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

Boncz, P.A.

Citation for published version (APA):
Chapter 2

DBMS Architecture

DBMS software offers much functionality and tends to be big and complex. The architecture of a relational DBMS, typically contains the following modules:

**query language parser** reads input from a user in a query language like SQL, checks it for syntactical and semantical inconsistencies, and if none are found translates the query to an internal representation (often, a *parse tree* data structure).

**query rewriter** takes the parsed query as an input and rewrites it to some standardized or *normal form*, while checking the query with respect to its authorization and semantic constraints. The query rewriter typically also takes care of expanding table-views used in the query to their full definition.

**query optimizer** translates the logical description of the query to a query execution plan. Often, many different translations are possible, each leading to a different query execution plan with different expected execution time and resource usage. The query optimizer is responsible for finding the best plan, or if that is not possible due to a huge search space with many alternatives, to at least avoid the bad plans. If parallel query execution is requested, query optimization also involves translating the execution plan into a parallel execution plan.

**query executor** executes the physical query plan, and produces the final result.

**access methods** are system services that provide access to the data stored in the database tables. Often, index structures like the B-Tree or hash-tables are used to speed-up access.

**buffer manager** is the system component that handles caching of table data stored on disk in the main memory. This is often done in a block-wise fashion, where the buffer manager maintains a buffer pool of cached disk blocks. The buffer manager interacts with the query executor in that it tries to adapt its caching strategy such that a minimal number of block I/Os is necessary.

**transaction manager** provides system services as locking for database transactions. Different grains of locking may be supported (page-level or row-level), and often deadlock-detection and -resolution is one of the additional tasks of this component.
**recovery manager** makes sure that whenever transactions commit, their results are made persistent, and whenever they abort, their results are erased as if they never happened.

Figure 2.1 shows how these modules interact.

Figure 2.1: Relational DBMS Architecture

In the following, we give a short chronological and thematic overview of DBMS architectures, in order to provide a context of reference for the ideas investigated in this thesis. We start discussing the early relational systems INGRES and System R, that have been hugely influential on the rest of the field. We then visit Database Machines, Parallel DBMSs, Extensible DBMSs and the increasing use of object-oriented techniques in database systems. Finally, we discuss database systems used for OLAP and data mining, which is the field of interest of this thesis. Here, we describe why traditional relation DBMSs have performance problems in this application area and describe a number of proposed techniques for improving this.

### 2.1 The Roots

In the early 1970s – some years after Codd published his paper that introduced the relational model [Cod70] – several simultaneous attempts where made to implement a first relational DBMS. Until that time, information systems were run on database systems based on networked or hierarchical data storage, like in the CODASYL standard [Oll78]. The first widely known relational implementations were INGRES, developed at UC Berkeley [SWKH76] and System R, developed at the IBM San Jose research facility [ABC+76].
2.1. THE ROOTS

INGRES was the first system to use UNIX as its implementation platform. It used the relational query language QUEL, that is not identical, yet very similar, to today's SQL. Several ideas from INGRES can still be found in current relational products, like the concept of a query rewriter for implementing views and integrity constraints, the use of relational tables to store meta-information (the data dictionary) and the idea of extensible query access methods. INGRES was commercialized into a product, and from it the Sybase DBMS evolved, later to be re-marketed and enhanced into Microsoft SQLserver.

System R introduced many important concepts like the SQL query language [CB74] (back then, it was spelled SEQUEL), and much exemplary work in logging and recovery was done in the implementation of the lower levels of the system [GMB+81]. System R was split into an upper layer called RDS (relational data system) that contained the query language parser, rewriter and optimizer, and a lower layer called RSS (relational storage system).

Join is the most complex relational operator and much research has been devoted to its most common variant: equi-join. INGRES used nested-loop algorithms for join that scan over the outer relation and perform index loop into the inner. System R has sort-merge as its main algorithm, in which both relations are first sorted on the join attribute, and subsequently merged. Later research into join algorithms showed, however, that hash-join algorithms perform at least as good, if not better than the algorithms employed by INGRES and System R. Specifically, the hybrid hash-join algorithm, proposed in 1984, is still considered the best general purpose join method [DKO+84].

The System R Team was the pioneer in the area of query optimization [SAC+79]. Its optimization algorithm restricts execution of multi-join queries to linear join trees, (i.e. one of the join operands is always a base relation), so that available access structures for the inner join operand can optimally be exploited. System R chooses the cheapest (in the sense of minimal total costs) linear tree that does not contain Cartesian products. This basic algorithm is still used in many of today's relational DBMS products.

Transactional performance was one of the primary concerns of the early relational systems, because when relational DBMSs were first proposed, an often heard critique from the opposing CODASYL camp was that relational transactional performance would never be sufficient. An important technique pioneered by System R was to amortize query interpretation and optimization effort in standard transactions over many executions, by compiling such frequently repeating queries into hard-coded query plans. Such a "canned" query would check whether the index structures and relations on which the plan depends would still exist (if not, re-compilation would occur), and directly execute a (typically small) sequence of calls to the RSS. While System R also allowed querying with ad-hoc query interpretation and optimization, its compiled queries greatly enhanced its transaction performance and therefore helped in making relational DBMS technology a commercial success.

Building on System R, additional research results were achieved in the area of distributed database systems (System R* [HSB+82]) and transaction management (the ARIES [MHL+92] methodology for efficient logging and recovery). Parts of System R made it into various IBM products like QBE, SQL/DS and indirectly DB2. Although not based on its source code, the Oracle system was designed to resemble System R very closely.
2.2 Database Machines

One of the avenues pursued to obtain high performance in early relational DBMSs was to work with specialized hardware, like disks with a specialized CPU mounted in each of its disk heads, as in the CASSM [HS81] and RAP [OSS77] systems. Such a CPU could evaluate a simple selection predicate inside the hard disk, limiting the amount of data to be transported over the bus to the central CPU. Hardware solutions were also tried in the area of parallel databases, like the DIRECT [DeW79] and PRISMA [AvdBF+92] systems. The latter – developed by our research group in collaboration with University of Twente and Philips – used a specialized interconnection network architecture, while the former also employed special-purpose CPUs for query processing. Such database systems, that consist of both software and custom-made hardware were called database machines.

Currently, the idea of database machines has been fully abandoned [BD83], for a number of reasons. The experience in the PRISMA project taught us that by the time the software to program the database machine was fully functional, the (then four year old) special-purpose hardware had already become obsolete in comparison with commodity hardware. Also, the development cost for specialized hardware that is used in only a very limited application area must be amortized over very few units sold. Therefore, specialized hardware has a bad price/performance characteristic compared to commodity hardware. Finally, DBMS software for a database machine is specifically designed for a highly particular architecture, and is therefore inherently non-portable. This seriously shortens the life-cycle of the DBMS as a piece of software.

Recently, we see two developments that might spell a partial come-back for some techniques from the database machine era. First, modern commodity processors like the Intel Pentium MMX and beyond, AMD K6/K7, but also Sun UltraSparc, provide SIMD (Single Instruction Multiple Data) instruction sets like VIS, MMX, 3dNOW and SSE [PWW97, OFW99, Die99], that perform simple arithmetic operations on 4, 8 or even 16 small data items (integer, float) in one CPU cycle. While these special-purpose instructions were originally targeted to multi-media applications only, hardware manufacturers currently are enriching and generalizing them in each new processor generation to re-target them to other domains, like streaming-internet applications. Therefore, at some point SIMD instruction sets may become usable in core DBMS algorithms.

A longer-shot development is the alternative IRAM computer architecture currently being proposed [KPP+97]. An IRAM (Intelligent RAM) can be best described as a memory chip that also contains a CPU, thus providing the possibility to execute operations inside the memory, where this local CPU has a huge memory bandwidth as it has direct access to it rather than through a bus. The idea behind IRAM is to create a bus-less computer system, in order to eliminate the Von Neumann bottleneck. A computer would consists of many such IRAM chips, possibly with (a) central coordinator CPU(s). Programming such a new computer model, implies for the DBMS that operations on database tables that are partitioned over various IRAM chips are executed locally, in parallel, in each memory chip, which brings us back to database machine architectures.

The important lesson learned from database machines, however, is that hardware-specific features should only be used when the hardware in question is a commodity.
2.3 Parallel DBMSs

There are two targets for increasing DBMS performance using parallelism: *speed-up* and *scale-up*. The former means that a problem of fixed-size complexity is solved quicker on parallel hardware than on sequential hardware. The latter means that a bigger problem can be solved on parallel hardware, in the same time it takes on sequential hardware. In parallel DBMSs, there is a distinction between *inter-query* and *intra-query* parallelism. Inter-query parallelism means that multiple queries are processed concurrently by different hardware units. In transaction systems, inter-query parallelism is often employed to obtain transaction scale-up (i.e., the ability to handle a greater number of transactions per minute using additional hardware). This is a relatively simple form of parallel query execution. In this discussion, we focus on intra-query parallelism, in which a complex and long-running query is split into multiple sub-tasks, that are executed in parallel. A successful parallel DBMS achieves linear speed-up and/or scale-up; obtaining a factor N of improvement on hardware consisting of N parallel units. This ideal case is generally not achieved as parallel query execution tends to suffer from three overheads: startup costs that cannot be parallelized, (memory,disk,cache) interference between concurrently executing sub-tasks, and poor load balancing which is often caused by uneven, skewed, distributions in the data.

![Parallel Hardware Architectures](image)

Figure 2.2: Parallel Hardware Architectures

The hardware architecture for a parallel DBMS can be classified as follows (see Figure 2.2):

*shared-everything*. All CPUs have access to the same memory and disks.

*shared-memory*. All CPUs have access to the same memory, but have individual disks.
shared-disk. All CPUs have individual memories, but share the same disks.

shared-nothing. All CPUs have separate disks and memory, and communicate exclusively via a network.

The first three architectural solutions only provide scalability to a handful of CPUs when standard hardware components are used. Therefore, parallel DBMSs pursuing these architectures needed to employ non-standard hardware and fall in the earlier discussed database machines category (e.g., DIRECT [DeW79] and GRACE [FKT86]). While it was generally agreed that shared-nothing architectures were the future [DG92], recent interest by high-end hardware manufacturers in ccNUMA [LL97] architectures spelled a come-back for the shared-memory approach. Shared-memory computers regained this popularity because they are easier to program, which is an argument more important for scientific super-computation than for parallel DBMSs. On the other hand, developments in network hardware (huge growth in bandwidth with much lower latencies [BCF+95]) combined with the rise of the internet, clearly favor shared-nothing architectures. In any case, the cheap and easy availability of dual- and quad-CPU configurations even in consumer PC hardware means that future parallel machines often will be hybrid systems, which consist of shared-nothing network of nodes, where each node is a shared-memory multiprocessor.

Techniques for parallel execution of relational queries are now well understood. Parallelism can both be obtained by pipelining streams of tuples through a query operator tree, where each operator in the tree is handled by a separate CPU executing in parallel. This scheme is also known as vertical parallelism. Vertical parallelism alone, though, warrants poor load balancing, as not all operators have equal cost, and the number of tuples flowing through the tree varies among operators. An alternative technique is horizontal parallelism. Here, the same query operators are then executed in parallel on different CPUs, where each CPU processes a different subset of all tuples. Horizontal parallelism requires horizontal partitioning the relations in order to distribute the partitions over the available CPUs. This partitioning can be done either using round-robin, range- or hash-partitioning algorithms.

Though parallel sort-merge was attempted in early database machines [FKT86], hash-join is the more popular algorithm for parallel join. Parallel hash-join using horizontal parallelism was pioneered in the GAMMA [DGS+90] system. In this approach, both relations to be joined are hash-fragmented and distributed according to hash-key over the available CPUs. Each pair of fragments that coincides in hash-key is joined using standard hash-join. The standard hash-join algorithm first builds a hash-table on the inner fragment. Incoming tuples from the outer fragment are probed to this hash-table, and if matches occur, join tuples are emitted. The PRISMA [WFA95b] parallel DBMS combined both horizontal and vertical parallelism, using a special pipelined hash-join that builds a hash-table on both its operands on-the-fly, and thus is able to directly start producing output tuples instead of having first to complete a build phase [WFA95a]. An alternative strategy is to work with right-deep linear join trees only, as this allows to execute the build phase of the simple hash-join algorithm in parallel on all base relations involved in the join tree [SD90]. The concept of encapsulation of parallelism with an exchange operator (sometimes also called split) in the Volcano system [Gra94] showed how both horizontal and vertical parallelism can be implemented elegantly and efficiently in a way that keeps the basic query operator implementations (i.e., select, project, join) free of hard-coded synchronization and
2.4. **EXTENSIBLE DATABASE SYSTEMS**

flow-control.

Figure 2.3: Parallel Query Execution Strategies

Figure 2.3 illustrates horizontal and vertical parallelism in a 4-way join query on four parallel CPUs, assuming a pipelined two-way hash-join without build phase (like in PRISMA). Join C incurs twice as much work as joins B and D, and four times as much as join A. The resource usage graphs show the letter A, B, C or D when a CPU is busy producing tuples for the corresponding join. The figure shows that in this case, simple vertical parallelism (uppermost) yields sub-optimal speedup due to bad load balancing, whereas horizontal parallelism (middle) yields perfect speed-up. The latter strategy, however, fully materializes intermediate results. If this does not fit in memory, a strategy that mixes horizontal and vertical parallelism (below) may perform best.

Almost all current relational DBMS products exploit inter-query parallelism on shared memory multiprocessors through multi-threading. Products like DB2, Informix and Oracle8 also offer parallel options to exploit intra-query parallelism on shared-nothing clusters. These implementations tend to be pragmatic and targeted to relatively simple queries. Massive parallelism is exploited in the products of Tandem [Gro87] and Teradata [BDS89], which build large-scale (e.g., 1000 CPU) shared-nothing systems from commodity hardware. Research interest in parallel query execution has shifted from the core issues towards parallelizing query execution in new application domains, such as GIS [DKL+94, SRKC95] and Data Mining [HKK97, LS98].

### 2.4 Extensible Database Systems

The early relational DBMS focused on supporting the needs of the “business” domain. Information systems in the business domain (used to) store data items with simple numerical and textual attributes only. Though “business” is probably the application domain with the most economic buying power (e.g. the financial sector), it is certainly not the only domain where large data volumes have to be managed. Other domains are: scientific information systems (e.g. storing tape-racks full of satellite measurements), geographical information systems (GIS) which contain both geometrical (point, polygons) and topological map data, multi-media information systems (storing images, sound, video), medical information systems (storing e.g. DNA sequences), etc.
Several research projects have focused specifically on making the DBMS extensible to such domains. The successor project to INGRES, called Postgres [SR86, SAH87], addressed these issues by opening up the DBMS kernel with a number of Abstract Data Type (ADT) interfaces. This extensibility interface allows for an extension module to be loaded into a running system. Such a module can introduce new atomic data types (e.g., polygon), new functions on these types (e.g., bool overlap(polygon, polygon)) and new supporting index structures (e.g., the R-tree [Gut84]). The driving domain behind the extensibility features of Postgres were the earth-sciences, which manage huge volumes of both geographic data like maps as well as satellite images that consist of bitmap data. The applicability to this domain was tested on a functional benchmark, called SEQUOIA [SFGM93]. Postgres was further developed into a product called Illustra, which was shortly after integrated into the commercial Informix DBMS [Ger95] (now an IBM product).

Once new data types, operators and index structures are added to the DBMS, somehow the query optimizer should also be told what to do when queries involving these types are optimized. The Starburst research system from IBM Almaden [HFLP89, SCF+86] addressed this problem, using a rule-based query optimizer and making it extensible with new rules. One important question concerning extensibility is: who does the extending? In the case of Starburst, this extending could only be done by a DBMS kernel programmer. Alternative extension programmer roles are: a DBMS extension-module developer (without access to the DBMS kernel code), a DBMS application developer, or even a DBMS end-user.

In extensible relational systems, the first alternative – third-party extension developers – has been the most popular choice. An example of an early DBMS prototype offering this is Gral [Güt89], which featured a fully extensible query optimizer, that was based on a configurable rule system for translating queries in a logical algebra into a physical query plan. The implementation of this system was targeted at the GIS domain. Commercial relational DBMS systems like Informix, DB2 and Oracle still refrain, however, from opening up the query optimizer to extension modules, because of the possible instability this might introduce: user-defined optimization rules may make faulty decisions, and could affect query execution in general.

Along the same lines, it has been argued [Fra84] that user-defined extension modules should execute in an address space that is fully separated from the DBMS kernel. A buggy extension – if run inside the same memory space as the DBMS kernel – may perform unwarranted calls to DBMS API functions, or do uncontrolled writes into
2.5. OBJECTS IN DATABASE SYSTEMS

Figure 2.5: Various ways to run extension modules in a DBMS

e.g. system buffers and corrupt DBMS tables, undermining the stability of a DBMS as a whole. On the other hand, executing all module-functions in a separate process implies that access to extension module functionality must always go through some inter-process communication mechanism, at the cost of at least one context switch per call. This can be highly expensive, especially if such overhead must be paid for each processed tuple; possibly millions of times in a single query. Commercial DBMS products have resolved this safety/performance trade-off in different ways: Informix allows its “data-blades” to execute in kernel-space, whereas Oracle8 separates its “data-cartridges” in a different process. An interesting development in this respect is integration of Java virtual machines into the DBMS kernel. The Java language has a higher level of abstraction than C or C++, and does not have the concept of pointers to memory regions; hence is easier to control. Therefore, a buggy piece of Java code can be prevented from writing into “random” memory locations, making it possible to safely run user-defined Java extensions inside the DBMS kernel. Therefore, we think that Java, especially when combined with native compilation, may be a key technology for achieving high performance DBMS extensibility, without sacrificing safety.

A modern extensible DBMS worth mentioning here is PREDATOR, which makes a case for E-ADTs (E for Enhanced), that open up an extension modules to the query optimizer by defining methods in a declarative way, rather than hard-coding them in a compiled extension module that basically is a black box for the query optimizer. In combination with additional optimization rules, queries involving expensive user-defined methods (like \texttt{sharpen(Image)}) can be fully pipelined and optimized. As an example, \texttt{clip(sharpen(Image))} can be rewritten into the faster \texttt{sharpen(clip(Image))}, which avoids loading the whole image from disk and reduces the sharpening work [SP97, Ses98]. Also, this system has showed promising results on the use of Java for creating safe extension modules and the performance trade-offs involved [GMSvE98].

2.5 Objects in Database Systems

Object-oriented database management systems (OODBMS) take the principle of extensible database systems even further, by working with a fully object-oriented data model. The concept of “relation” is replaced by “class”, “tuple” is replaced by “object”, and “table” by “collection”. An object-oriented database is the union of “extent” collections, which contain all existent objects of one class. The object-oriented paradigm brings enrichments with respect to the relational model in numerous as-
pects: classes can be specialized from each other in an inheritance hierarchy, classes can have methods defined on them, which can be overloaded in sub-classes, and often multiple collection types are supported (e.g., Set, Bag, List) that can also be nested. Many different flavors of object-oriented database technology exist. We limit ourselves in this discussion to three main types: object-oriented toolkits, “proper” OODBMSs and object-relational DBMSs.

**toolkits** One viewpoint to object-oriented database architecture was to create an extensible toolkit that provides all basic services needed to support a full-fledged DBMS. Such a toolkit could then easily be “finished” into a complete DBMS, by implementing the top layers. Such a top layer would define the specific application interface presented by the DBMS, specializing it into an image-DBS, document-DBMS, engineering-DBMS, etc. Object-orientation was used as the main vehicle for facilitating deployment of the basic components of the toolkit approach. That is, the inheritance mechanism is used to specialize the generic DBMS classes provided by the toolkit in the final domain-specific DBMS. Well-known toolkit systems are Genesis [BBG+88, Bat86], Exodus [GD87, CD87], DASDBS [Sch87] and more recently Shore [CDF98, Tea95]. Commercially, though, these systems have not made an impact. Developing a full-fledged DBMS product requires enormous effort – even in the case of the toolkit approach – and occurs infrequently. Therefore, there is not much market for a toolkit DBMS. Additionally, it has been found difficult to offer the correct level of abstraction in the toolkit interface. The experience of building specialized systems on top of EXODUS thought that the toolkit interface often hid too many details. In the case of EXTRA/EXCESS [VD91], the choices made in the EXODUS built-in client-server layer got in the way of efficiency, and the E programming language interface did not provide the tuning hooks that were actually desired [CD96]. On the other hand, the interface provided by a database toolkit is too low-level to directly support DBMS applications comfortably.

**object-oriented** The “proper” OODBMS approach specifically focuses on bridging the impedance mismatch that lies between the application programming language and the DBMS query language. In these systems, object-oriented programming languages like Smalltalk, C++ and more recently Java, can be used to directly access objects stored in the object-oriented database. That is, the object-oriented data model provided by the programming language has a one-to-one mapping with the data model in the DBMS: a database object is just a persistent programming language object. This removes the hassle and overhead of embedding SQL commands in a totally different programming language like C, and parsing out returned values from some result buffer, as typically is necessary when accessing a relational DBMS from an application program. The first application area where these systems got a foothold were engineering applications. In such applications, complex designs are made by combining many different parts that are categorized in a parts hierarchy. These designs are accessed by CAD/CAM software for interactive updates and visualization. Such applications greatly benefit from a tight coupling between application and database. The two standard OODBMS benchmarks OO1 [Gra93b], and its successor OO7 [CDN95], address such engineering problems. Well-known OODBMS implementations are ORION [KGBW90], O2 [Deu90], Gemstone [BOS91], Poet [Ple97], Object-Store [LLOW91], and Versant [WO98], of which the latter four are still available as commercial products. When these systems were designed, their language bindings and
2.5. OBJECTS IN DATABASE SYSTEMS

Object query facilities were all different. The ODMG was founded in order to establish a standard for object-oriented database systems. The ODMG-92 and ODMG-93 standards [CAB+94, CBB+97] define the ODL data model, language bindings for both Java and C++, and the object query language OQL. Until now, however, differences between the available systems remain, since no single system fully implements the ODMG standard.

The tight coupling with an object-oriented programming language in these OODBMSs is not only a strong-point, but also their weak-point. First of all, the principle that a programming language object is equal to a database object means that the OODBMS must adhere to the object storage layout imposed by the programming language. This eliminates the freedom for the OODBMS to use a physical structure that suits a specific access pattern, e.g. including index information, or table partitionings or clusterings. Secondly, the fast and direct language bindings stimulate programming in ODMG/C++ of e.g., foreign-key joins by following pointers through related objects. This may (in some cases) be efficient, but concerning data independence, such implementations are nothing better than the “pointer-chasing” DBMS application programming style from the CODASYL era. In contrast, the declarative nature of relational technology allows a relational DBMS to optimize, parallelize, change the physical storage and index structures on a table, or even change the relational schema. Freeing the user of thinking about these details and managing them automatically in an intelligent way is one of the very accomplishments of DBMS technology. OODBMSs are clearly weaker in this respect. A final problem is that the query language part of OODBMSs is generally underdeveloped, and either limited to very simple queries, or implemented without good query optimization. This is also reflected by the OO1 and OO7 benchmarks, that focus mostly on traversal performance and just contain trivial queries. Only the O2 system – which is currently not on the market anymore – had a full implementation of OQL. Optimization of OQL is much more difficult than SQL, because OQL is a nested language with multiple collection types, and must also handle the additional complexities introduced by method invocation and method overloading (what happens e.g. when methods have side-effects?). It is still unknown how OQL expressions can be fully translated to algebraic equivalents. Therefore, existing OQL implementations mostly resort to an easy translation into nested loop algorithms; which guarantees mediocre performance on complex queries. Some work has been done to remedy this situation [SdBB96, SAB94, SABdB94, GKKS97, CZ96, CZ98], but the problem is not solved yet, and research interest to solve it seems to be diminishing.

Object-relational When OODBMSs were proposed in the mid-eighties, a paradigm shift from relational to object-oriented databases seemed likely. This has not happened, because both object-oriented approaches described above had too many drawbacks to easily deploy them in areas where relational DBMSs are successful now. The reaction of the relational DBMS vendors has been to evolve their extensible relational technology, gradually adding more object functionality. One novelty is the introduction of row types, that enriches the concept of a relational tuple with certain object features. Like objects, row types can have methods defined on them, and be organized in an inheritance hierarchy. Such incremental addition of object features may not warrant an elegant DBMS architecture, but at least it preserves the achievements of relational technology, while still adding useful object-oriented functionality. All extensible relational DBMS products (Informix, DB2, Oracle) are moving in this object-relational direction, and functionality is aimed to converge in the SQL3 standard.
2.6 Main Memory Database Systems

During the mid-1980s falling DRAM prices seemed to suggest that future computers would have such huge main memories that most databases could entirely be stored in them. In such situations, it would be possible to eliminate all (expensive) I/O from DBMS processing. This seriously changes the architecture for a DBMS, as in a Main Memory DBMS (MMDBMS) there is no central role for I/O management.

An important question in a MMDBMS is how to do transactions and recovery in an efficient way. Some of the proposed algorithms [LC86b, Eic87], assume that a (small) stable subset of the main memory exists, a piece of memory whose content will not be lost in a power outage through a battery backup. These stable memories can be used to place e.g. a redo log. Others do not assume stable memories, and still use I/O to write transaction information to stable storage. These algorithms hence do not eliminate I/O (e.g. "logical logging" [JSS93]), but minimize it, as the critical path in a MMDBMS transaction only needs to write the log; not data pages from the buffer manager.

The main asset of a MMDBMS is its unparalleled speed for querying and update. Information on design and implementation of basic database data structures and algorithms can be found in the overviews by Garcia-Molina and Salem [GMS92] and Eich [Eic89]. Some specific research has been done in index structures for main memory lookup [Ker89, LC86a, DKO+84, AP92]. It turns out, that simple data structures like the binary AVL tree, called T-Tree, and simple bucket-chained hash outperform bread-and-butter disk-based structures like B-tree and linear hash, due to the fact that the only costs involved in index lookup and maintenance are CPU and memory access.

A specific problem in MMDBMS is query optimization. The lack of I/O as dominant cost factor means that it is much more difficult in a MMDBMS to model query costs, as they depend on fuzzy factors like CPU execution cost of a routine. Therefore, DBMS query optimization tends to make use of simple cost models that contain "hard" constants obtained by profiling [LN96, WK90]. One challenge in this area is to model the interaction between coding style, hardware factors like CPU and memory architecture and query parameters into a reliable prediction of main memory execution cost.

The end of popularity of MMDBMS techniques came in the early 1990s, when it became clear that not only DRAM sizes had grown, but also disk size, and problem sizes. MMDBMS were thereafter only considered of specific interest to real-time database applications, like e.g. encountered in embedded systems or telephone switches. Still, main memory sizes in commodity computers continue to increase, and for those application areas whose problem sizes do not grow as fast, it holds that at a certain time they will fit in main memory. Recently, prominent database researchers concluded in the Asilomar workshop [BBC+98] that MMDBMSs have an important future in such application areas.

Well known main memory systems are Smallbase [BHK+86, LN96] developed by HP, the object-oriented AMOS [FR97] system, the parallel MMDBMS PRISMA [AvdBFR+92], and Dalí [JLR+94, RBP+93] by Bell Labs. Smallbase and Dalí have been reshaped into commercial products, under the names Times Ten [Tea99] and Datablitz [BBG+99], respectively. Their main focus is highly efficient support of OLTP DBMS functionality on small or medium-size data sets. Also, all main relational vendors (IBM, Microsoft, Oracle) are offering small-footprint "ultra-light" versions of their DBMS servers for use in mobile computing devices and web PDAs.
2.7 Decision Support and Data Mining

Data Warehousing is an interesting area of business where an organization unites data from its various information systems in one large (think terabytes) database. A data warehouse is typically used for query-intensive applications, OLAP and data mining, whereas the production information systems where the data warehouse is filled with typically are the OLTP systems. This filling is typically done overnight, in the weekend or even once a month. As data from (multiple) operational information systems is uploaded, a data cleaning effort must first be done to integrate all sources into a coherent schema. Data cleaning techniques may be simple transformations (data migration), but can also include highly domain-specific knowledge (data scrubbing). A data warehouse fill hence includes data upload from the OLTP systems, data cleaning, and subsequent index creation, sorting and possible pre-computation of aggregate tables; all of which can be very time consuming. The main reason why data warehouses tend to become very big is that they are designed to include historical data, and therefore grow into a time series of concatenated snapshots of the OLTP systems.

Data warehouses are often used for complex analyses by OLAP or Data Mining tools. Table sizes in data warehouses tend to be very large. These two ingredients spell a performance disaster when standard relational DBMS queries are used to support OLAP and Data Mining [BRK98]. The common workaround for this problem deployed by commercial DBMS vendors, is a specialized Decision Support System (DSS). Such a system can either be a kind of intelligent middleware that sits in between the OLAP application and the relational DBMS, or an independent server that directly manipulates multi-dimensional data structures instead of relational tables, and whose only connection with the relational DBMS is an efficient upload mechanism. The former approach (middleware) is called Relational OLAP (ROLAP), whereas the latter is called Multidimensional OLAP (MOLAP). Currently, many OLAP products claim to combine (some) of the best features of both under the name Hybrid OLAP (HOLAP).

MOLAP The central data structure in a MOLAP system is a multidimensional array [Ken95] that contains the data cube in pre-computed form. This representation allows for fast “drill-down”, “roll-up” and “slice/dice” operations between aggregated results. Severe problems arise if the number of dimensions of interest is medium-size or large. The number of cells in the cube grows with the power of the number of dimensions, so the size of the cube data structure quickly gets out of hand. Real-world data is often skewed, so clusters with high data density from in the cube, leaving much of the other cells empty. An intelligent MOLAP system exploits this, both with a multi-dimensional representation that compresses sparse areas, and in the algorithm to build the cube [HRU96]. Examples of commercial MOLAP products are Essbase [Hyp00] and Oracle Express [Ora97b].

ROLAP In a ROLAP system, some middleware sits in between a relational DBMS and the OLAP tools and tries to optimize the query load using a combination of middleware caching, materialized views and clever use of indices in the relational DBMS. The question of what to cache, index and materialize is often facilitated in practice by working with precooked solutions for standard problems. In particular, products often assumes a particular topology of the database schema, like a star or snowflake. In such schemas, there is one big fact table that records “events”, and maintains (hierarchical) information on the fact table attributes in small tables around it. One of the techniques
used is to store pre-computed aggregates in the fact table itself in ‘phony’ rows that have NIL (or ALL) values for the aggregated attributes. Microsoft SQLserver introduced a CUBE operator to SQL that works this way [GBLP96]. Another approach is to store pre-computed views on frequently used subparts of the schema [CCH*98]. Well-known ROLAP vendors are MicroStrategy [Mic99], and Informix Metacube [Inf98].

The Problem... The problem with both approaches (or any hybrid combination of features) is that they are all based on the idea of saving query time by re-using static pre-computed results, while the application area – decision support – aims to facilitate knowledge discovery, which is an interactive, ad-hoc process.

In MOLAP, the static pre-computation lies in the decision which dimensions should be in the multi-dimensional array, and which aggregate functions to pre-compute for each cell. Similar static decisions must be made by ROLAP system administrators, again regarding which dimensions to index on, what tables to cache, and which views to pre-compute. Performance disaster strikes when the OLAP end-user, e.g. the insurance company analysts, who decides that they wants to see e.g. claim totals grouped by region and insurance products for claims made on rainy days, and the weather dimension is not a pre-computed dimension. In that case, all pre-computed structures are useless and the only way to answer the query is to scan the complete relational table, which is unacceptably slow. Recall that the required resources (memory size, build-up time) for all pre-computation techniques increase with the power of the number of pre-computed dimensions, therefore aggregates for typically only a handful of dimensions can be pre-computed.

While the above problems limit the ad-hocness of OLAP querying possibilities of MOLAP, ROLAP and HOLAP systems alike, it practically inhibits the use of these techniques for Data Mining. In Data Mining, it is always unknown a priori which combinations of attributes are to be queried together (discovering these interesting combinations is what Data Mining is actually supposed to do). In Data Mining, there is no pre-knowledge that can be used to make intelligent pre-computation decisions. Therefore, supporting ad-hoc decision support or Data Mining on large datasets remains an unsolved problem.

2.8 Conclusion

This chapter has given just a birds-eye view of DBMS architecture through the last three decades. More complete overviews of the various areas of DBMS architecture can be found in the following excellent literature:

- A comprehensive introduction to database systems in general is Stonebraker’s “red book” [Sto93].
- Another source of interest is the query processing survey by Graefe [Gra93a].
- Overview material about object-oriented database techniques and parallel techniques can be found in [Ze89, LOT94].

We have also discussed various data processing architectures for OLAP and Data Mining. Here, we have intended to show why current database technology has performance problems on these query-intensive applications, as proposed optimization strategies so far assume that all needed data can be covered by pre-computation, while we argue that this assumption is not sustainable in data mining and ad-hoc OLAP.