Monet And Its Geographic Extensions:

a Novel Approach to High Performance GIS Processing *

Peter A. Boncz, Wilko Quak, Martin L. Kersten
University of Amsterdam, CWI
{boncz,quak,mk}@fwi.uva.nl

Abstract

We describe Monet, a novel database system, designed to get maximum performance out of today's workstations and symmetric multiprocessors.

Monet is a type- and algebra-extensible database system using the Decomposed Storage Model (DSM) and employing shared memory parallelism. It applies purely main-memory algorithms for processing and uses OS virtual memory primitives for handling large data. Monet provides many options in memory management and virtual-memory clustering strategies to optimize access to its tables.

We discuss how these unusual features impacted the design, implementation and performance of a set of GIS extension modules, that can be loaded at runtime in Monet, to obtain a functional complete GIS server.

The validity of our approach is shown by excellent performance figures on both the Regional and National Sequoia storage benchmark.

1 Introduction

In recent years, consensus has been reached in the GIS community about the advantages of extensible database systems. In these systems, all data – thematic, geometric, and raster – is captured by a single datamodel. Queries containing both thematic and geometric primitives can be formulated in one language, and be optimized globally. Object-relational systems that extend their data- and query-model with GIS types and primitives are currently available [19, 5].

Storage and querying of geographic data in an extensible database system still poses severe performance challenges to current database technology. Data stored in a GIS is typically complex of nature (long polygons with topological inter-relationships) and large of size (raster data, sometimes even arriving in a continuous stream from satellites observing the earth).

The large datavolumes involved require the DBMS to use very efficient resource management strategies. Important aspects are I/O, multi-level cache performance, memory management and multiprocessor usage. Geographic query optimization requires an efficient use of spatial access paths and approximation steps [3], and a way to extend the query optimizer with geometric knowledge [10].

In this article we present Monet [2], a novel database server, intended to serve as backend in various application domains. It has already achieved considerable successes in Data Mining [12] and for supporting O-O traversals[1].

We discuss how Monet's architectural features provide opportunities for algebraic optimization and parallelization of queries, and how we extended Monet with geographical primitives. The performance effectiveness and scalability of our approach is demonstrated by excellent results obtained on respectively the Regional and National Sequoia Benchmark.

Our work on Monet forms the lower layer of the MAGNUM project – underway since 1994 – that aims at building a high performance parallel GIS database system with ODMG compliant O-O technology, employing a spatial reasoning system for query optimization, and a state-of-the-art user interface.

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

[1] For more details and actual information on Monet, see http://www.cwi.nl/cwi/projects/monet.html

2 Architecture of Monet

Monet is a novel database kernel under development at the CWI and UvA since 1994. Its development is based on both our experience gained in building PRISMA [1], a full-fledged parallel main-memory RDBMS running on a 100-node multi-processor, and on current market trends in database server technology.

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 over 50 MIPS. They rely more and more on efficient use of registers and cache, to tackle the ever-increasing disparity[2] between processor power and main memory bus speed. These hardware trends pose new rules to computer software – and to database systems – as to what algorithms are efficient. Another trend has been the evolution of operating system functionality towards micro-kernels, i.e. those that make part of the Operating System functionality accessible to customized applications. Prominent research prototypes are Mach, Chorus and Amoeba, but also commercial systems like Silicon Graphics' Irix and Sun's Solaris increasingly provide hooks for better memory and process management.

Given this background, we applied the following ideas in the design of Monet:

- *binary relation model.* Monet vertically partitions all multi-attribute relationships in Binary Association Tables (BATs, see Figure 1), consisting of `[OID,attribute]` pairs.

  This Decomposed Storage Model (DSM) [4] facilitates table evolution, since the attributes of a relation are not stored in one fixed-width relation. In a GIS setting, this means that Monet can easily choose to start maintaining precomputed functions on a table with geometric data, by creating a new – separate – BAT with this information.

  The price paid for DSM is small: the slightly bigger storage requirements are compensated by Monet's flexible memory management using heaps. The extra cost for re-assembling multi-attribute tuples before they are returned to an application, is negligible in a main-memory setting, and is clearly outweighed by saving on I/O for queries that do not use all the relation's attributes.

  Finally, maintaining all attributes in a different table enables Monet to cluster each attribute differently, and to precisely advice the operating system on resource management issues, for each attribute according to its access path characteristics.

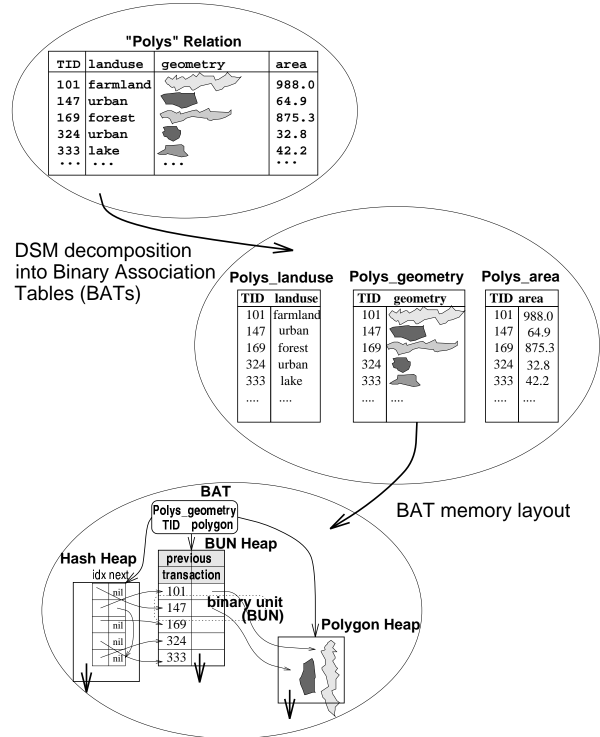

Figure 1: An example of Monet's decomposed storage scheme, in a GIS-extended application

  Figure 1 shows how relations are stored in BATs. The left column is referred to as *head*, the right column as *tail*. A BAT has at least 1 and at most 5 associated *heaps*, which form the basic memory structure of Monet. There is always a heap that contains the (fixed-size) atomic value pairs, called Binary UNits (BUNs). For atoms of variable size – such as `string` or `polygon` – both head and tail can have an associated heap (the BUNs then contain integer byte-indices into that heap). Finally, persistent search accelerators – for instance hash tables – may be stored in separate heaps, for both head and tail.

- *perform all operations in main memory.* Monet makes aggressive use of main memory by assuming that the database hot-set fits into main mem-

[2] In recent years this disparity has been growing with 40% each year

ory. All its primitive database operations work on this assumption, no hybrid algorithms are used. For large databases, Monet relies on virtual memory by mapping large files into it. In this way, Monet avoids introducing code to 'improve' or 'replace' the operating system facilities for memory/buffer management. Instead, it gives advice to the lower level OS-primitives on the intended behavior[3] and lets the MMU do the job in hardware.

Unlike other recent systems that use virtual memory [11, 21], Monet stores its tables in the same form on disk as in memory (no pointer swizzling), making the memory-mapping technique completely transparent to its main-memory algorithms.

Furthermore, Monet lets you specify a memory management strategy for each individual heap. Large heaps tend to be memory-mapped, while smaller heaps can be loaded in memory for speed. As for buffering strategies, BATs with a generally sequential access pattern can profit from DMA page prefetching, whereas this can be disabled for randomly accessed heaps.

- *extensible algebra*. As has been shown in the Gral system [10], many-sorted algebras have many advantages in database extensibility. Their open nature allows for easy addition of new atomic types, functions on (sets of) those types. Also, an SQL query calculus-to-algebra transformation provides a systematic framework where query optimization and parallelization of even user-extended primitives becomes manageable. Monet's Interface Language (MIL) interpreted language with a C-like syntax, where sets are manipulated using a *BAT-algebra*.

  The MIL has a sister language called MEL (Monet Extension Language), which allows you to specify extension modules. These modules can contain specifications of new atomic types, new instance- or set-primitives and new search accelerators. Implementations have to be supplied in C/C++ compliant object code.

- coarse grained *shared-memory parallelism*. Parallelism is incorporated using parallel blocks and parallel cursors (called "iterators") in the MIL. Unlike mainstream parallel database servers, like PRISMA [1] and Volcano [9], Monet 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 of the query plan. This approach benefits throughput at a slight expense of response time and memory resources.

  A version of Monet designed to exploit efficiently distributed shared-nothing architectures is described in [20]. A prototype runs on IBM/SP1.

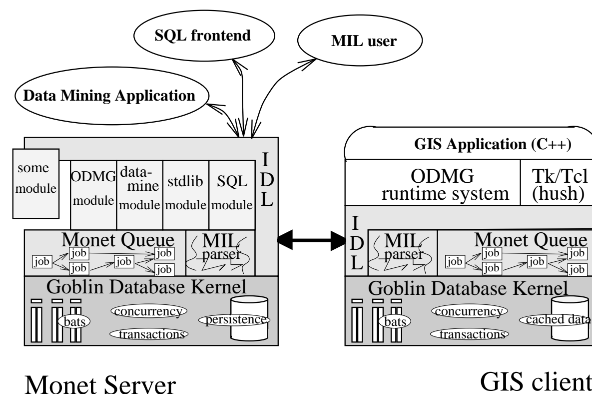

Figure 2: Monet Server and its Clients

Monet's design overview is shown in Figure 2. The low-level table-handling code supplying BATs, persistency and concurrency is called GDK[4]. The top layers consist of the Monet *request-queue*, from which multiple *interpreter threads* can take jobs for execution. Some of the actual MIL primitives are kernel-primitives, though most of them are placed in extension modules, that can be dynamically loaded, configuring Monet to ODMG, Data Mining or GIS functionality.

The algebraic MIL interface has been wrapped in an IDL specification, which also allows flexible interoperability using the CORBA mechanism, and encapsulates operations executed remotely or locally. Clients can either be normal applications doing function-shipping, or peer-to-peer Monet systems. These typically are applications, which come with simplified server-layer, allowing them to cache and manipulate Monet tables locally (for data-shipping situations).

2.1 Algebraic Interface

Monet has a textual interface that accepts a set-oriented programming language called MIL (Monet

[3]This functionality is achieved with the `mmap()`, `madvise()`,and `mlock()` Unix system calls.

[4]the Goblin Database Kernel: a predecessor system.

Interface Language). MIL provides basic set operations (BAT-algebra) and a collection of orthogonal control structures, including mechanisms to execute tasks in parallel. The MIL interface is especially apt as target language for high-level language interpreters (SQL or OQL), allowing for rule-based algebra translation [10], in which parallel task generation is easy. Algorithms that translate relational calculus queries to BAT algebras can be found in [13, 20]

We show in an example what the MIL looks like. Consider the following object relational SQL query on the relation `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 vertically 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. The SQL query gets translated to the following MIL block:

```
{
   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 with the end of the sequential block ({}) in which they were created. The three last joins 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. The dot notation "`a.oper(b)`" is equivalent to function call notation "`oper(a,b)`". 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)	$\{ab \mid ab \in AB \wedge b \geq Tl \wedge b \leq Th\}$
<AB>.select(T)	$\{ab \mid ab \in AB \wedge b = Tl\}$
<AB>.find(T)	$\{a \mid aT \in AB\}$

Note that `JOIN` projects out the join columns. The `MARK` operation introduces 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 [17]. First of all, new application domains typically require – complex – *user-defined data-types*, such as for instance `polygon` or `point`. Secondly, one often needs to define new *predicates and functions* on them (`intersect(p1, p2)` 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 *link additional server code* on top of it, the communication penalties of creating a separate server process, encapsulating the database (a "client-level" server), can be avoided.

3.1 Other Systems

Postgres [19] and GeoSabrina [7] are typical examples of an extended relational systems, allowing 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. In these systems, the programmer has more control, but to the point that data independence is compromised and the system gets hard to debug [8]. 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).

Monet extensions are packaged in modules, that can be specified in the Monet Extension Language (MEL). It requires you to specify ADT interfaces for new atomic types and accelerators, together with mappings to implementation functions in C compliant object code for all ADT operations and user-defined primitives.

Both module-specification and implementation object-code 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 ADT interface for atomic types assures that GDK's built-in accelerators will 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> , <byte-alignment> )
  FromStr  := <fcn>; # parse string to atom
  ToStr    := <fcn>; # convert an atom to string
  Compare  := <fcn>; # compare two atoms
  Hash     := <fcn>; # compute hash value
  Length   := <fcn>; # compute length of an atom
  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
  Heap     := <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

GDK provides passive support for user-defined search accelerators via an ADT interface that maintains user-defined accelerators under update and I/O operations. The support is "passive" since basic GDK operations only use the built-in accelerators for their own acceleration. An ADT interface always incurs some implementation overhead, and bearing in mind that accelerators in Monet have to retain their efficiency under main-memory conditions, the canonical access path trio `open()`, `findnext()` and `close()` [19] was left out[5]. 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
  Cluster := <fcn>; # cluster a BAT on accelerator order
END <name>;
```

As mentioned earlier, each accelerator resides in an individual heap, and hence can be be made persistent on disk, mapped into virtual memory and assigned a buffering strategy.

3.2.3 New Primitives

The MIL grammar has a fixed structure but depends on purely table-driven parsing. This allows for the run-time 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 MIL has been equipped with a polymorphism mechanism. A certain command, operation or iterator can have multiple definitions, with differing function signatures. Upon invocation, the Monet Interpreter decides which implementation has to be called, based on the types of the actual parameters.

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

The above shows the MEL syntax for specifying new primitives.

[5]extension code that 'knows" the accelerator, typically accesses it with a C-macro or C++ inline function.

4 GIS processing in Monet

Using the Monet database kernel to support the GIS applications foreseen poses the following challenges on its design and implementation:

- putting optimization and extensibility together.
- efficiently dealing with huge data, while keeping overhead on small data low.

On the first point, our approach is to use Monet as a flexible GIS backend. Though this paper does not seek to investigate geographical query optimization, the BAT-operators for specifying memory management physical clusterings, and caching strategies (see Section 4.2) show that Monet provides many opportunities for doing so – using algebraic transformation techniques. In such an algebraic translation, the vertical decomposition using DSM also saves I/O, and provide a means for inter-operation parallelism [20].

4.1 Managing GIS data in main memory

Regarding the huge datavolumes, Monet's main-memory oriented approach may at first sight seem unsound. The below table shows the sizes and cardinalities of the Sequoia benchmark (as specified in [18]), and occupied space in datastructures of Monet.

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.

Point			
specification	monet	cardinality	
2Mb	2.4Mb	60K	**Regional**
28Mb	13Mb	900K	**National**
300Mb		10M	**World**
Polygon			
specification	monet	cardinality	
20Mb	39Mb	60K	**Regional**
300Mb	407Mb	900K	**National**
3Gb		10M	**World**
Graph			
specification	monet	cardinality	
50Mb	-	300K	**Regional**
1Gb	-	6.5M	**National**
10Gb		65M	**World**
Raster			
specification	monet	cardinality	
1Gb	900Mb	500M	**Regional**
17Gb	15Gb	9G	**National**
2Tb		1T	**World**

SEQUOIA Benchmark Sizes

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].

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 1.2 Mb and 6 Mb, respectively, which can easily be handled in main memory. Even for the national benchmark these sizes are 18 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.

4.1.1 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[6] provide hooks for better memory and process management.

Stonebraker discarded the possibility of using memory-mapped files in databases [16], 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`[7]), 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 heaps into its main mem-

[6] These are the two platforms on which Monet is currently supported.

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

ory. 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 when needed.

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.

4.2 Memory Management in Monet

The memory-mapping implementation technique has a number of advantages:

- it provides *flexibility*. Orthogonally of what is stored in a Monet heap, one can decide to memory-map it, or not. Additionally, for each heap one can specify a different buffering strategy, which can be one of:
  - *prefetch*. After a page-fault, get one disk cluster (64K of pages), and start an async readahead through DMA for the next cluster.
  - *random*. Just get the pages faulted on – no prefetching.
  - *sequential*. Similar to prefetch, but the touched pages are immediately marked for swapout.
- the scheme is completely *transparent* in Monet[8]. This allows for a seamless transition from main-memory processing to disk-based processing.
- it is *efficient* in loading database tables, since only the pages needed are loaded. Under page-swapping conditions it holds out better than normal memory, because the file will swap on itself, and is not copied to swap space. Also, when saving a memory-mapped file, the MMU hardware guarantees that only dirty pages are written.

4.2.1 Clustering on Memory Pages

In effect, memory-mapped files bring Monet's approach back to the traditional, disk-based algorithms – but only where this is really necessary. This means, that in disk-dominated database configurations traditional methods such as clustering on a search accelerator order may be beneficial. Where in other systems this saves I/O, clustering will save page-faults in Monet. In Monet clustering is done by storing objects that are likely to be referenced at the time close to each other in memory – not necessarily, but probably, on the same disk-page. Remark that clustering can be performed orthogonally as an optional sorting operation (see Figure 3). It is transparent to Monet's main-memory algorithms, but will have an accelerating effect.

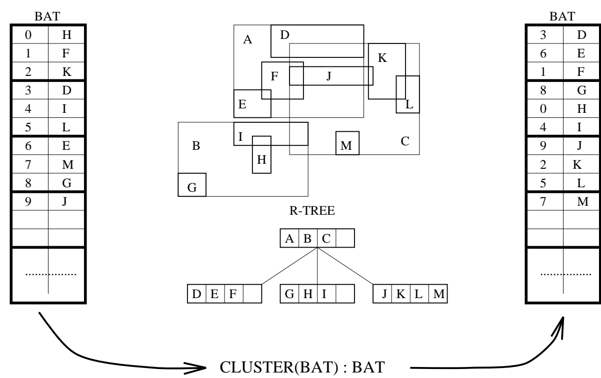

Figure 3: Clustering BATs in Memory Pages

All three different types of heaps in a BAT can be clustered independently:

- clustering *BUN-heaps*. The BUNs in a BAT can be clustered for optimal access via one of its associated accelerators. The result of the clustering depends on the accelerator-type. If, for instance, an R-Tree accelerator is used as access path, the resulting BUN-heap gets clustered spatially. In this case adjacent rectangles are likely to be on the same memory page. A hash-clustering ensures that items with the same hash value will be put together; this can significantly speed up equijoin operations.
- clustering *variable-sized atom heaps*. The variable-sized atoms of a BAT (which are stored in separate heap) can be clustered in the same order as the BUNs themselves. In some cases, however, it is better to cluster the BUN-heap and atom-heap differently, as we will see in the example of Section 4.2.2.
- clustering *search accelerator heaps*. Disk-page oriented systems use R^+ trees, putting effort in

[8]recall that Monet's datastructures take the same form on disk as in main memory

optimizing disk page occupancy, while minimizing tree-depth [15]. Monet is intended to be used in both disk-oriented as well as main-memory situations (in which simple algorithms tend to work best [14]). We solved this conflict by building a main-memory oriented R-Tree structure (with small nodes). For disk-dominated configurations we offer a clustering operation, that reorders a R-Tree, such that every memory page contains one subtree, which all together form the entire tree, where each dashed box contains about one disk-page of nodes (see Figure 4).

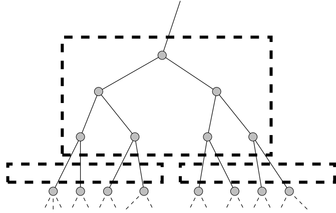

Figure 4: Clustering R-Tree nodes on memory pages

4.2.2 Performance Effects of Clustering

To give some idea of the impact of clustering on performance we have done some experiments with Query 6 of the Sequoia Benchmark (see Section 5). This query – which has a selectivity of about 1% – can be split into two parts: the first part selects all `[OID,box]` pairs within the given rectangle using an R-Tree. The boxes represent bounding boxes of polygons with the same OID. This query benefits from clustering of both the nodes in the R-Tree (in the way described above), and from clustering the `[OID,box]` BUNs on spatial proximity (R-Tree traversal order). The following table gives the results of this query with all possible clustering options:

time / #pageflt	nodecluster	no nodecluster
batcluster	0.234/30	0.775/89
no batcluster	0.388/39	0.843/97

The second part of Query 6 semijoins the selected OIDs with the `[OID,polygon]` BUNs, giving the requested set of polygons. Since Monet's semijoin is hash-based, it pays to cluster this BUN-heap, which contains only the OIDs – the polygons are in a separate variable-sized atom heap – on hash value. Because the OIDs were initially selected using the R-Tree, the polygon-heap can best be clustered on spatial proximity. This further reduced the number of page-faults on the semijoin from 336 to 274. So, we see here an example of a BAT, where the different heaps it consists of have differing optimal clustering orders.

4.2.3 Buffer Management Strategies

On memory-mapped files, Monet uses OS hooks to give advice on OS paging cache strategies. We distinguish three ways of accessing a BAT – all of which prosper by a different caching strategy.

- If no index exists on a BAT, a sequential scan is the only way of accessing a BAT, the OS can be told that to prefetch pages[9] using DMA, and mark them for release immediately after.
- If a BAT is accessed via an index on which the BAT is clustered, it is likely that pages near a page accessed will be accessed in the near future. By telling the OS to prefetch pages in DMA (but not to release them!) speedup will be obtained.
- If a BAT is accessed via a non-clustered index, the prefetching of adjacent pages is useless, because the memory pages are accessed in random order. By telling the OS to expect random access, time wasted by prefetching unnecessary pages is saved.

4.2.4 Performance Effects of different Cache Strategies

We ran Query 7 of the National Sequoia Benchmark (see Section 5) with different caching strategies, with the polygons clustered on R-Tree, or not:

Query 7	cluster	nocluster
random	2.29	2.35
prefetch	1.89	3.18

The above shows that in the nonclustered case, DMA *prefetching* is slower than *random* access (presumably because mostly unnecessary pages get fetched). In the clustered case, however, *prefetch* gives the best result, since spatially near polygons are stored in nearby memory pages.

Our results on clustering and buffer managament prove that current OS primitives are capable of supporting a database kernel built on main-memory datastructures and algorithms in a disk-bound setting (processing large datavolumes in virtual memory).

[9]this is cheaper than generating page-faults one-at-a-time

5 Monet Implementation of the Regional and National Sequoia 2000 Benchmarks

To demonstrate the feasibility and effectiveness of our approach, we decided to run both the Regional and National Sequoia 2000 Storage Benchmarks [18]. Results on the Regional benchmark have been published for both research and commercial GIS database servers [5]. Our implementation of the National benchmark sets a new mile-stone for further developments in this area.

The implementation of the Monet GIS datatypes makes extensive use of MEL modules. Several modules were implemented to provide for the necessary primitives and search accelerators. The Sequoia Regional Dataset was obtained, and a National Dataset was created from it. Then, the Sequoia queries were translated to Monet's MIL interface and run. The modules and the experimentations are described below.

5.1 Extension Modules

To support the Sequoia Dataset, we have implemented `Point`, `Box` and `Polygon` atomic types. The former two consist of fixed-size records of integers. The polygon type is a variable-sized type: BAT's containing polygons will store them in an associated heap. We provided ADT operations for these atoms, as well as predicates on them (such as `bool : Point_In_Box(p,b)`).

To optimize access on queries involving spatial proximity, we chose to implement R-Trees [15]. Since Monet accelerators must perform well in both main-memory as disk-based settings, we chose the most simple, lightweight approach, complemented by clustering operations to optimize on page-faults. Algebra-operations like `RTREEselect()` where added that use these R-Trees to optimize, in this case, an overlap-select. Similarly, Quad-Trees [15] were implemented, for fast access to points.

Rasters were implemented as unary tables: these are BATs which have the void type `any` in one of the columns. We interpret such an unary as a 2-D mesh of tiles. The tiles are adjusted – depending on the atom-size – to occupy one diskpage, and from left to right, row by row. This raster functionality was created by defining new commands, operating on unary BATs, that in effect perform raster-loads and -clippings.

All extensions were packaged in a set of MEL modules, from which an excerpt is shown below:

```
MODULE Box;
   ATOM Box(16,4);
      TOSTR   = BOXtoStr;
      FROMSTR = BOXfromStr;
      COMP    = BOXcomp;
      DEL     = BOXdel;
      HASH    = BOXhash;
      NULL    = BOXnull;
      PUT     = BOXput;
      GET     = BOXget;
      LENGTH  = BOXlength;
      HEAP    = BOXheap;
   END;
END Box;

MODULE RTree;
   USE Box;
   ACCELERATOR RTree ( Box );
      BUILD   = RTREEmake;
      DESTROY = RTREEdestroy;
      INSERT  = RTREEinsert;
      SAVE    = RTREEsave;
   END;

   OPERATOR (bat[any,ibox]) Join (bat[ibox,any])
       : bat(any,any)  = RTREEjoin;
   COMMAND RTreeSelect( bat[any,ibox], ibox )
       : bat(any,ibox) = RTREEselect;
   COMMAND RTreeCluster( bat[ibox,any] )
                       = RTREElogiccluster;
   COMMAND RTreeClusterNodes( bat[ibox,any] )
                       = RTREEfysiccluster;
END RTree;
```

5.2 Test Configuration

The Regional sequoia Dataset consists of point-, polygon-, directed graph- and raster-data about the State of California. We generated a National benchmark out of this data, by creating a grid of 5×3 States of California. We also expanded the queries with a factor 15, that is, our National Benchmark queries select on regions that are 15 times as big as the Regional selections, etc.[10]

Our benchmark platform was a Sparc 20, at 60 Mhz, 1MB secondary cache, with 128 MB of main memory. It has a Seagate ST15150W disk with 5Gb capacity, a throughput of 20Mbyte/s throughput and 0.85 msec access time. The client was a Sun IPX, connected to the server by 10Mbit/s ethernet. In this article, we compare our numbers with results published on Postgres, Illustra and Paradise [5]. The configuration described there has a CPU about twice slower, a disk with roughly twice less throughput, and 1.0 msec access time. For this reason, we hardware corrected their numbers for comparison purposes with a factor

[10] detailed information about the Sequoia Dataset scripts, the queries, and results can be found on `http://www.cwi.nl/projects/monet/sequoia/`.

0.6. This factor was taken by assuming that Sequoia performance is 50% I/O and 50% CPU bound, giving equal weight to differences in random access time and throughput. Of course, this is only a rough comparison: in random-access queries, we expect Monet to perform relatively better, whereas CPU bound results may be a bit inflated.[11]

5.3 Query Results and Analysis

The below tables display all Sequoia results:

Query 1: Database load	Monet National	Monet Regional	Paradise Regional	Illustra Regional	Postgres Regional
total	37190	712	2204	3506	5299
raster	34580	590			
point	656	14.5			
poly	1954	107			

Query	Monet National cold	Monet National warm	Monet National hot	Monet Regional cold	Monet Regional warm	Monet Regional hot	Paradise cold	Illustra cold	Postgres cold
Query 2: Select a raster for a given wavelength and rectangle, ordered by ascending time.	45.5			3.7	3.0	1.4	8.0	8.9	8.2
Query 3: Select a raster for a given time and rectangle, and calculate an average on the wavelengths for each cell.	14.2			0.87	0.75	0.38	1.20	2.88	3.44
Query 4: Select a raster for a given time, wavelength band and rectangle. Lower the resolution of the image by a factor 64 and store it in the DBMS.	2.4	1.6	1.2	0.24	0.18	0.09	0.36	1.44	0.78
Query 5: Find all points with a given name.	0.10	0.08	0.00	0.09	0.08	0.00	0.12	0.60	0.54
Query 6: Find all polygons intersecting a rectangle.	5.8	5.3	1.6	2.1	1.1	0.2	4.3	12.5	22.0
Query 7: Find all polygons larger than a certain size, and within a specific circle.	1.90	0.90	0.06	0.51	0.39	0.01	0.42	0.49	21.3
Query 8: Show the landuse/landcover in a 50km quadrangle surrounding a given point.	5.0	2.9	0.2	1.6	0.7	0.3	5.7	14.5	37.9
Query 9: Find the raster data for a given landuse type in a study rectangle for a given wavelength band and time.	14.7	0.7	0.4	1.4	0.2	0.0	1.7	0.7	1.7
Query 10: Find the names of all points of a specific vegetation type and create this a a new DBMS object.				4.7		1.4	-	0.4	196.2

[11] Interestingly, our larger memory (128MB vs 32MB) was no factor of influence, due to the simple selection-character of the Sequoia queries. The biggest memory user was Regional Query 6 with 3.0 MB of loaded memory plus 1.2 MB in memory mapped pages. Only database import takes more memory: point query 1 takes 26 MB.

The measured times, presented throughout this article are in seconds of elapsed time.[10] The table presents three times for both the National and Regional measurements:

- *cold:* the tables involved in the query have not been accessed since server startup.
- *warm:* the tables involved in the query have been accessed, but for a query that involved different data instances. This time captures typical database browsing access.
- *hot:* the same query was run as just before. This number characterizes execution time in a purely main-memory situation.

All Sequoia relations were vertically partitioned into separate `<relation>_<attribute>` BATs having OIDs in the head-columns, and a Sequoia attribute in the tail column.

The OID head-columns of all BATs had a hash-index built on them. The BUN-heaps of all BATs were clustered on these hash-tables, except for the `Points_location` BAT, which was clustered on the Quad-Tree indexing the points.

During polygon creation, we computed bounding box approximations for all polygons. On this `Polys_bboxes` BAT, an R-Tree was built, with which the polygon-heap of the `Polys_geometry` was clustered.

Monet's default buffering strategy on memory-mapped heaps is *random*. Since GIS data is large, we chose to memory-map all heaps of all BATs for both the Regional and the National Benchmark. The *prefetch* strategy was applied only to take advantage of clustering: on the tree-indices, and the polygon-heap (which was clustered on R-tree).

Query 2 requires the result to be exported back to the client. This was done by writing the resulting rasters to a disk on the client with NFS.

Most time in Query 1 (database load) is spent on parsing the ASCII files, making this query CPU bound. The high performance is achieved mainly by DSM, making the construction of index structures cheap, since the tuples involved are small. We should add, though, that the fast results are a bit offset by the fact that our topological datastructures were not yet finished, so we did not have to import graph data (and consequently we did not perform Query 11 either – but neither did the other systems).

The fast performance on raster queries show the benefit of a low overhead system. Raster data for

each time and wavelength band was stored in a separate – equally sized – raster, amounting to a total of 130 unary BATs. Since we did not have sufficient disk space to generate all 130 rasters for the National benchmark, the national raster queries accessed a subset of them multiply, calling a memory-flush utility between accessing them during query execution.[10]

As for the polygon- and point-queries, our results show that simple main-memory algorithms, enhanced by clustering and specific cache strategies on individual heaps, give good performance, even on very large datasets. More data than described here was obtained, but is omitted here because of space limitations.[10]

Note that we did not use Monet as a main-memory system here: no data was preloaded whatsoever. Our "cold" times therefore present the worst-case behaviour; when Monet's performance becomes disk-bound. This made a fair comparison with other systems possible. If we had used the potential of our 128 MB (e.g. by preloading all search accelerator heaps), thus blending main-memory with disk-bound processing, "cold" times would improve dramatically towards the "hot" numbers.

As a final point, we think that the Sequoia benchmark has limitations in both the simplicity of the queries – providing no need for complex optimization – and lack of serious thematic data, that would make relations even wider. A more complex benchmark would favor our algebraic approach, and wider relations would let Monet save even more I/O using its Decomposed Storage Model.

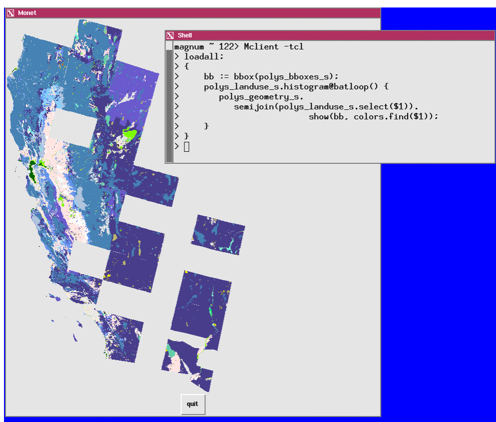

Figure 5: Provisional GEO-Monet Interface

6 Future Work

In the short term, our current provisional interface (Figure 5), just allows for drawing maps using algebraic commands – and zooming in/out with the mouse. It will be replaced soon by the GIS client depicted in Figure 2, that provides a better interface and caches results for supporting browsing sessions.

Our future actions involve further development of a full-fledged geometric and topological reasoning system on top of the current framework. Extension with CPU-intensive routines also requires further investments in parallel execution.

An OQL query optimizer and graphical user-interface are under development by other members of the MAGNUM project. This work takes place in the context of our ODMG compliant persistent programming system, that uses Monet and its – geographical – extensions as a backend.

7 Conclusions

We describe Monet, a MMDBMS that employs OS virtual-memory and buffer management primitives when large data volumes exceed main-memory. Its novel architecture presents a response to the trend of increasing main-memory sizes in custom hardware, that can be contrasted with the approach of just equipping a conventional DBMS with a large cache. To test the performance and extensibility of our system, we wrote GIS modules and ran the Sequoia benchmarks. Our implementation of the 15 GB National benchmark sets a new mile-stone for future developments in this area.

Monet's use of virtual memory primitives to avoid overhead typically induced by a DBMS tuple-manager. Its flexible storage structures (Decomposed Storage Model) and efficient (lightweight) algorithms, make it a highly efficient main-memory system, as shown by the "hot" Sequoia results. The "cold" experiments present the proof of concept for the employed OS techniques: Monet also obtains excellent results when huge datavolumes degrade performance to disk-bound processing.

By holding more and more data memory-resident, performance gradually shifts from Monet's "cold" towards the "hot" performance, enabling you to blend main-memory with disk-bound processing – a desirable property in times where many application domains are moving around the brink of the two situations.

Acknowledgements

We thank the members of the database research group of CWI and UvA for their continual support in making Monet a viable DBMS. The challenges posed by the MAGNUM project members have been a stimulus to prove our case.

References

[1] 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.

[2] P. A. Boncz and M. L. Kersten. Monet: An impressionist sketch of an advanced database system. In *Proc. IEEE BIWIT workshop, San Sebastian (Spain).*, July 1995.

[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] David J. DeWitt, Navin Kabra, Jun Luo, Jignesh M. Patel, and Jie-Bing Yu. Client-server Paradise. In *Proceedings of the 20th VLDB Conference, Santiago, Chile.*, pages 558–569, September 1994.

[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. G.Gardarin and M.Jean-Noël. Sabrina, a relational database system developed in a research environment. In *Technology and Sciences of Informatics.* AFCET-Gauthier Villard - John Willey and Sons Ltd., 1987.

[8] et al. Neuhold,E. and Stonebraker,M. Future directions in DBMS research. *ACM SIGMOD RECORD*, 18(1), March 1989.

[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 geometric applications". In *Proceedings of the 15th VLDB Conference, 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, M. L. Kersten, and A. Siebes. Data Surveyor: searching for nuggets in parallel. In *Knowledge Discovery in Databases.* MIT Press, Cambridge, MA, USA, 1995.

[13] 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.

[14] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *Proceedings of the 12th VLDB Conference, Kyoto*, August 1986.

[15] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison Wesley, 1990.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] Seth J. White and David J. DeWitt. Quickstore: A high performance mapped object store. In *ACM SIGMOD Conf. on the Management of Data*, pages 395–406, May 1994.