Monet

An Impressionist Sketch of an Advanced Database System

Peter A. Boncz *
University of Amsterdam
boncz@fwi.uva.nl

Martin L. Kersten
University of Amsterdam/CWI
mk@{fwi.uva.nl, cwi.nl}

November 1994

Abstract

Monet is a customizable database system developed at CWI and University of Amsterdam, intended to be used as the database backend for widely varying application domains. It is designed to get maximum database performance out of today's workstations and multiprocessor systems. It has already achieved considerable success in supporting a Data Mining application [12, 13], and work is well under way in a project where it is used in a high-end GIS application. Monet is a type- and algebra-extensible database system and employs shared memory parallelism. In this paper, we give the goals and motivation of Monet, and outline its architectural features, including its use of the Decomposed Storage Model (DSM), emphasis on bulk operations, use of main virtual-memory and server customization. As a case example, we discuss some issues on how to build a GIS on top of Monet; amongst others how Monet can handle the very large data volumes involved.

*Parts of this work are supported by SION grant no. 612-23-431

Monet *

An Impressionist Sketch of

an Advanced Database System

November 1994

Abstract

Monet is a customizable database system developed at CWI and University of Amsterdam, intended to be used as the database backend for widely varying application domains. It is designed to get maximum database performance out of today's workstations and multiprocessor systems. It has already achieved considerable success in supporting a Data Mining application [12, 13], and work is well under way in a project where it is used in a high-end GIS application. Monet is a type- and algebra-extensible database system and employs shared memory parallelism. In this paper, we give the goals and motivation of Monet, and outline its architectural features, including its use of the Decomposed Storage Model (DSM), emphasis on bulk operations, use of main virtual-memory and server customization. As a case example, we discuss some issues on how to build a GIS on top of Monet; amongst others how Monet can handle the very large data volumes involved.

1 Introduction

Developments in personal workstation hardware are at a high and continuing pace. Main memories of 128 MB are now affordable, and custom CPUs currently can perform at 50 MIPS, relying on efficient use of registers and cache to tackle the disparity between processor and main memory cycle time, that increases every year with 40% [16]. These hardware trends pose new rules to computer software – and to database systems – as to what algorithms are efficient. This article describes the design and implementation of an efficient, parallel, customizable, main memory-oriented database server called Monet. It has been under developed at the CWI and the University of Amsterdam since 1992.

Let us take a closer look at the design goals of Monet:

- *high performance.* Monet is designed to achieve optimal database performance on current hardware, with the above mentioned trends in mind.
- *customizability.* When a database server is to be used as the backend for applications in widely varying domains, it has to provide a way to customize its functionality. That is, not only it has to provide type-extensibility (allowing for instance the introduction of GIS datatypes and access methods), but also there is a need for adding application specific code to the server, that serves the need those specific applications.
- *basic database services.* While page-servers [6, 15] make efficient use of hardware facilities and provide flexibility, we feel that these systems make too much compromise to data independence [7]. Monet therefore has a table-oriented datamodel with value-based query facilities, but uses simple, basic algorithms to avoid the overhead typically found in a complex DBMS.

1.1 Principle Ideas

We now outline the principal ideas that we applied to achieve Monet's design goals.

- perform all operations in *main memory.* Monet makes aggressive use of main memory by assuming that the hot-set of the database fits into main memory. All its primitive database operations work on main memory, no hybrid algorithms are used. For a class of datasets, this assumption holds. We are, however, also interested in using Monet in application areas where it does not. In section 4.1 we propose a virtual memory management technique, which transparently converts Monet's behavior to the traditional disk-oriented approach when this is really necessary.
- employ inter-operation *parallelism.* The system exploits shared-store and all-cache architectures. A distributed shared-nothing approach is described in [23, 24]. Unlike mainstream parallel database servers like PRISMA [2] and Volcano [9], Monet

*Parts of this work are supported by SION grant no. 612-23-431

does not use tuple- or segment-pipelining. Instead, the algebraic operators are the units for parallel execution. Their result is completely materialized before being used in the next phase. This approach benefits throughput at a slight expense of response time and memory resources.

- use lean *bulk operations*. Studies like [16, 18] show that cache profiling can speed up a program by a factor of two. Deeply nested function calls cause CPUs to run out of register spaces, to similar performance penalties.

  Tuple-oriented database algorithms perform a sequence of different operations for every tuple in a set-operation, and thus reference many different memory locations repeatedly, easily causing register overflow and cache misses. In contrast, bulk operations iterate over a set of contiguous tuples in one go, executing one basic operation on all of them, typically without a doing a function call in the inner loop. This allows for a more efficient use of registers and (more intuitively) of cache memory[1].

- use one *simple datamodel*. Monet uses a simple data model based on Binary Association Tables (BATs). This allows for flexible object-representation using the Decomposed Storage Model (DSM) [4]. This vertical decomposition also helps partitioning the database such that the tables fit easier in main memory. The fixed-width tables allow for much optimization of kernel operations through heavy use of code-inlining.

- allow users to *customize* the database server. Since Monet is intended for use by different applications and programming paradigms, it does not provide one single user interface language. Its interface consist of an execution-level BAT algebra. Monet provides extensibility much like in the Gral system [10], where any new user command can be added to its interpreter, and its implementation linked into the kernel. The Monet grammar structure is fixed, but parsing is purely table-driven on a per-user basis. Users can change the parsing tables at run-time by loading and unloading modules.

- *Portability* The system is written in C and documented using a literate programming style. Programmers will find it easy to find their way around, modify it, and test the system for compliance with previous versions. However, the Monet algebraic programming language and its customizability interface will suffice for most application requirements.

Figure 1 shows the Monet Interpreter as a multi-threaded process, connected to its clients via TCP/IP links. Applications typically accompany themselves by

[1]It is often impossible to obtain hard data about CPU-cache performance. The sole option is to redo a run against a machine simulator.

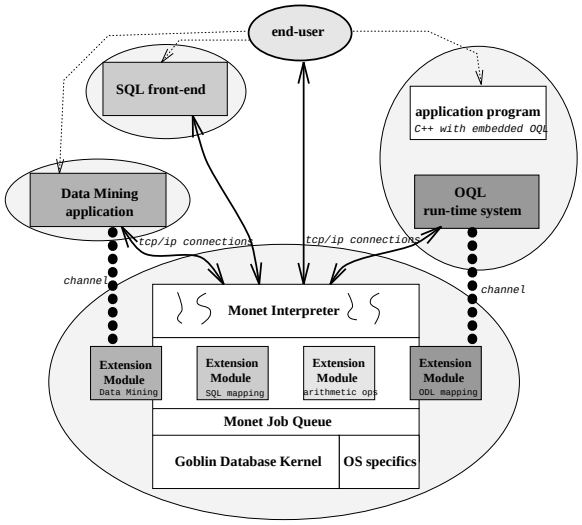

Figure 1: The Monet Architecture

a specific extension module, which provides the extra functionality needed by them. Extension modules can provide operations doing things ranging from arithmetic operations and GIS specific functionality to an SQL-to-BAT algebra query scheduler/optimizer.

2 Design Overview

The low-level functionality of Monet is implemented by the Goblin Database Kernel (GDK), which provides the binary datamodel, table persistence and basic concurrency and transaction mechanisms, comparable to [11]. The latter two mechanisms are not the topic discussed in this paper. On top of GDK are the – dynamically loadable – user-defined modules. The upper layer is formed by the multithreaded Monet Interpreter. We will now discuss these system components in some more detail.

2.1 GDK

The GDK datamodel partitions every relation vertically in Binary Association Tables (BATs). The left column of a BAT is called its *head*, the right column is called *tail*. A persistent BAT has its representation saved to disk. The first time a BAT is used, it is loaded from disk, only to be swapped back when GDK's heat-based BAT buffer manager decides so. Thus, when a BAT is used, it resides entirely in main memory, such that purely main memory algorithms can be used.

Traditional relational algorithms are disk-block oriented and try to achieve sequential access to ranges of tuples, since disk IO cost is dominant in query execution cost. In main memory systems it is necessary to work with different cost models. By absence of IO, query processing cost tends to be dominant [1]. This has important consequences to which auxiliary datastructures work best for optimizing access paths to data in main

memory tables. Algorithms typically have to be simple and lightweight – access to main memory is very fast – and are not block-oriented, because random access is almost equally expensive as sequential access in main memory [17].

For these reasons, the built-in GDK search accelerators are binary index trees (for range queries) and bucket-chained hashing, both of which are created on the fly during processing, when it is known to speed up a BAT operator.

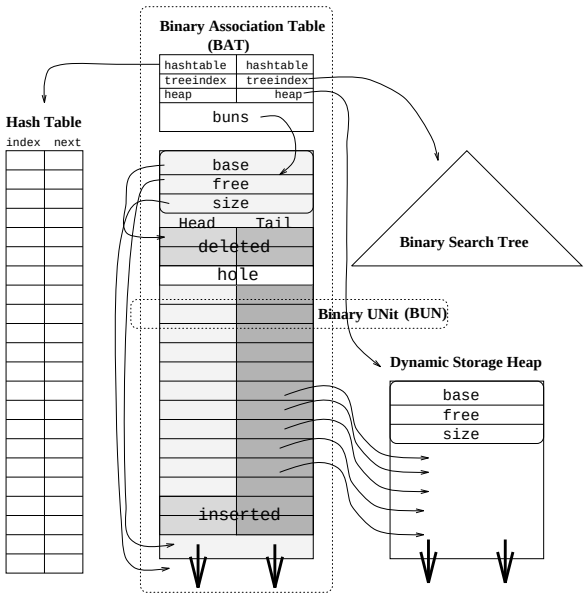

Figure 2: Binary Association Tables

2.1.1 Atomic Values

The data stored in the tables are atomic values. These can either be values of built-in types (integer, pointer, OID, string, character or float) or user-defined types. For the latter types, an Abstract Data Type (ADT) facility is provided, which will be discussed in more detail in Section 3.2.1.

GDK types can either be of fixed-size or variable-size. In the first case, atom values directly reside in the tuples (called *BUNs*) of a BAT. In the latter case, the BUN contains an integer index into a heap, where the variable-sized value can be found. A BAT can have a heap for any of its two columns.

For variable-sized atomic values there are two rules:

- every value resides in a – disjunct – contiguous range of memory. This makes it possible to manipulate loose – single – values as easily as groups of values, stored in a heap.

- a value may not contain any 'hard' pointers. Linked lists can still be implemented with integer indices or distances between nodes.

This design makes GDK's datastructures completely memory-location independent, such that no conversion overhead is involved when data is copied or moved. In effect, a heap stored on disk is completely identical to a heap in memory, which makes it possible to treat it as a memory-mapped file (see Section 4.1).

2.2 Monet Interpreter

The Monet Interpreter is a textual interface on top of GDK. It accepts a scripting language called the Monet Interpreter Language (MIL).

- MIL provides access to the database tables in the form of *persistent BAT-variables*, on which BAT algebra commands can be executed.

- The algebra *commands* and *operators* are comparable with "execution algebra" commands from Gral [10], in that they are simply executed without further optimizing transformations.

- MIL commands can be *overloaded* via the polymorphous type `any`. Depending on the actual parameters of a command invocation, a suitable implementation is chosen at run-time. This allows one to provide alternate implementations for different types, such that per-tuple typechecking within commands can be factored out.

- MIL is a block-structured language, in which every blocks creates a new variable scope. A *sequential block* is denoted by '{..}'. A *parallel block*, denoted by '[..]', specifies that the statements in the block can be executed in parallel. It is the programmer's responsibility to ensure that parallel statements do not interfere with each other[2].

- Apart from that, MIL provides standard *control structures* such as `IF. .ELSE` and `WHILE`.

- *Procedures* can be assembled from pieces of script. Recursion is allowed.

- A special construct are *iterators*, which iterate over some subset of a BAT expression, one BUN at a time. For each BUN, the head- and tail-values are supplied as parameters to a MIL-statement, which is then executed.

  Its syntax is: `<BAT-expr> @ <MIL-stmt>`. Inside the statement, the templates "$1" and "$2" mark the places where the head- and tail-values are substituted, on every tuple invocation. The parallel variant: `<BAT-expr> @[N] <MIL-stmt>`. executes the MIL-statement in parallel on a number of – maximally `N` – selected BUNs at a time.

- Since the Monet Interpreter identifies BATs with a special kind of OIDs, it is possible to have *nested relations*, by storing these "BAT-IDs" (BIDs) in a BAT.

[2] Monet provides BAT-level locking primitives.

In all, the MIL provides a rich environment for experimentation purposes. Figure 3 gives an inside look into the Monet Interpreter. Multiple interpreter threads can be active at any time, scheduled via a shared job-queue. A daemon allows for connections being made to Monet from the internet.

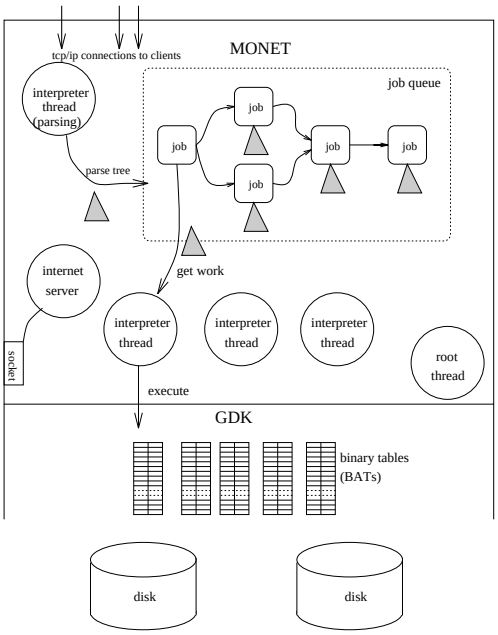

Figure 3: The Monet Interpreter

2.2.1 Rationale

We chose an execution-level BAT algebra as the query interface for Monet, since it is intended to be used as backend for very different applications. Those applications must address the issue of translating a user-level query (e.g. in SQL) to a sequence of BAT commands, in the process of which query optimization is done.

The BAT algebra works on binary tables only. The total vertical decomposition does not affect in any way the generality of the system. In [14, 23] algorithms have been given to transform queries on multi-attribute objects into sequences of primitive BAT operations. Moreover, such algorithms tend to significantly outperform multi-attribute systems, if the number of target attributes is smaller than the total number of attributes of a relation, while otherwise performance drops only slightly under the multi-attribute case [14]. Since BAT operations are executed on thin tables – which reside in main memory – normally complex operations (like join or semijoin) tend to be very cheap. For example, joining two 100K tables without supporting index won't take Monet more than 600ms on a SGI Indy. Vertical decomposition makes the data smaller, such that Monet is effective in only loading the database hot-set into memory, and better exploits the CPU cache.

2.2.2 Example

We show in an example what the BAT algebra looks like. Consider the following SQL query on relations `company [name,telephone]` and `supply [comp#, part#, price]`:

```
SELECT company.name,
       company.telephone,
       supply.quantity
FROM   company, supply
WHERE  supply.comp# = company.comp# AND
       supply.part# = part_no AND
       supply.price < 0.50)
```

In Monet's SQL frontend, the relational database scheme will be decomposed into five tables named `comp_name`, `comp_telephone`, `supply_comp`, `supply_part` and `supply_price`, where in each table the *head* contains an OID, and the *tail* contains the attribute value.

```
{
   VAR m_supply, m_comp;
   VAR m_name, m_telephone, m_quantity;

   m_supply := SEMIJOIN(supply_part.SELECT(part_no),
                        supply_price.SELECT(0.0, 0.50));
   m_supply := MARK(m_supply);
   m_comp := JOIN(m_supply, supply_comp);
   [
      m_name      := JOIN(m_comp, comp_name);
      m_telephone := JOIN(m_comp, comp_telephone);
      m_quantity  := JOIN(m_supply, supply_quantity);
   ]
   PRINT(m_name, m_telephone, m_quantity);
}
```

The variables created in the query cease to exist afterwards, because the whole is enclosed in a sequential block. The three last joins are executed in parallel on multiprocessor systems, because they are placed in a parallel block.

In all, the original double-select, single-join, three-wide projection SQL query is transformed in a sequence of 8 BAT algebra commands. We describe in short the semantics of the BAT commands used:

BAT command	result
<AB>.mark	$\{o_i a \mid ab \in AB \wedge unique\_oid(o_i)\}$
<AB>.semijoin(CB)	$\{ab \mid ab \in AB, \exists cd \in CD \wedge a = c\}$
<AB>.join(CD)	$\{ad \mid ab \in AB \wedge cd \in CD \wedge b = c\}$
<AB>.select(Tl,Th)	$\{ab \mid ab \in AB \wedge b \geq Tl \wedge b \leq Tl\}$
<AB>.select(T)	$\{ab \mid ab \in AB \wedge b = Tl\}$
<AB>.find(T)	$\{a \mid aT \in AB\}$

The dot notation "`a.oper(b)`" is equivalent to function call notation "`oper(a,b)`". Note that `JOIN` projects out the join columns. The `MARK` operation has the special semantics of introducing a column of unique new OIDs for a certain BAT. It is used in the example query to create the new – temporary – result relation.

3 Customizable Databases

When a database is used for more than administrative applications alone, the need for additional functionality quickly arises [20].

- First of all, new application domains typically require – complex – *user-defined data-types*, such as for instance Polygons or Points.
- Having these datastructures, one often wants to define new *predicates and functions* on them (Intersect(p_1, p_2) or Surface(p), for example).
- Also, new application domains often create a need for new *relational operators*, such as spatial join or polygon overlay.
- In order to evaluate queries using the new predicates, functions and relational operators, one needs new *search accelerators* (such as – for instance – R-Trees).
- Finally, applications using a database as backend want the option to perform certain application-specific operations near to the data. If a database server allows one to *add user-supplied pieces of code* to it, the communication penalties of creating a separate server process, encapsulating the database (a "client-level" server), can be avoided.

Putting all such new code in the kernel of a database that is intended to support widely varying applications – with widely different functionality – will make for a large, difficult-to-maintain system. Clearly, one wants an extensible design, in which the database can be modularly customized in different directions.

3.1 Other Systems

In the early days, relational systems only provided a rigid interface via SQL or QBE. Later versions of INGRES, and the Postgres system [22] are examples of relational systems that do allow for the introduction of new data types and access methods via prefixed ADT interfaces. This works fine for new datatypes, predicates on them, and their accelerators, but does not allow for addition of new relational operators. In recent years, database researchers have spent much effort on Object-Oriented databases [15]. In these systems, the programmer has more control, but to the point that data independence is compromised and the system gets hard to debug [7]. Another effort to achieve customizability has been the "extensible-toolkit" approach, where a database can be assembled by putting together a set of "easily" customizable modules (see [6]). Putting together such a system remains a serious work, however. One of the most appealing approaches to the problem we find in the Gral system [10], which accepts a many-sorted algebra. Such an algebra can by its nature easily be extended with new operations.

3.2 Extensibility in Monet

Monet's extension system most resembles Gral, supporting new data types, new search accelerators, and user defined primitives (embodying both new predicates and new relational operators). The support for new data types and accelerators is implemented in the GDK layer through ADT interfaces, whereas support for new operations is done in the Monet Interpreter.

Monet extensions are packaged in modules, that can be specified in the Monet Extension Language (MEL). Implementations must be provided in the form of precompiled C compliant object-code. Both module-specification and -implementation are fed into the `Minstall` utility (one of several special-purpose utilities coming with the Monet server), that parses the specification, generates additional code, updates Monet's module tables, and stores the object files in the system directories.

3.2.1 Atomic Types

The atomic types are part of the GDK layer. The possibility to include new types is implemented there, and simply funneled upward to the extension system. The ADT interface assures that GDK's built-in accelerators still work on user-defined types. For instance, one of the standard ADT operations is `AtomHash()`, which ensures that GDK's hash-based join works on BATs of any type. The ADT interface also contains routines to copy values to and from a heap, and to convert them to and from their string representations (for user interaction). Below we show how an atom can be specified, and which ADT operations should be defined:

```
ATOM <name> ( <fixed-size> )
  FromStr   := <fcn>; # parse string to atom
  ToStr     := <fcn>; # convert an atom to string
  Compare   := <fcn>; # compare two atoms
  Hash      := <fcn>; # compute hash value
  Null      := <fcn>; # create a null atom
  Put       := <fcn>; # put atom in a BAT
  Get       := <fcn>; # get atom from a BAT
  Delete    := <fcn>; # delete atom from a BAT
  CleanHeap:= <fcn>; # clean up the heap
  NewHeap   := <fcn>; # generate a new atom heap
END <name>;
```

In case of a fixed-sized atom, the `Put`, `Get` and `Delete` operations, perform the trivial task of updating some BUNs in the BAT. In case of a variable-sized atomic type, they have the additional task of updating the heap.

3.2.2 Search Accelerators

Just like the atom ADT interface, the accelerator ADT is implemented at the GDK level. GDK provides passive support for user-defined search accelerators via an ADT interface that maintains user-defined accelerators

under update and IO operations. The support is "passive" since basic GDK operations only use the built-in accelerators for their own acceleration.

The rationale behind this is to provide both flexibility and efficiency for the way in which search accelerators are used. The standard interface to relational access paths is the triple of functions: `open()`, `findnext(<predicate>)`, and `close()`. A single accelerator can be used to accelerate evaluation of multiple predicates (e.g. the same R-Tree can be used to accelerate polygon overlay, as well as various types of spatial joins). Hence, the `findnext()` operation – which is typically called at the deepest nested loop of database algorithms – has to do some check on its parameters in every invocation to see which predicate is being evaluated.

Bearing in mind that search accelerators in Monet accelerate for main memory access, one sees that the traditional approach may soon lead to performance degradation. We think that the access path interface for such a system is best implemented by a collection of C macros, and hence falls outside the reach of the (precompiled) GDK layer. User-defined primitives that "know" the accelerator semantics can just use such macros, other primitives cannot. The ADT interface merely serves to ensure that an accelerator remains up-to-date under GDK operations.

```
ACCELERATOR <name>
  Build   := <fcn>; # build accelerator on a BAT
  Destroy := <fcn>; # destroy accelerator
  Insert  := <fcn>; # adapt acc. under BUN insert
  Delete  := <fcn>; # adapt acc. under BUN delete
  Commit  := <fcn>; # adapt acc. for transaction commit
  Rollback:= <fcn>; # adapt acc. for transaction abort
  Load    := <fcn>; # load accelerator from disk
  Save    := <fcn>; # save accelerator to disk
END <name>;
```

Note that one can choose to make user-defined accelerators persistent (by implementing the `Load` and `Save` operations), though this is not the policy adhered for GDK's built-in accelerators.

3.2.3 New Primitives

The MIL grammar has a fixed structure but depends on purely table-driven parsing. This allows for the runtime addition of new commands, operators, and iterators. Moreover, every user has an individual keyword-table, such that different users can speak different "dialects" of MIL at the same time. All system tables have been implemented as BATs and are accessible to the user via persistent variables for debugging purposes.

In order to do type-checking at the highest possible level, the Monet Interpreter has been equipped with a polymorphism mechanism. One can specify a certain command, operation or iterator multiply, with differing parameter lists. Upon invocation, the Monet Interpreter decides which implementation has to be called, based on the types of the actual parameters. For invocations inside loops, optimizations have been made, such that – generally – name resolution is done on the first invocation only.

The MEL syntax for specifying new primitives is as follows:

```
COMMAND  <name> ( <type-list> )       : <type> := <fcn>;
ITERATOR <name> ( <type-list> )                := <fcn>;
OPERATOR <name> ( <type> )            : <type> := <fcn>;
OPERATOR ( <type> ) <name> ( <type> ) : <type> := <fcn>;
```

The implementations of all commands, operators, iterators and even ADT operations have to be supplied by the programmer in compiled C code. For now, Monet assumes database kernel programmers, who know what they are doing, and allows for the dynamic linkage of any C object-file or library into its running kernel.

4 Big Datasets

In the context of the MAGNUM project, Monet's extension mechanism is being used to support a high-end GIS application. This immediately leads to the question whether a main memory oriented approach is feasible in this field. In GIS, data volumes tend to be very large, as shown in the numbers below, taken from the Sequoia benchmark [21]. It shows the sizes of the three benchmarks it specifies, and the numbers of tuples in them (between parentheses). Similar requirements have been formulated in [8].

Point	Polygon	Graph	Raster	
2 Mb (76 K)	20 Mb (100 K)	50 Mb (300 K)	1 Gb	**Regional**
28 Mb (1 M)	300 Mb (1.5 M)	1 Gb (6.5 Mb)	17 Gb	**National**
300 Mb (10 M)	3 Gb (15 M)	10 Gb (65 M)	2 Tb	**World**

SEQUOIA Benchmark Sizes

On a workstation with 128 MB of main memory, Monet performs well until the database hot-set reaches 60 MB. Beyond that, the system will start swapping, until the BATs operated upon even exceed swappable memory.

Still, one should bear in mind that GIS algorithms typically employ filtering steps, in which much smaller relations are used, before using the voluminous polygon or graph data [3]. Raster data get decomposed in tiles, that are not loaded directly into main memory unless needed, such that the tuple-cardinality and size also stays modest.

Filtering algorithms in GIS use approximations, for example, minimum bounding rectangles (MBRs), for handling of polygon or graph data. Since a BUN consisting of a `<OID, MBR>` is 20 bytes long, we see that regional benchmark relations approximating the polygon and graph data would have sizes 2 Mb and 6 Mb, respectively, which can easily be handled in main memory.

Even for the national benchmark these sizes are 30 Mb and 130 Mb, which – possibly with the help of some fragmentation – can also be made to work in a large main memory.

The above reasoning shows that the approximation steps on relatively large GIS data often can be performed in main memory. However, after the filtering steps, such algorithms still need to access the big tables, in order to perform the final steps on the filtered objects. It is clear that these tables cannot economically be held in main memory. Therefore we use virtual memory primitives, supplied by modern operating system architectures.

4.1 Memory Mapped Files

In recent years, there has been an evolution of operating system functionality towards micro-kernels, i.e. those that make part of the OS functionality accessible to customized applications. Prominent prototypes are Mach, Chorus, and Amoeba, but also conventional systems like Silicon Graphics' IRIX and Sun's Solaris[3] provide hooks for better memory and process management.

Stonebraker discarded the possibility of using memory mapped files in databases [19], on the grounds that operating systems did not give sufficient control over the buffer management strategy, and the fact that virtual management schemes waste memory. Now – a decade later – we think the picture has changed. Operating systems like Solaris and IRIX do provide hooks to give memory management advice (`madvise`), lock pages in memory (`mlock`[4]), invalidate and share pages of virtual memory. This is why recently interest of the database community in these techniques has revived [11].

Monet uses the virtual memory management system call `mmap` to map big BATs into its main memory. The database table is mapped into the virtual memory as a range of virtual memory addresses. When addresses are accessed, page faults occur, and the pages are loaded on need.

This scheme has a number of advantages:

- the only upper limit to the size of the tables is the virtual address space. Monet currently runs on Sun and SGI machines that have a 32-bit addressing scheme. This leads to an address space of 4 Gb, which for the present is enough. Future CPUs will be equipped with 64-bits addressing, like DEC's Alpha already is.
- the scheme is totally transparent in Monet. Recall that Monet's datastructures are memory location independent, and take the same form on disk as in main memory.
- it provides flexibility. Orthogonally, one can decide to treat the table with BUNS and/or the associated heaps of a BAT as memory mapped files or not.
- it is efficient in loading database tables, since only the pages needed are loaded, and in saving them, because MMU hardware support guarantees that only dirty pages are written.

4.1.1 Block-Oriented Algorithms

In effect, memory mapped files bring Monet's approach back to the traditional, disk-block oriented algorithms – but only where this is really necessary.

In a block-oriented system, clustering of tuples which have values near to each other, can give considerable performance benefits, a principle on which both B^+-trees as R^+-trees are based. Such structures are very efficient in accelerating range queries, which is important in GIS.

A block-oriented strategy can also be applied in Monet's virtual memory environment. Memory mapped BATs can benefit greatly from clustering BUNs on memory pages. Figure 4 shows how an bulk-build R-Tree algorithm is used in the `cluster` command to reorder the BUNs in a memory mapped BAT. Both the supporting R-Tree as the BAT itself, are treated as memory mapped files. The `cluster` command rearranges the BUNs. The net effect of it all is very similar to the working of a traditional R-tree: a select on the range "G..I" causes three page faults before clustering, which after clustering is reduced to only one page fault.

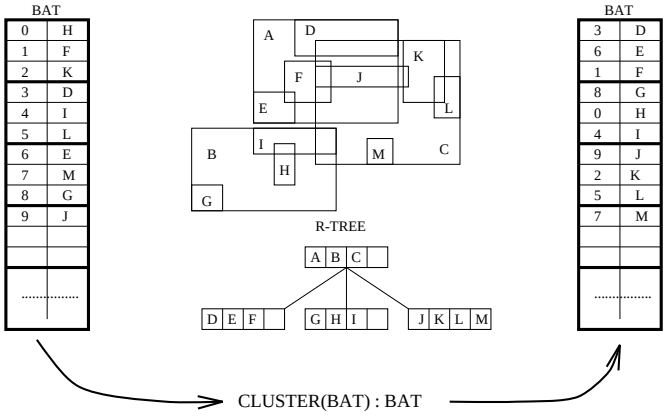

Figure 4: Clustering BATs in Memory Pages

One should note that the `cluster` operation is implemented as a separate BAT command. Its working is totally transparent to all other BAT Algebra commands – it just has an accelerating effect on them.

[3] These are the two platform on which Monet is currently supported.

[4] One has to have Unix root permission for this.

4.2 Case Example: supporting GIS in Monet

We demonstrate how Monet's extension facilities can be used to store GIS data, and to express GIS algorithms. This is done by writing a module that introduces GIS datatypes, new commands and operations working on those datatypes (or BATs containing such datatypes), and new search accelerators to aid in their execution.

4.2.1 Module Specification

We show the specification part in MEL only, since implementations have to be given in the form of C routines.

The GIS module introduces rectangles, circles, polygons (and polylines) to the system, the intersection operator, and some commands. We use a four-word representation for rectangles, and a three-word representation for circles, such that their fixed sizes in the BUNs are 16 and 12, respectively. Since polygons are variable-sized atoms, their representation resides in a heap – their fixed part in the BUN is only a 4-byte integer index into such a heap.

```
MODULE Gis;
    ATOM Polygon(4)
        ToStr    := polygonToStr;
        FromStr  := polygonFromStr;
        Hash     := polygonHash;
        Compare  := polygonCmp;
        ...
    END Polygon;

    ATOM Rectangle(16)
        ...
    END Rectangle;

    ATOM Circle(12)
        ...
    END Circle;

    ACCELERATOR RTree;
        Insert   := rtreeInsert;
        Delete   := rtreeDelete;
        Destroy  := rtreeDestroy;
        ...
    END RTree;

    OPERATOR (Polygon) "!!" (Polygon) : BIT
                := polygonIntersect;
    COMMAND ClipLine(Polygon, Rectangle) : Polygon
                := lineClip;
    COMMAND ApproxLine(Polygon, INT) : BAT(OID, Rectangle)
                := line2BoxSeq;
    COMMAND SJ_Intersect( ... ) ...
    COMMAND SJ_ExistIntersection( ... ) ...
    COMMAND SJ_ExistEnclosure( ... ) ...
    ...
END Gis;
```

The command `Approxline(`$poly, n$`)` produces an approximation of the polyline *poly* in just n rectangles which cover the entire polyline in minimal space.

The last three operations are new semijoin operators on BATs of rectangles or circles. The command `SJ_Intersect()` selects all BUNs that intersect with a certain polyline value, `SJ_ExistIntersection()` and `SJ_ExistEnclosure()` select all BUNs for which a BUN in a second BAT exists, that intersects with it, or is enclosed by it, respectively.

4.2.2 Data Model

The data model used in this example is a recursive notion of Area. Its ODL [5] syntax would be as follows:

```
interface Area {
   attribute Set<Area> subareas;
   attribute Polygon   poly;
   attribute Rectangle MBR;
   attribute Circle    MEC;
   attribute int       population;
   relation  City      capital;
}
```

An area has its form described by a polygon. Two approximations are availabe: a Minimum Bounding Rectangle (MBR), and a Maximum Enclosed Circle (MEC). These approximations will be used later on to provide a state-of-the-art spatial join [3]. Furthermore, thematic attributes like population and capital are present.

To demonstrate the use of nested relations in Monet, we mapped this definition to a nested set of BATs.

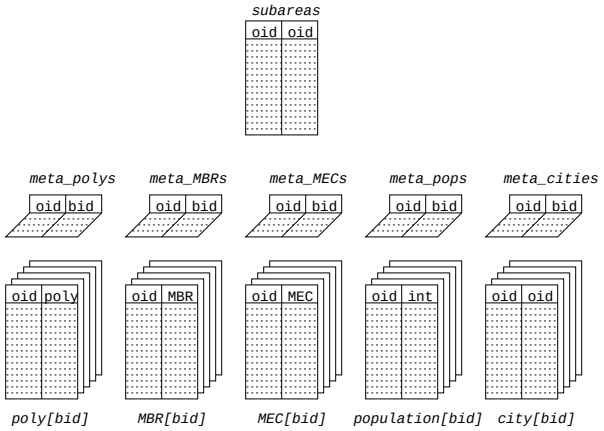

Figure 5: Recursive DSM Schema

The BAT *subarea* describes which relations are direct subareas of others. We assume hierarchical data to be stored in the model, e.g. states, containing counties, containing blocks. The *subarea* BAT would contain $< oid_{state}, oid_{county} >$ and $< oid_{county}, oid_{block} >$ BUNs. It can be used to either find out which areas are sub-area of a certain area, or to find its enclosing super-area.

We assume a syntetic root-area instance exists, which encloses the top-level areas (e.g. the country enclosing all states). Each set of sub-areas defined by some area is stored in a DSM-decomposed relation. Since any non-leaf area defines such a relation, there is a multitude of DSM-decomposed relations. Meta-relations consisting of

nested BATs describe which BAT contains the sub-area information for each non-leaf area.

The nice property of this model, is that is allow different resolutions in a hierarchical tree. If necessary, some counties may have no blocks, or some blocks may be decomposed to greated detail into yet another set of areas.

4.2.3 GIS Algorithms

Part of the GIS functionality has already been introduced via new BAT commands in the GIS module. On top of them, one can easily construct new commands using the scripted-procedure facility of the MIL. This has two advantages:

- It allows for fast scripting, which eases experimentation. Procedure interpretation cost is minimized, but one should still strive to use only bulk commands (operations on BATs) in the innermost loops of the procedures.
- It allows the programmer to use the MIL facilities for expressing parallelism, either via parallel blocks, or via parallel iterators

In this example we show how the MIL can be used to answer complex intersection queries. In particular, we want to answer queries like:

"which {states, counties, blocks, parcels} in the United States intersect with highway 66?"

The following MIL procedure call comes up with the answer, employing both parallelism and advanced GIS methods:

```
Decompose(#USA, #highway66, {0, 1, 2, 3}).
```

We now discuss both MIL scripted-procedures that we use to accomplish this:

- `AreaIntersectLine()`.

  This procedure computes a multi-step intersection semijoin on an area with a polyline. Its parameters are a set of polygons (with additional approximation relations using MBRs and MECs), and a polyline. It returns all polygons from the set that intersect with the polyline. In its computation, it first approximates the polyline by a set of (100) rectangles that overlap it. Then, it does an intersection semijoin on the MBR approximation with this set of rectangles, to determine the *candidate-set* of intersecting polygons (line 7). In parallel, an enclosure-semijoin is computed between the MEC approximation and the line representation, to obtain the set of *certain-hits* (line 8). Only on the candidates that survive the substraction of the certain-hits (line 12), the expensive polygon-to-polygon intersection operation is performed (line 14)

  The semijoin in line 10 can be a very big operation, since it works on the – typically voluminous – polygon BAT. This BAT might well be a memory mapped file. In that case, one could argue that a R-Tree traverse semijoin should be applied instead of a standard semijoin, in order to optimize sequential access to clustered polygons. This could be implemented by overloading `semijoin()`, or by providing a specialized `traverse()` command.

- `Decompose()`.

  This procedure recursively traverses the hierarchical data-model till *level* levels deep, and performs a multitude of spatial semijoins on the leaves. In line 24, the parallel `HASHLOOP(<value>)` iterator, finds all OIDs that are subarea of the current area. The statements from line 25 to 30 are executed in parallel, at most 20 at-a-time. Consequently, the `INSERT()` operation of line 28 needs to be locked against parallel interference. When arrived at the right level in the hierarchical tree, the meta-relations are used to find the BATs on which the actual work is to be transformed (line 33-35). Note that the clipping done in line 25-26, already provides a filtering step, such that no work is wasted in areas where there surely can be no intersection.

Note that there is an open issue which parameters (e.g. the granularity of parallelism, or the polyline approximation resolution in line 5) work best. This paper does not seek to investigate GIS algorithms, we just show that these are easily expressed and experimented with in Monet.

```
 1 PROC AreaIntersectLine (polys, mbrs, mecs, line)
 2 {
 3  VAR recheck, hits;
 4  {                          # new seqblock for local vars
 5    VAR filter, approx := ApproxLine(line, 30);
 6    [                        # parblock
 7      filter := SJ_ExistIntersection(mbrs, approx);
 8      hits   := SJ_ExistEnclosure(mecs, approx);
 9    ]
10    recheck := polys.SEMIJOIN(filter);
11    hits := recheck.SEMIJOIN(hits);
12    DELETE(recheck, hits);
13  }
14  hits.INSERT(SJ_Intersect(recheck, line));
15
16  RETURN hits;
17 }
18
19 PROC Decompose(AreaId, line, level)
20 {
21  IF (level > 0) {
22    VAR clip, result := NEW(OID, Polygon);
23
24    subareas@[20]HASHLOOP(AreaId) { # parallel iterator
25      clip := ClipLine(line,
26                       meta_MBRs.find(AreaId).find($2));
27      IF (clip != NIL) {
28          LOCK(INSERT(result, Decompose($2, clip, level-1));
29      }
30    }
31    RETURN result;
32  }
33  RETURN AreaIntersectLine(
34           meta_polys.find(AreaId), meta_MBRs.find(AreaId),
35           meta_MECs.find(AreaId), line);
36 }
```

5 Conclusions

Monet is a database system that takes an offbeat view on current design issues. It uses the Decomposed Storage Model (DSM) to vertically fragment all relations and to provide for a simple data model. It employs inter-operation parallelism and executes all – bulk – operations in main memory, allowing for high performance. Monet has a flexible extensibility interface, and uses virtual-memory techniques to handle very large data sets. The Monet Interpreter Language, together with the extensibility mechanism, provides powerful functionality to support new application domains, as we show in the GIS example. The system is already used on a daily basis for Data Mining application [12, 13] while work is still under way in the project to support an ODL/OQL C++ binding with Monet as backend.

References

[1] A. Analyti and S. Pramanik. Fast search in main memory databases. In *Proc. ACM SIGMOD Conf.*, page 215, San Diego, CA, June 1992.

[2] P. M. G. Apers, C. A. van den Berg, J. Flokstra, P. W. P. J. Grefen, M. L. Kersten, and A. N. Wilschut. PRISMA/DB: A parallel main memory relational DBMS. *IEEE Trans. on Knowledge and Data Eng.*, 4(6):541, December 1992.

[3] T. Brinkhoff, H. Kriegel, R. Schneider, and B. Seeger. Multi-step processing of spatial joins. In *23 ACM SIGMOD Conf. on the Management of Data*, pages 197–208, June 1994.

[4] G. Copeland and S. Khoshafian. A decomposition storage model. In *Proc. ACM SIGMOD Conf.*, page 268, Austin, TX, May 1985.

[5] R.G.G. Catell et al. *The Object Database Standard.* Morgan Kaufman, 1993.

[6] et al. Carey,M. and DeWitt,D. The EXODUS extensible DBMS project: An overview. In *In 'Readings in Object-Oriented Database Systems" edited by S.Zdonik and D.Maier, Morgan Kaufman.* 1990.

[7] et al. Neuhold,E. and Stonebraker,M. Future directions in DBMS research. *ACM SIGMOD RECORD*, 18(1), March 1989. Also published in/as: ICCS, Berkeley, TR-88-1, Sep.1988.

[8] A. U. Frank. Properties of geographic data: Requirements for spatial access methods. In *Advances in Spatial Databases, 2nd Symposium, SSD '91*, pages 225–235, Zurich, Switzerland, August 1991. Springer Verlag.

[9] G. Graefe. Encapsulation of parallelism in the volcano query processing system. In *19 ACM SIGMOD Conf. on the Management of Data, Atlantic City*, May 1990.

[10] R. H. Guting. Gral: An extensible relational database system for geomatric applications". In *Proceedings of the 15th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Amsterdam*, August 1989.

[11] D. Lieuwen H. V. Jagadish, R. Rastogi, A. Silberschatz, and S. Sudarshan. Dalí: A high performance main memory storage manager. In *Proceedings of the 20th VLDB Conference, Santiago, Chile.*, pages 48–59, September 1994.

[12] M. Holsheimer and M. L. Kersten. Arcitectural support for data mining. Int. Workshop on Knowledge Discovery in Databases, Seattle, 1994.

[13] M. Holsheimer, M. L. Kersten, and A. Siebes. Data surveyor: searching for nuggets in parallel. In *Knowledge Discovery in Databases.* MIT Press, Cambridge, MA, USA, 1995.

[14] S. Khoshafian, G. Copeland, T. Jagodits, H. Boral, and P. Valduriez. A query processing strategy for the decomposed storage model. In *Proc. IEEE CS Intl. Conf. No. 3 on Data Engineering, Los Angeles*, February 1987.

[15] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Comm. of the ACM, Special Section on Next-Generation Database Systems*, 34(10):50, October 1991.

[16] A.R. Lebeck and D.A. Wood. Cache profiling and the spec benchmarks: A case study. *IEEE Computer*, 27(10):15–26, October 1994.

[17] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *Proceedings of the 12th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Kyoto*, August 1986.

[18] A. Shatdahl, C. Kant, and J.F. Naughton. Cache conscious algorithms for relational query processing. In *Proceedings of the 20th VLDB Conference, Santiago, Chile.*, pages 510–521, September 1994.

[19] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 14(7), July 1981.

[20] M. Stonebraker. Inclusion of new types in relational database systems. In *Proc. IEEE CS Intl. Conf. No. 2 on Data Engineering, Los Angeles*, February 1986. Also published in/as: UCB/ERL memo M85/67, Jul.1985.

[21] M. Stonebraker, J. Frew, K. Gardels, and J. Meredith. The Sequoia 2000 storage benchmark. In *19 ACM SIGMOD Conf. on the Management of Data, Washington,DC*, May 1993.

[22] M. Stonebraker and G. Kemnitz. The POSTGRES next-generation database management system. *Comm. of the ACM, Special Section on Next-Generation Database Systems*, 34(10):78, October 1991.

[23] C. A. van den Berg. *Dynamic Query Optimization.* PhD thesis, February 1994.

[24] C. A. van den Berg and M. L. Kersten. An analysis of a dynamic query optimisation scheme for different data distributions. In J. Freytag, D. Maier, and G.Vossen, editors, *Advances in Query Processing*, pages 449–470. Morgan-Kaufmann, San Mateo, CA, 1994.