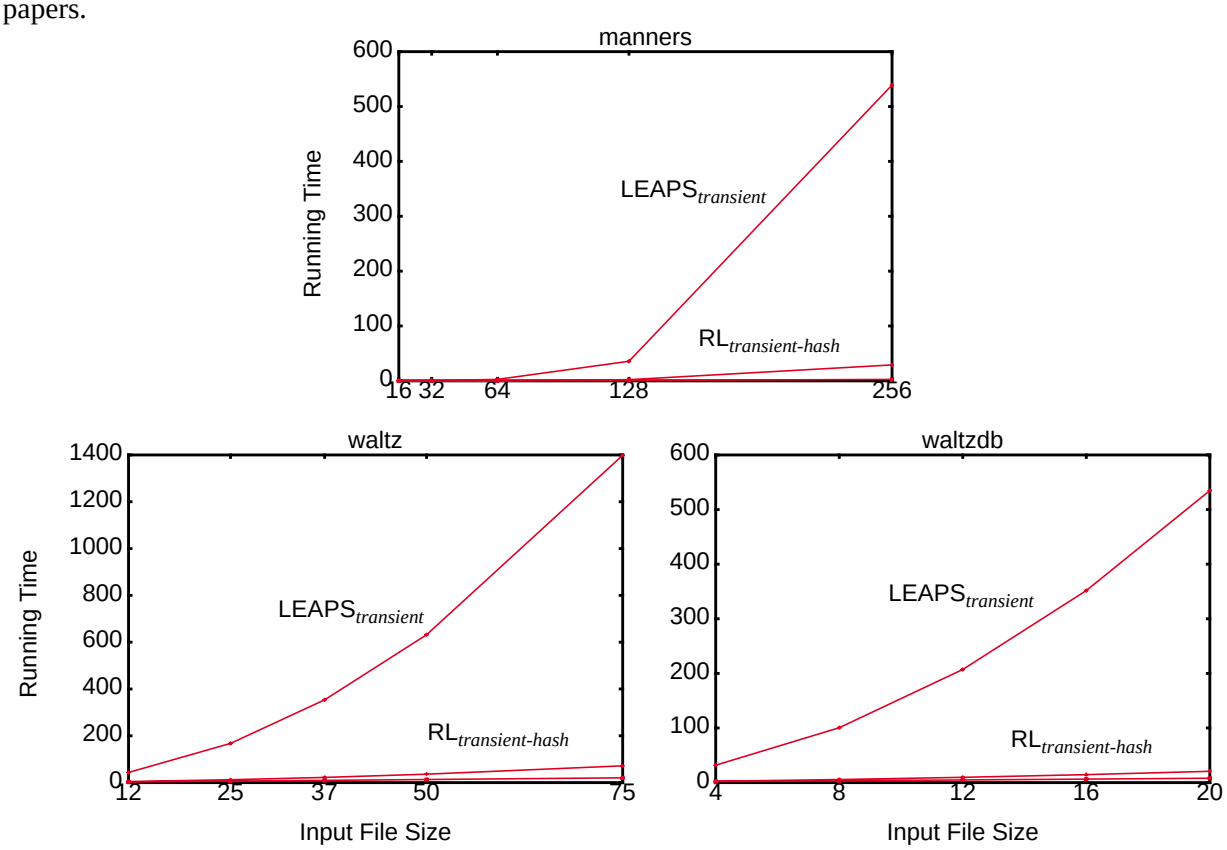



Figure 11: LEAPS$_{transient}$ and RL$_{transient-hash}$ execution times, showing LEAPS$_{transient}$ using nested loops join versus RL$_{transient-hash}$ using 13 and 1011 hash buckets.

There are two other aspects of our experiments that are worth noting. First, P2 is a general-purpose tool for generating customized LWDBs. Using P2 *substantially* reduced the effort to implement the LEAPS algorithms and to experiment with different LWDB features/implementations. Second, the files produced by P2 are nontrivial. The **`waltzdb`** rule set consists of approximately 40 rules, the RL generated P2 file consists of 3,000 lines, and the C file produced by P2 is almost 16,000 lines. (The source files produced by LEAPS$_{transient}$ and LEAPS$_{persistent}$ are comparable in size).

## 7 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.

We have described P2, an extensible lightweight DBMS that combines a relatively traditional data model and embedded data language with a powerful model of software system construction. 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 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 [Leh86] and sort-merge joins). This will allow us to use P2 to generate LWDBs for a broader range of applications. One feature in particular that would be important would be a SQL front-end to P2. This would allow us to provide LWDB support for existing applications expecting a SQL front-end and would facilitate performance comparisons of P2-generated LWDBs to other hand-crafted LWDBs.

We believe lightweight DBMSs have a wide applicability and practical importance. We feel that our work with the P2 demonstrates that extensible lightweight DBMSs are feasible. In the hope that P2 will benefit other researchers, we provide the source code and documentation for P2 via anonymous **`ftp`** from **`ftp.cs.utexas.edu:/pub/predator/`**.

**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.

## 8 References

[Bat88] 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, pages 1711-1730.

[Bat92] 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, pages 355-398.

[Bat93] D. Batory, V. Singhal, M. Sirkin, and J. Thomas, “Scalable Software Libraries”, *ACM SIGSOFT*, December 1993.

[Bat94a] D. Batory, “The LEAPS Algorithms”, Department of Computer Sciences, University of Texas at Austin, Technical Report 94-28.

[Bat94b] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin, “The GenVoca Model of Software System Generators”, *IEEE Software*, September 1994, pages 89-94.

[Bat94c] D. Batory, J. Thomas, and M. Sirkin, “Reengineering a Complex Application Using a Scalable Data Structure Compiler”, *ACM SIGSOFT*, December 1994.

[Big94] T. Biggerstaff, “The Library Scaling Problem and the Limits of Concrete Component Reuse”, *Proceedings of the Third International Conference on Reuse*, November 1994.

[Boo87] G. Booch, *Software Components with Ada*, Benjamin/Cummings, 1987.

[Bra91] 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.

[Bra93] D. Brant and D. Miranker, "Index Support for Rule Activation", *ACM SIGMOD*, May 1993.

[Car90] 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.

[Cog93] L. Coglianese and R. Szymanski, "DSSA-ADAGE: An Environment for Architecture-based Avionics Development", *Proceedings of AGARD 1993*.

[Coo88] T. Cooper and Nancy Wogrin, *Rule-based Programming with OPS5*, Morgan-Kaufmann, 1988.

[Haa90] 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, pages 143-161.

[Hey94] M. Heytens, S. Listgarten, M. Neimat, K. Wilkinson, "Smallbase: A Main-Memory DBMS for High-Performance Applications", HP Labs Technical Report, December 1994.

[Hei94] J.S. Heideman and G.J. Popek, "File-System Development with Stackable Layers", *ACM Transactions on Computer Systems*, February 1994.

[Hel94] J.M. Hellerstein, "Practical Predicate Placement", *ACM SIGMOD* 1994.

[Hut91] N. Hutchinson and L. Peterson, "The *x*-kernel: an Architecture for Implementing Network Protocols", *IEEE Trans. Software Engineering*, January 1991.

[Kel82] A. Keller, "Updates to Relational Database Through Views Involving Joins", in P. Scheuermann (editor), *Improving Database Usability and Responsiveness*, Academic Press, 1982.

[Leh86] T. Lehman and M. Carey, "Query Processing in Main Memory Database Management Systems", *ACM SIGMOD*, June 1986.

[Mir90] 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.

[Mir91] 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.

[Sha86] L.D. Shapiro, "Join Processing in Database Systems with Large Main Memories", *ACM Transactions on Database Systems*, September 1986.

[Sin92] 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, pages 11-33.

[Sto91] M. Stonebraker and G. Kemnitz, "The Postgress Next-Generation Database Management System", *Communications of the ACM*, October 1991, pages 78-92.

[Sto93] M. Stonebraker, "The Miro DBMS", *ACM SIGMOD*, 1993.

[Wel92] D. Wells, J. Blakeley, C. Thompson, "Architecture of an Open Object-Oriented Database Management System", *IEEE Computer*, October 1992, pages 74-82.