# List Comprehensions in AGNA, a Parallel Persistent Object System

Computation Structures Group Memo 326
March 10, 1991

**Michael L. Heytens** *and* **Rishiyur S. Nikhil**

Submitted for publication.

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-89-J-1988.

Support for Michael Heytens is provided in part by an Intel Graduate Fellowship.

# List Comprehensions in AGNA, A Parallel Persistent Object System

Michael L. Heytens*
Rishiyur S. Nikhil†
Massachusetts Institute of Technology

**Abstract**

List comprehensions in functional languages are structurally very similar to SQL, the standard declarative query language for relational databases. However, list comprehensions are more general, and have the advantage of being embedded seamlessly into a powerful programming language. Thus, functional languages with list comprehensions are attractive as query languages for persistent object systems, where database objects and database queries are not distinguished from other objects and computations.

We have implemented a parallel persistent object system called AGNA in which the query language is a general functional language that includes list comprehensions. To our knowledge, AGNA is the first implementation of a parallel persistent object system, whether based on functional languages or not.

In this paper, we describe list comprehensions and their optimization in AGNA. List comprehensions are optimized substantially, using techniques borrowed from the relational database literature, as well as techniques specific to our parallel implementation. Experimental results show that the optimizations have a dramatic impact on performance. A comparison of our results to results obtained on a commercial relational database system and to published results for an experimental parallel relational system indicates that the performance of AGNA approaches that of state of the art relational database systems.

## 1 Introduction

Many modern functional languages have a construct called the "list comprehension." While offering no fundamentally new expressive power, it provides an elegant and concise notation for expressing certain types of computations on lists.

The standard declarative database query language today is SQL, which evolved in the context of relational databases [6]. A strong similarity between SQL and list comprehensions has been recognized for some time [9]. List comprehensions are more powerful than SQL, however, because of their support for general computation. Also, they are well integrated with other parts of the language (*e.g.,* they may include arbitrary function calls and recursion, may be embedded within procedures, *etc.*), which is in marked contrast to the embedding of SQL in languages such as C or Ada. It is thus natural to consider a functional language with list comprehensions as a database query language ([9], see also [4]).

The close correspondence between list comprehensions and relational languages facilitates the application of relational optimization techniques to functional database systems. In his thesis, ([12]) Trinder shows how standard algebraic and implementation-based optimizations can be applied to list comprehension queries.

*Room 36-667, 77 Massachusetts Avenue, Cambridge MA 02139, USA; Phone: (617)-253-7811; Internet: heytens@caf.mit.edu
†Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA; Phone: (617)-253-0237; Internet: nikhil@lcs.mit.edu

In this paper, we describe the optimization and translation of list comprehension queries in AGNA, a parallel persistent object system that we have built. We describe a number of standard optimizations, and some that are designed specifically for a parallel implementation. Experimental uniprocessor and multiprocessor results indicate the effectiveness of the optimizations. We compare our results with results obtained on a commercial relational database system and published results for other parallel database systems.

In Section 2 we describe the AGNA language (or at least that part of it that is relevant for this paper). In Section 3 we describe the optimizations for list comprehensions based on standard SQL optimizations, and we compare the uniprocessor performance of AGNA with that of a uniprocessor commercial relational database system. In Section 4 we describe the optimizations for list comprehensions based on our parallel implementation, and compare the performance of AGNA on a parallel machine to published performance results of an experimental parallel relational database system. Finally, we conclude in Section 5 with some discussion of our results and prospects for the future.

## 2 Background

In AGNA, a database is viewed as a top-level persistent environment of bindings that associate names with persistent objects. The objects in the database may be of any type— scalars, complex objects, lists, procedures, indexed mappings, *etc.*

A transaction `(xact ...)` is a construct executed in this environment, and may contain definitions of new types, definitions of new bindings, declarative update specifications, and *queries* (expressions to be evaluated). In this paper we concentrate only on queries, which are pure functional expressions[1].

User-defined object types may be introduced into a database environment via the **type** form. For example, the following definitions introduce two new types and some of their fields (we currently use a Lisp-like notation to avoid detailed syntax design).

```
(type STUDENT (extent)              (type COURSE (extent)
 ((name     <=>  STRING)             ((name  *<=> STRING)
  (courses *<=>* COURSE)              (number <=> STRING)
  (gpa       =>  NUMBER)              (units   => NUMBER)))
  (year      =>  NUMBER)))
```

These declarations introduce new object type names **STUDENT** and **COURSE** along with the name and type of each field, whether they are single- or multi-valued, and whether they have single- or multi-valued inverses. For example, the student **name** field records a single string value, and supports a unique inverse mapping strings to students. The **courses** field is multi-valued, with a multi-valued inverse mapping a course to a collection of students. The basic operations available for manipulating structured objects, of both pre- and user-defined types, are allocation, field selection, and field definition. A field of an object may be read using the expression (**select** *object type field*).

The (**extent**) qualifications in the type definitions declare that persistent extents containing all students and courses are to be maintained automatically, *i.e.*, every construction of a **student** or **course** object is automatically inserted into the corresponding system-maintained extent (these insertions are not visible in the current transaction). The expressions `(all student)` and `(all course)` use these extents, and evaluate to lists of all students and all courses, respectively. Such lists (extents of declared types) are also called "base extents" because they are automatically maintained by the system, in contrast to lists that are produced as results of some computation. While the system chooses efficient internal representations for base extents, they are accessed in programs exactly like other lists.

New functions may be introduced into the database environment using the **define** form. For example, the following transaction extends the top-level environment with a function **courses-with-name** that maps a string **s** to a list of courses with that name (using the generic **invert** form for inverted fields):

[1]Briefly, an update specification is a declarative specification of the database visible to subsequent transactions, expressed as a function of its current state. For details, please see [11] and [8].

```
(xact
  (define courses-with-name
    (lambda (s) (invert course name s))))
```

List comprehensions in AGNA have the form:

```
(all  body-expression
      generator-or-filter
      ...
      generator-or-filter)
```

Each generator has the form (*identifier list-expression*), and each filter has the form (where *boolean-expression*). For example, here is a query to find the names of all first year students taking software engineering:

```
(xact
  (all (student-name s)
    (s (students-with-year 1))
    (c (courses-with-name "Software Engineering"))
    (where (member c (student-courses s)))))
```

While providing the conciseness and elegance of SQL, list comprehensions are more general because: (a) they allow arbitrary generator lists (that is, the lists over which identifiers such as s and c range may be arbitrary expressions, not just "base" extents), (b) they allow arbitrary predicates, not just arithmetic and string comparisons, and (c) the returned values can be of arbitrary type and computed using arbitrary functions, not just field projections. Further, since list comprehensions are expressions just like any other type of expression in the language, they may be embedded in procedures, embedded in recursive definitions, *etc.*

## Non-strict, implicitly parallel, eager semantics

AGNA pursues parallelism very aggressively:

- In a block:

  ```
  (letrec ((x1 e1)
            ...
           (xN eN))
     eBody)
  ```

  *all* expressions e1, ..., eN and eBody are evaluated in parallel, and the value of eBody may be returned as soon as it is available, even if the other expressions have not finished evaluating.

- In a primitive application:

  ```
  (+ e1 e2)
  (cons e1 e2)
  ...
  ```

  all arguments are evaluated in parallel, and some primitives may even return a result value before the arguments have finished evaluating (for example, cons and other object constructors).

- In a function application:

  ```
  (ef e1 ... eN)
  ```

  all expressions are evaluated in parallel; the function (value of ef) may be invoked as soon as it is known, and it may even return a result, before the arguments are known.

- In a conditional expression:

  ```
  (if e1 e2 e3)
  ```

  the predicate e1 is evaluated to a boolean value, after which one of the expressions e2 or e3 is evaluated and returned as the value of the expression.

In other words, *everything* is evaluated in parallel, except as controlled by conditionals and data dependencies. The semantics are:

- non-strict, but not lazy (we may evaluate expressions even if not needed for the result);
- eager, but not call-by-value (functions may be invoked before arguments are evaluated to values); and
- implicitly parallel (the programmer does not specify what must be done in parallel).

These semantics are borrowed from the Id programming language [10]. Programs evaluated under this regime often show massive amounts of parallelism. Id and AGNA are unique in this respect— we know of no other language that has these semantics or pursues parallelism this aggressively.

## 3 SQL-like optimizations

Compilation of AGNA transactions occurs in three major phases— source-to-source translation of the original transaction text, translation into dataflow program graphs, and translation into code for a multi-threaded abstract machine called P-RISC. Substantial code optimizations are performed at each stage.

The optimization and translation of list comprehensions takes place during the first phase of compilation. The primary function of this phase is to rewrite the input transaction text to a simpler form. While the AGNA query language is purely functional, some of the translations performed in this phase (and described in this paper) introduce non-functional constructs that are part of the internal intermediate languages.

### Combination of unary operations

A well-known algebraic transformation performed in relational database systems combines sequences of unary operations, applying them as a group, in order to avoid multiple traversals over large collections of data. This general transformation is also useful for improving list comprehensions. For example, consider the following query to find the names of all students with a GPA of at least 3.9:

```
(all (select s student name)
  (s (all student))
  (where (>= (select s student gpa) 3.9)))
```

A straightforward translation of this query first filters the list of all students, producing an intermediate list, over which the name selection function is then mapped.

```
(map (lambda (s) (select s student name))
     (filter (lambda (s) (>= (select s student gpa) 3.9))                    (T1)
             (all student)))
```

While this translation is simple and elegant, it can be improved by eliminating the construction and traversal of the intermediate list. This can be accomplished by combining the list filtering and mapping (two unary operations), and performing them both in a single pass over the student list. Here is a translation which includes this improvement.

```
(letrec (((p s)       (>= (select s student gpa) 3.9))
         ((t s)       (select s student name))
         ((f l p t r) (if (nil? l)
                          r                                                  (T2)
                          (let ((x (hd l)))
                            (f (tl l) p t (if (p x) (cons (t x) r) r))))))
  (f (all student) p t nil))
```

The three bindings define two unary functions p and t and one 4-argument, tail-recursive function f that performs the traversal.

We can summarize the principles behind this optimization in the following laws (expressed using a more readable notation than our Lisp-like transaction language), which are used as rewrite rules. Each rule converts two list traversals into a single traversal.

```
map f ∘ filter p       =    foldr (λ x ys.  IF p x THEN f(x):ys
                                            ELSE ys)
                                  []

filter p ∘ map f       =    foldr (λ x ys.  LET y = f(x)
                                            IN
                                              IF p y THEN y:ys
                                              ELSE ys)
                                  []

map f ∘ map g          =    map (f ∘ g)

filter p ∘ filter q    =    filter (λ x.  p x ∧ q x)
```

In addition, we use the following transformation which is only valid when the order of elements in a list is not relevant (this is indeed the case in our database queries):

```
foldr (λ x ys.  (f x)++ys) []      =      foldl (λ ys x.  (f x)++ys) []
```

This transformation converts `foldr` into `foldl`, which is sometimes preferable because it is iterative instead of recursive.

## Low-level filtering of base extents

For the filtering and transformation of an arbitrary list of objects, the translation scheme given above is optimal with respect to the number of cons operations performed to produce the result list (exactly one cons per output element). The value of the expression `(all student)` is obtained by scanning over the file that stores student objects and building a list in the volatile heap. When this list is available, it is then filtered and transformed in one scan.

However, a substantial improvement in performance can be obtained when the generator expression is a base extent, and the filters are simple predicates on the object fields. In this case, the filtering may be performed during file scanning, avoiding even the construction of the original list.

As in all database systems, AGNA is based on an "object storage system" that implements files and file scanning. The services provided by this module are similar to those provided in the Research Storage System (RSS) in System R [2], and WiSS, the Wisconsin Storage System [5]. The object storage system implements sequential object files, secondary BTREE and hash indexes, sequential and index object scans with predicates, and management of the cache of file pages. All persistent data access is performed through the object storage system, thereby insulating higher levels of the system from details of secondary storage such as data layout, whether access is through the OS file system or directly to a raw disk, *etc.*

The scan predicates supported by the object storage system are lists of conditions of the form $S\ \theta\ v$, where $S$ is a field name, $\theta$ is a relational operator such as equality, and $v$ is a value. We do not allow arbitrary AGNA predicates to be evaluated during the file scan because we would like predicate evaluation to be "quick", *i.e.,* matched to the speed at which the file scan is performed.

In the example query above, the condition describing the students of interest (GPA $\geq$ 3.9) is suitable for translation to such a low-level predicate, which we can then use in the scan of student objects. By performing the filtering within the storage system, a compact representation of the student extent (8K byte pages of objects) is scanned and filtered in an efficient manner. Here is a translation that incorporates this improvement:

```
(let ((pred (cons (make-condition gpa-slotid '>= 3.9) nil)))
  (map (lambda (s) (select s student name))
       (filter-extent STUDENT pred)))
```
(T3)

The extent filtering is performed by primitive procedure filter-extent, which takes a type identifier and a predicate object (a list of condition objects). In this case the predicate consists of a single condition, which is created by procedure make-condition. As we shall see soon, this use of low-level filtering provides a significant improvement in performance.

Another kind of improvement is possible for this query. Since the body expression simply projects the name field out of the student objects, we could easily pass this information down to filter-extent, which could then build a list of names directly, eliminating the need for the second pass (the map). While AGNA currently does not perform this optimization, we could add it easily.

## Use of indexes

One of the most important optimizations performed by relational systems is the use of efficient index structures. An index structure provides fast access to the records or objects in a file with a particular field value. Studies of relational systems have shown that the effective exploitation of indexes is essential for achieving good performance for a range of queries[3]. The experimental results presented in this paper indicate that the effective use of index structures is equally important for implementing list comprehension queries efficiently.

The object storage system in AGNA supports two types of secondary indexes— BTREE and hash. The decision to build an index on an object field is based on a user-supplied annotation in the type declaration. All user-defined object types have a "base" object file storing persistent objects. Index files for a field map field values to pointers to objects in the base file. By default, hash indexes are created for all fields with inverse mappings (*i.e.*, *<= and <= fields). This default behavior can be changed, however, by annotations in the type declaration. For example, if the student type were defined as follows:

```
(type STUDENT (extent)
  ((name   <=> STRING)
   (gpa     => FLOAT)
   (year  *<=> INTEGER (index btree))
   (bdate *<=> INTEGER (index btree))
   ...))
```

then three indexes would be created: a hash index on name, a BTREE index on year, and a BTREE index on bdate. In this example, BTREE indexes on year and birthdate may be preferred because of their ability to support range queries efficiently. For example, a BTREE index on bdate can be used to find all students with birthdates in a particular range, while a hash index can not be utilized in such a query. For the name field, on the other hand, a hash index may be preferred because of its ability to perform "exact match" name lookups more efficiently than a BTREE.

For a query which filters a base extent, there may be multiple "access paths" to the data. For example:

```
(all (select s student name)
  (s (all student))
  (where (and (== 1 (select s student year))
              (and (>= (select s student gpa) 3.9)
                   (< (select s student bdate) 720101)))))
```

There are at least three ways to implement this query: (1) scan the base student file with a predicate consisting of all three conditions; (2) use the index on year to locate first-year students, then apply the remaining conditions to the corresponding objects in the base file; and (3) use the index on bdate to locate students with birthdates before 1/1/72, then apply the remaining conditions to the corresponding objects in the base file.

Our compiler uses the following heuristics, listed in order of preference, to select an implementation strategy.

1. If a condition of the form $S = v$ exists on a field with a hash index, then use the index to find all objects with value $v$. Apply the remaining conditions to the objects returned. If conditions exist for more than one such field, then choose a <= field (unique inverse) over a *<= field. If more than one possibility still exists, then pick one arbitrarily.

2. If a condition of the form $S\ \theta\ v$ exists on a field with a BTREE index and $\theta$ is not the inequality operator, then use the index to find objects satisfying all such conditions on $S$. Apply the remaining conditions to the objects returned. If conditions exist for more than one such field, say $S_1$ and $S_2$, then use the following four steps to select one. (1) If a condition involving the equality operator exists for one field and not the other then choose the field with the equality condition. (2) If one field has a single-valued inverse (<=) and the other has a multiple-valued inverse (*<=), then choose the one with the single-valued inverse. (3) Choose more restrictive condition sets over less restrictive ones. For example, $S_1 > v_1$ and $S_1 < v_2$ is more restrictive than $S_2 > v_1$. (4) Pick a field arbitrarily.

3. If Rules 1 and 2 are not applicable, then simply scan the entire base file for objects which satisfy all conditions.

More sophisticated strategies are certainly possible, taking into consideration such things as the number of objects in the base extent, the number of blocks or units of disk allocation occupied by the base and index files, histograms describing distributions of field values, *etc.*; AGNA does not currently implement them.

The access path selected by the compiler is passed as arguments to `filter-extent-inverted` indicating the unique field identifier and the type of index to use (there may be more than one). For the example query above, all first year students are located, and then the conditions on GPA and birthdate are applied. All three conditions are applied within the object storage system, rather than explicitly materializing lists and filtering them. Here is the translation:

```
(let ((pred (cons (make-condition year-slotid '== 1)
                  (cons (make-condition gpa-slotid '>= 3.9)
                        (cons (make-condition bdate-slotid '< 720101)
                              nil)))))
  (map (lambda (s) (select s student name))
       (filter-extent-inverted STUDENT year-slotid BTREE pred)))          (T4)
```

## Experimental Results

To demonstrate the relative effectiveness of the optimizations discussed in this section, we executed the different list comprehension translations on a uniprocessor version of AGNA running on a Sun 4/490 workstation with 128 Mb of memory and an HP 97549 disk. The test database that we used contained 100,000 student objects. The student type was defined as follows:

```
(type STUDENT (extent)
 ((name    <=> (STRING 32))
  (gpa      => FLOAT)
  (bdate1 *<=> INTEGER (index btree))
  (bdate2   => INTEGER)))
```

The distribution of GPA values was such that 10000 students (10%) had GPAs of at least 3.9. The result lists were built in the heap and were not traversed by the top-level `print` procedure (*i.e.*, the timings do not include the time to print the result).

| | |
|---|---|
| Translation T1 (build student list, filter and transform it): | 163 seconds |
| Translation T2 (combine map and filter): | 161 seconds |
| Translation T3 (filter during object retrieval): | 20 seconds |

The final experiment that we performed on the uniprocessor version of AGNA finds all students with birthdates in a particular range:

```
(all s
  (s (all student))
  (where (and (> (select s student bdate1) v₁)
              (< (select s student bdate1) v₂))))
```

Values $v_1$ and $v_2$ were chosen so that 1000 students were selected by the query (1% of all students). The query was also repeated using **bdate2** instead of **bdate1**. Fields **bdate1** and **bdate2** contained identical values, so that the answers were the same. In the first case, the optimizer exploits the **bdate1** index, whereas in the second case, the system must scan the entire file of student objects.

| | |
|---|---|
| 1% indexed selection (using **bdate1**): | 0.25 secs |
| 1% non-indexed selection (using **bdate2**): | 9.00 secs |

The results of these tests indicate the importance of efficient filtering and indexing in the implementation of list comprehension queries in AGNA.

### Comparison with INGRES

We also ran these queries on INGRES version 6.3, a modern commercial relational database system. We built a corresponding database with the same size, and ran it on the same hardware platform (Sun 4/490, 128 Mb memory, HP 97549 disk). The SQL equivalent of the query on **gpa** is:

```
SELECT name
FROM   student
WHERE  gpa >= 3.9
```

The INGRES timings were:

| | |
|---|---|
| **gpa** query: | 5.9 secs |
| 1% indexed selection (using **bdate1**): | 0.4 secs |
| 1% non-indexed selection (using **bdate2**): | 5.0 secs |

These results indicate that even on a uniprocessor, AGNA is well within shooting distance of commercial relational systems on comparable queries. Further, we have an advantage over SQL in that AGNA's list comprehensions are part of a full functional language, so that the user can smoothly extend queries to perform arbitrary computation.

## 4 Optimizations for parallelism

Our implementation of AGNA is designed primarily for MIMD multi-processors consisting of processor-memory elements (PMEs), with or without attached disks, interconnected via a high-speed network (see Figure 1). The network may be a bus, in small multiprocessors, or a switching network in larger machines.

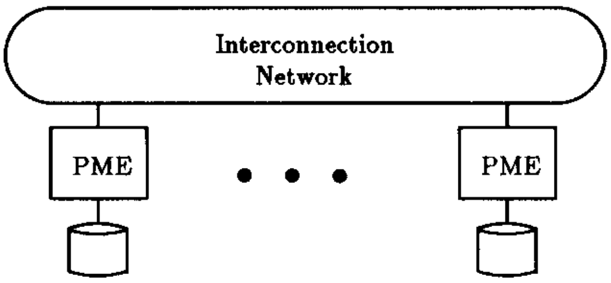


Figure 1: Machine organization.

While a number of object distribution strategies are possible, we currently distribute objects in a random fashion across all PMEs. In other words, (**allocate** *t*) allocates an object of type *t* on a PME chosen randomly. This strategy can be expected to provide a fairly uniform distribution of objects across the machine. The files at each PME (*i.e.*, the base and index files) are organized as described in the previous section.

## Non-strictness, parallelism, and open lists

The non-strict semantics in AGNA provide many opportunities for parallel execution. Consider translation T1 from the previous section, repeated here:

```
(map (lambda (s) (select s student name))
     (filter (lambda (s) (>= (select s student gpa) 3.9))
             (all student)))
```

In a strict implementation, we must build the entire student list (*i.e.*, evaluate `(all student)`), filter it, and then perform the map. Each operation must execute entirely before the next one may begin. With the non-strictness in AGNA, as soon as the first cons cell in the student list is allocated, a reference to it can be returned as the result of `(all student)` (shown in Figure 2). A reference to a student object is stored in the head of the list, while the tail is left empty ($\perp$ in the figure).

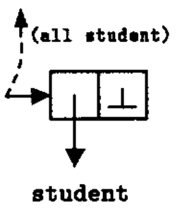


Figure 2: First cons cell in student list.

Construction of the remainder of the student list and the filter operation may then proceed in parallel. If the filter operation attempts to read the tail of a cons cell that is empty (*i.e.*, one that contains $\perp$), then it simply blocks, waiting for a value to be stored there. When the write finally arrives, the blocked read operation is notified, allowing the list traversal to continue. Cons cells and other data structures in AGNA have single-assignment semantics, so we can be assured that all readers of a data structure element will receive the same value.

A similar kind of parallelism also exists between the filter operation, which produces a filtered list of student objects, and the map operation, which consumes it. In fact, this form of parallelism is possible in AGNA between *any* computation which produces a list, and a computation that consumes it, not just those involving extent lists, or filtered extent lists. In [1], this producer/consumer parallelism, and other forms of parallelism due to non-strictness, are shown to be pervasive, even in programs that use traditional algorithms.

The non-strictness in AGNA can also be used to append lists efficiently. Say, for example, that we want to append lists L1 and L2. By choosing representations for L1 and L2 that embody two key ideas, we can append the lists in $O(1)$ time while using no additional storage. The first idea is to leave the tail of the last cons cell in each list empty. This will allow us to link the end of L1 directly to the head of L2 using no additional storage (*i.e.*, we don't have to build any intermediate lists). The second key idea is to maintain a direct reference not only to the first cons cell in each list, but also to the last one as well. This will allow us to locate the end of each list in constant time, so that we can link L1 to L2, and terminate L2 by storing `nil` in the tail of its last cons cell.

A representation for lists which includes these two ideas is called an *open list*. Open lists, which are closely related to difference lists in logic programming, can be represented as a cons cell containing two references A1 and A2, as shown in Figure 3. Internally, the structure consists of a list in which the tail of the last cons cell is empty. A1 points at the first cell and A2 points at the last cell (these may in fact be the same cell, if there is only one element in the list). With L1 and L2 represented as open lists, we can append them and return the result list as follows.

```
(par
  (set-cons-tl (tl L1) (hd L2))
  (set-cons-tl (tl L2) Nil)
  (hd L1))
```

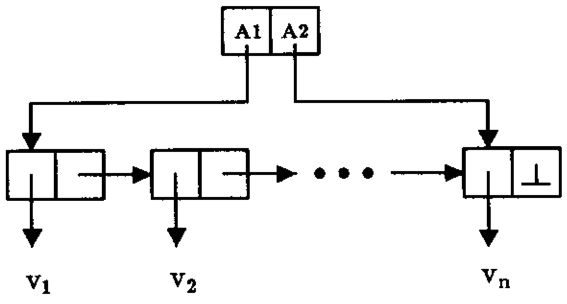


Figure 3: The structure of an open list.

Par is a construct which evaluates all component expressions in parallel, returning the value of the last one. The first call to procedure `set-cons-tl` stores a reference to the head of L2 in the last cons cell of L1. The second call to `set-cons-tl` terminates the list by storing `Nil` in the tail of the last cons cell in L2. The final expression returns the result list, which is not an open list.

## Exploiting the locality of data

In a parallel environment, performance of a query can often be improved dramatically if the locality of the data being queried can be exploited. This is especially true in a persistent object system, where the volume of data may be large (*e.g.*, terabytes).

We start with a very simple example:

```
(all student)
```

*i.e.*, an expression that simply evaluates to the list of all students. Student objects are distributed over the various PMEs of the parallel machine. The basic idea is for each PME to construct a list of all its students, and then to append these lists together. However, we would like the PMEs to work in parallel, and for the list-appending to be very efficient, *i.e.*, we wish to avoid constructing any intermediate lists.

Each PME executes the following procedure (we will see in a moment how it is persuaded to do this):

```
(local-extent2 STUDENT rest)
```

producing the result shown in Figure 4. Because of the non-strict semantics in AGNA we can construct

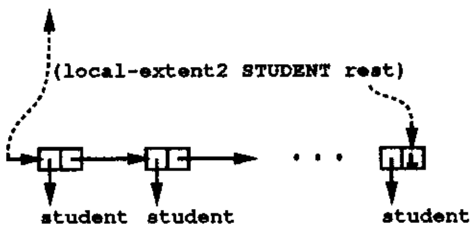


Figure 4: Local extent constructed on each PME.

the local list of student objects before we know the value of the `rest` argument. The last cons cell simply remains empty. As soon as the first cons cell is allocated, a reference to it can be returned as the result of `local-extent2`. When the value of `rest` arrives, which may be much later, it gets stored into the last cons cell.

Now, we step up one level to see the implementation of `(all student)`. Conceptually, the appending of the lists is accomplished by structuring the computation thus:

```
(foldr (lambda (j rest) (APPLY PME j local-extent2 STUDENT rest))
     Nil
     pmes)
```

Pmes is a list of PME numbers. The form:

```
(APPLY PME j f e1 ... eN)
```

evaluates the expression `(f e1 ... eN)` on PME *j*. Thus, the overall structure of the computation may be seen in Figure 5. The `foldr` computation runs on some PME *i*, and initiates the `local-extent2`

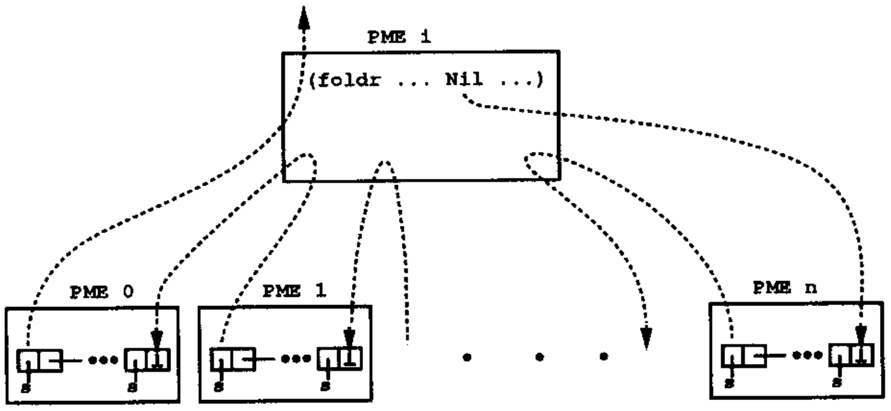


Figure 5: Local extent constructed on each PME.

computations on each PME (including its own PME). Because of non-strictness, all these computations can proceed in parallel, even though each one does not yet have the value of its `rest` parameter. Further, each PME *j* can return the reference to the head of its sublist as soon as it is allocated; this reference is passed on by `foldr` as the `rest` parameter for the computation in PME $j-1$, where it is stored in the tail of the last cons cell.

The basic strategy used to implement `(all student)` can also be used to implement filtering of base extents. Consider the following query from the previous section.

```
(all (select s student name)
  (s (all student))
  (where (>= (select s student gpa) 3.9)))
```

We can exploit the locality of student data by dispatching the predicate to all PMEs on which student objects reside, where local extent filtering may then proceed in parallel using the techniques described in the previous section. The local result lists are appended efficiently as described above. Here is a translation which implements this strategy:

```
(let ((pred (cons (make-condition gpa-slotid '>= 3.9) Nil)))
  (foldr (lambda (j rest) (APPLY PME j local-filter-extent STUDENT BASEFILE NOINDEX pred rest))    (T5)
         Nil
         pmes))
```

Procedure `local-filter-extent` performs the local filtering and linking of sublists. Its first argument identifies the type extent to filter. Its second and third arguments indicate the access path chosen by the compiler. Constants `BASEFILE` and `NOINDEX` indicate that use of an index in this query is not possible, so the filter must be performed by scanning all student objects in the base file. When an index is available, the second argument identifies the field on which the index exists, while the third argument identifies which index to use (there may be more than one). The fourth argument to `local-filter-extent` is the predicate, expressed as a list of conditions, and the fifth argument is the remaining result list.

Here is the definition of `local-filter-extent`:

```
(define (local-filter-extent t ap index p rest)
  (let ((res (filter-extent3 t ap index p)))
    (if (nil? res)
        rest
        (par                                        ;; in parallel
          (set-cons-tl (tl res) rest)
          (hd res))))
```

Local result lists are produced by primitive procedure **filter-extent3**, which is passed the type id, the access path, the index to use, and the predicate. Unlike procedure **filter-extent2** from the previous section (used in translation T4), **filter-extent3** returns either **nil**, if no local objects are found which satisfy the predicate, or an open list.

The use of an open list by **filter-extent3** allows all local result lists to be appended in an efficient manner by threading the head of the result list through each invocation of **local-filter-extent**. The overall structure of the computation is shown in Figure 6. Again, the **foldr** computation runs on some PME *i*, and initiates the **local-filter-extent** computations on each PME, including its own. The first invocation receives the initial result of **nil** through its **rest** argument, which it stores in the tail of its local result list. It then returns a reference to the head of the local list. The second invocation receives this reference, stores it in the tail of its local result list, and returns the new intermediate result list. This process continues until the last invocation is reached, which appends its local list and returns the final result to **foldr**. If a local inversion produces no result (*i.e.*, **filter-extent3** returns **nil**), then the incoming result list is simply passed along.

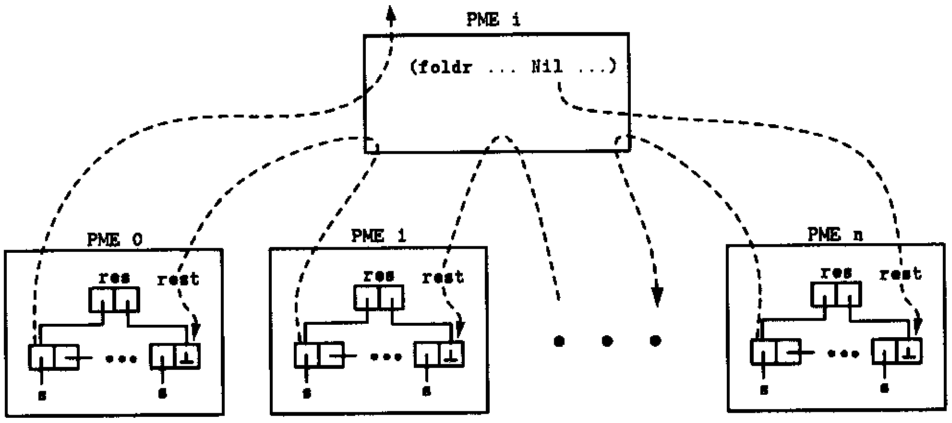


Figure 6: Local filtered extent constructed on each PME.

## Experimental Results

We conducted three sets of experiments to evaluate the performance of simple list comprehensions in AGNA in a parallel environment. The tests were performed on an Intel iPSC/2 with 32 nodes. Each node consisted of an Intel 80386 processor, 8 megabytes of physical memory, and a 330 megabyte MAXTOR 4380 disk drive.

The experiments that we conducted varied the machine size (*i.e.*, number of PMEs), the problem size, or both. The first set of experiments evaluated performance relative to extent size by keeping the machine size constant while scaling up the problem size. The second set evaluated the scalability of the system by maintaining a constant problem size while increasing the number of PMEs in the machine configuration. Finally, the third set of experiments evaluated the ability of the system to maintain a constant response time as the problem size and machine size were increased proportionally. The design of our experiments was motivated by the performance study reported in [7].

The queries that we used were based on the selection queries in the Wisconsin benchmark [3]. All three sets of tests used the same object structure and query template. The object structure was:

```
(type TESTOBJ (extent)
 ((unique1  => INTEGER)
    (unique2 <=> INTEGER (index btree))
    (filler   => (STRING 200))))
```

and the query template was:

```
  (all t
      (t (all testobj))
      (where (and (> (select t testobj unique1) v1)
                  (< (select t testobj unique1) v2))))
```

While the relation structure specified in the Wisconsin benchmark has more fields than the test object, only unique1 and unique2 were used by the part of the benchmark that we ran. Thus, we simply combined all unused fields into the `filler` field for simplicity. The total object length was the same as the Wisconsin benchmark relation width.

The unique1 and unique2 fields were assigned values from the range 1 - $n$, where $n$ was the number of objects used in a particular test. This allowed values $v_1$ and $v_2$ to be varied easily to test queries that returned different numbers of objects. The field used in the predicate expression was varied (between unique1 and unique2) to test indexed vs. non-indexed access. The result lists returned by the queries were built in the heap, and were not traversed by the `print` function, *i.e.*, the times for printing the results were not included. In all cases, the test objects were distributed uniformly across all nodes in the machine configuration used. All query times reported are elapsed time in seconds until all computation was complete.

## Performance relative to extent size

In this set of experiments, the machine configuration was kept constant at 32 PMEs while the extent size was increased from 100,000 to 10,000,000 objects. We ran two queries on unique1 (the non-indexed field), selecting 1% of the objects in the base extent in the first one, and 10% in the second one. We also ran two queries on unique2 (the indexed field), selecting 1% of the objects in the first one, and a single object in the second one.

While running the tests, we noticed that one of the PMEs was significantly slower than the others (about 25% slower). This PME was used only in the full machine configuration of 32 nodes, so that the results of all the experiments run on 32 nodes are somewhat skewed. For this set of experiments, all of the results are skewed equally. We have not yet isolated the cause of this problem.

Ideally, the increase in response time would not grow at a rate faster than the increase in extent size. The results are tabulated below.

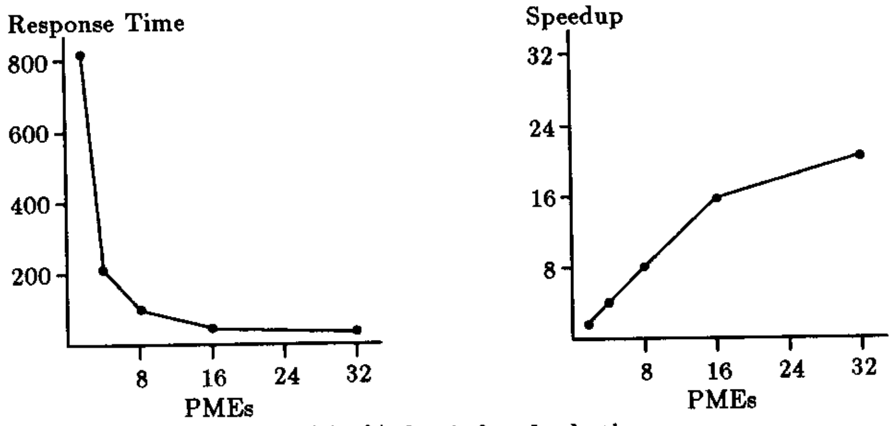
| Query Description | 100,000 | 1,000,000 | 10,000,000 |
|---|---|---|---|
| 1% non-indexed selection | 7.1 | 38 | 389 |
| 10% non-indexed selection | 7.2 | 40 | 410 |
| 1% indexed selection | 3.6 | 4.3 | 9.5 |
| 1 object using index | 3.6 | 3.6 | 3.6 |

For the non-indexed selections, the increase in response time is almost perfectly matched with the increase in extent size from 1,000,000 to 10,000,000 objects (38 to 389 seconds, and 40 to 410 seconds). For the increase in extent size from 100,000 to 1,000,000 objects, this is not the case because with 100,000 objects the time it takes to begin a transaction, dispatch the local filter operations, append the local result lists, and end the transaction becomes more significant (almost half of the total time) relative to the time spent filtering on each node. Response time for the indexed selections is completely dominated by this overhead.

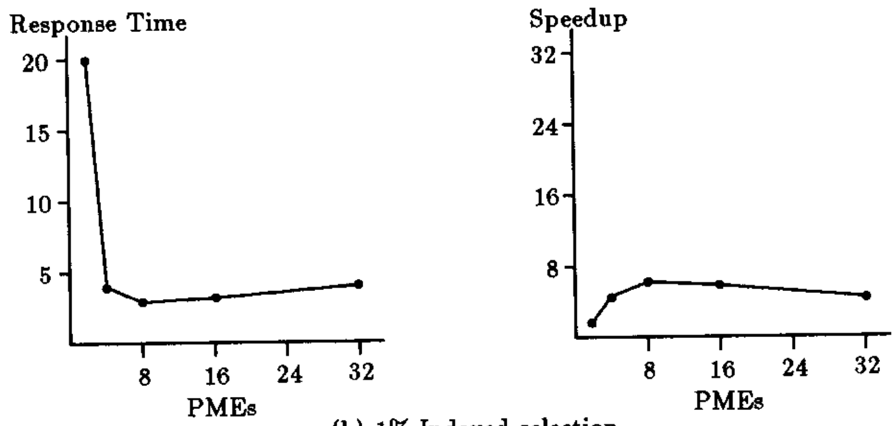
One area that we have identified for improvement that should reduce the overhead of short transactions is identifier lookups (such as `begin-transaction`, `local-filter-extent`, *etc.*) in the top-level environment. The top-level environment is spread over the entire machine, and identifier lookups currently involve searching on all PMEs. We are exploring an alternative strategy based on hashing which should make identifier lookups much more efficient.

## Speedup

In this set of experiments, we kept the extent size constant at 1,000,000 objects while the machine configuration was increased from 1 to 32 nodes. The ideal behavior in this case would be for response time to decrease (or speedup) proportionally with increases in machine size. Results for the 1% selection queries are shown in Figure 7. (We also ran the 10% selection query on a non-indexed field; the response time and speedup curves for it are almost identical to those for the 1% non-indexed selection.)



(a) 1% Non-indexed selection.



(b) 1% Indexed selection.

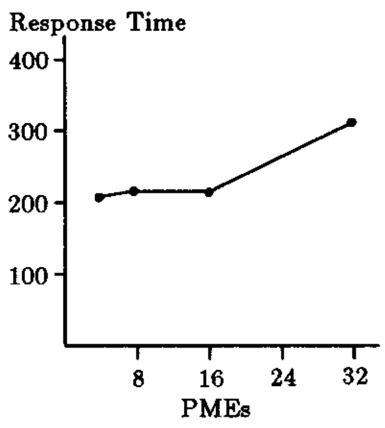
Figure 7: Speedup for 1% selection.

The speedup for non-indexed selections is almost perfectly linear from 1 to 16 nodes. The speedup from 16 to 32 nodes is not, however, due to the slowness of one of the nodes. For the indexed selection, the response time actually increases beyond 8 PMEs due to the increased transaction overhead. Again, we believe much of this is due to top-level identifier lookups. Also, with larger machine configurations, it takes longer to dispatch the local filter operations, and to append the local result lists.

## Scaleup

In the final set of experiments, we increased the machine and extent sizes proportionally. The ideal behavior in this case would be for response time to remain constant. The results are shown in Figure 8.

Response time remains relatively constant through 16 PMEs. The slight increases in response time from 4 to 8 and from 8 to 16 PMEs are due to increased transaction overhead and the increased overhead in performing the filter operation. The large jump in response time from 16 to 32 nodes is due, again, we believe, to the slowness of one of the nodes.



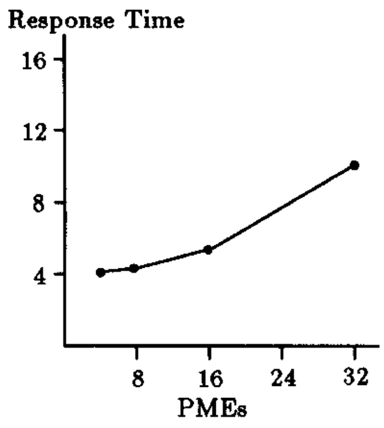


Figure 8: Scaleup for 1% selection with index (left) and without index (right).

**Comparison with Gamma**

The Gamma Parallel Relational Database Project (at the University of Wisconsin) has also reported results for the selection queries that we used in our experiments. A thorough performance evaluation of Gamma, on the same hardware platform, is reported in [7].

For queries that involve a significant amount of processing relative to the overhead of a transaction (*e.g.*, a non-indexed selection on all 32 nodes with an extent size of 10,000,000 objects), Gamma is anywhere from two to five times faster than AGNA. For very short transactions, the difference is more significant, often greater than a factor of ten. We believe that the differences in performance can be attributed to three things:

1. The P-RISC abstract machine, the target of the AGNA compiler, is currently emulated in software. Further, we know of numerous optimizations possible on P-RISC code that we have not yet had time to incorporate. From hand coded examples, we believe that these optimizations and better compilation can easily increase performance by a factor of 10 or more.
2. AGNA implements objects in a segmented, paged, virtual heap. The management of this heap, including its paging to disk, is implemented entirely in software which has not been optimized very much yet. Further, our heap structure is more complicated due to the more complex object model supported by AGNA.
3. Much less effort has gone into AGNA to date to tune the system.

We are very encouraged by our results, and believe that we are within shooting distance of parallel relational systems on comparable queries. Again, we have the advantage that in AGNA, the programmer can smoothly escalate to more complex objects and queries that involve general networks of objects and arbitrary computation.

## 5 Concluding Remarks

To our knowledge, AGNA is the first implementation of a parallel persistent object system, whether based on functional languages or not. Further, although functional languages have been proposed and prototyped as database query languages before, we believe that AGNA is the first implementation whose performance approaches that of state of the art relational database systems. The performance of AGNA relies heavily on:

- Optimization of list comprehensions, based on reducing the number of intermediate lists, use of indexes to combine generators and filters, and moving filters as close to disk i/o as possible.

- Aggressive pursuit of parallelism, based on fine grain, non-strict evaluation.

We plan to implement additional list comprehension optimizations in AGNA in the near future. One area of focus will be nested generators, the equivalent of join queries (cross-products) in relational systems. We plan to investigate traditional methods for implementing such queries, as well as new ones that can exploit the direct object references that are part of the AGNA object model.

We are also looking for other benchmark programs, especially ones that involve complex objects and general computation, and benchmark results from other persistent systems.

**Acknowledgements**

We thank Prof. David DeWitt of the University of Wisconsin, Madison, for giving us access to the Intel iPSC/2 at Wisconsin, and for assisting us in the actual use of the machine. Funding for this work is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-89-J-1988. Support for the Michael Heytens is provided in part by an Intel Graduate Fellowship.

## References

[1] Arvind, D. E. Culler, and G. K. Maa. Assessing the Benefits of Fine-grained Parallelism in Dataflow Programs. *International Journal of Supercomputer Applications*, 2(3), 1988.

[2] M. Astrahan et al. System R: Relational approach to database management. *ACM Transactions on Database Systems*, 1(2), June 1976.

[3] D. Bitton, D. J. DeWitt, and C. Turbyfill. Benchmarking database systems: A systematic approach. In *Proceedings of the 1983 Conference on Very Large Data Bases*, August 1983.

[4] O. P. Buneman, R. E. Frankel, and R. S. Nikhil. An implementation technique for database query languages. *ACM Transactions on Database Systems*, 7(2):164–186, June 1982.

[5] H.-T. Chou, D. J. DeWitt, R. H. Katz, and A. C. Klug. Design and implementation of the wisconsin storage system. *Software—Practice and Experience*, 15(10):943–962, October 1985.

[6] C. J. Date. *A Guide to the SQL Standard.* Addison-Wesley, Reading, Massachusetts, 1987.

[7] D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, March 1990.

[8] M. L. Heytens. *The Design and Implementation of a Parallel Persistent Object System.* PhD thesis, MIT, 1991 (expected).

[9] R. S. Nikhil. Functional Databases, Functional Languages. In *Proceedings of the 1985 Workshop on Persistence and Data Types, Appin, Scotland.* Department of Computing Science, University of Glasgow, and Department of Computational Science, University of St. Andrews, Scotland, July 1987 (revised).

[10] R. S. Nikhil. Id (Version 90.0) Reference Manual. Technical Report CSG Memo 284-1, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA, July 1990.

[11] R. S. Nikhil and M. L. Heytens. Exploiting Parallelism in the Implementation of AGNA, a Persistent Programming System. In *Proc. 7th IEEE Intl. Conf. on Data Engineering, Kobe, Japan*, April 8-12 1991.

[12] P. Trinder. *A Functional Database.* PhD thesis, Oxford University, Cambridge, England, May 1989.