Monet; a next-Generation DBMS Kernel For Query-Intensive Applications

Boncz, P.A.

Citation for published version (APA):
Chapter 3

Monet

The complex interdependencies that come into play when designing DBMS software make it intractable to use a purely theoretical methodology. DBMS architecture therefore is an experimental science, or maybe even some kind of software art. Our research group has studied database architecture for the past 10 years. In our work, we take the empirical approach of trying out architectural ideas by creating a prototype DBMS, with which real-life experiments can be performed in order to gain true insight into the (lack of) merits of our ideas. In the late 1980s and early 1990s, our group participated in research into massively parallel database systems, which created the PRISMA system [AvdBF+92] and continued with the Goblin system [KPvdB92, vdB94] in the area of dynamic query optimization. The work in the course of this PhD research from 1994 on has also resulted in a prototype system, called Monet [BK95, BKK95, BKK96, BWK98, BRK98, KK98, BK99, BMK99, BMK00, MBK00, KMBN01, MBK02].

In our short review of DBMS architecture in the previous chapter, we stated that query-intensive applications like data mining or ad-hoc decision support cannot be supported efficiently on large datasets with currently available software solutions. MOLAP and ROLAP are not apt as both rely on pre-knowledge and pre-computation techniques which are of no use in ad-hoc query loads. Standard relational DBMS techniques are generic, but cannot provide sufficient performance. Our Monet system specifically sets out to provide high performance on large data sets and ad-hoc queries. We describe the Monet design goals in Section 3.1.

In Section 3.2, we analyze in detail why performance of existing relational DBMS products suffers on query-intensive applications. It turns out that causes can be found both in the access pattern of query-intensive applications for which relational DBMS products were not initially designed, as well as continuing radical change in commodity hardware components, which has changed trade-offs in DBMS architecture.

Finally, in Section 3.3, we discuss the main design ideas that were applied in the architecture of Monet, and explain how they circumvent the performance problems faced by relational DBMS products on query-intensive applications.

3.1 Monet Goals

In the design and implementation of Monet, we try to create a DBMS architecture that answers the following questions:
• how to get best performance out of modern CPU and memory hardware on query-intensive DBMS applications?

Past DBMS research first and foremost has focused on minimizing the amount of random I/O and put CPU or memory access optimization on second place (or disregarded these aspects entirely). In the design of Monet, we take up the challenge of coming up with a DBMS architecture that is optimally suited for fully utilizing the power of modern CPUs and optimizing main-memory traffic. We target query-intensive DBMS applications and specifically do not want to compromise query-intensive performance with system features that are only relevant for supporting OLTP.

As high performance is our primary goal, Monet is designed to exploit parallel execution. We aim at facilities in the system to easily exploit shared memory multiprocessors with a few CPUs as found in commodity hardware including communication and control mechanisms for exploiting parallel resources of shared-nothing computers on a network, as well as ccNUMA architectures with shared distributed memory and many processors.

• how to support multiple (complex) data models?

The relational model and its standard SQL query language is currently most popular in DBMS software. Other data models like the object-oriented and object-relational model, however, are increasingly popular. We would like our system to be usable for these and emerging models. Also, own experience in the area of data mining [BRK98, KSHK97] has shown that there are applications that prefer DBMS query languages other than SQL or one of its object-relational or object-oriented variants. Therefore, we designed Monet in such a way that it provides all DBMS services needed, but in a manner that is neutral to what the end-user sees as the data model and query language.

• how to provide sufficient extensibility to new domains?

Though we have primarily mentioned OLAP and data mining, query-intensive applications are or not limited only to the business domain. On the contrary, the driving applications of many new domains are query-intensive. Examples are (text) information retrieval [dVW98], XML database management [SKWW00, SWK+01, SKW01], similarity matching in image or video databases [NK97, NQK98, NK98, KNW98, dVB98, WSK99, dV99, Nes00, dV00, dWAK00, BdVK01, dVMK02], and geographic querying both using topological and geometrical models [BK95, BQK96, WZT+98]. Monet was designed with such new domains in mind, and is currently being used by collaborating researchers in all these new application domains.

3.2 Relational Performance Problems

Most currently popular DBMS products were designed in the early 1980s, almost two decades ago at the time of this writing. As stability is one ground concern for these commercial products, their innermost system functionality is rarely revised, hence the original design decisions made there have lasting impact, and may start to conflict with fulfilling the system requirements when the original design assumptions are invalidated by changing circumstances.
In the following, we outline two such changes in circumstances that affect the efficiency of DBMS technology today, and which motivated the research in this thesis. In the first place, we discuss how the access pattern of OLTP has influenced the design of physical database storage structures and how these design choices collide with the needs of new query-intensive applications. Secondly, we outline how changes in commodity hardware components, on which DBMS software runs, change the efficiency trade-offs and as such interact with DBMS architecture.

### 3.2.1 Problem 1: Column-Access to Row-Storage

The implementation of current relational DBMS products stores database tables clustered by row, as this favors the access pattern of OLTP queries, which were dominant in the DBMS applications at the time of their design. They employ storage schemes like the Flattened Storage Model (FSM) and the Normalized Storage Model (NSM), that both store the data of a each relational tuple as a consecutive byte sequence [VKC86]. The complete relational table is stored in one database file that holds the concatenation of all these byte-sequences representing rows, with some free space left in between to accommodate updates. Such a storage scheme optimizes I/O traffic on OLTP queries: all data of one row is consecutive and therefore can be found in the same disk block, hence a minimum of I/O activity (i.e. one block read/write) is necessary for executing a typical OLTP query.

Query-intensive applications, however, typically need to access many rows, of which only a few column values are used. As a substantial portion of all rows is needed, a relational DBMS is forced to scan the complete database file for executing such queries. In data warehouses, tables tend to have many columns and a vast amount of rows, which results in a database file that can total gigabytes or even terabytes in size. Scanning such a huge file requires a lot of I/O, and can take significant time. However, because only a small number of columns is actually needed in OLAP, the great majority of the table scan is wasted effort, because almost all data read in, gets projected out right away as it contains data of non-needed columns. This simple yet devastating phenomenon is the main reason why relational DBMS products tend to be slow on OLAP an data mining queries.

Notice that no DBMS index structure, like for example the often suggested bit-map index [O'N87], can make the DBMS avoid doing these wasteful table scans, because OLAP and data mining queries often have a low selectivity (e.g., “all sales after march 21” is not very selective; it probably involves more than half of all sales records). This a-priori decreases the benefits of any index structure: an index structure may yield a list of selected record identifiers quickly, but the select phase is usually not the end of query execution. In general, an OLAP query has to perform some action on the selected tuples – in our example, it must sum claim-amounts while grouping by product and city – hence the query needs additional column values for the selected tuples (in our example: Claim, Product and City). These columns are chosen ad-hoc and therefore cannot be supposed to be stored in the index on the date-of-claim column. Therefore, they must be fetched from the main database file in a project phase.

The problem for indices is that in this project phase, a full table scan often tends to be faster than fetching all blocks identified by the index individually, unless the number of blocks needed is indeed very much smaller than the total number of disk blocks in the database file. The selectivity percentage for which this is the case is usually fairly
small – like 3% or less; which is not the case in most low-selectivity queries found in OLAP and data mining applications. This threshold percentage mainly depends on the cost ratio of random vs. sequential I/O, and can roughly be calculated as follows: in the 8 milliseconds it takes one random I/O to execute, a disk with a sequential access bandwidth of 20MB/second can read 40 disk blocks of 4KB, hence the break-even point for a full scan vs. random reads in this configuration is when 1/40th of all blocks must be read (which corresponds to selecting than 2.5% of the tuples). In the following, we will see that due to trends in hardware this break-even point shifts even lower every year, making (non-clustered) indices less and less beneficial in query-intensive DBMS applications.

<table>
<thead>
<tr>
<th>year</th>
<th>computer model</th>
<th>processor</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>type</td>
<td>MHz</td>
<td>STREAM/opy (bandwidth)</td>
</tr>
<tr>
<td>1989</td>
<td>Sun 3/40</td>
<td>6920</td>
<td>20</td>
</tr>
<tr>
<td>1990</td>
<td>Sun 3/80</td>
<td>6930</td>
<td>20</td>
</tr>
<tr>
<td>1991</td>
<td>Sun 4/280</td>
<td>Sparc</td>
<td>17</td>
</tr>
<tr>
<td>1992</td>
<td>Sun ss10/31</td>
<td>superSparc I</td>
<td>33</td>
</tr>
<tr>
<td>1993</td>
<td>Sun ss10/41</td>
<td>superSparc II</td>
<td>40</td>
</tr>
<tr>
<td>1994</td>
<td>Sun ss207/1</td>
<td>superSparc II</td>
<td>75</td>
</tr>
<tr>
<td>1995</td>
<td>Sun Ultra170</td>
<td>ultraSparc I</td>
<td>167</td>
</tr>
<tr>
<td>1996</td>
<td>Sun Ultra2200</td>
<td>UltraSparc II</td>
<td>200</td>
</tr>
<tr>
<td>1997</td>
<td>SGI PowerCh.</td>
<td>R10000</td>
<td>195</td>
</tr>
<tr>
<td>1998</td>
<td>SGI Origin 2000</td>
<td>R12000</td>
<td>300</td>
</tr>
<tr>
<td>1999</td>
<td>Intel PC</td>
<td>00486</td>
<td>66</td>
</tr>
<tr>
<td>2000</td>
<td>Intel PC</td>
<td>Pentium</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pentium</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pentium</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pentium</td>
<td>133</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PentiumPro</td>
<td>200</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PentiumII</td>
<td>300</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PentiumIII</td>
<td>350</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PentiumIII</td>
<td>450</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PentiumIII</td>
<td>600</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PentiumIII</td>
<td>750</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Athon</td>
<td>500</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Athon</td>
<td>800</td>
</tr>
</tbody>
</table>

Figure 3.1: Hardware Trends in the Last Two Decades.

### 3.2.2 Problem 2: Commodity Hardware Has Changed

In the past decades, commodity computer hardware has grown tremendously in power. Figure 3.1 summarizes through the years characteristics of those hardware components most relevant for DBMS performance, like CPU, memory and magnetic disks.
The upper-left table lists characteristics of popular Sun and SGI workstations, and mid-range PC hardware (i.e. PCs worth around $2500 at introduction), ordered by their year of introduction. The table lists for the CPU of each computer its clock frequency in MHz and the maximal number of instructions that the CPU can execute in parallel. Both characteristics together determine the power of a CPU. For the computer memory it shows latency, as well as bandwidth and typical memory size in the computer configuration. We use the results of the COPY test from the STREAM Benchmark [McC95] for characterizing real-life memory bandwidth.

The lower-left table shows hard disk products of medium price (around $700) at their year of introduction, with their official manufacturer specifications concerning capacity, random-access latency and sequential read bandwidth (which we take from their specified internal sustained bandwidth).

The right-hand graph contains a plot of the data from the left-hand tables in exponential scale, revealing remarkably consistent trends in the development of each kind of component. This mini-survey shows that size and bandwidth of both main memory and hard disk space, as well as CPU power have been growing steadily with about 50% each year (a.k.a. Moore’s law). The dissonants to this trend are disk latency, which improved just about 10% a year, and main-memory latency, which stayed more or less the same. Consequently, for each and every piece of software – including DBMS software – these latency components are becoming ever more expensive relative to the other components that make up overall performance. We analyze the effects of these hardware trends on DBMS performance in more detail below.

**Disk Latency**

The relative rise in cost of disk latency mostly affects OLTP queries as they have a block-at-a-time access pattern and thus are limited mostly by cost of random I/O. OLTP systems have countered this performance problem in three ways. First, RAID [PGK88] systems have become commonplace. A response time of a couple of milliseconds, as imposed by disk latency, is acceptable for each individual OLTP query. The challenge in OLTP is sustaining an ever growing rate of queries per second. RAID devices partition the database file over many disks, so many OLTP queries can be executed in parallel on different disks, which provides scalability in throughput despite lagging disk latency.

A second trend of the recent years in disk hardware has been the introduction of memory caches inside the hard drive (see the right-most column of the disk table in Figure 3.1). Hot areas on the disk, like the end of the database file, where new records are inserted, will reside in this cache and requests on cached blocks can be executed at very high speed. Such hard disk cache memory is protected by error-checking hardware and a battery, making it fully failsafe and persistent. Hard drives with memory caches only bring performance improvement, though, if the I/O access pattern exhibits locality of reference.

Fueled by the growth of main-memory sizes, a third way to enhance OLTP performance on applications for which the problem size is in order of a few gigabytes has been to switch to main-memory database technology. Performance of a main-memory database system (MMDBMS) is not constrained by disk latency at all, and

---

1 Memory latency is the time it costs to read a byte in main memory that is not in one of the memory caches. We determined this latency using the Calibrator Tool described in Chapter 4.
can therefore achieve throughput far superior to a normal DBMS. Several new commercial products have surfaced that target these new main-memory OLTP applications [Tea99, BBC+99, LPR99].

The above issues, however, are not so important for query-intensive applications, as OLAP and data mining usually generate (large) sequential I/O reads and their performance thus depends more on disk bandwidth than on disk latency.

**Bandwidth**

Though we noted in Section 3.2.1 that query-intensive applications cause a huge I/O load, it is *memory access* that becomes the bottleneck for OLAP and data mining if hardware is scaled to its limits. The reason for this is that disk bandwidth has increased with Moore’s law through the years, and can additionally be scaled using parallel I/O devices, like RAID. This additional scaling stops, however, when so many disks have been attached to the computer that their accumulated disk bandwidth exceeds the bus or memory bandwidth. Figure 3.1 shows that memory bandwidth has also been increasing with Moore’s law, but unlike disk bandwidth, it cannot be scaled arbitrarily in a single computer installation, and thus becomes a bottleneck in installations with RAID devices [NBC+94, RvIG97].

**Memory Latency**

The most urgent reason why memory access is becoming a bottleneck for query-intensive DBMS applications like OLAP and data mining, however, is the lack of progress in memory latency. In 1990, one CPU cycle (supposing a 25MHz CPU) lasted 40ns, so accessing memory that has a latency of 160ns cost 4 CPU cycles. In 2000, one CPU cycle of a 1GHz CPU lasts 1ns, so accessing 160ns memory that is not in any cache, will make the CPU stall (i.e. do nothing) for no less than 160 CPU cycles. Main memory of a computer is made of cheap DRAM chips. DRAM technology development focuses on creating chips with ever higher density (=capacity). DRAM latency has not improved in the past decades and is not expected to improve either. The only way hardware manufacturers have found to hide the ever increasing slowness of main-memory access relative to the other hardware components, has been to equip computers with ever more *cache memory*, that is made of fast but much more expensive SRAM chips. Cache memory was first separately placed on the computer board, but is now also integrated in the CPU chip in order to minimize the physical distance between cache and CPU. A typical computer now has a small L1 memory cache (e.g., 64KB) on the CPU and a larger L2 cache (e.g., 2MB) on the computer board. Any application that has a memory access pattern that achieves a low hit rate in these memory caches and thus depends on memory latency is, or will in the near future, be facing a huge *memory access bottleneck*. Low cache hit-rates occur when the application repeatedly makes random accesses to more memory than fits the cache. DBMS algorithms that have this property are hash-join and sorting, which are common in OLAP and data mining queries.

**CPU Utilization**

Studies into CPU utilization under various DBMS workloads have shown that modern CPUs tend to be stalled for the great majority of their time, achieving a low utilization
of their true power [BGB98, KPH+98, TLPZT97, ADHW99]. A significant portion of this stall time is caused by memory latency, for the reasons described above. It turns out, however, that this high stall rate has the additional cause that the machine instructions of typical DBMS algorithms tend to be highly interdependent, causing so-called dependency stalls. Modern CPUs are not only more powerful than their predecessors due to a higher clock speed, but increasingly also due to more potential for parallel execution within the CPU. The idea is that if instructions are independent, they can be executed at the same time by replicated execution units. The upper-left table in Figure 3.1 shows that this number of replicated execution units in CPUs is steadily rising: where the 680X0 and 80486 CPUs just could one instruction at a time, a modern CPU like the Athlon can in theory execute 9 instructions in parallel in one clock cycle. This peak performance can only be reached if the performance-intensive parts of the program always contain at least 9 independent instructions. This is a tough requirement and makes it hard to achieve this peak CPU performance, posing challenges both to compiler technology as well as application programming. Only specifically optimized scientific computation programs seem currently to take full advantage from these advances in processor parallelism. DBMS software, in contrast, is doing specifically bad in this area.

### 3.3 Monet Architecture

The architecture of a DBMS involves many interacting design decisions on the macro and micro scale, encompassing both data structures for DBMS storage and algorithms for query execution, as well as putting to practice many programming skills. DBMS architecture therefore is not only about applying new techniques, but also about finding the right mix of already existing ones.

This is certainly the case in the architecture of Monet: some of the ideas are new, many other ideas had already been described. The combination of these ideas into one unique yet coherent design, with a well-stated purpose (high-performance support for query-intensive applications) is what brings the scientific “added value”.

In the following, we discuss the mix of ideas applied in Monet with which we tried to achieve our research goals.

#### 3.3.1 idea 1: provide DBMS back-end functionality

Figure 3.2 shows how the modules that make up a full-fledged DBMS can be separated in a front-end and and a back-end. The Monet system is intended to provide back-end functionality here.

**Serve Multiple Front-Ends**

The primary advantage of this separation is that multiple front-ends can operate on one identical back-end. Therefore, this front-end/back-end design facilitates our goal of supporting multiple diverse data models. We can have, for example, a relational front-end accepting requests in the SQL query language, as well as an object-oriented ODMG front-end accepting OQL queries, as well as front-ends like an OLAP or data mining tool. This is also the current architecture of the Data Surveyor product line of Data Distilleries B.V. that uses Monet as back-end [HKMT95].
The intermediate language in which front-end and back-end communicate is one of the prime design challenges of the Monet system. The requirement for this intermediate query language, called Monet Interpreter Language (MIL), is to provide the minimal set of primitives to support all Monet front-ends efficiently. Chapter 4 describes MIL in detail.

Figure 3.2: Front-end/Back-end architecture.

**Explicit Transaction Management**

Figure 3.2 depicts a relational DBMS front-end that consists of an SQL parser, SQL (view) rewriter, and query optimizer. The Monet back-end contains services for buffer management, recovery management and query execution. Lock management and recovery management are shaded gray in Figure 3.2, as we have actually pushed them out of the Monet kernel into some of its extensibility modules, which can be used optionally. When designing MIL, we used our freedom in defining new language primitives to explicitly separate transaction management facilities in the language from query processing facilities. MIL contains explicit transaction commands, which provide the building blocks for ACID transaction systems, instead of implicitly implementing one specific transaction management protocol hard-coded in the query execution engine. Monet front-ends use these explicit transaction primitives to implement a transaction protocol of their own choice. That gives them the freedom to decide upon issues like the granularity of locking needed, and more importantly, it also de-couples transaction management from the query-processing primitives. Query processing primitives in MIL assume that all locks for executing the query are already held and just execute without any transaction overhead. This design has the advantage that it enables DBMS application areas that do not need stringent transaction management to avoid paying a performance price during query processing for transaction overhead they do not need. In particular, OLAP and data mining generally can do with a simple transaction protocol that uses a very coarse level of locking (a read/write lock on the database or table level), and their query execution performance can greatly profit if transaction management effort is eliminated or reduced.
3.3. MONET ARCHITECTURE

Run-Time Query Optimization

As we target query-intensive applications rather than OLTP, the emphasis in Monet functionality is put on query execution. Query execution in Monet also includes some tasks normally done in the query optimizer, as we propose to divide query optimization into a strategic and a tactical phase (this is further elaborated in Chapter 4). The strategic optimization is performed in the front-end and determines a good order of the operations in a query plan. The tactical phase is done at run-time in the Monet query executor and entails finding the optimal algorithm and buffer management setting (e.g., choosing between merge-join, hash-join, or partitioned hash-join). This design can take table characteristics into account that are only known at run-time, as well as the system state at that moment, which improves the quality of the optimization decisions. Also, hiding details (like a plethora of different join variants and parameter settings) limits the search space during strategic query optimization, reducing the time needed to find a good query plan.

3.3.2 idea 2: you can do everything with just binary tables

Monet uses the binary table model, where all tables consist of exactly two columns. These tables we call Binary Association Tables (BATs). This data model was introduced in literature as the Decomposed Storage Model (DSM) [CK85, KCJ+87]. The binary table model is a physical model in Monet, as the BATs are the sole bulk data structure it implements. DSM can be used to store a variety of logical data models, like the relational data model, the object-oriented data model, or even networked data models. Each front-end uses mapping rules to map a logical data model as seen by the end-user onto binary tables in Monet. In the case of the relational model, for example, relational tables are vertically fragmented, by storing each column from a relational table in a separate BAT. The right column of these BATs holds the column value, and the left column holds a row- or object-identifier (see Figure 3.3).

While the concept of mapping a user language (like relational calculus) to an internal language (like relational algebra) is standard practice in database systems [Dat85, UW97], a similar remapping of the data model from a user data model onto an internal system data model, as done by the Monet front-ends, is not.

![Figure 3.3: Decomposition of the Relational Model onto BATs.](image-url)
CHAPTER 3. MONET

We now go deeper into the advantages we see for the binary table model in Monet.

Access Pattern

We showed in Section 1.1.2 that query-intensive applications have a column-wise access pattern, and argued in Section 3.2.1 that such access patterns create wasteful table scans when table storage is clustered by row. Monet decomposes relational tables by column, storing each column in a separate BAT, which comes down to full vertical fragmentation. This has the advantage that queries that access many rows but only few columns just need to scan the BATs of the columns they need. The BATs belonging to non-used columns remain untouched. Full vertical fragmentation therefore greatly reduces the I/O and memory bandwidth generated by OLAP and data mining queries, resulting both in much less I/O and memory access costs.

The derived idea of partial vertical fragmentation has also been proposed [GGMS97], but while this may be a viable technique to decide which data resides where in a distributed database, we reject this idea for OLAP and data mining. Partial vertical fragmentation supposes pre-knowledge of which columns are frequently accessed together and employs attribute usage matrices to determine optimal clusterings of columns into vertical fragments. However, OLAP and certainly data mining are application areas that imply ad-hoc querying. A good OLAP and data mining system must be able to swiftly answer queries that involve any combination of attributes. The TPC-H benchmark (previously TPC-D) reflects this view by banning vertical fragmentation, as this would allow benchmark implementations to place all attributes that stay unused in the TPC-H query set into one dummy fragment, effectively throwing them out of the benchmark. We would, however, like to make a case here for allowing full vertical fragmentation in TPC-H, as full vertical fragmentation does not favor particular attribute combinations and actually pays a performance price for its savings in I/O bandwidth, namely: during query execution extra joins must be done to reconstruct tuples from values stored in different vertical fragments (i.e., BATs).

A few commercial systems are around that already use vertical fragmentation physically, while offering a standard (non-fragmented) relational data model on the logical level. One way to implement this is to view vertical fragments as an additional index structure. This approach is called a projection index [OQ97]. Systems that use projection indices are Compaq's NonstopSQL/MX [CDH+99] and Sybase IQ [Syb96]. Another way is to perform the vertical fragmentation at the deepest possible level: storage on file. This results in transposed files [Bat79], a technique known from the 1970s. Systems that use transposed files are ADABAS [Coh78, Bat85] and Teradata [BDS89]. In the case of Monet, we consider vertical fragmentation so crucial that we make it explicit in the logical data model (BATs are vertical fragments). By doing so, we can concentrate in our algorithms on working well with fragmented data and make sure that not only base relations are vertically fragmented, but also intermediate results. Vertical fragmentation not only greatly reduces I/O access cost, but also reduces main memory access cost on sequential access (by making optimal use of the cache lines), hence it also helps in accelerating handling of intermediate data. This explicit approach is also offered in Tantau's Infocharger [Tan99], which is based on an early version of Monet.
3.3. MONET ARCHITECTURE

Key-Joins For Free

It is important in Monet to execute key-joins with high efficiency, because foreign-key joins occur frequently in OLAP and data mining, and because vertical fragmentation causes extra key-joins needed for tuple reconstruction.

Monet employs two basic techniques that make key-joins free in terms of lookup effort. First, BATs that contain columns of the same relational table typically contain these tuples in the same order, and this knowledge is exploited in Monet. A join of two BATs that contain the same OID sequence is resolved by iterating in parallel a cursor through both BATs; which is a linear process requiring no lookup effort. Second, when tables are decomposed into BATs, new OIDs are generated for the left-hand columns. These OIDs are typically dense and ascending integers (e.g., 1000, 1001, 1002, ..). In such cases Monet does not materialize the OIDs in the BAT, but makes the column of type VOID (for virtual-OID), and just retains the base OID (in this case, 1000). This avoids repeating the same column of OIDs in every BAT, which would almost double the storage requirements. It also makes foreign-key joins highly efficient, because lookup in a VOID column can be done just by position (i.e. if we look for OID 1002 in a VOID column that starts at 1000, we know by subtraction that the tuple can be found at position $2 = 1002-1000$).

Simplicity And Elegance

Having a fixed table format both eases many design aspects of the query language and facilitates (optimization of) its implementation. Still, the binary table model is a relational table model, so Monet can build on previously obtained insights in relational query processing, like in the area of relational algebras and calculi.

As an example of elegance, we return to the subject of foreign-key relationships. A relational column that is a foreign key to some other table gets stored as a BAT that contains [OID,OID] pairs. Such a structure happens to be known in database literature as a join-index (in case of 1-N joins, the 1-side can even be of type VOID). This also enables Monet to store a "reverse" relationship only once, in one BAT that is a join index. Reverse relationships are a feature of object-oriented data models that allow to give a foreign-key relationship an attribute name in both join directions (e.g., Set<Item> Order.items would be a reverse of Item.order). In other storage schemas, reverse relationships are tricky because both sides must be kept consistent under updates. Monet automatically solves this problem due to the very nature of BAT storage.

Mapping Flexibility

Each front-end can make its own mapping rules, both for mapping its logical model onto BATs, as well as for mapping its logical query language onto operations on BATs (a.k.a. query optimization). This idea makes it possible to store a diversity of logical data-models in Monet, and plays an important role in achieving both our goals of multi-model support and extensibility to new domains.

As an example, the MOA project at University of Twente [BWK98] has implemented an extensible high level front-end on top of Monet. MOA implements among others an ODMG-like object model on top of Monet that allows for arbitrary nesting. Collections
of such complex objects all get mapped onto simple BATs; no complex data structures are needed for its implementation.

An important contribution of MOA is that it takes DBMS extensibility a step further, by introducing the concept of structural extensibility in addition to the traditional base type extensibility. Base type extensibility is well-known in relational DBMS systems [SAH87, HFLP89, SLR96] and allows for the addition of new data types (e.g., point, polygon) and operations on them (intersect, overlap, etc.), and possibly even addition of new index structures for accelerating these new operations (e.g., an R-tree). For base type extensibility, MOA directly relies on the base type extensibility in Monet, that provides all these kinds of extension possibilities. In structural extensibility, however, it are the type structuring primitives themselves that are being extended. In object-oriented database systems, canonical structuring primitives are Tuple, Bag and Set. MOA maps instances of such type structures (i.e., tables, classes) onto BATs in Monet using a set of mapping rules. MOA, however, also allows to add new structuring primitives, simply by defining new mapping rules both for data and operations onto BATs and MIL.

In the field of information retrieval, structural extensibility has successfully been used to introduce network structures as a new structuring primitive called INFNET. The MIRROR [dV99] DBMS, implemented using MOA, uses bayesian inference networks to store and query document collections. These document collections have, apart from a number of traditional attributes, an INFNET attribute that represents a belief network over all words in the collection. Such a belief network can be stored in three BATs that are join indices. Queries over the network mainly involve foreign-key joins, which are cheap in Monet.

In a similar way, the MAGNUM project [WvZF98] uses MOA and Monet to integrate topological GIS structures called triangulated interconnection networks (TINs) together with geometrical primitives like polygons, in one query processing system. The TIN structure contains a triangulated map as a graph onto BATs, storing the vertices in of the graph in join-index BATs. This mapping makes it possible to translate e.g., spatial adjacency queries, again, into cheap foreign-key joins.

The advantage of structural extensibility over a data cartridge or data-blade solution [Ora97a, OHUS96] (which would implement an INFNET or a TIN as a complex blob-like object), is that in the case of MOA, the DBMS can “look inside” the objects. Fully integrating complex domain-specific structures (like network structures) in the heart of the database kernel, makes it possible to perform cross-domain query optimization and parallelization. This can also be contrasted with the currently popular “wrapper” approach to multi-media database systems [RS97], where such integration is not possible, and which forces mixed-media queries to be split in a standard media query part sent to a standard relational DBMS and an image query part that is sent to some wrapped special-purpose image retrieval system. Consequently, the wrapped retrieval system must always evaluate the image query predicate on its entire image collection, whereas an integrated query processing system like MOA is able to restrict the search a priori with the most selective condition first (as determined by query optimization), which in our view is one the very accomplishments of DBMS technology [dVW98].
3.3.3 idea 3: do not re-invent the OS

In 1986, Stonebraker defined operating system (OS) services desired by database systems and identified a number of areas where OS support was lacking [Sto81, Sto84]. As a result of the lacking OS functionality two decades ago, most DBMS products have opted to re-implement some OS services in user space. Examples are: having a separate buffer pool that caches data on top of virtual memory where the OS caches data from disk (found in all DBMSs), implementing raw disk I/O that bypasses the OS file system (in most), or having a built-in thread-package to meet DBMS scheduling constraints (in some).

In each of these problem areas, we discuss how OS functionality has evolved:

Addressing Space

In the early 1980s, most operating systems still used a 16-bit addressing scheme, which imposed a maximum of 64KB of addressable memory. This affected database architecture: the implementation of the INGRES system was split into six separate processes just to make its program code fit. Later, all operating systems switched to 32-bit addressing, which normally implies a usable memory range of 2GB or 4GB. Currently, main-memory sizes are approaching this limit, which again creates a shortage of address space. Currently, all RISC hardware manufacturers, and the UNIX variants that use them already offer 64-bit operating systems that raise the usable memory range to 256TB (i.e., internally, they actually use 48 bits addresses). That should be sufficient for now, and 64-bit addressing provides room for growth in the future (at least 60 years, if Moore’s law would continue). In the year of publication of thesis, we also expect the 64-bit versions of PC hardware and the Windows NT and Linux OS-es that run on them.

Virtual Memory Management

The mmap UNIX system call allows to access data on disk using a memory interface, and works as follows: the OS gives back a memory range that represents a file stored on disk. Accesses by the application to the returned memory range will cause page faults, that are caught by the OS, which loads the demanded blocks from the file into memory, transparent to the application. Stonebraker discarded the possibility of using OS support for virtual memory as an implementation of DBMS buffer management, however. The problem with making the OS responsible for buffering of DBMS files, is that the virtual memory management algorithm of the OS does not have knowledge of the access pattern of each application. Therefore the OS typically resorts to a simple Least-Recently-Used (LRU) scheme. This scheme has been shown to perform badly on DBMS loads [CD85].

Some prototype operating systems, like Chorus [RAA88] and Mach [ABG86], have been designed in the last decades to allow applications to influence OS services like virtual memory management [CRRS93]. More importantly, most commercial UNIX flavors have been extended virtual memory management services with the madvise call, that allows to this very thing and solves the problem. Unfortunately, Windows NT does not provide a similar mechanism yet.
CHAPTER 3. MONET

Threads And Scheduling

In OLTP loads, it is highly important that DBMS locks are held for as short time as possible in order to favor concurrent execution of many queries. One specific cause for holding a lock during a long time is when a process is scheduled out by the OS just when it is holding a lock. INGRES therefore sometimes exhibited “convoy” behavior, where each of its six processes would execute requests sequentially instead of in parallel, because all would wait for the same locks. To prevent this from happening the OS process scheduling needs to be aware of threads of control in a process, their synchronization primitives and allow the DBMS to influence the scheduling priorities of its threads. These issues have all been resolved now in the POSIX pthreads standard, which provides all desired functionality (Windows NT provides similar facilities).

File I/O

OS file systems do not provide the file I/O facilities that support the concept of atomic transactions, which is what a database system needs. Also, the minimum I/O granularity (the disk block) tends to be of fixed size, hence cannot be adapted dynamically to the needs of a DBMS application (smaller in order to reduce contention, or larger for clustering purposes). These OS deficiencies still persist, and therefore raw-disk I/O still can provide better DBMS consistency and performance (Oracle, which offers both OS and raw disk I/O, states a performance gain for the latter of 15% [Ora97a]). However, these problems mostly involve OLTP applications and their needs (random I/O performance). Query-intensive applications mainly use bulk I/O, where the main DBMS demand simply is high throughput, and this usually has been implemented satisfactorily in the OS.

We think that the OS situation has improved till the point that it is feasible to build a system like Monet without duplication of effort between the DBMS and OS. To be more precise, while we started development on 32-bits platforms, Monet also exploits these new 64-bit platforms and their large addressing spaces; it uses OS facilities for multi-threading, file I/O (for bulk data access) and virtual memory buffering (for random access). This approach allowed us to achieve a fully functional system with relatively few lines of code (about 30,000) and makes the system less dependent on hardware characteristics, and therefore more portable – which is the reason of existence of an OS in the first place.

3.3.4 idea 4: optimize main-memory query execution

In Section 3.2.2, we identified memory access and CPU utilization as the main performance bottlenecks for execution of OLAP and data mining queries. As a consequence, many issues in the design of Monet are closely related to the field of main-memory database systems (MMDBMS), although residence in main memory of the entire database is not our assumption.

This orientation towards main-memory execution is notable in various areas of the Monet architecture, and involves data structures, buffer management facilities, query processing algorithms and implementation techniques.
3.3. **MONET ARCHITECTURE**

**Basic Data Structures**

Monet stores data exclusively in BATs. The physical design for the basic BAT table structure is ultimately simple: a memory array of fixed-length records, without holes for unused space. This structure optimizes memory cache usage on sequential scans, as there is no wasted space hence 100% of all data that is loaded into the memory caches is actually used.

Concerning auxiliary structures, we build upon past results into index methods in main-memory databases [LC86a], which identified simple hashing and a binary tree structure called the T-tree to perform best. Monet uses direct-hashing with a bucket-chained implementation for handling collisions and a variant of T-trees for accelerating value lookups.

**Buffer Management**

In the design of Monet we took great care to ensure that systems facilities that are only needed by OLTP queries do not slow down the performance of query-intensive applications. This principle has been applied both to transaction management – by offering explicit transaction primitives separate from query processing – as well as to buffer management.

Relational DBMS products tend to be centered around the concept of a disk block or page, where a pivotal role is played by the buffer manager component, that reads parts of the database file into memory one disk block at a time, and the algorithmic focus in query execution is to do the work in the least number of I/Os as possible. The needs of query-intensive applications are different, though: they mostly use sequential read bandwidth, and the main optimization challenges are in the area of memory access and CPU utilization. As a result, from the standpoint of query-intensive applications, much of the relational buffer management just serves the needs of OLTP, and is superfluous or even a hindrance for achieving high performance.

Buffer management in Monet is done on the coarse level of a BAT (it is entirely loaded or not at all), both to eliminate buffer management as a source of overhead inside the query processing algorithms and because all-or-nothing I/O fits the hardware trends much better than random I/O. Recall from our example in Section 3.2.1 that sequential I/O currently gives about 40 times more bandwidth than reading block-for-block. In case of very large datasets, it is not possible to load an entire BAT into memory. In those cases we reduce the buffer management grain size by using explicit horizontal fragmentation, loading fragments of a BAT at a time (which still are relatively large).

**Query Processing Algorithms**

Query processing algorithms, like hash-join and sorting, until now assumed that access to the main memory was cheap and memory access cost was uniform, independent on locality of reference. This is not true anymore. Clearly the strongest hardware trend in Figure 3.1 is the lagging performance of memory access latency. Whereas memory access in 1992 cost 7 CPU cycles, the typical costs at the turn of the millennium are between 70 and 100 CPU cycles and rising.

We now shortly discuss the effect of the *memory access bottleneck* on algorithms for common operators in query-intensive DBMS loads:
• **selections.** In OLAP and data mining loads, the selectivity is typically low, which means that most data needs to be visited and this is best done with a scan-select. Sequential access makes optimal use of the memory subsystem and does not pose any problems.

• **grouping and aggregation.** Two algorithms are often used here: sort/merge and hash-grouping. In sort/merge, a table is first sorted on the GROUP-BY attribute(s) followed by scanning. Hash-grouping scans the relation once, keeping a temporary hash-table where the GROUP-BY values are a key that give access to the aggregate totals. This number of groups is often limited, such that this hash-table fits the memory caches. This makes hash-grouping superior to sort/merge concerning main-memory access; as the sort step has random access behavior and is done on the entire relation to be grouped, which probably does not fit any cache.

• **equi-joins** Hash-join has long been the preferred main-memory join algorithm. It first builds a hash table on the smaller relation (the inner relation). The outer relation is then scanned; and for each tuple a hash-lookup is done to find the matching tuples. If this inner relation plus the hash table does not fit in any memory cache, a performance problem occurs, due to the random access pattern. Merge-join is not a viable alternative as it requires sorting on both relations first, which would cause random access over even a larger memory region.

Consequently, we identify equi-join as the most problematic operator with respect to memory access and introduce in Chapter 4 a join algorithm called **radix-partitioned hash join** that optimizes the memory access pattern of hash-join.

We think the ever growing memory access bottleneck has shifted the “algorithmic battleground” from I/O access optimization towards memory access pattern optimization. Notice that if an algorithm has a high cache hit-ratio in one level of the memory hierarchy, it automatically also has high hit-ratios on the lower levels. This memory hierarchy starts at the top in the CPU registers, with various levels of cache memory below it, followed by the physical main memory and finally down to the virtual memory swap file on disk. That is, we consider virtual memory paging to be essentially the same as memory access, where a page fault is nothing else than a cache miss, only with a large cache line size (a memory page) and a latency that is in the milliseconds. The similarity between memory and I/O is complete now that new memory technologies like Rambus [Ram96] have a sequential bandwidth that is much higher than the bandwidth achieved with random access (that is, cache line size times latency), which has always been characteristic to I/O.

In Monet, we thus concentrate on main-memory execution, as we think it is in main memory where the crucial access pattern tuning must take place. I/O is done with the help of the OS by mapping large BATs into virtual memory.

**Implementation Techniques**

Implementation techniques are a somehow vague, yet highly important issue in the construction of a DBMS. In Monet, we focus on techniques that aid the compiler to create code that is efficient with respect to memory access and CPU performance.

One notable technique is to eliminate function calls from the inner loops of performance-critical routines, like hash-join. Function calls incur a significant penalty in terms of
3.3. MONET ARCHITECTURE

CPU cycles (at least 25 on current architectures) and memory access, but also make it more difficult for a compiler to perform optimizing code transformations. In the C language, a function call might modify any of the areas pointed to by pointer variables in the program, hence calling a function disables many compiler optimizations. Eliminating function calls from the inner loops of operator is only feasible if the query execution infrastructure of the DBMS is specifically prepared for this, which is the case of Monet. Recall that Monet operators have direct access to the data in a BAT, and do not need to call the buffer manager or lock manager while accessing tuples. Still, an operator like hash-join needs to perform bread-and-butter actions on join column values, which can be of any type, like computing a hash-number, or comparing two values for equality. Such functionality on the various data types supported by the DBMS is typically implemented using an Abstract Data Type (ADT) interface or – when using object-oriented DBMS implementation language – through class inheritance with late binding on methods. Both approaches require multiple routine or method calls for each tuple processed in the hash-join.

This function call overhead is eliminated in Monet by providing specialized implementations of the join-algorithm for the most commonly used types (a.k.a. compiled predicates). These specialized implementations compare two values using a direct value comparison in the programming language. The Monet source code is kept small by generating both the default ADT instantiation and the specialized ones with a macro package from one template algorithm.

As a result, a sequential scan over a BAT comes down to a very simple loop over a memory array of fixed-size records, whose values can be processed without making function calls. This makes Monet’s query operator implementations look very much like scientific programs doing matrix computations. Such code is highly suitable for optimization by aggressive compiler techniques, like loop unrolling [Sil97] or memory prefetching [Mow94]. Additionally, we apply main-memory database programming techniques like logarithmic code expansion [Ker89] to explicitly unroll loops in situations where the compiler does not have sufficient information.
CHAPTER 3. MONET

3.4 Conclusion

In this chapter, we have given a short overview of the goals we set out to achieve with the Monet system:

- adapt database architecture to the characteristics of modern hardware in order to improve the CPU and memory cache utilization.

- investigate the possibilities for a database kernel that is able to store multiple data models and query languages defined on those, such as the relational, the object-oriented model as well as e.g. semi-structured data (XML).

- investigating new data model and query language directions with respect to the established (object) relational model and OQL/SQL languages, to better address the needs of (emerging) extended domains that are characterized by complex data structures and query-intensive applications (data mining, multi-media, GIS).

In our motivation for these goals, we focused specifically on (performance) problems now encountered in relational database technology on modern hardware and on query-intensive applications. Here, we remarked that the basic data storage in relational technology tends to be optimized for OLTP applications rather than query-intensive applications. As for hardware utilization, we observed that I/O and memory latencies are an exponentially increasing bottleneck in query processing as these characteristics are the only ones not to follow the law of Moore in commodity hardware. Concerning I/O, this trend makes sequential scans (at exponential pace) increasingly more attractive than random based table (index) access, changing currently known trade-offs, especially in query-intensive applications where queries tend to have lower selectivities than in OLTP. The memory latency bottleneck poses new challenges concerning algorithms and data structures found in database processing – a fact that is still relatively unknown to the database community at the time of this writing. The increasing importance of compile-time optimizations for achieving super-scalar CPU efficiency [ACM+98], and the specific problems DBMS software exhibits in this respect [BGB98, KPH+98, TLPZ97, ADHW99] are another such overlooked challenge.

As for the question how the architecture of Monet addresses these challenges, we outlined a mix of ideas. The first is to focus on a small kernel of database functionalities, hoping to so capture better a common set of features that can be shared over multiple data models, query languages and extensibility domains. The second idea is to use vertical fragmentation as a cornerstone, as this physical data storage strategy fits well the query-intensive domain, in that it helps optimize I/O and memory cache access, as well as forms a simple and clean data abstraction that can be used as building block for a variety of data models. The third idea is to do everything possible in the database architecture in order to facilitate the creation of database processing algorithms that can squeeze optimum CPU/memory performance out of current commodity hardware, which translates into total absence of a buffer manager and transaction system from the critical path of its query operators, as well as creating a "RISC" query algebra where the operators have a very low degree of freedom as to enable in their implementation the use of a number of programming techniques normally only found in scientific computation programs inside their performance-critical inner loops, with the goal of allowing compilers to produce more efficient code for modern, super-scalar, CPUs.