Monet

A Next-Generation DBMS Kernel For Query-Intensive Applications

Peter Alexander Boncz

Monet:

A Next-Generation DBMS Kernel For Query-Intensive Applications

Academisch Proefschrift

ter verkrijging van de graad van doctor

aan de Universiteit van Amsterdam,

op gezag van de Rector Magnificus

prof. mr. P. F. van der Heijden

ten overstaan van een door het

college voor promoties ingestelde commissie

in het openbaar te verdedigen

in de Aula der Universiteit

op vrijdag 31 mei 2002 te 11.00 uur

door Peter Alexander Boncz
geboren te Amsterdam

Promotiecommissie

Promotor:	prof. dr. M. L. Kersten
Overige commissieleden:	Prof. dr. P. Valduriez Prof. dr. P. M. G. Apers Prof. dr. L. O. Hertzberger Prof. dr. ir. A. W. M. Smeulders dr. P. van Emde Boas

Faculteit

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

The research reported in this thesis has been initiated while the author was at the Science Faculty of the University of Amsterdam, at the Intelligent Sensory Information Systems (ISIS) research group of the Informatics Institute.

The research reported in this thesis was continued while the author was at Data Distilleries B.V., a CWI research spin-off company that uses the Monet system – subject of this thesis – for data mining functionality in its analytical Customer Relationship Management (aCRM) products.

The research reported in this thesis was finished at current position of the author at CWI, the Dutch national research laboratory for mathematics and computer science, within the theme Data Mining and Knowledge Discovery, a subdivision of the research cluster Information Systems.

The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Graduate School for Information and Knowledge Systems. SIKS Dissertation Series No-2002-07.

ISBN 90 6196 512 8

Cover: impression of Figure 3.1 (hardware latencies do not obey the law of Moore).

Acknowledgments

At the time I was studying computer science at the Vrije Universiteit Amsterdam, I used to regard databases (which I mostly identified with conceptual modeling and relational normal forms) as an extremely dull and uninspiring subject. My computer science interests were rather in computer architecture and systems engineering, in which area I completed the M.Sc. program in 1992 together in a joint project with my best friend Hylke Sprangers. It is therefore ironical that I ended up doing a Ph.D. in databases. This episode started in 1994 when I came across an OIO position with Martin Kersten in the MAGNUM project, where a new database system was to be developed to power – among others – a new generation of geographical information systems. The challenge of architecting and engineering a database management system had not occurred to be before, but immediately attracted me.

Martin Kersten at that time headed the database research group at CWI, and had just started at a part-time position as professor at the University of Amsterdam, where I became stationed. Martin has played a crucial role in all my research. He has a tremendous creativity that produces a continuous stream of ideas, ranging from brilliant to impossible, coupled with the courage to go against the tide of accepted scientific opinion when he believes in something. Many ideas underlying Monet – which were rather unusual at their inception – come from him, as well as the initial implementation. As we share many of our interest, our professional relationship has always been stimulating, sometimes by competitiveness in the positive sense.

In the MAGNUM project, I enjoyed working together with Carel van den Berg, Wilko Quak, Jan Flokstra and Annita Wilschut. In that time, Annita took charge of most of the organizational hassles, and made MAGNUM one of the few inter-university projects I know of where intense and fruitful cooperation occurred. Investing in hard work directed at objectives only in the mid- or longer-term was a characteristic of Annita, who was also invaluable in helping the junior project members like Wilko and me in acquiring basic research skills. Meanwhile, the daily work routine in my room at the university, whose distance to the CWI made me a physically rather isolated member of the CWI database research group, were enlivened by the companionship of my room mate Arnold Jonk. I specifically recall his (black) humor and our always continuing conversation on a broad range of topics.

An important event in my professional career was the foundation of Data Distilleries as a data mining spin-off company by members of the CWI database research group, including Marcel Holsheimer and Fred Kwakkel. Data Distilleries, which later specialized into aCRM (analytical Customer Relationship Management), is special in the Dutch IT sector because it is a software product company, while almost all IT companies here are service-oriented and "just" customize software that generally comes straight out of the USA. Software product companies have an immense potential in terms of growth and profitability, but are much more difficult to make into a success than IT service companies due to the stiff competition in the software product market, which has a winner-takes-all characteristic.

Quickly, Data Distilleries started using the Monet prototype to power their data mining products, which is why they asked me to join in, initially for one day a week. As a consequence of working just four days at the university, my Ph.D. contract extended until June 1999. By that time, I had produced what I consider my best research results

through an extremely effective and pleasurable cooperation with Stefan Manegold in the area of cache-conscious query processing. This experience really showed me the power of teamwork as well as the possibilities of inter-disciplinary research.

Due to the frenetic development of Data Distilleries and the pressure this put on Monet as a software product instead of a research prototype, I decided to join Data Distilleries full-time when my OIO contract finished. Working at Data Distilleries was an eye-opener for me, as it was my first experience working in the "real world" and the circumstances were highly stimulating: a booming high-tech startup where the possibilities as well as the demands were endless. Here I learned to broaden my horizon from only one specialized subject to a complex interplay of many factors (among others facing the many challenges of going from finding a viable business proposition to overall system architecture into managing large-scale software development by cooperating teams of persons, with all its social aspects as well). But most of all, at Data Distilleries I had the privilege of working with an inspiring set of individuals including Tim Rühl, Donald Kwakkel, Hans Boëtius, Edith Kanters, Ronald Wanink, Maarten Smeets, Frank Bos, Marijn Deé and Martin van Dinther. They put up with many of my Monet bugs and the best of them are still fluent in MIL.

Currently I am back at the database research group at CWI, where Monet continues to be used and improved, among others by the tireless work of Niels Nes and enthusiasm of Arjen de Vries. Monet has always been a team effort which would have been impossible to achieve without all those who contributed, notably among them Menzo Windhouwer and Jonas Karlsson.

During my time at Data Distilleries, I was always intending to finish my thesis, but the endless list of urgent company issues would always prevent me from doing any work. In the end, after coming back at CWI, it took four months to complete. Finally getting there is in many senses thanks to my family and especially Cecilia, who has always stimulated me to finish this thesis and lovingly accepted my late working hours in doing so.

Contents

Chapter 1

Introduction

In the last decades, almost all sectors of society have come to depend strongly on information technology. The amount of data stored worldwide in information systems has continuously experienced exponential growth. The increased availability of both large data volumes, now often accessible by means of the internet, and commodity computer hardware that is every year cheaper yet more powerful, has created new ways in which information systems are used. The primary goal of early large-scale information systems was the preservation and lookup of data items (recording account balances, tax payments, addresses, etc..). Now that many organizations have created an information system infrastructure and have gathered huge amounts of data over time, information systems are starting to be used more and more for extracting insight from this accumulated data and – in the digital economy – a business advantage for the organization. Therefore, the purpose of many new applications of information systems involves complex analyses over large data volumes, rather than simply preservation and lookup.

1.1 The DBMS and its Applications

Most information systems in operation today are implemented by using a commercially available standard *Database Management System* (DBMS) product. The DBMS is an – often complex – piece of software that manages the data stored in an information system. It facilitates concurrent access by multiple users to a database, limits access to data to authorized users only, and recovers from system failures without loss of integrity [Ull89a, Ull89b]. The *relational* DBMS is the most widely used kind of DBMS, that organizes the database in regular tables that consist of rows and columns, and provides access to these tables through the high level query/data manipulation language SQL (Structured Query Language) [Cod70]. An extensive description of relational DBMS concepts is beyond the scope of this thesis, and we assume the reader to be familiar with them [Dat85, UW97].

1.1.1 On-Line Transaction Processing (OLTP)

When relational DBMSs became popular in the early 1980s, their main use was to support *on-line transaction processing* (OLTP). Typical queries in an OLTP system for e.g., running an insurance company are "show all characteristics recorded about

customer X", for example to provide the employee receiving a telephone call with basic customer data, or simple updates like: "insert a new record for a sale to customer X of a certain product Y". The mix between read-only and update queries varies among OLTP applications, but updates tend to form a significant fraction and can even dominate query processing.

OLTP applications involve little or no analysis and serve the use of an information system for data preservation and lookup.

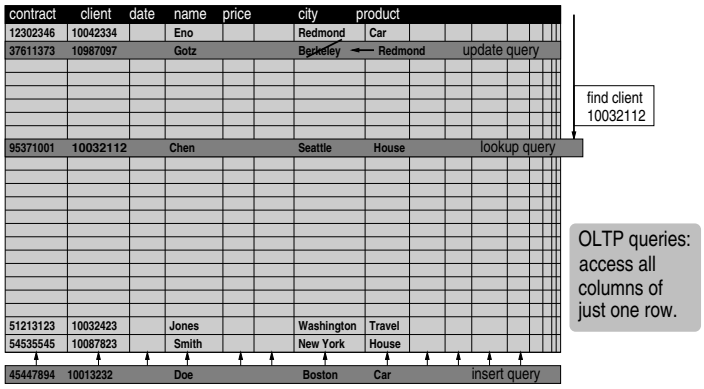

Figure 1.1: OLTP access pattern.

Figure 1.1 depicts three OLTP queries: an update, a lookup and an insert that each access just one row in the table (grey-shaded). These examples illustrate the access pattern of an OLTP query to a relational database table: the query accesses one or only a few rows, but uses most of its columns. An OLTP query typically is short-running and requires little resources from the DBMS. However, OLTP systems often have a heavy load of many (e.g., hundreds) queries per second, which can be challenging to sustain [GR91].

1.1.2 Query-Intensive Applications

In the 1990s, new DBMS applications became popular that differ from OLTP applications, as they involve complex analysis on large data volumes. Update queries are infrequent here and, if any, typically batch-oriented rather than on-line. We will refer to such applications as *query-intensive* DBMS applications. Examples of query-intensive DBMS applications are *on-line analytical processing* (OLAP) and *data mining*. Let us now discuss both application areas in more detail.

OLAP

OLAP tools summarize and group large amounts of data into small yet insightful results, typically using 2- or 3-D graphics to visualize answers. An example OLAP query is "give accumulated totals per city and per insurance product group of the insurance claims made last year after march 21". The result of this query could be a 3-D bar-chart with x-axis City, y-axis Product, and vertical bars denoting Claims in the z-axis. Answering this query usually requires the DBMS to go through the entire claims table to filter out all records after march 21, and compute for these

selected records a series of sums representing total claims, with a separate result for each occurring combination of city and insurance product.

Such 3-D visualization of summarized data is called a "data cube", and OLAP data is often thought of as multi-dimensional [Ken95]. One typical operation is "roll-up" of e.g., City along the location hierarchy: City-Region-Country-Continent, which results in increasingly summarized totals with less detail. The inverse operation of roll-up is called "drill-down". Other OLAP operations are "slice-and-dice" (i.e., selection combined with projection in the dimensions), and "pivot" (i.e., a re-orientation of the dimensions).

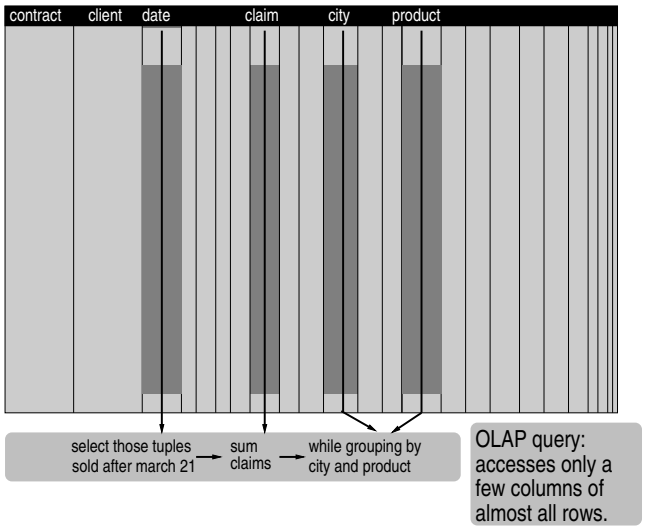

Figure 1.2: OLAP access pattern.

Figure 1.2 shows that the access pattern of OLAP is quite different from OLTP, as one single query from an OLAP tool often requires analysis of a substantial portion of the rows of the database table(s). Answering the query therefore requires the DBMS to do much more work, hence OLAP queries are medium to long-running, as data volumes tend to be large. The load on OLAP systems, however, tends to be limited in terms of number of queries per second in comparison to OLTP loads [CD97, Wid95].

The *knowledge discovery process* for which OLAP tools are used, is an interactive process. OLAP queries therefore should exhibit interactive response, rather than medium to long running times observed in today's DBMS technology.

Data Mining

Data mining is an even more demanding application area for a DBMS, as the query load generated by a data mining tool can be considered a kind of "repeated OLAP". In data mining, the objective is to locate sub-groups that have statistically significant differences from the mean and are interesting in some respect. An example query on a car insurance customer database could be: "what are profiles of dangerous drivers?". It is left to the data mining tool to determine what are the characteristics of those dangerous customer groups. This is typically done using a combination of statistical measures and automated search techniques from the field of artificial intelligence

(AI). The answers found by data mining can be unexpected, i.e., a user never would have found them using an OLAP tool manually, as OLAP tools just give answers to specifically asked questions. A data mining tool finds these unexpected and interesting groups by building a hypothesis (e.g., identifying "male drivers" as a sub-group in the database whose members have a higher probability of causing traffic accidents) and step-by-step improving it (e.g., "male drivers in company-leased cars"). This *search for the best hypothesis* can be compared to the search process a computer chess program performs while looking for the best chess move. In data mining, however, judging each "move" (i.e., hypothesis describing a sub-group) on its merits, corresponds to executing an OLAP-like DBMS query that checks the statistical validity of the hypothesis by looking in the historical data. In our example case, such a query would determine the percentage of accidents among male drivers with respect to the form of car-ownership. Answering one data mining query implies finding a good model and this typically requires testing many hypotheses, hence can amount to hundreds or thousands of OLAP-like queries, all of which together are supposed to finish in interactive time. Therefore, data mining is a truly query-intensive application area.

We should note here, that we consider that data mining tools should use a DBMS to execute queries on-the-fly. The extreme intensity of query loads has until now scared away many data mining tools vendors for implementing such solutions [Cha98]. DBMS use of such tools is limited to importing a (small) DBMS table into the tool, which performs the data mining analysis, while all data is memory resident in a data structure specific to the data mining algorithm chosen. We reject this approach as it goes against the main advantages of using a DBMS, which is providing efficient management of shared data and data independence. In particular, the before mentioned ad-hoc approach of processing data outside the DBMS in algorithm-specific in-memory structures is both limited in scalability (all data should fit in memory) and hard to extend with new algorithms, as each new algorithm requires specific data structures for handling mass data. In our view, one should use DBMS technology to support the *information need* [Cha98] of the data mining tool to provide both scalability for mining huge data volumes, and re-use of the same DBMS mass-data manipulation infrastructure for all data mining algorithms. Therefore, when we discuss data mining as an DBMS application area, we refer to solutions that perform statistical validation of hypothesis inside using DBMS queries.

1.2 Thesis Outline

We use the term *DBMS architecture* with the meaning of the design of a DBMS from the software engineering standpoint. This thesis is about DBMS architecture, where we investigate the question:

- *how to design DBMS software that is capable of supporting query-intensive applications with high performance?*

This thesis is structured as follows. We start in Chapter 2 with a short historic overview of DBMS architecture. Here, we motivate our choice to focus on the issue of performance, by showing why achieving good performance on query-intensive applications is a major problem for current DBMS products. In Chapter 3 we formulate specific research goals, and describe a number of ideas that we decided to try out in our *Monet* research DBMS. The first papers published on Monet were:

- P. Boncz and M. Kersten. Monet: an impressionist sketch of an Advanced Database System. *Proc. Basque International Workshop on Information Technology*, San Sebastian, Spain, July 1995.
- P. Boncz and F. Kwakkel and M. Kersten. High Performance Support for OO Traversals in Monet. *Proc. British National Conferences on Databases*, Edinburgh, Scotland, July 1996.

We then describe the Monet system in detail: Chapter 4 introduces the algebraic MIL language, which is the query language for the Monet system, designed to support query-intensive loads in extended database application areas. Chapter 5 describes how this language was implemented in Monet to provide high performance. Both chapters are based on:

- P. Boncz and M. Kersten. MIL Primitives For Querying a Fragmented World. *The VLDB Journal*, 8(2), November 1999.
- P. Boncz and W. Quak and M. Kersten. Monet and its Geographical Extensions: a Novel Approach to High-Performance GIS Processing, *Proc. EDBT Conference*, Avignon, France, June 1996.

In Chapter 6, we focus on the question how the relational join operator can best be implemented in Monet, and formulate hardware-conscious join algorithms, where we provide detailed insight[1] into the reasons why Monet is successful in exploiting the power of modern hardware, through cost modeling and experimentation. This research has been published in:

- P. Boncz and S. Manegold and M. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. *Proc. VLDB Conference*, Edinburgh, Scotland, July 1999.
- S. Manegold and P. Boncz and M. Kersten. Optimizing Database Architecture for the New Bottleneck: Memory Access. *The VLDB Journal*, 9(1), August 2000.[2]
- S. Manegold and P. Boncz and M. Kersten. What happens during a Join? Dissecting CPU and Memory Optimization Effects. *Proc. VLDB Conference*, Cairo, Egypt, July 2000.

[1] Joint research with Stefan Manegold, certain parts will overlap in his Ph.D. thesis.
[2] Re-published extended version of the VLDB 1999 Conference paper, as Best-of-VLDB 1999 paper.

- S. Manegold and P. Boncz and M. Kersten. Optimizing Join On Modern Hardware. *IEEE TKDE*, 14(4), July 2002.

In Chapter 7, we show how design and implementation of MIL in Monet provide the required back-end functionality to construct a full-fledged and well-performing SQL-speaking RDBMS front-end.[3] While not directly based on them, this chapter is related to other papers published on using Monet to support an object-oriented query language and a data mining front-end.

- P. Boncz and A. Wilschut and M. Kersten. Flattening an Object Algebra to Provide Performance. *Proc. ICDE Conference*, Orlando, FL, February 1998.
- P. Boncz and T. Rühl and F. Kwakkel. The Drill Down Benchmark. *Proc. VLDB Conference*, New York, NY, June 1998.

The thesis is concluded in Chapter 8 (Conclusion), which contains a retrospective on Monet and recommendations for future work.

[3]This SQL front-end will be part of the upcoming open-source release of Monet.

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 adapts 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.

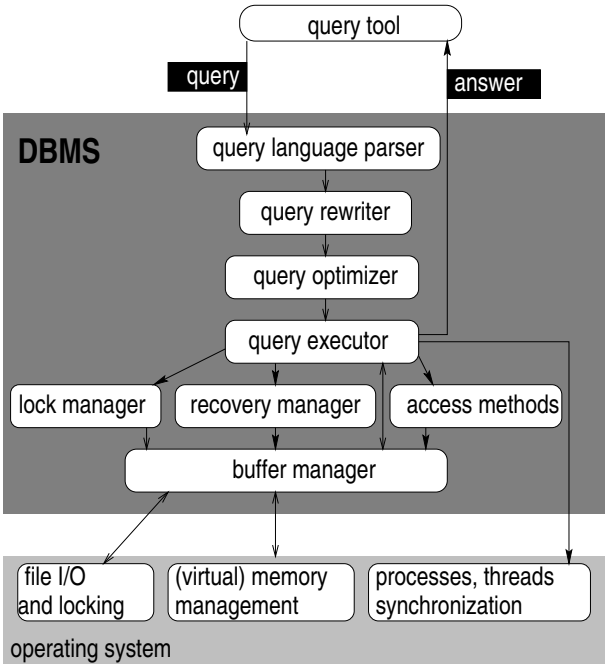

Figure 2.1: Relational DBMS Architecture

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].

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.

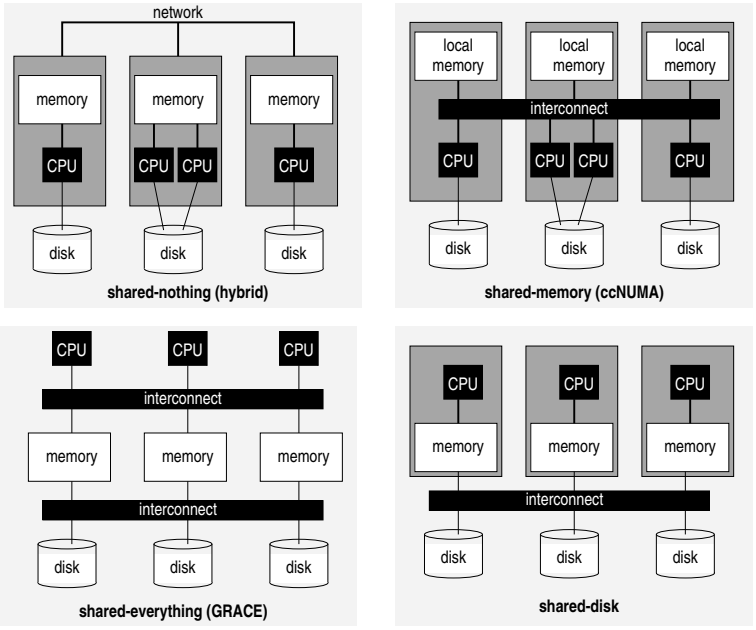

Figure 2.2: Parallel Hardware Architectures

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

flow-control.

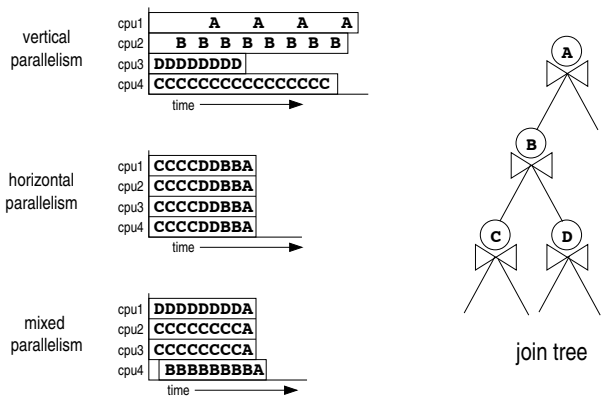

Figure 2.3: Parallel Query Execution Strategies

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.

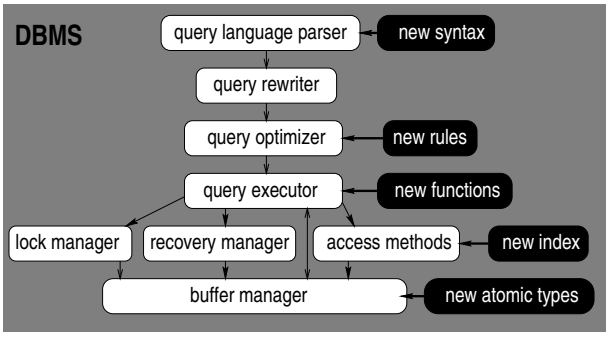

Figure 2.4: Opening up relational DBMS architecture for extension module

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

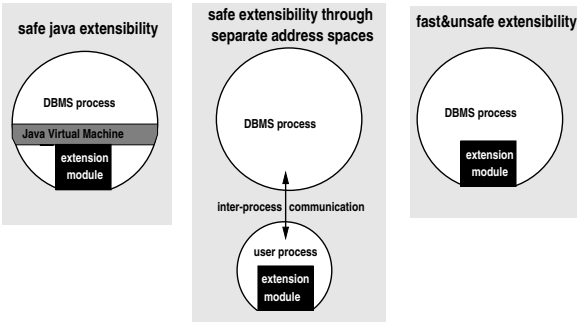

Figure 2.5: Various ways to run extension modules in a DBMS

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 `sharpen(Image)`) can be fully pipelined and optimized. As an example, `clip(sharpen(image))` can be rewritten into the faster `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 [CDF+98, 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 thaught 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], ObjectStore [LLOW91], and Versant [WO98], of which the latter four are still available as commercial products. When these systems were designed, their language bindings and

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 [AvdBF+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 multidimensional 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 analists, 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.

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, dVWAK00, BdVNK01, dVMNK02], and geographic querying both using topological and geometrical models [BK95, BQK96, WvZF+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.

year	computer model	processor			memory		
		type	MHz	#par. units	STREAM/copy (bandwidth)	typical size (MB)	latency (ns)
1989	Sun 3/60	68020	20	1	6.5	16	
1990	Sun 3/80	68030	20	1	4.9	16	
1991	Sun 4/280	Sparc	17	1	9.6	16	160
1992	Sun ss10/31	superSparc I	33	3	42.9	32	
1993	Sun ss10/41	superSparc I	40	3	48.0	32	
1994	Sun ss20/71	superSparc II	75	3	62.5	64	870
1995	Sun Ultra1 170	ultraSparc I	167	5	225.2	128	225
1996	Sun Ultra2 2200	ultraSparc II	200	5	228.5	256	225
1996	SGI Power Chall.	R10000	195	5	172.7	128	610
1997	SGI Origin 2000	R10000	250	5	332.0	256	424
1998	SGI Origin 2000	R12000	300	5	336.0	256	404
1992	Intel PC	80486	66	1	33.3	8	
1993	Intel PC	Pentium	60	2	47.1	8	161
1994	Intel PC	Pentium	90	2	46.4	8	161
1995	Intel PC	Pentium	100	2	85.1	8	161
1996	Intel PC	Pentium	133	2	84.4	16	161
1996	Intel PC	PentiumPro	200	5	140.0	16	203
1997	Intel PC	PentiumII	300	5	188.2	32	145
1998	Intel PC	PentiumII	350	5	279.3	32	145
1998	Intel PC	PentiumII	400	5	304.0	32	145
1999	Intel PC	PentiumIII	600	5	379.2	64	135
2000	Intel PC	PentiumIII	733	5	441.9	128	135
1999	AMD PC	Athlon	500	9	373.5	64	217
2000	AMD PC	Athlon	800	9	387.9	128	217

year	hard disk model		size (MB)	latency (ms)	rotations /minute	bandwidth (MB/s)	cache (MB)
1979	Tandon	TM–602	5	75	3600	0.62	0
1981	Tandon	TM–252	10	85	3600	0.62	0
1983	Seagate	ST–225	20	65	3600	0.62	0
1984	CMS	LT LD20	21.4	75	2640	0.9	0
1984	Mitsubishi	MR522	25.5	85	3536	0.9	0
1985	Quantum	Q540	40	45	3550	0.9	0
1985	Seagate	ST–251	42	38	3600	0.9	0
1986	CDC	94205–5	43	28	3597	0.9	0
1986	Microscienc	HH–1050	43	28	3436	0.9	0
1987	Rodime	RO3055	45	36.3	3600	0.9	0
1988	Rodime	RO5090	74.6	36.3	3600	0.9	0
1988	Micropolis	1325	67.1	28	3600	0.9	0
1991	CDC	94221–190	170	19	3597	1.1	0
1992	CDC	94171–344	344	19	3597	1.5	0
1993	Tandy	250–4168	540	12	4498	1.5	0
1996	Quantum	Empire	4500	9.5	5400	3.6	0
1998	Quantum	Fireball	6400	10	5400	14	0.5
1998	Quantum	Atlas II	9100	8	7200	14	2
2000	Quantum	Atlas10K II	18400	5.0	10000	24	8

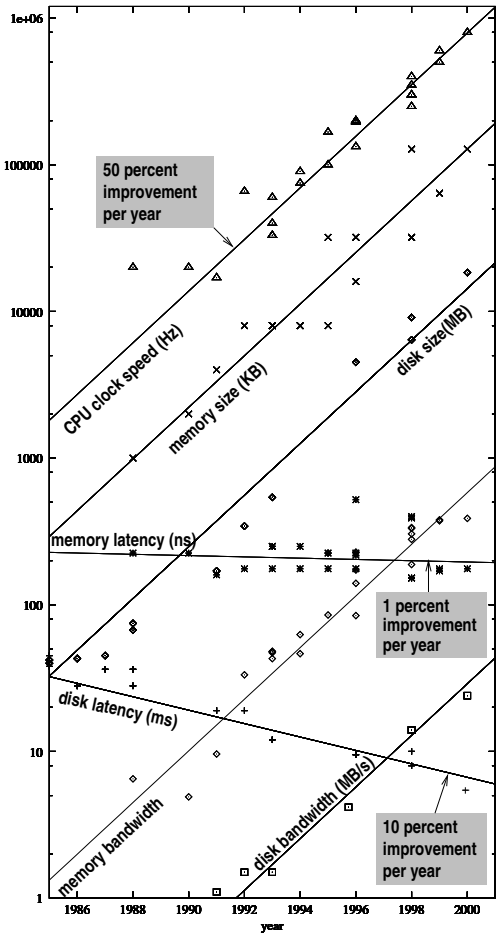

Figure 3.1: Hardware Trends in the Last Two Decades.

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.[1] 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, BBG+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.

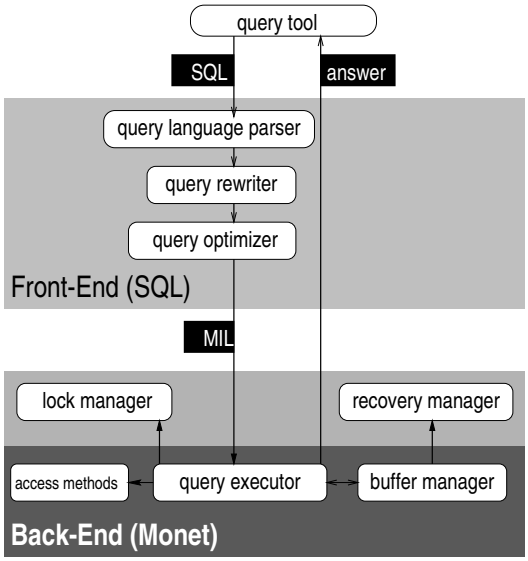

Figure 3.2: Front-end/Back-end architecture.

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 uopn 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.

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.

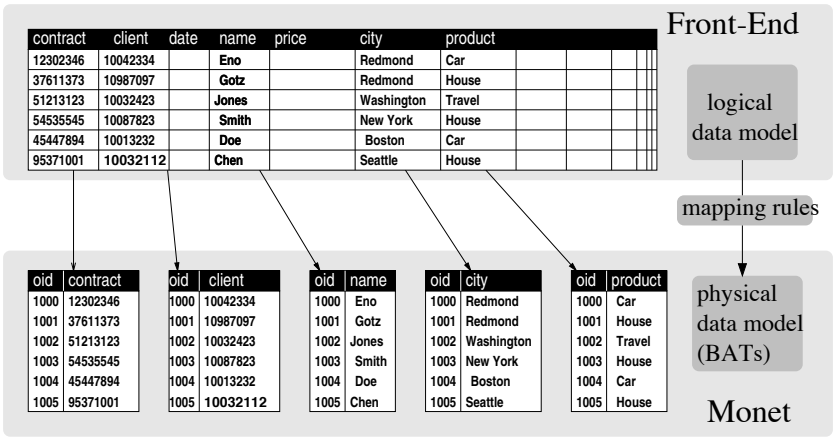

Figure 3.3: Decomposition of the Relational Model onto BATs.

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.

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 [WvZF+98] 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 [RAA+88] and Mach [ABG+86], 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.

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.

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

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.

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, TLPZT97, 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 data 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.

Chapter 4

The MIL Language

Monet was designed to work in a front-end/back-end system architecture, in order to reach its design goals of extensibility and support for multiple logical data models. It can be seen as the back-end that provides a kernel of DBMS facilities to multiple front-ends (Figure 4.1). A relational front-end maps SQL queries onto Monet requests. An ODMG front-end does the same for OQL and object access. Other front-ends, targeted to specific applications such as data-mining, may co-exist.

Front-end systems built on top of Monet, such as a relational or an object-oriented DBMS, communicate with the Monet back-end through the MIL language. This does not mean to say that MIL itself is an object-oriented or even a relational language; MIL just provides the minimal, but complete set of primitives, such that each front-end can adequately map operations on its logical model to the underlying Monet primitives. Adequately here means that our objectives for the design of Monet (performance and extensibility) should not be compromised.

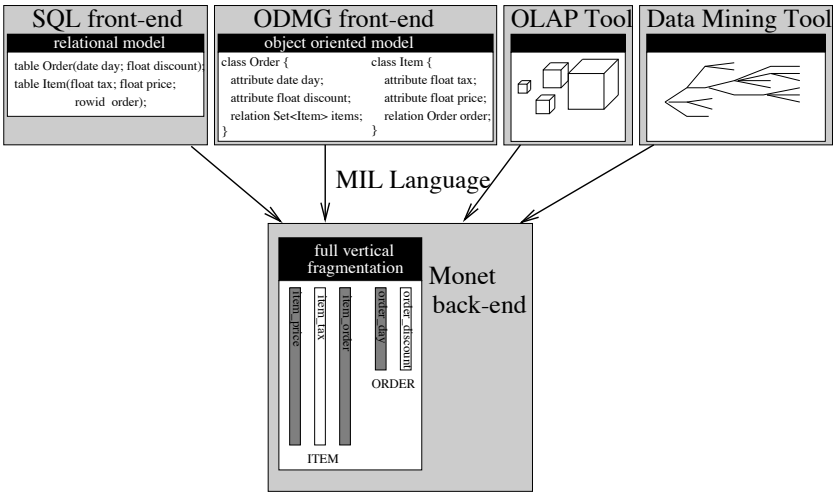

Figure 4.1: The front-end/back-end architecture.

Figure 4.2 shows the general structure of MIL. The language consists of a number of control structures (the black area) and extensibility features (the concentric areas), and has the following characteristics:

- it provides all DBMS services needed by the front-ends.
- it uses one simple bulk data type, the binary table.
- its table manipulation operations form a closed algebra on this binary table model.
- it provides constructs to express various kinds of parallelism.
- it is extensible with new primitives, data types, and associated search accelerator structures.
- it is a computationally complete procedural language.

In the design of MIL, we applied valuable lessons learned in previous work on database languages. The idea of using *query algebras* as intermediate languages for relational query execution dates back to [Cod70]. In the context of extensible relational systems, this idea was generalized to allow modular extensibility using ADT interfaces. The extensibility interface of MIL was inspired by Gral [Güt89], an early system that offered extensibility on all relevant levels (data types, algebra operators and search accelerators). The Fad [DV92] language is well-known for its consequent functional operator style. The expressive genericness of this language proved a hindrance to run Fad programs efficiently on bulk data and perform parallelization [FGN+95]. Its successor language, Flora, used as an intermediate language in the IDEA [LVZZ94] system, solved this problem by focusing on a simple kernel of bulk operators. This decision we also followed in MIL, as well as the decision in Flora to introduce explicit language primitives for specifying parallelism.

4.1 Data Model

The MIL data model consists of an extensible set of atomic values, and one collection type, the BAT (Binary Association Table). The formal definition of the set of all types $\mathcal{T}$ in MIL is:

1. $t \in \mathcal{A}_f \cup \mathcal{A}_v \Rightarrow t \in \mathcal{T}$
2. $T_1, T_2 \in \mathcal{T} \Rightarrow \texttt{bat}[T_1, T_2] \in \mathcal{T}$

The first rule defines atomic data types $\mathcal{A}$ (both of fixed and variable size), and the latter defines the BAT type. A BAT value is a multi-set that contains binary tuples, called Binary UNits (BUNs). The left column of a BAT is called the *head* column and the right is called the *tail* column. The `bat`$[T_1, T_2]$ type is parametrized by the types of its head and tail columns, and may be *nested*, as those types might again be BATs.

As a starting point, we have the collection of fixed-size atoms $\mathcal{A}_f = \{$`bit`, `chr`, `sht`, `int`, `lng`, `flt`, `dbl`, `oid`$\}$, respectively denoting boolean values, single characters, small, normal and long integers, normal and double floating point numbers, and object identifiers. The standard collection of variable-sized atoms $\mathcal{A}_v = \{$`str`$\}$ just contains the string type.

This initial set of atomic types is focused on supporting standard business applications, but the MIL data model can be extended with new atomic types. We have

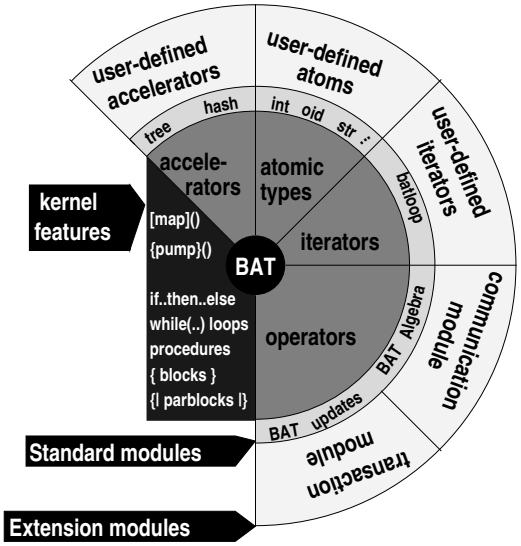

Figure 4.2: The structure of MIL

implemented many new atomic types and operations on them. These types encompass enrichments in the business area (like currency and temporal types) [BWK98], the GIS domain (points, polylines, polygons) [BQK96] and multi-media (images, video, audio) [NK98].

The MIL syntax for values of the standard atomic types follows that of the C/C++ programming languages. Values can be cast to another type with conversion functions `t(value)`, $t \in T$, that implicitly exist for each atomic type. Casting is necessary to distinguish longs from integers and doubles from floats (e.g., `lng(42)`, `dbl(3.14)`). The `bit` type has two values, denoted `true` and `false`.

Each type has one additional special value, called *nil*, that expresses the "don't know" value. We use the `nil` as a shorthand for the `oid(nil)`, as this value is often used. For nil values of other types, we use casts (e.g., `bit(nil)`, `int(nil)`, `dbl(nil)`).

4.1.1 Example Data Mapping

Suppose that we have a relational schema with tables `Order` and `Item` in which the attribute `order` identifies the order to which the item belongs:

```
table Order(int id; date day; float discount);
table Item(int order; float price; float tax);
```

A relational data model can be stored in Monet by splitting each relational table by column[CK85]. Each column becomes a BAT that holds the column values in its tail (right column). The head (left column) holds an object identifier (`oid`). We use the naming convention *table-name_column-name* for such BATs. The relational tuples can be reconstructed by taking all tail values of the column BATs with the same head `oid`.

This mapping scheme decomposes our ORDER table into `order_id`, `order_day` and `order_discount`, and the ITEM table into `item_order`, `item_price` and `item_tax`. Sometimes it may be possible to use one of the unique columns as the head column in the

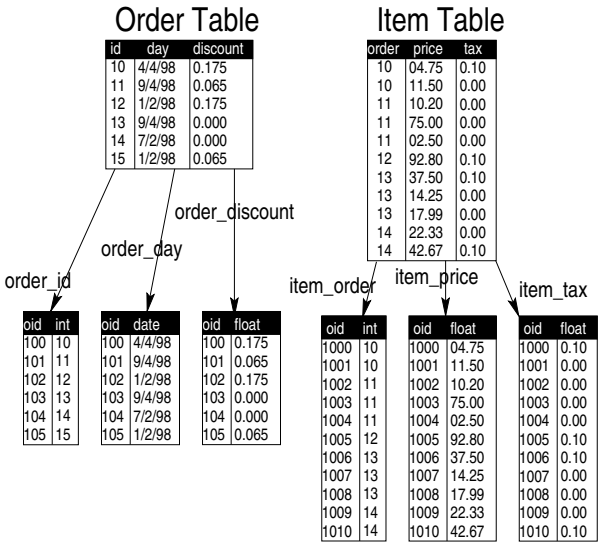

Figure 4.3: Mapping Relational Tables onto BATs.

BATs (like `id` for the `Order` table field), but if this is not possible, we use system-generated `oid` numbers; as depicted in Figure 4.3.

4.2 MIL Execution

MIL is a procedural block-structured language, with standard control structures like if-then-else, and while-loops. The BAT *iterator*, denoted `bat-expr@`*iterator*, provides another way of looping. This cursor-like construct visits elements (BUNs) from a BAT, and for each element executes a MIL statement. This statement can contain the special variables `$h` and `$t`, that represent the head- and tail- value of the current element, respectively. The most commonly used iterator in MIL is the `batloop`, that sequentially visits all elements of a BAT.

The basic unit of MIL execution is the *operator*. MIL operators receive a number of input values and produce a single output value. All operators can be called like `op(`*expr1*`,..,`*exprN*`)` but MIL also allows infix notation `(`*expr1* `op` *expr2*`)` for binary operators as well as object-oriented dot-notation *expr1*`.op(`*expr2*`,..,`*exprN*`)`.

Multiple operators with the same name, but with different *signatures* may exist (overloading). An operator signature consists of the operator name, followed by a comma-separated list of parameters between parentheses, a colon and then the return type. Each parameter definition consists of the parameter type and a parameter name. Operators may have a variable number of parameters. In the signature, such parameters are denoted $\cdots$*type-expr*$\cdots$.

Most operators are *polymorphic*, which means that their signature contains (free) type-variables, denoted in this document with capital single-letter italics.[1]

[1] In MIL we use the syntax `any::<`*tag*`>` for free type variables. When such free variables have the same tag number, this implies that the types of the actual parameters passed must match each other.

MIL is a dynamically typed language, so function resolution is a run-time task. The execution mode of operators is to simply first interpret all parameters and materialize their results. If an operator exists with a signature that matches these actual parameter, it is invoked (else a run-time error occurs).

MIL also accepts the C/C++ shorthand for assignment combined with operator execution(e.g., `i :+= 1` means `i := i+1`). In order to provide flexible execution of variable functions, a string name of an operator can be *dereferenced* to call it as in `(*str-expr)(expr1, ...,exprN)`.

MIL *extension modules* introduce new atomic data types, operators, iterators, and search accelerators (Figure 4.2). Search accelerators are data structures related to BAT columns, that are maintained under updates by the system. They do not introduce semantics, but are generally used to accelerate execution of certain operators (e.g., a hash table accelerates equi-selections and equi-joins). The core of the language is introduced by the standard module collection, which database extenders can augment with their own. Extension modules are implemented in C/C++. Alternatively, new operators can be defined run-time in MIL as scripted *procedures*:

```
proc operator-signature <MIL statement>
```

The statement is called the *body* of the procedure. When a procedure is executed, its body – in which the formal parameters from the signature have their scope – is interpreted. A value can be returned from a procedure body with the `return` keyword.

As procedures may be defined to receive a variable number of arguments, the MIL body can access those with the special construct `$(int-expr)`, that retrieves the i-th parameter. The actual number of parameters is obtained with `$0`. All arguments from the i-th until the last can be passed into another operator that with the `$(int-expr..)` construct.

As an example, let us define the `avg(`T`,..`T`..):`T. operator that averages its arguments, using a procedure:

```
proc avg(any::1 v, ..any::1..) : any::1 {
        var i := 1;
        while(i <= $0)
                v :+= $(i :+= 1);
        return v / $0;
}
```

Though procedures are interpreted instead of executed as a C/C++ function – like normal MIL operators – this distinction is hidden on the language level, and procedures can be used like any other operator.

This extensibility mechanism is comparable to extension mechanisms used in relational systems like [SAH87, OHUS96] and differs from [Ora97a] in its choice to run extension code directly in the DBMS process space, as performance is a primary concern in Monet.

4.2.1 Atomic Value Operators

The minimal set of operators $=, \neq, <, >, \leq, \geq,$ `hash`() is present on all atomic types.[2] Each atomic type brings with it an additional interface of specific operations:

[2]The `hash(any):int` returns an `int` 32-bit hash-number.

numerical types `sht,int,flt,dbl` and `lng` have arithmetic operators (+, −, ∗, /), as well as the `between(`*value, low, high*`):bit` that checks whether $low \leq value \leq high$.

floating point `flt` and `dbl` have math operators `cos, sin, tan`, etc..

strings have a series of (sub)string and matching operations.

object identifiers the `newoid(int size):oid` operator requests a system-wide unique range of fresh `oid`-s. The function returns the start value of this consecutive range.

booleans the `bit` type `and`, `or`, `not` operators defined on it.

Note that the convention for all MIL operators is to respect the "don't know" semantics of the `nil` value. For example, `int(nil)+2` and `int(nil)>10` evaluate to `int(nil)` and `bit(nil)`, respectively.

4.2.2 BAT Algebra

The focus of MIL execution is to enable efficient bulk operations on mass data stored in BATs. This core functionality is offered by a *BAT algebra* of MIL operators. These operators

- have an algebraic definition, as provided below.
- are free of side-effects, which makes the algebra apt as a language for optimization with rewrite systems.
- form a closed algebra with BATs as the only collection-type.

We formally define the semantics of the BAT algebra operators using tables that show in their left column the *operator signature*, and in the right column a definition of its result. Just like in C++, if trailing parameters in a signature have a specified default value, when one omits these trailing parameters when an calling an operator, their default values are used. The notation of a BUN is $[a,b]$. We denote BATs as bags $\langle \cdots \rangle$, and – if we know that no double elements will occur – as sets: $\{\cdots\}$. In the case of bags, we use a pragmatic bag-union notation $\cup\langle x \rangle$:<*cartesian−product*>:<*filter−condition*> where the defined bag consists of those x corresponding to each element in the cartesian product (given as a – compound – for-all quantifier) for which the filter condition holds. Note that where $\cup$ and $\setminus$ are used in bag definitions, the loss-less bag-union resp. the bag-minus are meant. $|S|$ indicates the size of a collection. BATs can always be seen as a list of BUNs, that is, with some *BUN-ordering*. Only a few MIL algebra operators define such an ordering, so for most BATs we just do not know anything about their BUN-ordering. The notation of a BAT as a certain BUN-list $B = [[h_1,t_1],...,[h_{|B|},t_{|B|}]]$ we shorten as $B = \left[\stackrel{|B|}{i=1}[h_i,t_i]\right]$.

MIL operator signature	**Definition of the Result**
find(bat[*H*,*T*] *AB*, *H a*)→ *T*	*b* `if` ∃[*a*,*b*]∈*AB*,`else` *T*(`nil`)
select(bat[*H*,*T*] *AB*,*Tv*$_1$,*Tv*$_2$=*v*$_1$,	∪⟨[*a*,*b*]⟩: ∀[*a*,*b*] ∈ *AB* :
`bit` r_1 = `true`,r_2=r_1)	((`isnil`(v_1)∨v_1<*b*)∧(`isnil`(v_2)∨*b*<v_2)∧¬(`isnil`(v_1)∧`isnil`(v_2)))
→bat[*H*,*T*]	∨(r_1∧v_1=*b*)∨(r_2∧v_2=*b*)∨(`isnil`(v_1)∧r_1∧`isnil`(v_2)∧r_2∧`isnil`(*b*))
range and equi-selections	

The `select` supports the selection predicates `isnil`(t,v), $t \leq v, t < v, t = v, t > v, t \geq v$ as well as $v_1 \leq t \leq v_2, v_1 < t \leq v_2, v_1 \leq t < v_2$, and $v_1 < t < v_2$:

```
select(b,nil);              # select on isnil(tail(b))
# other selects do not select the nils
select(b,42);               # select on tail(b) = 42
select(b,10,20);            # select on 10 <= tail(b) <= 42
select(b,10,20,false,true); # select on 10 < tail(b) <= 42
select(b,10,20,true,false); # select on 10 <= tail(b) < 42
select(b,10,20,false,false);# select on 10 < tail(b) < 42
select(b,10,nil);           # select on 10 <= tail(b)
select(b,10,nil,false);     # select on 10 < tail(b)
select(b,nil,20);           # select on tail(b) <= 20
select(b,nil,20,false);     # select on tail(b) < 20
```

The BAT algebra is closed on the BAT type, so the result of the join is again a binary table. This is achieved by projecting out the join columns; the result consists of the outer columns of the left and right BAT where their inner columns matched.

MIL operator signature	**Definition of the Result**
join(bat[T_0,T_1] AB, bat[T_1,T_2] CD)→bat[T_0,T_2]	$\cup\langle$ [a,d]$\rangle$: $\forall$[a,b]$\in AB$,$\forall$[c,d]$\in CD$: $b=c$
outerjoin(bat[T_0,T_1]AB, bat[T_1,T_2] CD)→bat[T_0,T_2]	join(AB,$CD \cup$ {[b,`nil`]|[a,b]$\in AB$: $\nexists$[b,c]$\in CD$})
relational equi-join	

Figure 4.4 provides a flavor of MIL execution in the example from Section 4.1.1. The depicted sequence of MIL statements retrieves all item-IDs from orders with a certain discount, and the tax paid over them. As Monet never materializes N-ary tables, the result of our query is a relational table that is again decomposed, in the BATs `UNQ_ITM` and `UNQ_TOT`. We use the standard `printf(str format, ...)` printing operator for printing results to standard output. This operator is part of the (standard) `io` extension module.

SQL example query

```
SELECT item.id AS id,
       item.price*item.tax AS total
WHERE order.id = item.order AND
       order.discount BETWEEN 0.00 AND 0.06
```

MIL translation (*annotated below*)

```
ORD_DSC := select(order_discount,0.0, 0.06)
                                        a bat[oid,flt]
IDS_NIL := join(order_id.reverse,ORD_DSC)
   creates a bat[oid,oid] with selected order-IDs in head, nil tail
ITM_NIL := join(item_order,IDS_NIL)
    creates a bat[oid,oid] with selected item-IDs in head, nil tail
UNQ_ITM := mark(ITM_NIL).reverse
creates a bat[oid,oid] fresh oids in head, selected item-IDs in tail
UNQ_PRI := join(UNQ_ITM,item_price)
    creates a bat[oid,flt] with selected item-IDs and their prices
UNQ_TAX := join(UNQ_ITM,item_tax)
     creates a bat[oid,flt] with selected item-IDs and their taxes
UNQ_TOT := [*](UNQ_PRI,UNQ_TAX)
         creates a bat[oid,flt] with selected item-IDs and totals
[printf]("%d %s\n", UNQ_ITM, UNQ_TOT)
```

Figure 4.4: A simple SQL query and a MIL translation.

Much like in the definition of the `select`, which selects on tail, the operators in the BAT algebra have fixed semantics on which columns of their BAT parameters they work, and from which columns result values are derived. If an operator needs to work on the opposite column of some BAT, MIL allows to view each BAT with head and tail

column swapped. This *reverse view* on a BAT is delivered by the `reverse` operator. The `mirror` allows to view a BAT as if it had its head column superimposed on the tail column, yielding a BAT with two identical columns. The rationale to have these column-swapping operators is that it allows the other BAT algebra operators to make fixed assumptions about their parameter formats – thus reducing the degree of freedom in the operator signatures and simplifying their implementation and optimization.

MIL operator signature	**Definition of the Result**
reverse(bat[H,T] AB)→bat[T,H]	$\left[_{i=1}^{N}[b_i,a_i]\right]$, $AB = \left[_{i=1}^{N}[a_i,b_i]\right]$
mirror(bat[H,H] AB)→bat[H,H]	$\left[_{i=1}^{N}[a_i,a_i]\right]$, $AB = \left[_{i=1}^{N}[a_i,b_i]\right]$
mark(bat[H,T] AB, `oid` $o=0$)→bat[H,oid]	$\left[_{i=1}^{N}[a_i,o+i-1]\right]$, $AB = \left[_{i=1}^{N}[a_i,b_i]\right]$
project(bat[H,T_1] AB, $T_2 c$)→bat[H, T_2]	$\left[_{i=1}^{N}[a_i,c]\right]$, $AB = \left[_{i=1}^{N}[a_i,b_i]\right]$
slice(bat[H,T] AB, `int` lo,hi)→:bat[H, T]	$\left[_{i=1+max(0,lo)}^{min(N,lo+hi)}[a_i,b_i]\right]$, $AB = \left[_{i=1}^{N}[a_i,b_i]\right]$
order(bat[H,T] AB)→bat[H,T]	$\left[_{k=1}^{N}[h_k,t_k]\right] = AB \quad \wedge \forall 1 \le k < N : t_k \le_n t_{k+1}$
order(bat[H,T_1] AB, bat[H,T_2] AC) →bat[H,oid]	$\left[_{k=1}^{N}[h_k,t_k]\right] = \cup\langle[a,\mathtt{id}([b,d])]\rangle : \forall[a,b]\in AB, \forall[c,d]\in AC : a=c$ $\wedge \forall 1 \le k < N : t_k \le_n t_{k+1}$
MIL operators with list-semantics, using the below definitions:	
`id`*(x) respects* $=_n, \le_n$	$\mathtt{id}([a,b]) = \mathtt{id}([c,d]) \Leftrightarrow [a,b] =_n [c,d]$, $\mathtt{id}([a,b]) \le \mathtt{id}([c,d]) \Leftrightarrow [a,b] \le_n [c,d]$
lexicographic $=_n, \le_n$	$[a,b] =_n [c,d] \Leftrightarrow a =_n c \wedge b =_n d$, $[a,b] \le_n [c,d] \Leftrightarrow (a =_n c \wedge b \le_n d) \vee \mathtt{!}(c \le_n a)$
$=_n, \le_n$ `nil`-*proof*	$a =_n b \Leftrightarrow (a =_n b) \vee (\mathtt{isnil}(a) \wedge \mathtt{isnil}(b))$, $a \le_n b \Leftrightarrow (a \le_n b) \vee \mathtt{isnil}(a)$

The `mark` also introduces a new tail column, but fills it with an ascending range of `oid`-s that starts with the second parameter value. Note that `nil+`x`=nil`, so passing the `nil` value as second parameter will yield a BAT with `nil` in the entire tail column. This same effect can be reached for any constant value with the `project`. The `mark` is often used for introducing a new column of unique `oid`-s. As we saw in Section 4.1.1, such unique columns are used in MIL to couple BATs that represent a decomposed table. This not only goes for persistent tables, but also for intermediate results of query processing, as those can be viewed as temporary tables. In line 4 of Figure 4.4, the result of the join from line 2 gets a new unique column using the `mark`. This unique column is present in all temporaries of Figure 4.4 whose name starts with `UNQ`.

The `slice` returns a positionally selected subset of its parameter BAT. If the BUN-ordering of this BAT is unknown, the result can be considered a random selection without putting back. The unary `order` returns a bag-wise identical BAT as its only parameter, but guarantees that the tail column is ordered. The binary `order` is much similar to the binary `group` discussed later, but guarantees that the generated `oid`-s tail column is ordered and respects the lexicographical $\le_n$ ordering on tail value combinations (See Section 7.3.5 for a detailed discussion how this operator facilitates multi-column ORDER BY).

By careful design of the BAT data structures (see Section 5.3.2), the `reverse`, `mirror`, `slice` and `mark` actually do not have to materialize their results, which makes them zero-cost operators.

MIL operator signature	**Definition of the Result**
count(bat[H,T] AB)→`int`	$\lvert AB\rvert$
sum(bat[H,T_0] AB)→ T_1	$\sum_{[a,b]\in AB} b$
max(bat[H,T] AB)→ T	$b : [a,b] \in AB \wedge \nexists y > b, [x,y] \in AB$
min(bat[H,T] AB)→ T	$b : [a,b] \in AB \wedge \nexists y < b, [x,y] \in AB$
aggregates; on boundary condition $AB = \emptyset$: count,sum return zero, others return t(`nil`)	

The above collection of *aggregates* is by no means complete. Extension modules with new ones can be introduced easily in MIL.

MIL operator signature	Definition of the Result
unique(bat[H,T] AB)→bat[H,T]	{ [a,b] | [a,b]∈AB }
diff(bat[H,T] AB, bat[H,T] CD)→bat[H,T]	{ [c,d] | [c,d]∈CD∧ ∄[c,d]∈AB }
union(bat[H,T] AB, bat[H,T] CD)→bat[H,T]	{ [a,b] | [a,b]∈AB∨[a,b]∈CD }
intersect(bat[H,T] AB, bat[H,T] CD)→bat[H,T]	{ [a,b] | [a,b]∈AB∧[a,b]∈CD }

The classical operations on sets, formed by the BUNs of a BAT, are displayed above. If only one column of the parameter BATs is of interest, one should first make the other column constant (with `mark(nil)`) or equal (with `mirror`).

MIL operator signature	Definition of the Result
group(bat[oid,T] AB)→bat[oid,oid]	∪⟨[a,`id`(b)]⟩ : ∀[a,b]∈AB : `true`
group(bat[oid,oid] AB, bat[oid,T] DC) →bat[oid,oid]	∪⟨[a,`id`([b,c])]⟩ : ∀[a,b]∈AB,∀[d,c]∈DC : a=d
group encoding into head values: `id`(x)=`id`(y) ⇒ x=$_n$$y$	

Relational `group by`, or object-oriented `nest/unnest`, require specific support on the flat binary algebra. Such groupings may involve multiple attributes. In MIL, groupings are materialized in a *grouping* BAT that holds in the head column identifiers of all objects of interest, and in the tail a unique group identifier. The `group` operators construct such grouping-BATs.

The unary `group` operator is executed on a first `bat[oid,any]` and creates a new equivalence group for each different value from the tail column. The result is formed by a BAT with the same head column as the input, with a group-id in the tail column for each BUN. These group-id's from a dense collection of `oid`s [0,1,..,N-1], with N the number of distinct tail values in the original BAT. Such groupings can be refined using the binary `group` operator that subdivides the groups into new equivalence subgroups, taking into account an additional `bat[oid,any]`.

MIL operator signature	Definition of the Result
fragment(bat[H,T]AB,bat[H,H]CD)→bat[H,bat[H,T]]	{[h,select(AB,l,h)]|[l,h]∈CD}
split(bat[H,T] AB,int $nranges$)→bat[H,H]	{[l,h]|l ≤ h ∧ ∃[l,x],[h,y]∈AB}
S=split(AB,n) creates range-partition⇒∀[a,b]∈AB:∃$_{unique}$[l,h]∈S : l≤a≤h	

The MIL data model supports nested BATs, as produced by the *fragmentation* operator `fragment`. This operator performs a range-fragmentation of a BAT according to the head column. The range-BAT containing the split boundaries – that is passed as a second parameter – can be produced with the `split` operator. The `nranges` is only a target; the actual number of ranges depends on the distribution of the values in the head column of AB.

4.2.3 Operator Constructors

The $\{f\}$ and $[f]$ are special MIL syntax constructs that implicitly define a new operator for each already defined operator f.

MIL operator signature	Definition of the Result
[f](bat[H,T1] AB_1,··,bat[H,T_n] AB_n)→bat[H,T_r]	∪⟨[a_1, f(b_1,···,b_n)]⟩: ∀[a_1,b_1]∈AB_1,···,∀[a_n,b_n]∈AB_n :a_1=…=a_n
multi-join map of operator $f(T_1,\cdots,T_n) \rightarrow T_r \in$ {=,≠,<,>,≤,≥,+,−,*,/,`and`,`or`,`not`,···}	

The *multi-join map* constructs an operator that does an implicit equi-join on the head columns of multiple BATs and executes the operator that was passed between the square brackets on the result of this join (all matching combinations of tail values). The result of the multi-join map is again a BAT, that contains the head value for each match and in the tail the result of the corresponding operator execution.

The multi-join map of the `*(flt,flt)→flt` operator was demonstrated in line 6 of Figure 4.4. It produces a new BAT with for all selected items the multiplied price and tax.

Though not shown in the definition, we can also type the `const` keyword in front of an actual parameter and pass any kind of value (not necessarily a BAT) into the map operator. In this case, that parameter is not taken into the multi-join, and this value is passed as a constant into all operator executions. For example, `[*](item_tax, const 0.07)` multiplies all prices by .07. Typing `const` is actually not necessary for non-BAT values; e.g., `[*](item_tax, 0.07)` will do as well. In order to keep the multi-join map meaningful, at least one parameter should still be a (non-`const`) BAT.

MIL operator signature	**Definition of the Result**
$\{f\}(\text{bat}[T_0,T_1]\ AB, \text{bat}[T_2,T_1]\ CB, \text{bat}[T_2,T_3]\ CD,$ $\text{bat}[T_2,\text{bit}]\ CE{=}CB.\text{project(true).unique)}{\rightarrow}\text{bat}[T_0,T_4]$	$\cup\langle[a,f(S_b)]\rangle : \forall[a,b]{\in}AB: \texttt{true},\ \ S_b = \cup\langle[d,d]\rangle :$ $\forall[c_1,d]{\in}CD, \forall[c_2,b]{\in}CB, \forall[c_3,\text{true}]{\in}CE : c_1{=}c_2{=}c_3$
BAT-pump with operator $f(\text{bat}[T_3,T_3]) \rightarrow T_4 \in \{$ count,min,max,sum,$\cdots\}$	

The *pump* `{f}` constructs a new operator that works on a set of bags, where each bag is represented by a BAT. On each such BAT, the operator that was between accolades, is executed. Its first parameter is called the "extent" as a result BUN is produced for all elements in this bag: for each BUN, the head value is placed in the head of the result, together with the result of `f()` in the tail, computed as explained in the following. The second parameter is called the "grouping" and associates each group (i.e. each tail value of the extent) with a possibly empty set of identifying values (typically, this is a unique set of `oid`-s). By joining this set of identifying values[3] with the head of the third parameter (referred to as "attribute"), we arrive at a bag of attribute-values. For this bag, a BAT constructed that contains all values in both head and tail, and on this BAT, the operator `f()` is evaluated.

The fourth, optional, parameter of the complex pump imposes a selection on all attribute-values. This variant was introduced for computing *data cubes* in data mining MIL queries, such as those generated by the tools of Data Distilleries [BRK98]. Such query loads work with a large number of different selections at the same time (that correspond to the data mining hypotheses under statistical validation), such that it becomes interesting to re-use the full BATs storing the grouping and the attributes for all tuples instead of producing intermediate subsets of these BATs for each selection:

```
sub_oid := oid_sel.select(true).mark.reverse;
sub_val := sub_oid.join(oid_val);
sub_grp := sub_oid.join(oid_grp);
```

[3] A past version of the pump `{f}(`BD`,` BX`)` had two parameters: a BD attribute-group BAT that contained attribute values in the head column, and group-values in tail. The extent BX was the second parameter of which only the head column was used, both as group-value and as head column for the result. The pump was changed to the current definition as we found that in many query processing situations, the group- and attribute-values are already placed in distinct BATs, and joining them together is not really necessary as this join can be handled inside the pump by a trivial scan over both (e.g. the case of equal, unique and ordered head columns), and this past pump can without loss of performance or expressiveness be rewritten to `{f}(`BX`.mirror,` BD`.reverse.mark.reverse,` BD`.mark.reverse)`.

```
res_agr := {f}(sub_val, sub_grp, res_grp);
```

⇒

```
res_agr := {f}(res_grp, oid_grp, oid_val, oid_sel);
```

This again supposes that it is trivial for the pump to construct the equi-join between selection-, attribute- and group-BATs. As we will see in Section 5.3.5, the Monet implementations of pump as well as the implementation of the multi-join map is based on this assumption, and uses pre-processing with the MIL equi-join for handling the more complex kinds of joins.

4.2.4 BAT Updates

The BAT update operators modify their BAT operands and are therefore separated from the BAT algebra.

MIL operator type signature	**informal semantics**
bat(str *h*, str *t*, int size=100) → bat[*H*,*T*] *AB*	*AB* := AB_{prev} := ∅
info(bat[*H*,*T*] *AB*)→bat[str,str]	*return [property,value] information on this bat*
copy(bat[*H*,*T*] *AB*, int mode=CP_NORMAL)→bat[*H*,*T*]	*CD* := CD_{prev} := *independent copy of AB*
mode ∈ { CP_NORMAL, CP_ENUMERATE }	

A newly created BAT is an empty bag. The `info` operator produces a meta-BAT that contains various properties (see Section 5.7) and statistics on a BAT. The reason why the `copy` operator is in the update interface is that copying has no meaning in an algebra. The copy produced is an identical set of BUNs, but may use an enumerated representation (`CP_ENUMERATE`, see Section 5.2.1).

MIL operator type signature	**informal semantics**
create(bat[*H*,*T*] *AB*, str *acc*, ..X_i..)	*create search accelerator on head of AB*
destroy(bat[*H*,*T*] *AB*, str *acc*)	*destroy search accelerator from head of AB*
standard accelerators *acc* ∈ { hash,tree }	

Search accelerators are part of Monet's extensibility interface. MIL comes standard with the bucket-chained `hash` table structure and the T-`tree`, both successful main-memory data structures for accelerating value lookup [LC86a].

MIL operator type signature	**informal semantics**
modes ∈ { BAT_READ, BAT_APPEND, BAT_UPDATE, BAT_WRITE}	
insert(bat[*H*,*T*] *AB*, *H a*, *T b*) → bit	*AB* := *AB* ∪ ⟨ [*a*,*b*] ⟩
insert(bat[*H*,*T*] *AB*, *H a*, *T b*) → bit	*AB* := *AB* ∪ *CD*
delete(bat[*H*,*T*] *AB*, *H a*, *T b*) → bit	*AB* := *AB* \ ⟨ [*a*,*b*] ⟩
delete(bat[*H*,*T*] *AB*, bat[*H*,*T*] *CD*) → bit	*AB* := *AB* \ *CD*
update(bat[*H*,*T*] *AB*,*H a*, *T b*) → bit	*AB*.delete(*AB*.select("=",*b*).insert([*a*, *b*])
update(bat[*H*,*T*] *AB*, bat[*H*,*T*] *CD*) → bit	*AB*.delete(*CD*.mirror.join(*AB*,"=").insert(*CD*)
access(bat[*H*,*T*] *AB*, int *mode*)	*change the update access mode of a BAT*

New BAT elements can be inserted and deleted, or values can be replaced in a straightforward way. Considering the fact that each BAT is a BUN-list (in an often unknown ordering), the `insert` guarantees that the only effect is that BUNs are *appended* to this list. Similarly the `replace` guarantees that the BUN-ordering is left intact. All update primitives exist both in single-value and bulk (bag-at-a-time) versions.

In order for updates to succeed, write access (`BAT_WRITE`) has to be granted with the `access` operator. BATs constructed with the BAT algebra operators have default read-only (`BAT_READ`) access, though, as this permits maximum optimizations in the Monet implementation, in particular, the sharing of memory resources between different BATs. For this same reason of enabling specific resource sharing optimizations, there also exist the append- and update-modes (`BAT_APPEND`, resp. `BAT_UPDATE`). The former allows only `insert`-s, while the latter also allows `replace`-s.

As the MIL update operators do not perform any locking, the MIL users need to ensure themselves that a BAT is not being updated concurrently (that is a hard rule). The Monet implementation can do BAT resource sharing between a read-only BAT and a writable BAT if the latter is append-only (in that case the already existing part of a BAT can be shared, since inserts only append). As for concurrency guarantees, multiple MIL users may read an append-only BAT. The same even goes for an update-only BAT, if the readers *themselves* ensure that they only read elements that are not being replaced. Thus, in case of append-only and update-only BATs, we can have concurrent reads while `insert`-s and in the latter access mode also `replace`-s are ongoing. In their Monet implementation, however, `insert` and `replace` both can potentially cause a re-allocation of memory resources, which would make such concurrent sharing unsafe.

All update operators therefore return a `bit` success status. Any update operator on read-only BATs fails always, just like `delete`-s on all but `BAT_WRITE` BATs. In fact, all updates on such fully writable BATs always succeed.[4] Inserts into append-only and inserts and replaces into update-only BATs *may or may not* succeed, that depends on whether the implementation needs to reallocate. In case of such a failure, it is the responsibility of the MIL user to change the BAT access mode manually into `BAT_WRITE`. It is then also up to the MIL users (e.g. a transaction system) to take explicit steps first, that prevent concurrent reads.

MIL operator type signature	**informal semantics**
persists(bat[H,T] AB, str s) : bit	*add to persistent store. name must not be in use.*
load(str s, int mode=MALLOC) → bat[H,T]	*read() or mmap() from persistent store into memory*
remove(str s) : bit	*remove from persistent store at next commit*
commit() : bit	*global database commit*
mode ∈ { MALLOC, VM_NORMAL, VM_RANDOM, VM_SEQUENTIAL }	

BATs normally exist until their last MIL variable reference disappears. By use of the `persists` operator, however, BATs are associated with a persistent (disk image) name. This means that before releasing the memory resources of a persistent BAT, it is written to disk (if dirty or not yet saved). As such, a Monet database consists of a collection of persistent BATs. The `load` operator allows one to obtain a persistent BAT by name.

At a successful `commit`, the set of persistent BATs is guaranteed to be permanent, and all its BAT images clean. Also, all persistent resources (i.e. disk images) of previously persistent BATs that were `removed` since the previous `commit` are released. A database crash and subsequent restart brings back the exact same collection of persistent BATs with the same content, as well as cleans up disk images that did not make the last `commit`. To make this recovery possible, Monet keeps the exact disk image at last commit (i.e. writes before the next commit goes to different disk files).

[4] Resource exhaustion causes an exception, not just a failed return status.

This transaction functionality is intended to support only the most basic atomicity and durability needs. The rationale is that in the Monet and MIL design philosophy, query processing is separated as much as possible from transaction processing, in order never to let transaction overhead stand in the way of query operator performance, as there are quite a few query-intensive application areas that need very little transaction functionality and very much performance (e.g. data mining, multi-media retrieval, GIS). Moreover, in Section 7.4, we show that with two simple extension modules that introduce explicit locking and I/O, sophisticated and high performance transaction systems *can* be build on top of the basic MIL transaction functionality.

4.3 Object-Oriented Example

We now use part of the decision support database from the TPC-D and TPC-H benchmarks [Tra95, Tra99] to illustrate how an object-oriented data model can be stored in Monet BATs, and how query processing can be implemented using MIL primitives.

4.3.1 Mapping The Object Model

The object-oriented model supplants the flat relational tables with a nested type system in which `Object` and `Set` form the basic building blocks for database types. Both concepts can be refined using inheritance, and methods can be defined on them. A standardized object-oriented data model has been defined by OMG [CAB+94, CBB+97] together with an object-oriented equivalent of the SQL query calculus, named OQL. We rephrase our example schema from Section 4.1.1 in an object-oriented way, as follows:

```
class Order {
 attribute date day;
 attribute float discount;
 relation Set<Item> items;
}
```

```
class Item {
 attribute float price;
 attribute float tax;
 relation Order order inverse Order.items;
}
```

The object-oriented data mapping into BATs is similar to the relational example. Simple object attributes are mapped just like relational columns into *table_attribute* BATs. Relation attributes, i.e. those that refer to an object, simply store an `oid` in the tail of such a BAT. Relation attributes allow to avoid one level of indirection as compared to the relational mapping (for instance, we can now directly join orders with items on its "order" attribute, instead of first having to join on the relational "order.id" attribute). The object-oriented data-model also allows to specify referential consistency using *inverse* relationships.

The possibility to nest collection types, however, leads to one extra BAT in the mapping of a class. This BAT is called the *extent*, and holds the `oid`-s of all objects in the collection. Set-valued attributes are stored in *table_attribute* BATs just like ordinary attributes, with the difference that each `oid` in the extent can occur zero or more times in the head of this BAT (instead of exactly one time). The set-value of such an attribute is formed by all tail values in this BAT with its `oid` in the head. In this way, nested collections are *flattened* into flat binary tables. Note that the empty set is encoded by the absence of an `oid` (the extent is necessary to detect its absence).

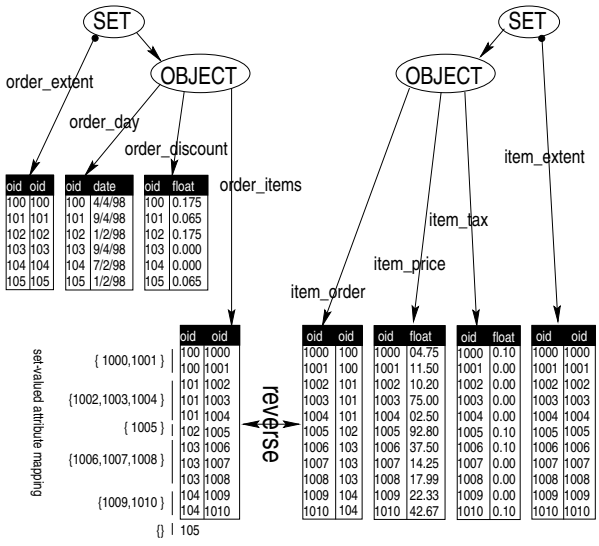

Figure 4.5: Mapping Objects onto BATs.

We represent the extent of a class with a `bat[oid,oid]` of which the head holds the `oid`-s of all objects in the class. Its tail column can be used to store the identifiers of the objects in the direct superclass. In top-level classes like `Item` and `Order`, we could store system-wide object identifiers in the tail.[5] Making a difference between *local* and system-wide object identifiers is interesting, as local identifiers need to be unique only in their (sub)class, and hence might be implemented with a smaller data type. In this example, we use the same identifiers in both columns.

A final note concerns relation attributes with an inverse relationship, like `Order.items` and `Item.order`. Here, we refrain from materializing an `order_items` BAT. Whenever it is used, we can instead use the `reverse` view on the `item_order` BAT. This way, the problem of keeping both inverses consistent is implicitly solved by the data structure.

4.3.2 Query Translation

An example OLAP query on our schema asks per year totals of tax paid on discounted items:

```
SELECT year, sum(total)
FROM
(   SELECT   price * tax AS total,
             year(item.order.day) AS year
    FROM     item
    WHERE    order.discount BETWEEN 0.00 AND 0.06)
GROUP BY year;
```

The MIL equivalent of this OQL single join query contains six BAT joins.[6] The join in line 7 actually corresponds with the OQL join between orders and items, the other

[5] Alternatively, one could choose to always store system-wide object identifiers in the tail of the extent.

[6] Each intermediate result can be destroyed right after its last use. This is done by assigning its MIL variable e.g. to `nil`. Statements doing so have been omitted here for brevity.

five joins are a consequence of the vertical fragmentation applied in Monet. While this may seem a waste of effort, we will see later how the MIL operators keep track of the relatedness of vertical fragments and how they avoid doing unnecessary work when joining those.

OQL query

```
SELECT year, sum(total)
FROM ( SELECT price * tax AS total,
              year(item.order.day) AS year
       FROM   item
       WHERE  item.order.discount BETWEEN 0.00 AND 0.06)
GROUP BY year;
```

MIL translation (*annotated below*)

```
01 ORD_DIS  :=  select(order_discount, 0.0, 0.06)
   bat[oid,oid] select subset [order,discount]
02 SEL_ORD  :=  ORD_DIS.mark.reverse
   bat[oid,oid] renumber tail and reverse into [selID,order]
03 SEL_DAY  :=  join(SEL_ORD.order_day)
   bat[oid,date] get [selID,day] values
04 SEL_YEA  :=  [year](SEL_DAY)
   bat[oid,int] extract [selID,year] values
05 SEL_GRP  :=  group(SEL_YEA)
   bat[oid,oid] group on year:  [selID,grp]
06 GRP_GRP  :=  unique(SEL_GRP.reverse.mirror)
   bat[oid,oid] get all unique [grp,grp]
07 ITM_SEL  :=  join(SEL_ORD, item_order.reverse).reverse
   bat[oid,oid] [item,selID] oid combinations
08 SUB_ITM  :=  ITM_SEL.mark.reverse
   bat[oid,oid] renumber tail column into subIDs
09 SUB_SEL  :=  ITM_SEL.reverse.mark.reverse
   bat[oid,oid] renumber head column into subIDs
10 SUB_PRI  :=  join(SUB_ITM,item_price)
   bat[oid,flt] get bat[subID,price] values
11 SUB_TAX  :=  join(SUB_ITM,item_tax)
   bat[oid,flt] get bat[subID,tax] values
12 SUB_TOT  :=  [*](SUB_PRI,SUB_TAX)
   bat[oid,flt] compute bat[subID,price*tax]
13 SUB_GRP  :=  join(SUB_SEL,SEL_GRP)
   bat[oid,oid] get bat[subID,grp]
14 RES_GRP  :=  GRP_GRP.mark.reverse
   bat[oid,oid] renumber tail into resIDs
15 RES_SUM  :=  {sum}(RES_GRP, SUB_GRP, SUB_TOT)
   bat[oid,flt] bat[resID,sum(tot)] results
16 RES_SEL  :=  {min}(RES_GRP, SEL_GRP, SEL_GRP.mirror)
   bat[oid,oid] get one selID for each result
17 RES_YEA  :=  join(RES_SEL, SEL_YEA)
   bat[oid,int] bat[resID,year] results
18             [printf]("% 9d % 6.3f \n", RES_YEA, RES_SUM)
```

Many optimizing query execution engines on the relational model have been built successfully in the past decades, by following the strategy of transformation of relational calculus to relational algebra with optimizing rewrite systems. The MIL binary algebra on the data model of Monet can be seen as an instance of such a physical algebra. This makes translation of SQL into MIL a special case of the traditional relational execution path. A specific translation technique for the decomposed model can be found in [KCJ+87, vdB94].

For supporting object-oriented systems, database researchers have tried to repeat the successes in the relational field by proposing a number of object-oriented query algebras [SLVZ95, CZ96]. They offer a nested object data model for supporting complex objects and support multiple collection types like Set, List and Bag. These languages were designed as input languages for algebraic optimizer systems that produce a physical query plan. Their implementation, however, turned out to be difficult due to the combination of a large number of operations and the complex storage model. To our

knowledge, no efficient implementations of object-algebras have been reported on large databases.

With the object-oriented MOA front-end [BWK98] for Monet we showed that despite the additional mapping of a logical data model (object- oriented) to the physical binary tables, ad-hoc query processing can be supported efficiently.

The fragmentation of the nested object-oriented data model onto binary tables brings some additional intrinsic optimizations. Traversing a relation attribute (see line 7) boils down to executing a MIL `join` operator on a `bat[oid,oid]`. This is very efficient as it comes down to the join optimization technique of *join indices*, proposed in [Val87]. Additional optimizations are achieved on set-operator expressions on nested sets. Intersecting two set-valued attributes on a collection of objects, for instance, is executed with just one MIL `intersect` operator, e.g. the following OQL query may intersect many sets:

```
select intersect(items1, items2) from Orders
```

but translates in MIL to the single bulk operator:

```
intersect(order_items1, order_items2)
```

4.3.3 Optimized Translations

MIL operators have the execution policy of full materialization of their result. This choice was made mainly to allow for more main-memory specific optimization in the implementation of MIL operators. If intermediate results are larger than the available memory, this simple policy quickly becomes suboptimal. A *pipelined* execution, where chunks flow through an operator tree, then performs better.

In a "pipelined" MIL program, we use on-the-fly horizontal fragmentation of tables in chunks, and let the MIL program iterate over these chunks. Standard decomposition rules for fragmented query processing must be applied in order to produce correct results using this additional horizontal fragmentation. For instance, a selection on a fragmented table can be executed on each chunk but results must afterwards be collected with a union. Aggregate computations must use decomposition rules of the aggregate in a local function and a global function [GBLP96].

Below we show a "pipelined" version of our sample MIL program, that was created by fragmenting the `Order` table on `oid`. Balanced chunk-sizes are guaranteed by the `split` MIL operator. All `Order` BATs are then fragmented with these split boundaries (lines 1-4).[7] The main MIL program body is then placed inside a loop over all chunks (lines 8-23). Wherever one of the `Order` BATs was used, it is substituted by the current chunk from this BAT. The SUM aggregate gets decomposed in a local {`sum`} and a global {`sum`}. The local aggregate results are grouped and re-aggregated using the global function after the loop terminates (lines 25-29).

[7] In Section 5.3.2 we describe how the **`split`** and **`fragment`** operators can be made to consume just constant time.

MIL Statement	signature of created bat
00 BOUNDS := split(order_extent, N);	bat[oid,oid]
01 FRG_EXT := fragment(order_extent, BOUNDS);	bat[oid,bat[oid,oid]]
02 FRG_DIS := fragment(order_discount, BOUNDS);	bat[oid,bat[oid,flt]]
03 FRG_DAY := fragment(order_day, BOUNDS);	bat[oid,bat[oid,date]]
04 FRG_I_O := fragment(item_order.reverse, BOUNDS);	bat[oid,bat[oid,oid]]
05 GRP_YEA := new(oid,int);	
06 GRP_SUM := new(oid,flt);	
07 BOUNDS@bat[P]batloop() {	
08 ORD_DIS := select(FRG_DIS.find($h), 0.00, 0.06);	bat[oid,flt]
09 ORD_SEL := ORD_DIS.mark(oid(count(ORD_DIS));	bat[oid,oid]
10 SEL_DAY := join(ORD_SEL.reverse,FRG_DAY.find($h));	bat[oid,date]
11 SEL_YEA := [year](SEL_DAY);	bat[oid,date]
12 SEL_GRP := group(SEL_YEA);	bat[oid,oid]
13 GRP_GRP := unique(SEL_GRP.reverse.mirror);	bat[oid,oid]
14 GRP_SEL := {min}(GRP_GRP, SEL_GRP, SEL_GRP.mirror);	bat[oid,oid]
15 GRP_YEA.insert(join(GRP_SEL.reverse.mirror, SEL_YEA));	
16 ITM_SEL := join(FRG_I_O.find($h), ORD_SEL);	bat[oid,oid]
17 SUB_ITM := ITM_SEL.mark(0).reverse;	bat[oid,oid]
18 SUB_SEL := ITM_SEL.reverse.mark(0).reverse;	bat[oid,oid]
19 SUB_PRI := join(SUB_ITM, item_price);	bat[oid,flt]
20 SUB_TAX := join(SUB_ITM, item_tax);	bat[oid,flt]
21 SUB_TOT := [*](SUB_PRI, SUB_TAX);	bat[oid,flt]
22 SUB_GRP := join(SUB_SEL, SEL_GRP);	bat[oid,oid]
23 GRP_SUM.insert({sum}(GRP_SEL.reverse, SUB_GRP, SUB_TOT));	
24 }	
25 GRP_GLB := group(GRP_YEA);	bat[oid,oid]
26 RES_GLB := unique(GLB_GRP.reverse.mirror).mark.reverse;	bat[oid,oid]
27 RES_GRP := {min}(RES_GLB, GRP_GLB, GRP_GLB.mirror);	bat[oid,oid]
28 RES_YEA := join(RES_GRP, GRP_YEA);	bat[oid,int]
29 RES_SUM := {sum}(RES_GLB, GRP_GLB, GRP_TOT);	bat[oid,flt]
30 [printf]("% 9d % 6.3f\n", RES_YEA, RES_SUM);	

The next step is to parallelize the pipelined program, by letting MIL work on multiple chunks in parallel. This is simply achieved by using a parallel batloop in line 7 with some parallelism degree P (`BOUNDS@[P]batloop(){ ··· }`). Not shown are lock statements that are now required around the `insert` statements. As an additional optimization, independent statements like line 1-4 could be placed in a parallel block ({| ··· |}).

4.4 MIL Extensibility

When a database is used for more than administrative applications alone, the need for additional functionality quickly arises [Sto86]. First of all, new application domains typically require – complex – *user-defined data-types*, such as for instance `polygon` or `point`. Secondly, one often needs to define new *predicates and functions* on them (`intersect(`p_1`, `p_2`)` or `surface(p)`, for example). Also, new application domains often create a need for new *relational operators*, such as spatial join or polygon overlay. In order to evaluate queries using the new predicates, functions and relational operators, one needs new *search accelerators* (such as for instance R-Trees). Finally, applications using a database as back-end want the option to perform certain application-specific operations near to the data. If a database server allows one to *link additional server code* on top of it, the communication penalties of creating a separate server process, encapsulating the database (a "client-level" server), can be avoided.

4.4.1 Other Systems

Postgres [SAH87] and Informix [Ger95] are examples of an extended relational systems, allowing for the introduction of new data types and access methods via prefixed

ADT interfaces. This works fine for new data types, predicates on them, and their accelerators, but does not allow for addition of new relational operators. In recent years, database researchers have spent much effort on Object-Oriented databases. In these systems, the programmer has more control, but to the point that data independence is compromised and the system gets hard to debug [eaENS89]. Another effort to achieve customizability has been the "extensible-toolkit" approach, where a database can be assembled by putting together a set of "easily" customizable modules (see [CD87]). Putting together such a system remains a serious work, however. One of the most appealing approaches to the problem we find in the Gral system [Güt89], which accepts a many-sorted algebra. Such an algebra can by its nature easily be extended with new operations.

4.4.2 MIL Extension Modules

Monet's extension system most resembles Gral, supporting new data types, new search accelerators, and user-defined primitives (embodying both new predicates and new relational operators).

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

Both module-specification and implementation object-code are fed into the `mel` parser and glue code generator. Implementation and glue code together form a shared library that can be dynamically loaded from MIL with the `load(module,..);` and unloaded with the `drop(module,..)` primitives.

Atomic Types

The ADT interface for atomic types assures that the MIL query algebra operators will work on user-defined types. For instance, one of the standard ADT operations is `AtomHash()`, which ensures that hash-based join works on BATs of any type. The ADT interface also contains routines to copy values to and from a heap, and to convert them to and from their string representations (for user interaction). Below we show how an atom can be specified, and which ADT operations should be defined:

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

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

Search Accelerators

Monet provides passive support for user-defined search accelerators via an ADT interface that maintains user-defined accelerators under update and I/O operations. The support is "passive" since basic MIL operators only use the built-in accelerators for their own acceleration. An ADT interface always incurs some implementation overhead, and bearing in mind that accelerators in Monet have to retain their efficiency under main-memory conditions, the canonical access path trio `open()`, `findnext()` and `close()` [Sto86] was left out[8]. The ADT interface merely serves to ensure that an accelerator remains up-to-date under MIL operators.

```
ACCELERATOR <name>
  Build   := <fcn>; # build accelerator on a BAT
  Destroy := <fcn>; # destroy accelerator
  Insert  := <fcn>; # adapt acc. under BUN insert
  Delete  := <fcn>; # adapt acc. under BUN delete
  Commit  := <fcn>; # adapt acc. for transaction commit
  Rollback:= <fcn>; # adapt acc. for transaction abort
  Cluster := <fcn>; # cluster a BAT on accelerator order
END <name>;
```

New Primitives

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

In order to do type-checking at the highest possible level, the MIL has been equipped with a polymorphism mechanism. A certain command, operation or iterator can have multiple definitions, with differing function signatures. Upon invocation, the Monet Interpreter decides which implementation has to be called, based on the types of the actual parameters, and matching operator and MIL procedure signatures in reverse order of definition.

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

The above shows the MEL syntax for specifying new primitives.

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

4.5 Conclusion

In this chapter, we have formally defined the MIL language, which is a computationally complete, dynamically typed, polymorphic, parallel programming language that stores (large) data collections in Binary Association Tables (BATs), and defines on this data model a 25-operator BAT algebra, an update interface and an extension language (MEL). Query algebras are not exactly new [Cod70], and have been used in a wide variety of database systems (e.g. [Güt89, DV92, LVZZ94, FGN+95]). Yet, MIL is a new query algebra as it is defined on the decomposed storage model (DSM) [CK85, KCJ+87] with the focus on extensibility and query-intensive applications.

In MIL, we have sought to define the minimal set of primitives that is powerful enough to allow a wide range of application scenarios. This has resulted in an algebra where the operators manipulate one, two or at most a few BATs (i.e. columns) at a time. This is a new direction for database query languages, which normally manipulate larger collections of columns (i.e. organized in tables or objects) – a direction pursued both to keep MIL neutral in the sense of data structures (and hence as easily applicable as possible to widely different logical data models) and because algebra operators with few parameters have a low degree of freedom, which opens up certain memory and CPU optimization opportunities in their implementation.

The fact that MIL is a column-at-a-time algebra typically leads to query graphs where the crucial nodes – such as those MIL operators computing selections and joins – are shared by multiple nodes higher in the query graph, that e.g. project columns through these joins and selections. In the generic iterator-model for query processing operators [Gra93a] the nested-iterator execution is broken for such nodes in order to materialize a sharable result. Given that such "forced materializations" are bound to happen often in MIL query plans, plus the fact that the nested iterator model might negatively impact CPU efficiency due to its recursive function calling, we decided to try a policy of full materialization of results in MIL operators. We showed in this chapter, that the resource consumption problems that are caused by such an approach can be handled by generating iterative "pipelined" MIL queries that process data in fragments. The advantage of such MIL queries is that these are directly apt for parallel query processing.

Some anecdotal evidence of the expressive power of MIL has been given in the various examples, where RDBMS and OODBMS data is stored in BATs and SQL/OQL queries on this data are translated into MIL.

Chapter 5

The Implementation of MIL

In this chapter, we describe the experience gained from implementing MIL as the primary interface language to the Monet system [BK95], and provide details of this implementation. We give an overview of its novel data structures and main-memory based algorithms. Special attention is paid to parts of the MIL language crucial for performance of the object-oriented and relational front-end applications.

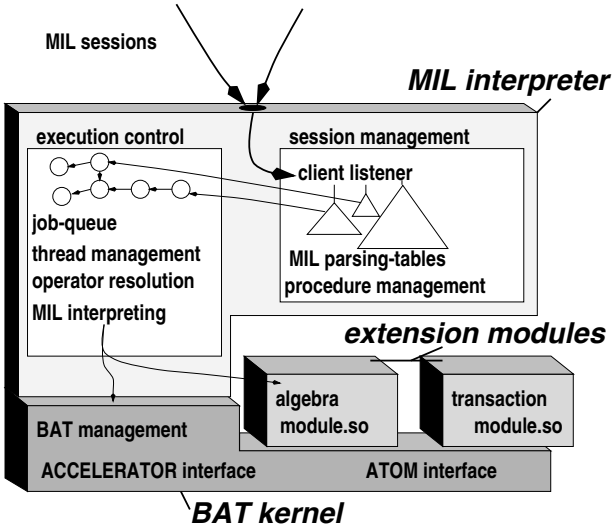

Figure 5.1: Monet software architecture.

Figure 5.1 shows the Monet software architecture. The basic data structures and primitives for management of BATs are provided by the *BAT kernel*. This kernel includes support for persistence, transaction management and data access. The MIL operator primitives themselves are found in *extension modules* that can dynamically be loaded into the system. The other MIL language features, like parsing, variable handling, procedure management, resolution of overloaded operators, etc. are provided by the *MIL interpreter*; which coordinates execution of client applications.

5.1 Main-Memory System Design

Our design decision to target Monet towards main-memory execution of mostly read-only queries has consequences for its implementation. CPU instruction time and memory access cost are the dominant costs in main-memory systems, rather than I/O. In-memory data movement and predicate evaluation tend to take up most time during query processing [TLPZT97, ADHW99]. As main-memory optimization and cost modeling are largely unexplored research areas, main-memory system design still depends on intuitive programmer notions about what kind of coding style makes well use of memory cache, CPU registers, etc. In the design of Monet we therefore adhered to a number of rules of thumb:

1. *keep it simple.* Having a complex software architecture that offers powerful (generic) operations can easily lead to a high percentage of CPU overhead (e.g., in interpretation cost, parameter passing or buffer copying) when there is no overshadowing I/O cost. Straightforward processing algorithms and data structures, like bucket-chained hash tables or T-trees, have proven to work best in main memory [LC86a].

2. *use large granularities.* Implementation functions that work on the granularity of one tuple-at-a-time introduce a fixed amount of interpretation overhead for each tuple (stack operations, context switch). Using large granularities in the basic processing functions is an effective way to decrease the effects of such interpretation overhead.

3. *sequential memory access.* Historic cost-models for main-memory systems could safely assume absence of locality of reference on memory access. Modern custom hardware, however, has three memory levels, and uses pipelined memory transfer over the bus to enhance memory bandwidth. This makes sequential memory access significantly faster than random access. This holds for simple PC hardware, but is even more true for the new generation of scalable shared memory multi-processor computers [Sil97].

The result of applying these rules in the design of Monet are reflected in the simple sequential array structure for BAT storage, the bulk nature of the MIL operators, and the straightforward algorithms applied for their implementation.

5.2 Data Storage in Monet

The BAT data structure (Figure 5.2) is seen by database code as a pointer to a *BAT descriptor*. A BAT descriptor points to two *column descriptors*, one for the *head* column, the other for the *tail*. Each column descriptor contains column-specific information, like the type stored, and pointers to search accelerators. The bulk data structure of the BAT is the *BUN heap*, a main-memory array of binary tuples (BUNs). It is reachable from the BAT descriptor via a *BUN descriptor*. BUNs are fixed-size records that consist of a head- and a tail-field.

The heaps of a BAT are stored on disk in their exact memory layout, which enables us to map these files into virtual memory. The algorithms of Monet do not see the difference between mapped memory and normal memory. To make this direct mapping

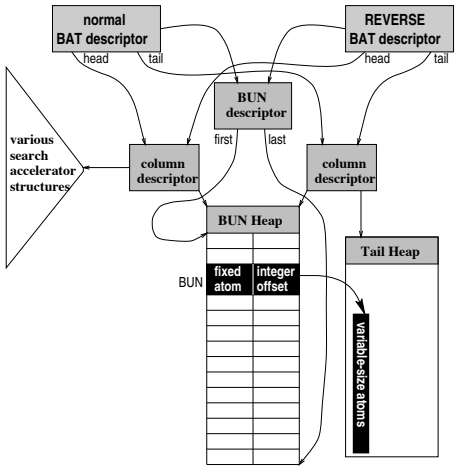

Figure 5.2: the BAT data structure.

possible, our storage scheme is carefully kept free of hard pointers. Absence of hard pointers implies that *pointer swizzling* [WD92] is performed lazily, on each data access. This policy only works well if data access is cheap and simple, and swizzling cost can be factored out in bulk operations. Monet therefore provides only a limited number of ways (3) to store atomic data in a BAT:

- *fixed-size atoms* are stored directly in the BUN record.
- *variable-sized atoms* store an integer in the BUN record. The integer is a byte-offset into a separate heap. This heap is a linear memory space just like the BUN heap and is reachable from the column descriptor.
- *implicit storage* virtual `oid`-s, defined by the additional `void` type, require no storage. A `void` column implicitly defines a column of densely ascending `oid` values (e.g., $100, 101, 102, 103, ..$). These values are computed on-the-fly by adding the array index number of the BUN in the BUN heap to some `oid` base number, called "seqbase". This seqbase (in our example 100) is stored in the column record.

The different treatment of variable-size atoms is necessary to keep the BUN heap an array of fixed-size BUNs. Implicit data storage was introduced deeply into the BAT data structure, as it is an optimization that is both greatly beneficial and often applicable. Many Monet applications map data into BATs that have one column with system-generated `oid`-s, and these are often dense and ascending. Virtual `oid`-s optimize both memory usage and value lookup: BAT sizes are cut by more than half[1] and lookup can simply be done by position: when looking for `oid` 102 in a `void` column with seqbase=100 we calculate by subtraction that it is located at array index 2 in the BUN heap.

[1] Alignment restrictions require a 4-byte type like `oid` to be aligned on 4-byte boundaries. A BUN `[oid,chr]` therefore occupies 8 bytes, whereas a `[void,chr]` occupies just 1 byte! Not aligning data would require values to be swizzled and copied on each access, at considerable extra CPU cost.

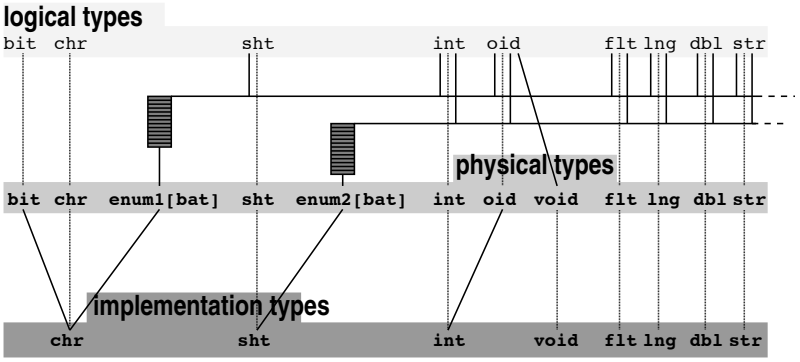

Figure 5.3: Type remappings on three levels.

5.2.1 Storage Type Remappings

The simple nature of data storage in a BAT can be contrasted with more flexible data storage schemes that would allow a more compact data representation, for instance by using bit-wise integer encodings on low-cardinality columns [Tan99]. Such flexibility is added in Monet on a level higher than the direct data structures, by storing a BAT with a different physical type signature than it is logically perceived. Virtual `oid`-s are one example of such *type remappings*, as they implement the logical `oid` type in a different way. These type-differences in the BAT implementation are hidden by the MIL interpreter.

There are three type levels in the MIL implementation between which type-remappings exist (see Figure 5.3):

logical types are the types known in the MIL language. These are called logical as they are not tied to one specific implementation.

physical types are a superset of the logical types (a logical type may be stored in alternative ways). A physical type defines how a type is implemented. For instance, BATs with an `oid` column may be stored either using `oid`-s or `void`-s. All physical types mapped on the same logical type have exactly the same MIL semantics.

implementation types are a subset of the physical types, as the implementation of some physical types may re-use the implementation from others. Such a *derived* physical type only implements the string representation functions of Monet's atom interface, but copies all other behavior of the type it is derived from. For instance, `bit` is implemented by `chr`, and `oid` is (currently) implemented by `int`.

An *enumeration type* is a specific case of a logical-to-physical type remapping. The idea is to represent all values in an enumerated domain as (small) integers. In OLAP and data mining, column values often have a low cardinality. If 256 or fewer different values occur, 1 byte would suffice to encode the values (resp. 2 bytes for 65536 or less). A lookup table is used to translate the encoding back to the original value. The parametrized physical types `enum1[BAT]` and `enum2[BAT]` provide generic encodings into 1- and 2-byte integers. Their parameter is an *encoding BAT* that contains the lookup table.

The advantage of enumeration types is compact storage, which is achieved especially if the other column is `void`. In those cases, the BUN heap becomes a dense array of 1- or 2-byte values. Enumeration types preserve the value ordering on the encoded values in the integer codes. By doing so, operators like range-select can work directly on the encoded values. This policy, however, makes the enumeration types expensive to update, as an insert of a new value in the domain may trigger a recoding of all values in the BAT. For this reason, enumeration type storage should only be applied to BATs when updates are infrequent or bulky.

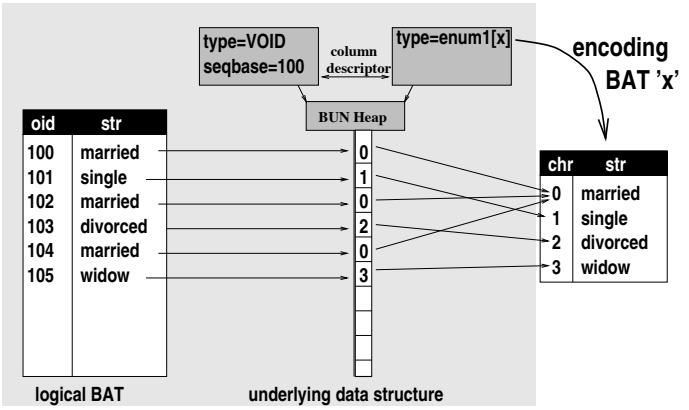

Figure 5.4: `bat[oid,str]` implementation as `bat[void,enum]`.

Enumeration types in MIL are part of the physical data design (i.e., they should be explicitly created), but in their use they are transparent, because the MIL interpreter hides all storage type remappings from the user. This mainly happens during *operator resolution*, where the MIL interpreter looks for an operator with a signature that matches the actual parameters. Enumerated types generally do not have the same MIL operators defined on them as the types they encode. This often causes operator resolution to fail. The MIL interpreter catches such *resolution misses* and tries to take corrective action. Its last resort is to convert all enumerated BAT parameters back to their non-enumerated representation; though this is expensive and avoided where possible. For this reason we created specific implementations that directly handle enumeration types for certain operators (e.g., pump and multi-join map – see Section 5.3.5). In other cases, like `bat.select("=",val)`, the MIL interpreter first tries to *encode* the non-BAT parameters (i.e. `val`), as this is cheaper than decoding the BAT.

5.3 MIL Operator Implementations

MIL operators are defined in an algebraic way; independent of the algorithms that implement them. Still, MIL is the target language for query-optimizing front-ends. For this reason, we introduce here the distinction between **strategical** and **tactical** query optimization, rather than the well-known distinction between logical and physical query-optimization. Query-optimizing front-ends produce MIL programs, so they decide the execution order of logical operations (the query execution "strategy"). Choosing a suitable algorithm (determining the run-time "tactics") is done automati-

cally by the MIL operator implementations.

5.3.1 Tactical vs. Strategical Optimization

In normal query optimization, the physical algebra contains algorithm-specific primitives; the query optimizer chooses both strategy and tactics. MIL separates these two concepts, which alleviates (though not eliminates) a number of problems found in classical query optimization:

1. queries are optimized to be executed in isolation. The real situation of the execution system, however, is determined by a load of multiple queries and the database status at time of execution (including buffer management and available search accelerators), which might favor altogether different decisions [KdB94].

2. errors in estimates of intermediate result characteristics quickly propagate in complex queries where the estimates of one operator are calculated from parameters that are themselves the result of previous estimates [IC91]. Such estimation errors lead directly to wrong decisions made by the optimizer.

3. a very detailed model of query processing creates a huge search space for complex queries, whose search itself gets to be resource-consuming [GLPK94].

Problems 1 and 2 are dealt with by the tactical phase at run-time, so it can take into account the system state. Monet's policy of materializing all results now becomes a benefit. When an operator starts, all information about its parameters is known. The optimization decisions are based on real information, not on estimates. This is the main difference between our approach and the so-called 'choose-plan operator' dynamic query optimization approach of [GW89, CG94, KD98]. In the 'choose-plan operator' approach, just before query execution, the estimates on the base operators are updated, and variables in the query are bound; then a new query optimization is done on the entire tree to see which alternative is best. This approach hence makes a decision based on much more actual information than normal QO – hence alleviates problem one – but as it is just an optimization closer to the moment of execution, it still suffers in errors made by estimation functions in the model (problem 2).

It is important to note that separating the query optimization in a strategical and tactical phase assumes that the strategical phase can do without physical details. The target of optimization cannot be formulated in terms execution time, as this depends on the (physical) algorithms chosen. A useful alternative target is minimization of the number of intermediate tuples generated. In this case, the price paid for our simplification is making the assumption that the best plan corresponds to the strategy that generates the least number of intermediate tuples.

The notion of strategical and tactical query optimization should not be confused with the classical notion of logical and physical query optimization [Gra93a], in which the logical phase depends on heuristics and the physical phase on a cost model. In the strategical phase we already determine the eventual execution order of logical algebra operations, so this optimization process includes both logical and physical optimization. We might for instance use cost models to estimate the selectivities of various MIL operators. The abstraction of all physical alternatives (e.g., merge-, hash- and nested loop-join) into their single logical operator (join) just causes a strong reduction of the search space, hence alleviates problem three.

As strategic optimization is a task of the front-end, further discussion of it falls outside the scope of this thesis.[2]

5.3.2 Data Structure Optimizations

Tactical optimizations imply that the implementations of the MIL operations themselves decide at run-time how they will produce their logical result. Some MIL operators can exploit the decomposed nature of the BAT data structure (Figure 5.2) and actually produce their result without doing any real work.

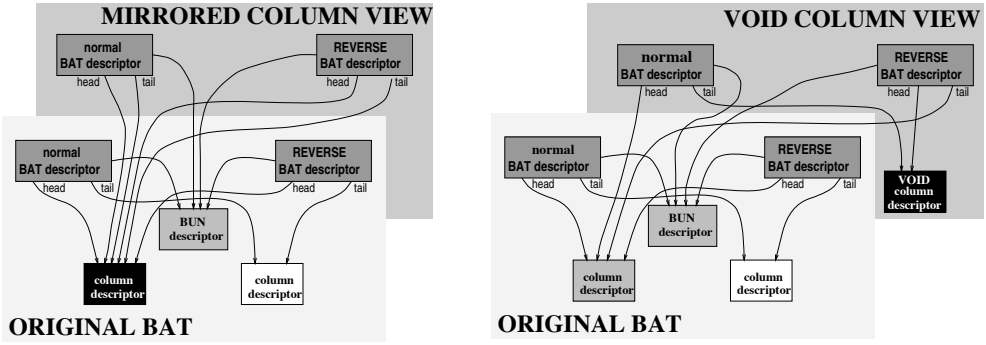

Figure 5.5: `mirror` (left) and `mark` implementations (right).

reversed view the BAT data structure contains two BAT descriptors (see Figure 5.2); one *normal* and one *reversed*. These two descriptors differ only in that they have their column descriptor pointers swapped. As such, they represent two different *views* on the same BAT. The implementation of the MIL `reverse` operator on a BAT makes use of these views. It just jumps from one view to the other; making this operation free of cost.

mirrored view The `mirror` MIL operator creates a new BAT descriptor that has both head and tail column descriptor pointers pointing to the original head column descriptor. The resulting BAT appears to have two identical columns.

void view Virtual `oid`-s are introduced by the `mark` operator, that creates a new column descriptor stating the column data type to be `void`. As `void` values are computed just by position, the data in the BUN heap is not looked at, and can therefore share the BUN heap from the operand BAT.

slice views The `slice` operator always returns a view BAT, whose BUN heap points to a horizontal slice of BUN heap of its first parameter. The same technique is used when a BAT column contains ascending values and a range-`select` or `fragment` is done on it. In such cases, we just provide an alternative BUN descriptor that points at that subset (see Figure 5.6).

enumeration views BATs with enumeration types provide various opportunities for view optimizations. Consider the unary `group` operator, that replaces the tail

[2] In [MPK00] one can find a discussion of such a system based on Monet, with the additional aim of exploiting synergy between multiple queries; i.e. multi-query optimization.

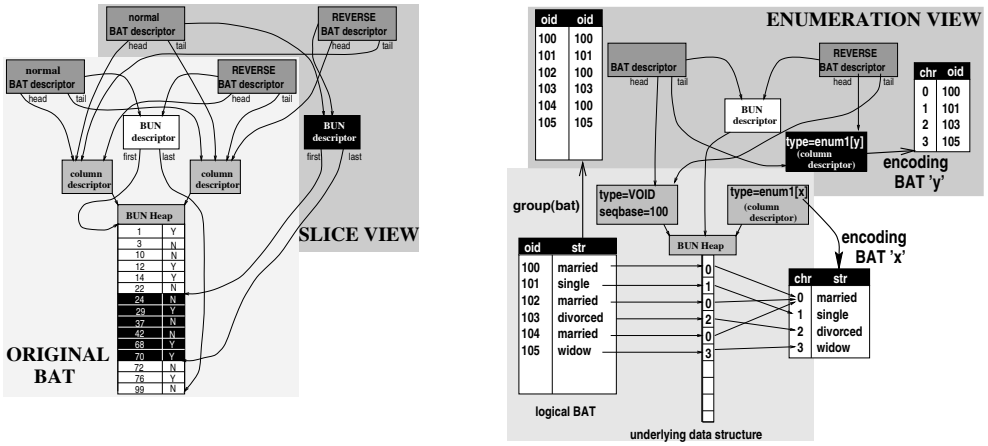

Figure 5.6: Slice and Enumeration Views.

column with `oid`-s. Each such `oid` uniquely identifies a tail value. It therefore suffices to replace the *encoding BAT* of the enumeration type with an alternative encoding BAT that maps onto `oid` values (see Figure 5.6). We then just create a view with a different enumeration type in the column descriptor; this enumeration type points to our new encoding BAT.

The unary multi-join map can use a similar optimization (e.g., `[*]( tax, 0.007)` can just be executed on the encoding BAT) if the resulting values are unique. The one-column version of the `unique` operator also can represent its result by a view on this encoding BAT.

cast views a *map cast* occurs when an (implicit) conversion function is passed into the multi-join map (e.g., `[bit](bat[any,chr])`). If the cast target has the same implementation type as the tail of the BAT (as `chr` and `bit` indeed have), the multi-join map can create a view with a tail column descriptor that contains the target type.

All these optimizations are highly efficient and exploit the freedom that MIL has in choosing the best way an operator can be implemented at run-time.

property	semantics	implementation
	column properties	
`int` **type**	(physical) type number	field in the column descriptor
`bit` **enum**	`true` ⇔ enumerated type	derived from the physical type
`bit` **dense**	the column contains a densely ascending range	field in the column descriptor
`bit` **sorted**	`true` ⇒ ascending value sequence	field in the column descriptor
`bit` **constant**	`true` ⇒ all equal values	sorted(b) and min(b)=max(b)
`oid` **align**	unique identifier for this value sequence	field in the column descriptor
`bit` **key**	`true` ⇒ no duplicates this value sequence	field in the column descriptor
`bit` **hash**	`true` ⇔ hash-table on this column exists	derived from the accelerator-list
`bit` **Ttree**	`true` ⇔ T-tree on this column exists	derived from the accelerator-list
	BAT properties	
`bit` **set**	`true` ⇒ no duplicate BUNs exist in this BAT	field in the BAT descriptor
`bit` **mirrored**	`true` ⇒ head and tail column are identical	short for **align**(h)=**align**(t)
`int` **count**	the exact number of BUNs in the BAT.	computed from BUN descriptor

Figure 5.7: BAT and Column Properties.

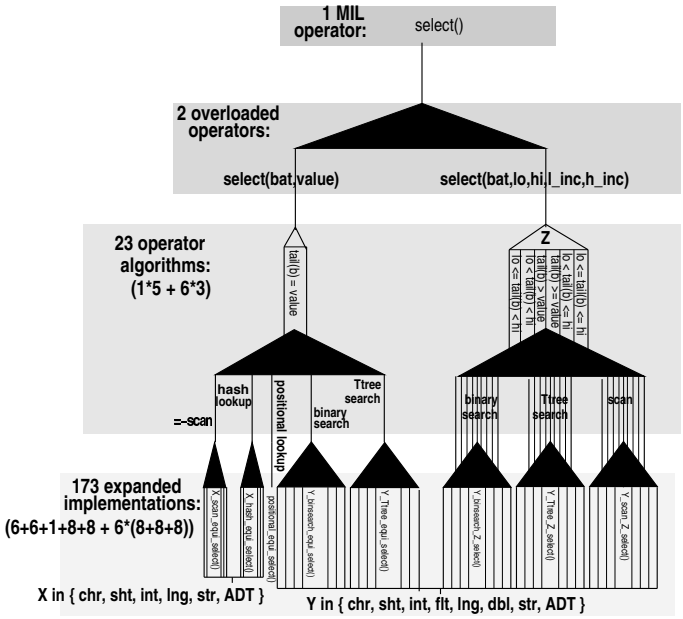

Figure 5.8: One MIL Operator, Many Implementations.

5.3.3 Property-Driven Tactical Optimization

Not all MIL operators get a free ride in terms of their implementation. For these operators the Monet implementation contains a multitude of algorithms. Selecting a good alternative at run-time happens in three levels:

operator overloading Some of the selection work is off-loaded to the command resolution in the MIL interpreter. Figure 5.8 shows that specific implementations for the equi- and range- predicates are available for the MIL `select` operator. The binary select handles the `=(tail(b),value)` and `isnil(tail(b))` predicates. The other select handles all range-conditions.

algorithm selection For the equi- and range-selects, the MIL interpreter can choose between hash-lookup, T-tree search, binary search and sequential scan (Figure 5.8). The tactical decisions made here are partially based on general system information about the CPU load, I/O activity and memory consumption. The most important information, though, comes from the *properties* that Monet maintains on all BATs (Figure 5.7).

Note that in the equi-select both the `=(tail(b),value)` and `isnil(tail(b),value)` predicates can be handled by the same algorithm, due to the fact that in Monet, the `nil` is implemented as a normal domain value (i.e., the smallest value). In the case of the range-predicates (which must filter out `nil`-s), this also means that in the physical C implementation, all range predicates have a lower bound that at least is `nil` (thus the range-selects for $<$`(bat(tail),value)` and $\leq$`(bat(tail),value)` need not be implemented).

The current tactical optimization procedures try to make best use of the information provided by the properties using heuristics, sometimes supplemented

```
01  PROC select(BAT[ANY::1,ANY::2] b, ANY::2 val) :  BAT[ANY::1,ANY::2]{
02              VAR i := b.info;
03
04              IF (i.find("dense").bit) {
05                      RETURN positional_equi_select(b,v);
06              } ELSE IF (i.find("tail.hash").bit) {
07                      RETURN hash_equi_select(b,v);
08              } ELSE IF (i.find("tail.sorted").bit) {
09                      RETURN binsearch_equi_select(b,v);
10              } ELSE IF (i.find("tail.Ttree").bit) {
11                      RETURN Ttree_equi_select(b,v);
12              }
13              RETURN scan_equi_select(b,v);
14  }
```

Figure 5.9: MIL procedure for tactical optimization of equi-select.

by a simple cost model. For selections, positional lookup is the most effective method, followed by hash-lookup, binary search and T-tree search. These rules are optimal under main-memory conditions.

We implemented these tactical optimizations as scripted MIL procedures, like in Figure 5.9. This makes it easy to experiment with more complex cost models (e.g., by using virtual memory usage statistics and result size estimates, or even sampling).

An important feature of property-driven tactical optimization is that each operator implementation *propagates* all relevant properties onto its result BAT. If the `scan_equi_select` is executed on a BAT that has a sorted head column, it will propagate this property on its result.

type-specific expansions When an algorithm has been selected, the Monet implementation makes an additional automatic choice for a type-specific routine.

MIL operators are generally type-generic. This means that data access (for instance, comparing two values) goes through some atomic ADT function interface. Calling functions in the inner loop of an algorithm should be avoided in programs optimized for main-memory. In order to optimize main-memory performance, Monet has for each algorithm multiple implementation routines that are specific to a certain type. We call such type-specific implementations *macro-expansions* as we generate them automatically from one source base using a macro package.

Monet is an extensible system and new atomic types may appear at run-time, so there always needs to be one generic implementation that still uses the ADT routines. All type-specific expansions are optional. Code expansion is an optimization technique that trades off code size for performance, so only those cases that benefit most and are likely to be used should be expanded.

Figure 5.8 shows, that the equi-select hash implementation has macro-expansions for the types `chr`, `sht`, `int`, `lng`, and `str`. These are called `chr_hash_equi_select`, `sht_hash_equi_select`, etc. There is also one `ADT_hash_equi_select` that goes through the ADT interface, that is used for all other types. Note that the `int` expansion is also used for selecting `oid` values, as `int` is the *implementation type* of `oid`.[3]

[3]Moreover, as a special optimization, the equi-select remaps `flt` selections to the `int` implemen-

Type-specific implementations are selected automatically, and are therefore not visible on the MIL level.

5.3.4 MIL Operator Implementation Overview

The different algorithms implemented for unary MIL operators are shown in bold text by Figure 5.10. This table shows per row which properties need be set for them to be chosen (leftmost column), as well as the macro-expansions applicable for each algorithm (rightmost column). As the decision which code-expansions to generate largely depends on the nature of the algorithm, these code-expansions and algorithm type share the horizontal dimension in Figure 5.10. The logical operators listed at the top of each column have a parameter h or t, indicating the column (head or tail) of the BAT-parameter on which the properties should apply.[4]

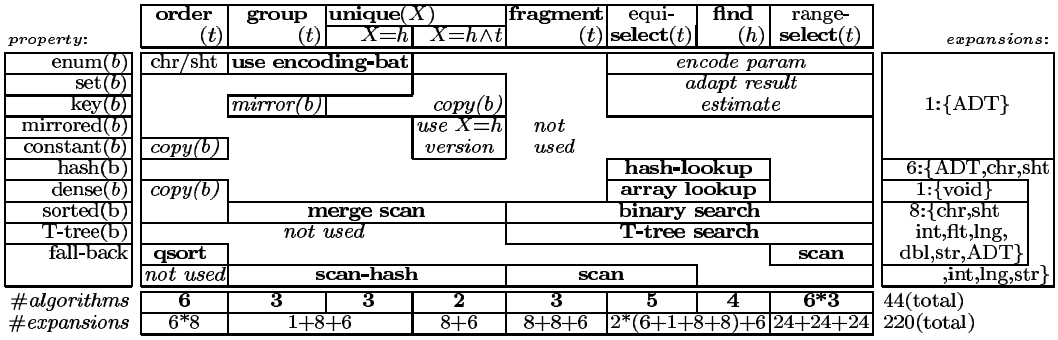

Figure 5.10: Algorithm Overview for **unop**(b).

The MIL-script for the equi-select of Figure 5.9 can be reconstructed by checking the equi-select column from top to bottom. The algorithm of the first row in which the property condition holds is chosen for execution.

A more complex example is the `unique` operator, which is split in two cases. The left column (labeled $X = h$) singles out the special case that only the head column is relevant for the uniqueness of the BUNs: if we know that both columns are equal (**mirrored(b)**), or one contains all the same values (**constant(b)**). In those cases, the standard two-column `unique` implementation invokes the single-column one, which is more efficient.

Figure 5.11 gives a similar algorithm overview for binary operators. Similar to `unique`, the binary set operators `intersect`, `union` and `diff` are implemented both in their 1- and 2-column versions.

We can read Figure 5.11 from top to bottom to find out which algorithm is chosen. Note that all binary operators except `diff` and `group` are symmetrical, in which cases the possibility of execution with swapped parameters is also taken into account. In contrast to the unary operators, there are no fall-back algorithms that are executed when no properties are set. The reason for this is that it is more efficient to *enforce* one property (e.g., by creating a hash-table or by sorting) than to execute a nested loop

tations and `dbl` selections to `lng` as these types share the same ADT hash function.

[4]Some simple-scan operators that are not shown are `sum`, `max` and `min`. The latter two take advantage of the sorted(b) and T-tree(b) properties to fetch their result directly, if they are set. The `count` operator just reads the *count*-property, while the `split` operator is based on sampling.

algorithm. The decision which property to enforce and – for symmetrical operators – on which BAT, depends on memory statistics and result size estimates.

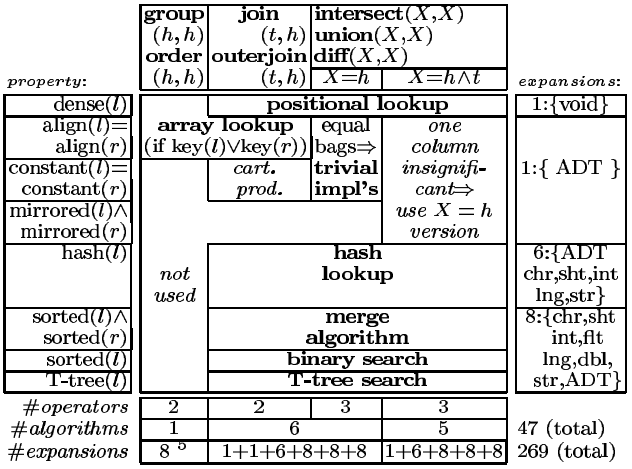

Figure 5.11: Algorithm Overview for **binop**(l, r).

5.3.5 Operators With Implicit Multi-Joins

The binary `group`, binary `order`, pump `{op}` and multi-join map `[op]` all define a result in terms of a multi-way equi-join on their parameters. Putting some hard-coded equi-join algorithm inside their implementations is not a good idea, because this leads to code duplication, but more importantly, such equi-joins would prevent any extended, optimized implementations of `join` to be used. In Section 6.5.2 we will see a particularly relevant example, where the MIL equi-`join` is overloaded to use the cache-friendly radix-cluster/decluster algorithm if certain conditions are met (i.e. other algorithms would thrash the caches).

Therefore, the implementation strategy for such operations is to assume the equi-join is trivial, i.e. each parameter is a densely ascending sequence, such that the join parameters already represent the join result that can be processed by a simple scan. For any other case, scripted MIL procedures perform the equi-join as a pre-processing step and thereafter call the "trivial-join" based implementations.

The `order` Operator

Figure 5.12 shows that the binary `order` is implemented by a C implemented operator `CTorder` that assumes all its three parameters to be trivial equi-joins. A MIL procedure is used to handle the case where real equi-joins are necessary, in a pre-processing step. We also show details of the C implementation of `CTorder`, which assumes that the second parameter (group-bat) is already sorted on tail. This in turn implies that all groups are consecutive and can be processed by a merge algorithm.

[5] 1 would be expected from the table. The binary group, and order though, are code-expanded on 8 tail types.

MEL Declaration Of `CTorder`

```
OPERATOR CTorder (bat[oid,any::0] o, bat[oid,any] s, bat[oid,any] v) :  bat[oid,int]
```

MIL Procedure That Resolves Equi-Join If Necessary

```
# dense() is used to assure a void column.  always falls into the second (free) case
proc dense(bat[oid,any] b) :  bat[void,b] { return b.reverse.order.mark.reverse; }
proc dense(bat[void,any] b) :  bat[void,b] { return b; } # normal case
proc order(bat[any::1,any] b, bat[any::1,any] v) :  bat[any::1,int] {
  if (v.info.find("dense").bit) {
    var s := b.order; # free if already sorted
    var sh := s.mark.reverse,
    var st := s.reverse.mark.reverse,
    var sa := sh.join(v);
    if (sa.count = sh.count) {
      return CTorder(sh, st, sa); # general, optimized, case
    }
    # unexpected misses occurred in dense join!
    var xa := a.reverse.mark.reverse;
    var xs := a.mark.reverse;
    return CTorder(xs.join(sh).dense, xs.join(st).dense, xa);
  }
  # generic 2-way equi-join
  var s := b.order;
  var xo := s.mark.reverse;
  var xs := s.reverse.mark.reverse;
  var yv := v.mark.reverse;
  var yo := v.reverse.mark.reverse;
  var xy := join(xo, yo.reverse);
  var jx := ji.mark.reverse;
  var jy := ji.reverse.mark.reverse;
  var js := jx.join(xs);
  var jr := js.order.mark;
  var rx := jr.reverse.join(jx);
  var ry := jr.reverse.join(jy);
  return CTorder(rx.join(xo).dense, rx.join(xs).dense, ry.join(yv).dense);
}
```

C Pseudo Code For `CTorder` Operator Implementation

```
typedef struct { oid head, int tail } bun; // orderby result tuples
// CTorder helper function
int sort_renumber(bun *start, bun *end, char* base, int ID) {
  // put cluster in sorted order
  qsort_offset_longcopy_<T2>(start, end, sizeof(oid), base);
  // replace offsets by ascending IDs
  <T2> prev = (<T2>) (base + start.tail);
  for(ID++; start < end; start++) {
    <T2> cur = (<T2>) (base + start.tail);
    if (cur != prev) { ID++; prev = cur; }
    start.tail = ID;
  }
  return ID;
}
```

```
bun Result[N]
CTorder(oid O[N], <T1> S[N], <T2> V[N]) {
  int clusterStart = 0, ID = 0;
  <T1> prev = S[0];

  for(int i=0,offset=0; i<N; i++,offset+=sizeof(<T2>)) {
    if (S[i] != prev) {
      ID = sort_renumber(Result+clusterStart, Result+i, V, ID);
      clusterStart = i; // new cluster starts here
    }
    Result[i].head = O[i];
    Result[i].tail = offset;
  }
  sort_renumber(Result+clusterStart, Result+N, V, ID);
  return Result;
}
```

Figure 5.12: Implementing `order` as a MIL Procedure around `CTorder`

```
proc multijoin(bat[int,bat] cols) :  bat[int,bat] {
  var n := cols.count - 1, i := n, nres := 0;
  var res := bat(oid,bat), pivots := bat(bat,bat);
  if (n < 0) return res;
  # join phase (execute linear tree of 'real' joins)
  var joinids := cols.find(n).mark.reverse;
  while((i :-= 1) >= 0) {
    var cur := cols.find(i).mark.reverse;
    var pivot := join(last, cur.reverse);
    pivots.insert(pivot.mark.reverse, pivot.reverse.mark.reverse);
    joinids := pivot.reverse.mark.reverse.join(cur).dense;
  }
  # project phase (only positional joins)
  res.insert(0, joinids);
  if ((i := n) = 0) {
    res.insert(1, cols.find(0).reverse.mark.reverse);
  } else {
    var outer := pivots.reverse.mark.reverse;
    var inner := pivots.mark.reverse;
    var cur := outer.find(i - 1);
    var nxt := inner.find(i - 1);
    while((i :-= 1) > 0) {
      res.insert(nres :+= 1, join(cur, cols.find(n - (i+ 1)).reverse.mark.reverse));
      cur := join(nxt,outer.find(i - 1));
      nxt := join(nxt,inner.find(i - 1));
    }
    res.insert(nres :+= 1, join(cur, cols.find(n - 1).reverse.mark.reverse));
    res.insert(nres :+= 1, join(nxt, cols.find(n).reverse.mark.reverse));
  }
  return res;
}
```

Figure 5.13: MIL Implementation Of Multi-Way Equi-Join In a Scripted Procedure

This merge algorithm assembles per group all BUNs in the output relation, where the `int` tail initially contains a byte-offset to the corresponding values. Each group forms a consecutive chunk in the result-BAT, and is sorted by a type-specific (code-expanded) Quick-Sort inside Monet. The sub-sorted chunk is then scanned, and for each different `tail` value, a new (ascending) `oid` is generated, which is placed *over* its old value (which was the beforementioned byte-offset). This algorithm is also illustrated later in this thesis in Figure 7.9.

The unary `CTorder` is directly implemented by the Monet Quick-Sort, as listed in Figure 5.10. Monet does not use the standard library `qsort` as this one is buggy on some OSes[6] and much performance can be gained by exploiting the knowledge on data types and tuple layout inside the database kernel, again using the technique of code-expansions. Monet implements Quick-Sort for all base types (8) with versions with or without offsets (2) using different variants for record swapping: 4-byte word swap, 8-byte word swap, and byte iteration (3) making for a total of 8 * 2 * 3 = 48 expanded `qsort` implementations. In Figure 5.10, the swapping method and addressing mode (offset or not) are counted as 6 separate algorithms. By sorting on a type and record-width known at compile-time, more than a factor 5 of performance is gained with respect to (non-buggy) standard library `qsort`-s, that compare each pair of values with a dereferenced (late binding) comparison routine that is passed to it as a parameter and swaps records with `memcpy`.

[6] On Solaris, `qsort` never returns on medium sized sorts with many duplicates.

#params	*type expansions*			*expansions*$^{\#params+1}$
3	enum1	fixed-size	6	216
2	enum2	variable-	6	36
1	normal	size	6	6
1	normal	ADT	1	1

Figure 5.14: 253 multi-join map implementations.

pump {op} and multi-join map [op]

The multi-join map implementation assumes that all its BAT-parameters have dense head columns, such that they already represent a multi-join result. If that is not the case, the MIL interpreter first invokes the `multijoin` shown in Figure 5.13, which similarly to `order` uses the MIL `join` to create BATs with dense head columns. This multi-join receives a nested-BAT that contains all $1 \leq n \leq N$ `BAT[any::0,any::n]`-parameters that must be joined on head column. The result $N+1$ result "columns" of the form `BAT[void,any::n]` for $0 \leq n \leq N$. That is, the first column holds the head value that matched, and all remaining BATs the "columns" of the table that is the result of the multi-join. The `multijoin` is implemented as a MIL procedure, and follows a two phase approach that puts all BATs to be joined in a linear join tree, and then creates join indices from the bottom to the top. The second phase then uses the join indices to project the columns from the leaves up to the root level.

group-type (T_1)	*algorithm*	*value-type* (T_3)	
ADT	hash-group	enum1	fixed-size
chr	sel-hash-group	enum2	
sht	merge-group	normal	variable-size
int	sel-merge-group		
4	4	3	2

Figure 5.15: 96 pump implementations.

The pump and multi-join map parameters often receive BAT parameters that have enumerated tail columns. In order to avoid conversions, their optimized implementations decode such types on the fly. We again use macro-expansions (see Section 5.3.3) to avoid having to check for each value whether it is enumerated (into either 1 or 2-byte integers) or not. Another expansion dimension that speeds up data access to the tail columns is created for fixed or variable-size atoms.

The multi-join map `[op]` has the additional complication that a variable number of BAT parameters may be passed. To eliminate looping overhead, and overhead for assembling a variable number of arguments for each invocation of `op`, code-expansions are made for the multi-join map with one, two and three parameters, always also code-expanded on their result type (hence $\#params+1$).

The pump `{op}` combines "attribute" values with a common "grouping" value. Consequently, the grouping columns of such BATs tend to have a relative low cardinality, and are therefore often represented with an enumeration type that exploits this. For this reason, in addition to the generic ADT version, type-specific expansions are made for the `chr`, `sht` and `int` types (these are the implementation types for the `enum1[b]`, `enum2[b]` and `oid` physical types, respectively). These are combined with four algorithms: merge- and hash-grouping, both in variants without and with a selection (the fourth optional parameter of the pump).

5.4 Conclusion

This chapter has given an overview of the implementation of Monet, and particularly of the MIL interpreter and the BAT algebra. This also entailed a detailed look at the data storage methods employed. Monet combines the decomposed storage model with a physical representation as a dense BUN array stored in main memory or mapped in virtual memory. The data type representation in the array is of fixed size and possibly compressed using *enumeration types*. The possible data storage optimizations are hidden in a data type layering of *logical types* (the MIL types), that can be stored in multiple *physical types*. Some physical types share the same implementation, hence *implementation types* form the lowest data type layer in Monet.

A number of techniques have been applied in Monet in order to optimize the performance of the BAT algebra operators. The most spectacular of such optimizations are the *view*-implementations, that reduce the operator cost to constant time, by constructing a result that shares its underlying BAT memory resources with its operands. Some of the most often used MIL operators (`reverse`, `mirror`, and `mark`) fall into this category.

One of the advantages of the full materialization in MIL is that when an operator starts executing, its parameters are completely available. This allows for a number of dynamic query optimization strategies that are hard to realize in the operator model [GW89, Gra93a], such as those that depend on the actual size, uniqueness, or ordering of intermediate query results. Also, it enables the MIL interpreter to choose itself which physical algorithm is most efficient to execute a particular (logical) algebra operator. This optimization process is called *tactical* optimization, and is separated here from *strategical* optimization, which aims at determining a good order of logical operators. We argue that abstracting away from physical algorithms reduces the query plan search space, which is an important problem in query optimization. We consider strategical optimization a front-end task (i.e. outside the scope of Monet), while tactical optimization is automatic in Monet.

Tactical query optimization in MIL is driven in the first place by the *properties* which are maintained on each BAT, and *propagated* by all operators. Some of the tactical optimizations can be done simply by MIL operator overloading (i.e. on the operator signature), others by explicit checking in MIL procedures that wrap around the various operator alternatives. The lowest level of tactical optimization is the use of the programming technique of *code-expansions* [Ker89]. Monet is implemented in a macro language from which C routines are extracted. Typically, each BAT algebra operator is code-expanded on the data type being processed by it. The main effect of code-expansions is that expensive constructs as dereferenced function calls (vs. C++ methods with late binding), variable-size data structure boundaries, and interpreted predicate evaluation, which lead to CPU-wise poorly predictable and thus inefficient code, can be eliminated. Thanks to the low degree of freedom in BAT algebra operator signatures, the amount of code expansion generated stays limited. In all, the 25 BAT algebra operators are implemented with 99 algorithms – which we summarized for all operators in this chapter. These algorithms are macro-expanded into 838 highly efficient implementation routines that can be invoked by the MIL interpreter. As these functions are all fairly simple, the cost in binary code size of all these expansions still remains moderate: when compiled with space-optimization on PC hardware, the Monet binary occupies about 1.5 MB.

Chapter 6

Memory/CPU Optimized Query Processing

Custom hardware—from workstations to PCs—has experienced tremendous improvements in the past decades. However, as discussed in Section 3.2.2, this improvement has not been equally distributed over all aspects of hardware performance and capacity. Figure 3.1 shows that the speed of commercial microprocessors has been increasing roughly 70% every year, while the speed of commodity DRAM has improved by little more than 50% over the past decade [Mow94]. A partial reason for this trend is that there is a direct trade-off between capacity and speed in DRAM chips, and the highest priority has been for increasing capacity. The result is that from the perspective of the processor, memory has been getting slower at a dramatic rate. This affects all computer systems, making it increasingly difficult to achieve high processor efficiency. Another trend is the ever increasing number inter-stage and intra-stage parallel execution opportunities provided by the multiple execution pipelines and speculative execution in modern CPUs. It has been shown that current database systems on the market make poor use of these new features [ADHW99, KPH+98, BGB98, TLPZT97]. This poor use is embodied by low IPC scores (instructions executed per CPU cycle) and high number of memory cache misses observed during database query execution. In contrast, other computational fields, like scientific computation, show over the generations of hardware increasing IPC scores, achieving near optimum performance out of modern CPUs and memory subsystems.

One rationale behind Monet was to reverse this trend in database systems; therefore, its design focuses on efficient use of main memory and CPU resources in query intensive database applications. One research question in Monet therefore was how data structures and algorithms should be designed in order to make efficient use of these resources. This chapter treats this subject, in particular the efficient processing of large joins in the context of (standard) relational query processing.

6.1 Related Work

Database system research into the design of algorithms and data structures that optimize memory access, has been relatively scarce. Our major reference here is the work by ShatDahl et al. [SKN94], which shows that join performance can be improved using

a main memory variant of Grace Join, in which both relations are first hash-partitioned in chunks that fit the (L2) memory cache. While being a landmark paper, there were various reasons that led us to explore this direction of research further. First, after its publication, the observed trends in custom hardware have continued, deepening the *memory access bottleneck*. For instance, the authors list a mean performance penalty for a cache miss of 20-30 cycles in 1994, while a range of 200-300 is typical in 2001 (and rising). This increases the benefits of cache-optimizations, and possibly changes the trade-offs. Another development has been the introduction of so-called level-one (L1) caches, which are typically very small regions on the CPU chip that can be accessed at almost CPU clock-speed. The authors of [SKN94] provide algorithms that are only feasible for the relatively larger, off-chip L2 caches that were in existence at that time. Finally, [SKN94] uses standard relational data structures. We will argue, in contrast, that the impact of memory access is so severe, that vertically fragmented data structures should be applied at the physical level of database storage.

Though we believe in the universal relevance of memory-access optimization to database performance, it is especially important for main memory databases [LC86a, LC86b, Eic89, Wil91, AP92, GMS92]. In the case of Monet, we use aggressive coding techniques for optimizing CPU resource utilization [BK99], that go much beyond the usual MMDBMS implementation techniques [DKO+84]. As described in Section 5.3.3, all Monet implementation code is written in a macro language, from which C language implementations are generated. The macros implement a variety of techniques, by virtue of which the inner loops of performance-critical algorithms like join are free of overheads like database ADT calls, data movement and loop condition management. These techniques were either pioneered by our group (e.g., logarithmic code expansion [Ker89]) or taken from the field of high performance computing [LL97]. In this work, we will show that these techniques allow compilers to produce code that better exploits the parallel resources offered by modern CPUs.

Past work on main-memory query optimization [LN96, WK90] models the main-memory cost of query processing operators on the coarse level of a procedure call, using profiling to obtain some 'magical' constants. As such, these models do not provide insight in individual components that make up query cost, limiting their predictive value. Conventional (i.e., non main-memory) cost modeling, in contrast, has I/O as dominant cost aspect, making it possible to formulate accurate models based on the amount of predicted I/O work. Calibrating such models is easy, as statistics on the I/O accesses caused during an experiment are readily available in a database system. Past main memory work has not been able to provide such cost models on a similarly detailed level, for two reasons. First, it has been found difficult to model the interaction between low-level hardware components like CPU, Memory Management Unit, bus and memory caches. Second, it was impossible to measure the status of these components during experiments, which is necessary for tuning and calibration of models. Modern CPUs, however, contain counters for events like cache misses, and exact CPU cycles [BZ98, ZLTI96, Yea96, Adv00, GBC+95, GT96, LH99, HSU+01, SA00, TDF01]. This enabled us to develop a new main-memory cost modeling methodology that first mimics the memory access pattern of an algorithm, yielding a number of CPU cycle and memory cache events, and then scores this pattern with an exact cost prediction. Therefore, the main contribution of the algorithms, models and experiments presented here is to demonstrate that detailed cost-modeling of main memory performance is both important and feasible.

6.2 Outline

In Section 6.3, we take a look at the aspects of modern computer architecture most relevant for the performance of main memory query execution, identify ongoing trends in custom hardware, and outline the consequences of these trends for database architecture. We also describe our *calibration tool*, that extracts the most important hardware characteristics like cache size, cache line-size, and cache latency from any computer system, and provide results for our benchmark platforms (Sun, SGI, and PC hardware).

In Section 6.4.2, we introduce the *Radix-Cluster* algorithm that improves the partitioning phase in Partitioned Hash-Join, as it allows to trade memory access cost for extra CPU processing. We perform exhaustive experiments where we use CPU event counters to obtain detailed insight in the performance of this algorithm. First, we vary the partition sizes, to show the effect of tuning the memory access pattern to the memory caches sizes. Second, we investigate the impact of code optimization techniques from the field of main memory databases. These experiments show that improvements of almost an order of magnitude can be obtained by combining both techniques (cache tuning and code optimization) rather than by each one individually. We explain our results with detailed models of both the partition (Radix-Cluster) and join phase of Partitioned Hash-Join, and show how performance can exactly be predicted from hardware events like cache and TLB misses.

The partitioned join and clustering strategies solely focus on determining for an equi-join which tuples join together. Their end-result hence is a join-index; i.e. projection of columns other than the join key columns from both input tables into the result was disregarded. In traditional relational DBMSs, this is not a critical omission, as including projection columns in the join algorithms is a trivial extension on the Radix-Cluster and Partitioned Hash-Join (the relations processed just get a bit wider due to the extra "luggage" of the projection columns). This is not the case for Monet query processing with vertically fragmented relations, where column projection is a separate phase, posterior to computing the join itself. In Section 6.5, we describe a new algorithm called *Radix-Decluster*, that executes such single-column projections very efficiently with regards to CPU usage and memory access, and use it in extensive benchmarking to compare the performance of overall join processing (including projections) of Monet with "traditional" and cache-conscious join processing in relational systems.

6.3 Modern Hardware and DBMS Performance

First, we describe the technical details of modern hardware relevant for main memory query performance, focusing on CPU and memory architectures. We perform experiments to illustrate how the balance between CPU and memory cost in query processing has shifted through time, and discuss a calibration tool that automatically extracts the hardware parameters most important for performance prediction from any computer system. We then look at what future hardware technology has in store, and identify a number of trends.

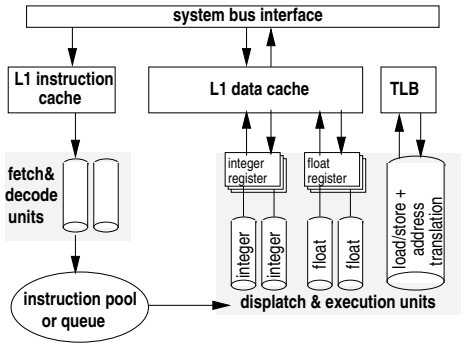

Figure 6.1: Modern Out-Of-Order CPU

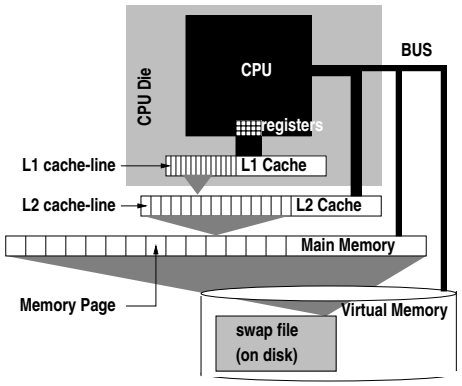

Figure 6.2: Hierarchical Memory System

6.3.1 A Short Hardware Primer

While CPU clock frequency has been following Moore's law (doubling every three years), CPUs have additionally become faster through parallelism *within* the processor. Scalar CPUs separate different execution stages for instructions, e.g., allowing a computation stage of one instruction to be overlapped with the decoding stage of the next instruction. Such a *pipelined* design allows for inter-stage parallelism. Modern *super-scalar* CPUs add intra-stage parallelism, as they have multiple copies of certain (pipelined) units that can be active simultaneously. Although CPUs are commonly classified as either RISC or CISC, modern CPUs combine successful features of both. Figure 6.1 shows a simplified schema that characterizes how modern CPUs work: instructions that need to be executed are loaded from memory by a fetch-and-decode unit. In order to speed up this process, multiple fetch-and-decode units may be present (e.g., the PentiumIII has three, the R10000 two [Die99, KP99, ZLTI96, Yea96]). Decoded instructions are placed in an instruction queue, from which they are executed by one of various functional units, which are sometimes specialized in integer-, floating-point, and load/store pipelines. The PentiumIII, for instance, has two such functional units, whereas the R10000 has even five. To exploit this parallel potential, modern CPUs rely on techniques like *branch prediction* to predict which instruction will be next before the previous has finished. Also, the modern cache memories are *non-blocking*, which means that a cache miss does not stall the CPU. Such a design allows the pipelines to be filled with multiple instructions that will probably have to be executed (a.k.a. *speculative execution*), betting on yet unknown outcomes of previous instructions. All this goes accompanied by the necessary logic to restore order in case of mis-predicted branches. As this can cost a significant penalty, and as it is very important to fill all pipelines to obtain the performance potential of the CPU, much attention is paid in hardware design to efficient branch prediction. CPUs work with *prediction* tables that record statistics about branches taken in the past.

Modern computer architectures have a hierarchical memory system, as depicted in Figure 6.2, where access by the CPU to main memory, consisting of DRAM chips on the system board, is accelerated by various levels of cache memories. Introduction of these cache memories, that consist of fast but expensive SRAM chips, was necessary due to the fact that DRAM memory latency has progressed little through time, making

its performance relative to the CPU become worse exponentially. First, one level of cache was added by placing SRAM chips on the motherboard. Then, as CPU clock-speeds kept increasing, the physical distance between these chips and the CPU became a problem, as it takes a minimum amount of time per distance to carry an electrical signal over a wire. As a result, modern CPUs have cache memories inside the processor chip. Without loss of generality, we assume one on-chip cache called L1, and a typically larger off-chip cache on the system board called L2. We identify three aspects that determine memory access cost:

latency Our exact definition of *memory latency* (l_{Mem}) is the time needed to transfer one byte from the main memory to the L2 cache. This occurs, when the piece of memory being accessed is in neither the L1 nor the L2 cache, so we speak of an *L2 miss*. It is important to note that during this time, all current hardware actually transfers multiple consecutive words to the memory subsystem, since each cache level has a smallest unit of transfer (called the *cache line*). During one memory fetch, modern hardware loads an entire cache line from the main memory[1] in one go, by reading data from many DRAM chips at the same time, transferring all bits in the cache line in parallel over a wide bus. Similarly, with *L2 latency* (l_{L2}) we mean the time it takes the CPU to access data that is in L2 but not in L1 (an *L1 miss*), and *L1 latency* (l_{L1}) is the time it takes the CPU to access data in L1. Each L2 miss is preceded by an L1 miss. Hence, the total latency to access data that is in neither cache is $l_{Mem} + l_{L2} + l_{L1}$. As L1 latency cannot be avoided, we assume in the remainder of this chapter, that L1 latency is included in the pure CPU cost, and regard only memory latency and L2 latency as explicit memory access cost.

To give an idea of the typical latencies, the Origin2000 used in our experiments has a L1 latency of 1 cycle, a L2 latency of 6 cycles and a memory latency of 100 cycles (it uses the 250MHz R10000 processors [ZLTI96, Yea96], so 1 cycle = 4ns). Thus, loading an L1 line (32 byte) form memory to the CPU takes $l_{Mem} + l_{L2} = 106$ cycles, loading an L2 line (128 byte, 4 L1 lines) from memory to CPU takes $l_{Mem} + 4 * l_{L2} = 124$ cycles, or 496ns.

bandwidth We define *memory bandwidth* as the number of megabytes of main memory the CPU can access per second. Sometimes there is a difference between read and write bandwidth, but this difference tends to be small. Bandwidth is usually maximized on a sequential access pattern, as only then all memory words in the cache lines are used fully. In conventional hardware, the memory bandwidth used to be simply the cache line size divided by the memory latency, but modern multiprocessor systems typically provide excess bandwidth capacity.

For instance, when fetching cache lines sequentially our Origin2000 would provide 128/ (496*10^{-9}) = 246Mb/s, but its maximum bandwidth is in fact 555MB/s.

address translation The Translation Lookaside Buffer (TLB) is a common element in modern CPUs (see Figure 6.1). This buffer is used in the translation of logical virtual memory addresses used by application code to physical page addresses in

[1] To which cache line the memory is loaded, is determined from the memory address. An X-way associative cache allows to load a line in X different positions. If X>1, some *cache replacement* policy chooses one from the X candidates. Least Recently Used (LRU) is the most common replacement algorithm.

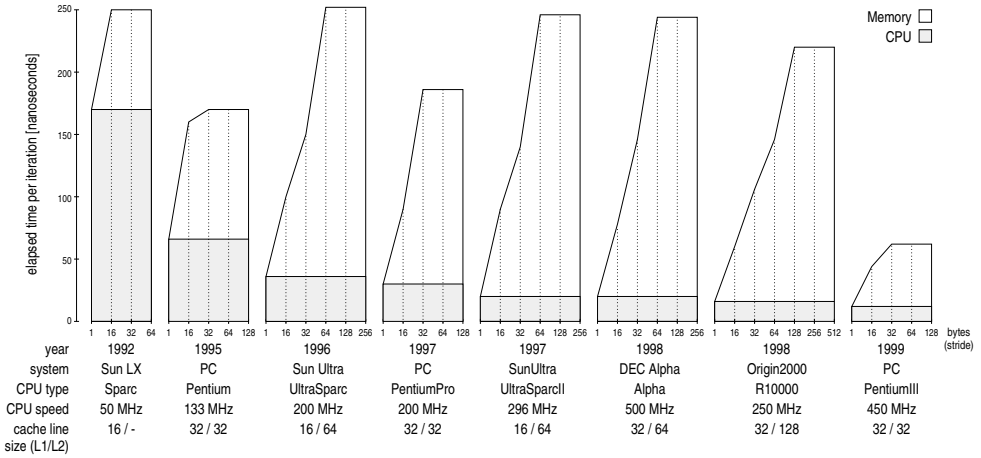

Figure 6.3: CPU and memory access cost per tuple in a simple table scan

the main memory of the computer. The TLB is a kind of cache that holds the translation for (typically) the 64 most recently used pages. If a logical address is found in the TLB, the translation has no additional cost. However, if a logical address is not cached in the TLB, a *TLB miss* occurs. A TLB miss is handled by trapping to a routine in the operating system kernel, that translates the address and places it in the TLB. Depending on the implementation and hardware architecture, TLB misses can be more costly even than a main memory access (on our Origin2000, it costs 57 cycles = 228ns). Moreover, handling a TLB miss often involves a lookup in a large memory array, whose access can itself trigger additional memory cache misses. The more pages an application uses (which is also dependent of the often configurable size of the memory pages), the higher the probability of TLB misses.

6.3.2 Experimental Quantification

We use a simple scan test to demonstrate the severe impact of memory access cost on the performance of elementary database operations. In this test, we sequentially scan an in-memory buffer, by iteratively reading one byte with a varying *stride*, i.e. the offset between two subsequently accessed memory addresses. We made sure that the buffer was in memory, but not in any of the memory caches. This experiment mimics what happens if a database server performs a read-only scan of a one-byte column in an in-memory table with a certain record-width (the stride); as would happen in a in a simple aggregation (e.g. SELECT MAX(column) FROM table).

Figure 6.3 shows results of this experiment on a number of popular workstations of the past decade. The X-axis shows the different systems ordered by their age, and per system the different strides tested. The Y-axis shows the absolute elapsed time for the experiments. For each system, the graph is split up to show which part of the elapsed time is spent waiting for memory (upper), and which part with CPU processing (lower, gray-shaded).

All systems show the same basic behavior with best performance at stride 1, increasing to some maximum at a larger stride, after which performance stays constant. This

is explained as follows: when the stride is small, successive iterations in the scan read bytes that are near to each other in memory, hitting the same cache line. The number of L1 and L2 cache misses is therefore low, and the memory access costs are negligible compared to the CPU costs. As the stride increases, the cache miss rates and thus the memory access costs also increase. The cache miss rates reach their maxima, as soon as the stride reaches the cache line size. Then, every memory read is a cache miss. Performance cannot become any worse and stays constant.

When comparing the Sun LX to the Origin2000, we see that CPU performance has increased 10-fold, of which a factor 5 can be attributed to faster clock frequency (from 50MHz to 250MHz), and a factor 2 to increased processor parallelism (the CPU cost has fallen from 160ns at 50MHz = 8 cycles to 16ns at 250MHz = 4 cycles). While this trend of exponentially increasing CPU performance is easily recognizable, the memory cost trend in Figure 6.3 shows a mixed picture, and has certainly not kept up with the advances in CPU power. Consequently, while our experiment was still largely CPU-bound on the Sun from 1992, it is dominated by memory access cost on the modern machines (even the PentiumIII with fast memory is 75% of the time stalling for memory). Note that the modern machines from Sun, Silicon Graphics and DEC actually have a memory access cost that in absolute numbers is even higher than the Sun from 1992. This can be attributed to the complex memory subsystem that goes with multi-processor SMP design, resulting in a high memory latency. These machines do provide a high memory bandwidth—thanks to the ever growing cache line sizes[2]—but this does not do any good in our experiment at large strides (when data locality is low).

This simple experiment also makes clear why database systems are quickly constrained by memory access, even on simple tasks like scanning, that seem to have an access pattern that is easily cacheable (sequential). The default physical representation of a tuple is a consecutive byte sequence (a "record"), which must always be accessed by the bottom operators in a query evaluation tree (typically selections or projections). The record byte-width of typical relational table is measured in the hundreds of bytes. Figure 6.3 makes quite clear that such large strides lead to worst-case performance, such that the memory access bottleneck kills all CPU performance advances.

To improve performance, we strongly recommend using **vertically fragmented** data structures. In our Monet system, we *fully* decompose relational tables on all columns, storing each in a separate Binary Association Tables (BAT). This approach is known in literature as the Decomposed Storage Model [CK85]. A BAT is represented in memory as an array of fixed-size two-field records `[oid,value]` – called Binary UNits (BUN) – where the `oid`-s are used to link together the tuples that are decomposed across different BATs. Full vertical fragmentation keeps the database records thin (8 bytes or less) and is therefore the key for reducing memory access cost (staying on the left side of the graphs in Figure 6.3).

6.3.3 Calibrator Tool

In order to analyze the impact of memory access cost in detail, we need to know the characteristic parameters of the memory system, like memory sizes, cache sizes, cache line sizes, and access latencies. Often, not all these parameters are (correctly) present

[2]In one memory fetch, the Origin2000 gets 128 bytes, whereas the Sun LX gets only 16; an improvement of factor 8.

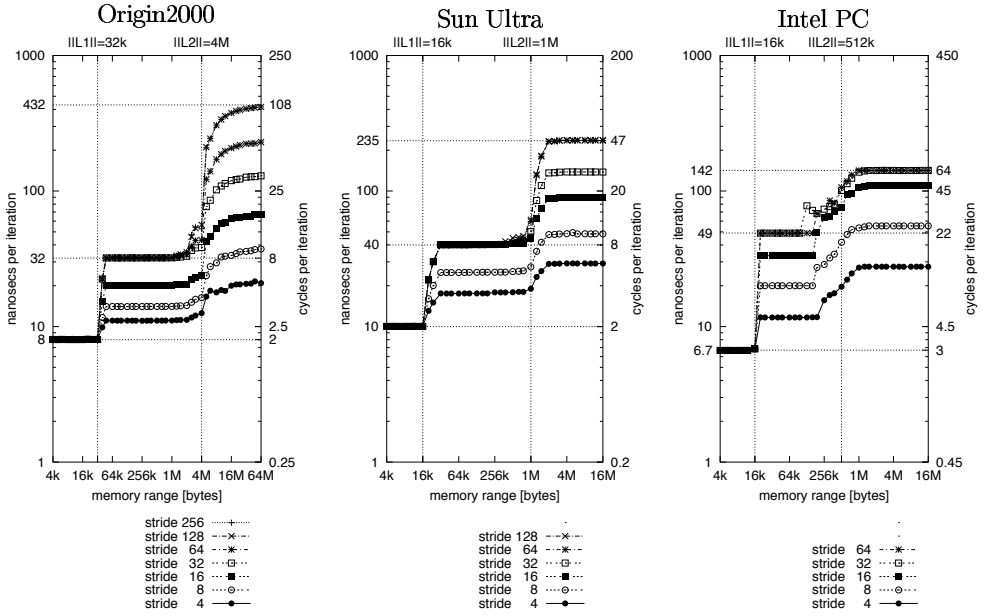

Figure 6.4: Calibrator Tool: Cache sizes, lines sizes, and latencies

in the hardware manual or from the vendor. Thus, we need to calibrate them ourselves. In the following, we describe a simple but powerful *calibration tool* to measure the (cache) memory characteristics of an arbitrary machine.

Calibrating the Memory System Our calibrator is a simple C program, mainly a small loop that executes a million memory reads. By changing the stride and the size of the memory area, we force varying cache miss rates. Thus, we can calculate the latency for a cache miss by comparing the execution time without misses to the execution time with exactly one miss per iteration. This approach only works, if memory accesses are executed purely sequential, i.e. we have to make sure, that neither two or more load instructions nor memory access and pure CPU work overlap. We use a simple pointer chasing mechanism to achieve this: the memory area we access is initialized such that each load returns the address for the subsequent load in the next iteration. Thus, modern super-scalar CPUs cannot benefit from their ability to hide memory access latency by speculative execution.

To measure the cache characteristics, we run our experiment several times, varying the stride and the array size. We make sure, that the stride varies at least between 4 byte and twice the maximal expected cache line size, and that the array size varies from half the minimal expected cache size to at least ten times the maximal expected cache size. The leftmost plot in Figure 6.4 depicts the resulting execution time (in nanoseconds) per iteration for different array sizes on our Origin2000 (MIPS R10000, 250 MHz = 4ns per cycle). Each curve represents a different stride. From this figure, we can derive the desired parameters as follows: Up to an array size of 32 KB, one iteration takes 8 nanoseconds (i.e. 2 cycles), independent on the stride. Here, no cache misses occur once the data is loaded, as the array completely fits in L1 cache. One of the two cycles accounts for executing the load instruction, the other one accounts

for the latency to access data in L1. With array sizes between 32 KB and 4 MB, the array exceeds L1, but still fits in L2. Thus, L1 misses occur. The miss rate (i.e. the number of misses per iteration) depends on the stride (s) and the L1 cache line size (LS_{L1}). With $s < LS_{L1}$, $\frac{s}{LS_{L1}}$ L1 misses occur per iteration (or one L1 miss occurs every $\frac{LS_{L1}}{s}$ iterations). With $s \geq LS_{L1}$, each load causes an L1 miss. Figure 6.4 shows, that the execution time increases with the stride, up to a stride of 32. Then, it stays constant. Hence, L1 line size is 32 byte. Further, L1 miss latency (i.e. L2 access latency) is 32ns − 8ns = 24ns, or 6 cycles. Similarly, when the array size exceeds L2 size (4 MB), L2 misses occur. Here, L2 line size is 128 byte, and L2 miss latency (memory access latency) is 432ns − 32ns = 400ns, or 100 cycles. Analogously, the middle and the rightmost plot in Figure 6.4 show the results for a Sun Ultra (Sun UltraSPARC [GBC+95] 200 MHz = 5ns per cycle) and an Intel PC (Intel PentiumIII! [Die99, KP99] 450 MHz = 2.22ns per cycle).

The *sequential memory bandwidth* for our systems, listed in Table 6.1, is computed from the cache line sizes and the latencies as follows:

$$bw_{seq} = \frac{LS_{L2}}{l_{Mem} + \frac{LS_{L2}}{LS_{L1}} * l_{L2}}.$$

We will discuss *parallel memory bandwidth* in the next section.

Analogously, we derive the parallel memory bandwidth from our scan experiment (cf. Figure 6.3) by dividing the largest cache line side by the execution for the respective stride. We get 555 MB/s for the Origin2000, 244 MB/s for the Sun, and 484 MB/s for the PC.

Calibrating TLB We use a similar approach as above to measure *TLB miss cost*. The idea here is to force one TLB miss per iteration, but to avoid any cache misses. We force TLB misses by using a stride that is equal to or larger than the systems page size, and by choosing the array size such that we access more distinct spots than there are TLB entries. Cache misses will occur at least as soon as the number of spots accessed exceeds the number of cache lines. We cannot avoid that. But even with less spots accessed, two or more spots might be mapped to the same cache line, causing conflict misses. To avoid this, we use strides that are not exactly powers of two, but slightly bigger, shifted by L2 cache line size, i.e. $s = 2^x + LS_{L2}$.

Figure 6.5 shows the results for our three machines. The x-axis now gives the number of spots accessed, i.e. array size divided by stride. Again, each curve represents a different stride. From the leftmost plot (Origin2000), e.g., we derive the following: Like above, we observe the base line of 8 nanoseconds (i.e. 2 cycles) per iteration. The smallest number of spots where the performance decreases due to TLB misses is 64, hence, there are 64 TLB entries. The decrease at 64 spots occurs with strides of 32KB or more, thus, the page size is 32KB. Further, TLB miss latency is 236ns−8ns = 228ns, or 57 cycles. In the rightmost plot, the second step at 512 spots is caused by L1 misses as L1 latency is 4 times higher than TLB latency on the PC. On the Origin2000 and on the Sun, L1 misses also occur with more than 1024 spots access, but their impact is negligible as TLB latency is almost 10 times higher than L1 latency on these machines.

Table 6.1 gathers the results for all three machines. The PC has the highest L2 latency, probably as its L2 cache is running at only half the CPU's clock speed, but it has the lowest memory latency and an incredibly low TLB miss latency. The Ori-

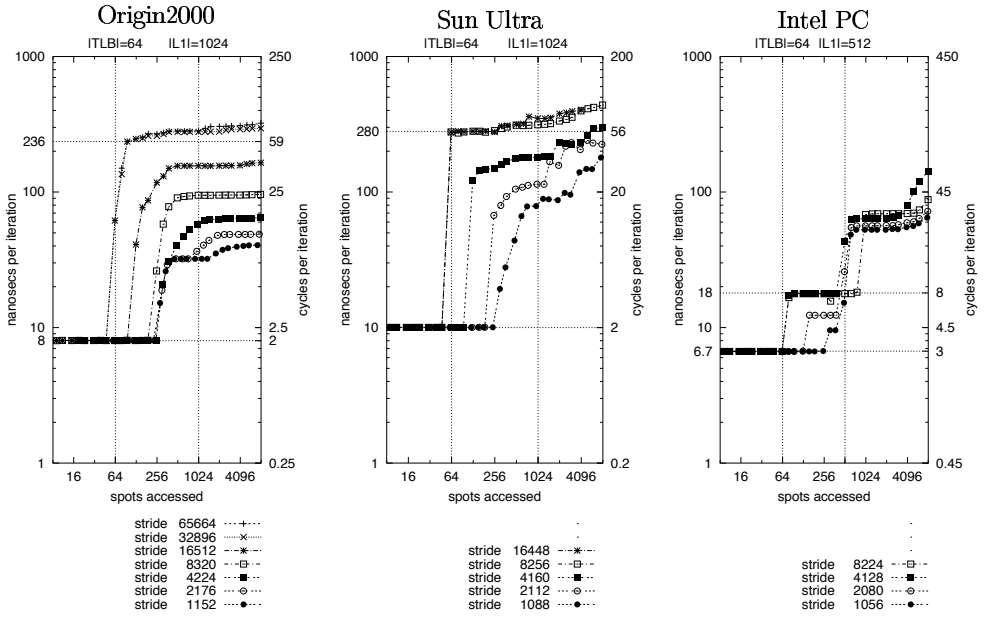

Figure 6.5: Calibrator Tool: TLB entries and TLB miss costs

gin2000 has the highest memory latency, but due to its large cache lines, it achieves the best sequential memory bandwidth.

6.3.4 Parallel Memory Access

It is interesting to note that the calibrated latencies in Table 6.1 do not always confirm the suggested latencies in the sequential scan experiment from Figure 6.3. For the PentiumIII, the access cost per memory read of 52ns at a stride of 32 bytes, and 204ns at a stride of 128 bytes for the Origin2000, are considerably lower than their memory latencies (135ns resp. 424ns), where in the case of the Sun Ultra, the scan measurement at L2 line size almost coincides with the calibrated memory latency.

		SGI Origin2000	Sun Ultra	Intel PC
OS		IRIX64 6.5	Solaris 2.5.1	Linux 2.2.5
CPU		MIPS R10000	Sun UltraSparc	Intel PentiumIII
CPU speed		250 MHz	200 MHz	450 MHz
main memory size		48 GB (4 GB local)	512 MB	512 MB
L1 cache size	$\|\|L1\|\|$	32 KB	16 KB	16 KB
L1 cache line size	LS_{L1}	32 bytes	16 bytes	32 bytes
L1 cache lines	$\|L1\|_{L1}$	1024	1024	512
L2 cache size	$\|\|L2\|\|$	4 MB	1 MB	512 KB
L2 cache line size	LS_{L2}	128 bytes	64 bytes	32 bytes
L2 lines	$\|L2\|_{L2}$	32,768	16,384	16,384
TLB entries	$\|TLB\|$	64	64	64
page size	$\|\|Pg\|\|$	32 KB	8 KB	4 KB
TLB size	$\|\|TLB\|\|$	2 MB	512 KB	256 KB
L1 miss latency	l_{L2}	24 ns = 6 cycles	30 ns = 6 cycles	42.2 ns = 19 cycles
L2 miss latency	l_{Mem}	400 ns = 100 cycles	195 ns = 39 cycles	93.3 ns = 42 cycles
TLB miss latency	l_{TLB}	228 ns = 57 cycles	270 ns = 54 cycles	11.1 ns = 5 cycles
seq. memory bandwidth	bw_{seq}	246 MB/s	193 MB/s	225 MB/s
par. memory bandwidth	bw_{par}	555 MB/s	244 MB/s	484 MB/s

Table 6.1: Calibrated Performance Characteristics

The discrepancies are caused by *parallel memory access* that can occur on CPUs that feature both speculative execution and a non-blocking memory system. This allows a CPU to execute multiple memory load instructions in parallel, potentially enhancing memory bandwidth above the level of cache-line size divided by latency. Prerequisites for this technique are a bus system with excess transport capacity and a *non-blocking cache* system that allows multiple outstanding cache misses.

To answer the question what needs to be done by an application programmer to achieve these parallel memory loads, let us consider a simple programming loop that sums an array of integers. Figure 6.6 shows three implementations, where the leftmost column contains the standard approach that results in sequential memory loads into the buf[size] array. An R10000 processor [ZLTI96, Yea96] can continue executing memory load instructions speculatively until four of them are stalled. In this loop that will indeed happen if buf[i],buf[i+1],buf[i+2] and buf[i+3] are not in the (L2) cache. However, due to the fact that our loop accesses consecutive locations in the `buf` array, these four memory references request the same 128-byte L2 cache line. Consequently, no parallel memory access takes place. If we assume that this loop takes 2 cycles per iteration[3], we can calculate that 32 iterations cost 32*2 + 124 = 190 cycles (where 124 is the memory latency on our Origin2000); a total mean cost of 5.94 cycles per addition.

Parallel memory access can be enforced by having one loop that iterates two cursors through the buf[size] array (see the middle column of Figure 6.6). This causes 2 parallel 128 byte (=32 integer) L2 cache line fetches from memory per 32 iterations, for a total of 64 additions. On the R10000, the measured maximum memory bandwidth of the bus is 555MB/s, so fetching two 128-byte cache lines in parallel costs only 112 cycles (instead of 124 + 124). The mean cost per addition is hence 2 + 112/64 = 3.75 cycles.

It is important to note that parallel memory access is achieved only if the ability of the CPU to execute multiple instructions speculatively spans multiple memory references in the application code. In other words, the parallel effect disappears if there is too much CPU work between two memory fetches (more than 124 cycles on the R10000) or if the instructions are too much interdependent, causing a CPU stall before reaching the next memory reference. For database algorithms this means that random access operations like hashing will not profit from parallel memory access, as following a link list (hash bucket chain) causes one iteration to depend on the previous; hence a memory miss will block execution. Only iterative algorithms with independent iterations and CPU processing cost per iteration that is less than the memory latency, will profit, like in the simple scan experiment from Figure 6.3. This experiment reaches optimal parallel bandwidth when the stride is equal to this L2 cache line size. As each loop

[3]As each iteration of our loop consists of a memory load (buf[i]), an integer addition (of "total" with this value), an integer increment (of i), a comparison, and a branch, the R10000 manual suggests a total cost of minimally 6 cycles. However, due to the speculative execution in the R10000 processor, this is reduced to 2 cycles on the average)

normal loop	**multi-cursor**	**prefetch**

for(**int** tot=i=0; i<N; i++) {
  tot += buf[i];
}
	for(**int** tot_0=tot_1=i=0, C=N/2; i<C; i++) {
	  tot_0+= buf[i];
	  tot_1+= buf[i+C];
	} **int** tot = tot_0+ tot_1;
		for(**int** tot=i=0; i<N; i++) {
		#**prefetch** buf[i+32] **freq**=32
		  tot += buf[i];
		}

5.94 cycles/addition	3.75 cycles/addition	3.88 → 2 cycles/addition

Figure 6.6: Three ways to sum an int-array, and their cycles per addition (Origin2000)

iteration then requests one subsequent cache line, modern CPUs will have multiple memory loads outstanding, executing them in parallel. Results are summarized at the bottom of Table 6.1, showing the parallel effect to be especially strong on the Origin2000 and the PentiumIII. In other words, if the memory access pattern is **not** sequential (like in equi-join), the memory access penalty paid on these systems is actually much higher than suggested by Figure 6.3, but determined by the latencies from Table 6.1.

6.3.5 Prefetched Memory Access

Computer systems with a non-blocking cache can shadow memory latency by performing a memory fetch well before it is actually needed. CPUs like the MIPS R10000, the Intel PentiumIII, and the newer Sun UltraSPARC II [GT96] processors have special *prefetching instructions* for this purpose. These instructions can be thought of as memory load instructions that do not deliver a result. Their only side-effect is a modification of the status of the caches. The work in [Mow94] studies compiler techniques to generate these prefetching instructions automatically. These techniques optimize array accesses from within loops when most loop information and dependencies are statically available, and as such are very appropriate for scientific codes written in FORTRAN. Database codes written in C/C++, however, do not profit from these techniques as even the most simple table scan implementation will typically result in a loop with both a dynamic stride and length, as these are (dynamically) determined by the width and length of the table that is being scanned. Also, if table values are compared or manipulated within the loop using a function call (e.g., comparing two values for equality using a C function looked up from some ADT table, or a C++ method with late binding), the unprotected pointer model of the C/C++ languages forces the compiler to consider the possibility of side effects from within that function; eliminating the possibility of optimization.

As a way around such situations, the MipsPRO compiler for the R10000 systems of Silicon Graphics allows the programmer to pass explicit prefetching hints by use of pragma's, as depicted in the rightmost column of Figure 6.6. This pragma tells the compiler to request the next cache line once in every 32 iterations. Such a prefetch-frequency is generated by the compiler by applying loop unrolling (it unrolls the loop 32 times and inserts one prefetch instruction). By hiding the memory prefetch behind 64 cycles of work, the mean cost per addition in this routine is reduced to 2 + ((124-64)/32) = 3.88 cycles. Optimal performance is achieved in this case when prefetching two cache lines ahead every 32 iterations (#**prefetch** buf[i+64] **freq**=32). The 124 cycles of latency are then totally hidden behind 128 cycles of CPU work, and a new cache line is asked every 64 cycles. This setting effectively combines prefetching with parallel memory access (two cache lines in 128 cycles instead of 248), and reduces the mean cost per addition to the minimum 2 cycles; three times faster than the simple approach!

6.3.6 Future Hardware Features

In spite of memory latency staying constant, hardware manufacturers have been able to increase memory bandwidth in line with the performance improvements of CPUs, by working with ever wider lines in the L1 and L2 caches. As cache-lines grew wider,

buses also did. The latest Sun UltraII workstations, for instance, have a 64-byte L2 cache-line which is filled in parallel using a PCI bus of 576 bits wide (576 = 64*8 plus 64 bits overhead). The strategy of doubling memory bandwidth by doubling the number of DRAM chips and bus lines is now seriously complicating system board design. The future Rambus [Ram96] memory standard eliminates this problem by providing an "protocol-driven memory bus". Instead of designating one bit in the bus for one bit of data transported to the cache-line, this new technology serializes the DRAM data into packets using a protocol and sends these packets over a thin (16-bit) bus that runs at very high speeds (up to 800MHz). While this allows for continued growth in memory bandwidth, it does not provide the same perspective for memory latency, as it is still DRAM that is being accessed, and there will still be the relatively long distance for the signals to travel between the CPU and the memory modules on the system board; both factors ensuring a fixed startup cost (latency) for any memory traffic.

An interesting proposal worth mentioning here has been "smarter memory" [MKW+98], which would allow the programmer to give a "cache-hint" by specifying the access pattern that is going to be used on a memory region in advance. In this way, the programmer is no longer obliged to organize his data structures around the size of a cache line, rather lets the cache adapt its behavior to the needs of the application. Such a configurable system is in some sense a protocol-driven bus system, so Rambus is a step in this direction. Configurable memory access has not yet been considered for custom hardware, however, let alone in OS and compiler tools that would need to provide the possibility to incorporate such hints for user-programs.

In a development that started from 2000, however, the memory subsystems for AMD PC platforms (the DDR SDRAM-based NVidia Nforce chipset [NVI01]), as well as the Pentium 4 [HSU+01], AMD Athlon XP [Adv01] and Sun SPARC III [LH99] processors themselves, do implement "hardware prefetching". In all cases, this entails a simple algorithm that detects pure consecutive memory access, and in those cases automatically issues memory prefetch requests. In a sense, this is a way to exploit the available memory bandwidth, which is an alternative to the traditional measure of just increasing the cache line width (which has the disadvantage that the cache becomes coarse-grained, hence less efficient for random-access loads). If the algorithms prefetches sufficient cache lines ahead, this new hardware feature has the potential of eliminating all memory access cost from algorithms that exhibit pure sequential access (at a pace lower than the maximum memory bandwidth) automatically.

Concerning CPU technology, it is anticipated [Sem97] that the performance advances dictated by Moore's law will continue well into at least 2010. However, performance increase will also be brought by more parallelism within the CPU. The upcoming IA-64 architecture has a design called Explicitly Parallel Instruction Computing (EPIC) [ACM+98, SA00], which allows instructions to be combined in bundles, explicitly telling the CPU that they are independent. The IA-64 is specifically designed to be scalable in the number of functional units, so while newer versions are released, more and more parallel units will be added. This means that while current PC hardware depends relatively less on parallel CPU execution than the RISC systems, this will most probably change in the new 64-bit PC generation.

Another ongoing CPU trend is tied to the trend of ever higher CPU clock speeds. The simplest way to increase CPU speed is to use more advanced (i.e. smaller) silicon process technology. The trend of process technology decreasing the transistor gate length with 50% every 2 years is one of the basic enablers of Moore's law. On top of

that, though, hardware designers tend to increase the length of the processor pipeline in every new design. For example, the MIPS R10K has 5 stages, the Pentium III has 7, the AMD Athlon has 11, and the Intel Pentium 4 has 20 [ZLTI96, Yea96, Die99, KP99, Adv99, Adv00, HSU+01]! The reason for doing so is that by splitting up the work of instruction execution in the CPU into more stages, each individual stage has to do less work, completes faster, making possible a higher clock speed on the same process. The side-effect of this design strategy is that new processors depend ever more strongly on speculative execution. In other words, if such processors have to execute difficult-to-predict code (i.e., non-Monet database code), the performance price paid for branch mis-predictions (i.e. flushing the pipeline) becomes ever greater.

A final trend in CPU design to be mentioned here is multiprocessing-on-a-chip. The simplest form, found in the latest IBM Power4 [TDF01] and Sun MAJC [Sun99] chips, is to simply place two identical CPUs on the same chip, together with SMP cache coherency logic between them. The motivation for doing so is to allow more efficient cache coherency, and – more importantly – simply because the ever-shrinking process technology sizes give CPU designers such a huge "transistor budget" (i.e., currently in the order of hundreds of millions) that they are losing the race of thinking up designs in time for effectively using such huge amounts of transistors. Putting multiple incarnations of a smaller existing CPU design on one chip is a way to sensibly efficiently use current chip manufacturing possibilities. A more subtle approach to multiprocessing-on-a-chip is called Simultaneous Multi-Threading (SMT) [EEL+97]. Here, one observes that the currently executing thread of "difficult" code often starves due to branch mis-predictions and memory wait cycles, while the OS might have other threads ready, waiting to execute (e.g. as in an SMP parallel database, executing a query on a single-CPU machine). By adding only a small bit of extra logic (i.e. replicating the register sets and some memory management support), one could make a CPU capable of executing 2, 4 or more threads simultaneously. Therefore, SMT is a more intelligent and efficient way of increasing performance than simply putting identical CPUs on the same chip (which replicates full, poorly used, CPUs). SMT will appear in 2002 in the new Intel Pentium 4 Xeon [Int01], with other hardware manufacturers to follow. As a multiprocessing-on-a-chip CPU (be it SMT or not) essentially makes for an SMP machine, this ongoing trend means that in the long term parallel processing will become ubiquitous (finally), in the sense that any machine will be an SMP machine (at least).

Summarizing, we have identified the following ongoing trends in modern hardware:

- CPU performance keeps growing with Moore's law for years to come.
- a growing part of this performance increase will come from parallelism within the chip.
- new bus technology will provide sufficient growth in memory bandwidth.
- memory latency will not improve significantly.

This means that the failure of current DBMS technology in properly utilizing memory and CPU resources of modern hardware [ADHW99, KPH+98, BGB98, TLPZT97] will not go away by itself, rather will grow worse. Modern database architecture should therefore take into account this new hardware environment. With this motivation, we investigate in the following new approaches to large main memory equi-joins, that specifically aim at optimizing resource utilization of modern hardware.

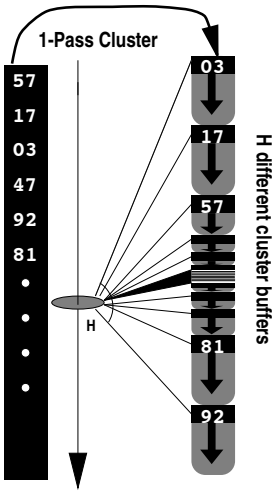

Figure 6.7: Straightforward clustering algorithm

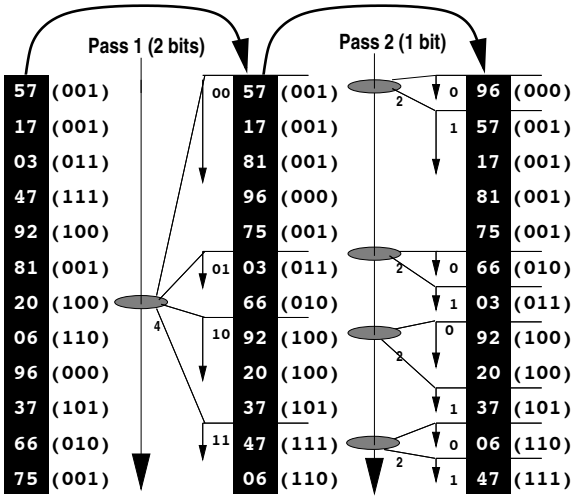

Figure 6.8: 2-pass/3-bit Radix-Cluster (lower bits indicated between parentheses)

6.4 Partitioned Hash-Join

Shatdahl et al. [SKN94] showed that a main-memory variant of Grace Join, in which both relations are first partitioned on hash-number into H separate *clusters*, such that each fit the memory cache, performs better than normal bucket-chained hash join. This work employs a straightforward clustering-algorithm that simply scans the relation to be clustered once, inserting each scanned tuple in one of the clusters, as depicted in Figure 6.7. This constitutes a random access pattern that writes into H separate locations. If this H is too large, there are two factors that degrade performance. First, if H[4] exceeds the number of TLB entries each memory reference will become a *TLB miss*. Second, if H exceeds the number available cache lines (L1 or L2), *cache trashing* occurs, causing the number of cache misses to explode.

As an improvement over this straightforward algorithm, we propose a clustering algorithm that has a memory access pattern that requires less random-access, even for high values of H.

6.4.1 Radix-Cluster Algorithm

The **Radix-Cluster** algorithm divides a relation into H clusters using multiple passes (see Figure 6.8). Radix-Clustering on the lower B bits (called the "Radix-Bits") of the integer hash-value of a column is achieved in P sequential passes, in which each pass clusters tuples on B_p bits, starting with the leftmost Radix-Bits ($\sum_1^P B_p = B$). The number of clusters created by the Radix-Cluster is $H = \prod_1^P H_p$, where each pass subdivides each cluster into $H_p = 2^{B_p}$ new ones. When the algorithm starts, the entire relation is considered as one single cluster, and is subdivided in $H_1 = 2^{B_1}$ clusters.

[4] If the relation is very small and fits the total number of TLB entries times the page size, multiple clusters will fit into the same page and this effect will not occur.

The next pass takes these clusters and subdivides each in $H_2 = 2^{B_2}$ new ones, yielding $H_1 * H_2$ clusters in total, etc. Note that with $P = 1$, Radix-Cluster behaves like the straightforward algorithm.

For ease of presentation, we did not use a hash function in the table of integer values displayed in Figure 6.8. In practice, though, it is better to use such a function even on integers in order to ensure that all bits of the table values play a role in the lower bits of the radix number.

The interesting property of the Radix-Cluster is that the number of randomly accessed regions H_x can be kept low; while still a high overall number of H clusters can be achieved using multiple passes. More specifically, if we keep $H_x = 2^{B_x}$ smaller than the number of cache lines and the number of TLB entries, we totally eliminate both TLB and cache trashing.

After Radix-Clustering a column on B bits, all tuples that have the same B lowest bits in its column hash-value, appear consecutively in the relation, typically forming chunks of $C/2^B$ tuples (with C denoting the cardinality of the entire relation). It is therefore not strictly necessary to store the cluster boundaries in some additional data structure; an algorithm scanning a Radix-Clustered relation can determine the cluster boundaries by looking at these lower B Radix-Bits. This allows very fine clusterings without introducing overhead by large boundary structures. It is interesting to note that a Radix-Clustered relation is in fact *ordered* on Radix-Bits. When using this algorithm in the Partitioned Hash-Join, we exploit this property, by performing a merge step on the Radix-Bits of both Radix-Clustered relations to get the pairs of clusters that should be Hash-Joined with each other.

6.4.2 Quantitative Assessment

The Radix-Cluster algorithm presented in the previous section provides three tuning parameters:

1. the number of Radix-Bits used for clustering (B), implying the number of clusters $H = 2^B$,
2. the number of passes used during clustering (P),
3. the number of Radix-Bits used per clustering pass (B_p).

In the following, we present an exhaustive series of experiments to analyze the performance impact of different settings of these parameters. After establishing which parameters settings are optimal for Radix-Clustering a relation on B Radix-Bits, we turn our attention to the performance of the join algorithm with varying values of B. For both phases, clustering and joining, we investigate, how appropriate implementation techniques can improve the performance even further. Finally, these two experiments are combined to gain insight in overall join performance.

Experimental Setup

In our experiments, we use binary relations (BATs) of 8 bytes wide tuples and varying cardinalities (C), consisting of uniformly distributed random numbers. Each value occurs three times. Hence, in the join-experiments, the join hit-rate is three. The result of a join is a BAT that contains the `[oid,oid]` combinations of matching tuples (i.e.,

category	MIPS R10K	Sun UltraSPARC	Intel PentiumII
memory access	L1_data_misses * 6 cy L2_data_misses * 100 cy	STALL_LOAD	cycles_while_DCU_miss_outstanding
		STALL_STORBUF	
	TLB_misses * 57 cy	M_{TLB} * 54 cy	M_{TLB} * 5 cy
	L1_inst_misses * 6 cy L2_inst_misses * 100 cy	STALL_IC_MISS	cycles_instruction_fetch_pipe_is_stalled
			ITLB_misses * 32 cy[5]
CPU stalls	branches_mis-predicted * 4 cy	STALL_MISPRED STALL_FPDEP	taken_mis-predicted_branches_retired * 17 cy[5]
			cycles_instruction_length_decoder_is_stalled cycles_during_resource_related_stalls[6] cycles_or_events_for_partial_stalls
integer divisions	$C * 2 * 35$ cy	$C * 2 * 60$ cy	cycles_divider_is_busy ($= C * 2 * 35$ cy)
real CPU	"the rest"	"the rest"	"the rest"

Table 6.2: Hardware Counters used for Execution Time Breakdown

a join-index [Val87]). Just like in [SKN94], we do not include tuple reconstruction in our comparison. The issues arising from taking into account the cost of projecting columns from both join input relations into the output are later described in detail in Section 6.5. The experiments were carried out on the machines presented in Section 6.3.3, an SGI Origin2000, a Sun Ultra, and an Intel PC (cf. Table 6.1).

To analyze the performance behavior of our algorithms in detail, we breakdown the overall execution time into the following major cost categories:

memory access In addition to memory access costs for data as analyzed above, these also contain memory access costs caused by instruction cache misses.

CPU stalls Beyond memory access, there are other events that make the CPU stall, like branch mis-predictions or other so-called resource related stalls.

divisions We treat integer divisions separately, as they play a significant role in our Hash-Join (see below).

real CPU The remaining time, i.e. the time, the CPU is indeed busy, executing the algorithms.

The three architectures we investigate, provide different hardware counters [BZ98] that enable us to measure each of these cost factors accurately. Table 6.2 gives an overview of the counters used. Some counters yield the actual CPU cycles spent during a certain event, others just return the number of events that occurred. In the latter case, we multiply the counters by the penalties of the events (as calibrated in Section 6.3.3). None of the architectures provides a counter for the pure CPU activity. Hence, we subtract the cycles spent on memory access, CPU stalls, and integer division from the overall number of cycles and assume the rest to be pure CPU cost.

In our experiments, we descovered that in our algorithms, branch mis-predictions and instruction cache misses do not play a role on either architecture. Thus, for simplicity of presentation, we omit them in our further considerations.

[5] Taken from [ADHW99].

[6] This counter originally includes "cycles_while_DCU_miss_outstanding". We use only the remaining part after subtracting "cycles_while_DCU_miss_outstanding", here.

Radix-Cluster

To analyze the impact of all three parameters (B, P, B_p) on Radix-Clustering, we conduct two series of experiments, keeping one parameter fixed and varying the remaining two.

First, we conduct experiments with various numbers of Radix-Bits and passes, distributing the Radix-Bits evenly across the passes. Figure 6.9 shows an execution time breakdown for 1-pass Radix-Cluster (C = 8M) on each architecture. The pure CPU costs are nearly constant across the all numbers of Radix-Bits, taking about 3 seconds on the Origin, 2.5 seconds on the PC, and a about 5.5 seconds on the Sun. Memory and TLB costs are low with small numbers of Radix-Bits, but grow significantly with rising numbers of Radix-Bits. With more than 6 Radix-Bits, the number of clusters to be filled concurrently exceeds the number of TLB entries (64), causing the number of TLB misses to increases tremendously. On the Origin and on the Sun, the execution time increases significantly due to their rather high TLB miss penalties. On the PC however, the impact of TLB misses is hardly visible due to its very low TLB miss penalty. Analogously, the memory costs increase as soon as the number of clusters exceeds the number of L1 and L2 cache lines, respectively. Further, on the PC, "resource related stalls" (i.e. stalls due to functional unit unavailability) play a significant role. They make up one fourth of the execution time when the memory costs are low. When the memory costs rise, the resource related stalls decrease and finally vanish completely, reducing the impact of the memory penalty. Or, in other words, minimizing the memory access costs does not fully pay back, as the resource related stalls partly take over their part.

Figure 6.10 depicts the breakdown for Radix-Cluster using the optimal number of passes. The idea of multi-pass Radix-Cluster is to keep the number of clusters generated per pass low—and thus the memory cost—at the expense of increased CPU cost. Obviously, the CPU cost are too high to avoid the TLB cost by using two passes with more than 6 Radix-Bits. Only with more than 15 Radix-Bits, i.e. when the memory cost exceed the CPU cost, two passes win over one pass.

The only way to improve this situation is to reduce the CPU cost. Figure 6.11 shows the source code of our Radix-Cluster routine. It performs a single-pass clustering on the D bits that start R bits from the right (multi-pass clustering in $P > 1$ passes on $B = P * D$ bits is done by making subsequent calls to this function for pass $p = 1$ through $p = P$ with parameters $D_p = D$ and $R_p = (p - 1) * D$, starting with the input relation and using the output of the previous pass as input for the next). As the algorithm itself is already very simple, improvement can only be achieved by means of implementation techniques. We replaced the generic ADT-like implementation by a specialized one for each data type. Thus, we could inline the hash function and replace the `radix` by a simple assignment, saving two function calls per iteration.

Figure 6.12 shows the execution time breakdown for the optimized 1-pass Radix-Cluster.

Figure 6.13 shows the execution time breakdown for the optimized multi-pass Radix-Cluster. The CPU cost has reduced tremendously, by almost factor 4. Replacing the two function calls has two effects. First, some CPU cycles are saved. Second, the CPUs can benefit more from the internal parallel capabilities using speculative execution, as the code has become simple and more predictable. On the PC, the resource stalls have doubled, neutralizing the CPU improvement partly. Probably, the simple loop does not offer enough "meat" to fill the pipelines efficiently.

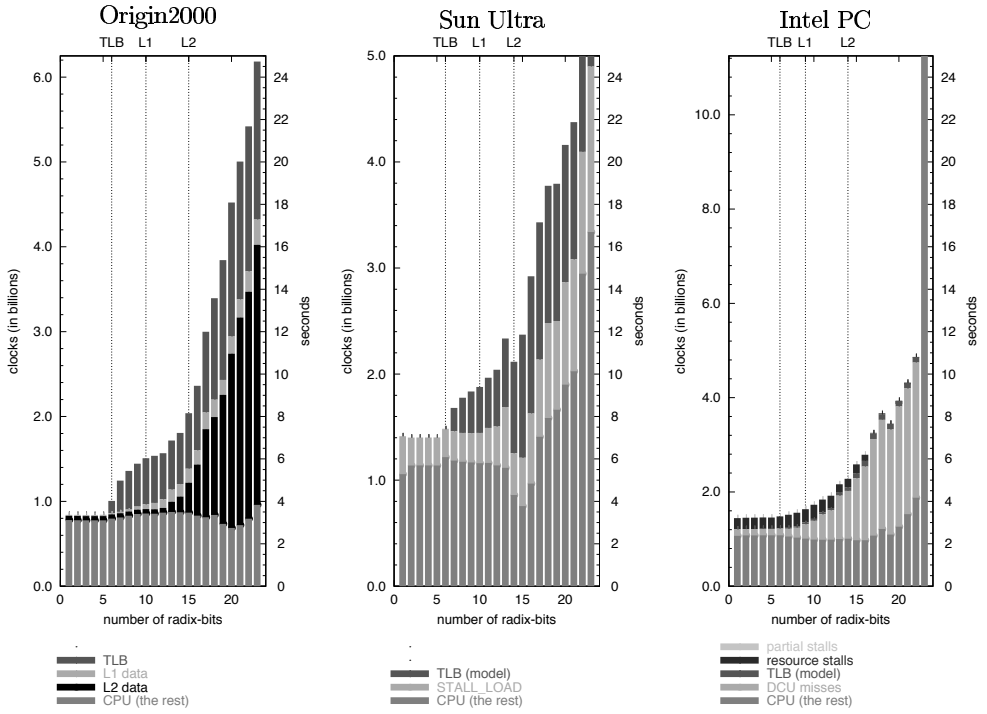

Figure 6.9: Radix-Cluster ($C = 8$M, 1 pass)

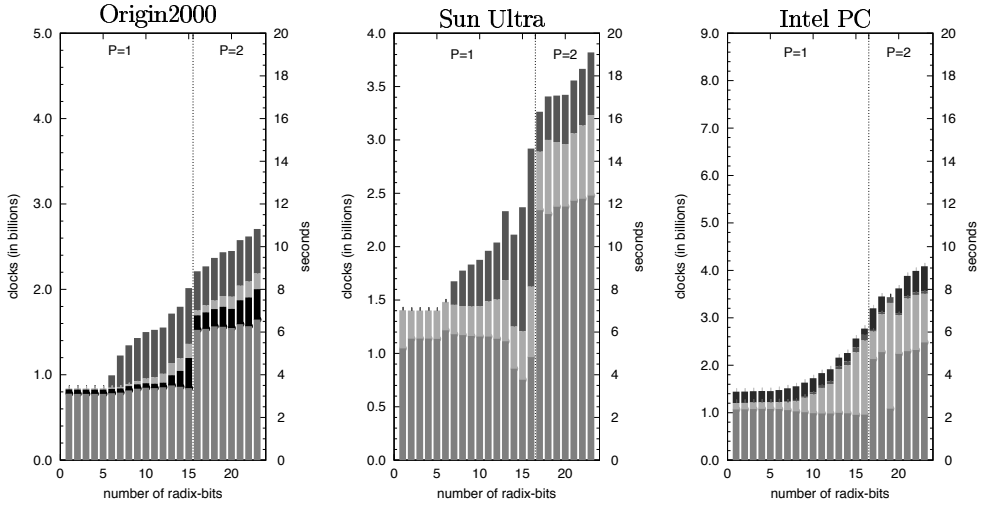

Figure 6.10: Radix-Cluster ($C = 8$M, best)

```
#define HASH(v) ((v≫7) XOR (v≫13) XOR (v≫21) XOR v)
typedef struct {
 int v1,v2; /* simplified binary tuple */
} bun;

radix_cluster(bun *dst[2^D], bun *dst_end[2^D] /* output buffers for created clusters */
 bun *rel, bun *rel_end,                       /* input relation */
 int R, int D                                  /* radix and cluster bits */
){
  int M = (2^D - 1) ≪ R;
  for(bun*cur=rel; cur<rel_end; cur++) {
    int idx = (*hashFcn)(cur→v2)&M;          ||int idx = HASH(cur→v2)&M;
    memcpy(dst[idx], cur, sizeof(bun));      ||*dst[idx] = *cur;
    if (++dst[idx]≥dst_end[idx]) REALLOC(dst[idx],dst_end[idx]);
  }
}
```

Figure 6.11: C language Radix-Cluster with annotated CPU optimizations (right)

With this optimization, multi-pass Radix-Cluster is feasible already with smaller numbers of Radix-Bits. On the Origin, two passes win with more than 6 Radix-Bits, and three passes win with more than 13 Radix-Bits, thus avoiding TLB trashing nearly completely. On the PC, the improvement is marginal. The severe impact of resource stalls with low numbers of Radix-Bits makes the memory optimization of multi-pass Radix-Cluster almost ineffective.

In order to estimate the performance of Radix-Cluster, and especially to predict the number of passes to be used for a certain number of Radix-Bits, we now provide an accurate cost model for Radix-Cluster. The cost model takes the number of passes, the number of Radix-Bits, and the cardinality as input and estimates the number of memory related events, i.e. L1 cache misses, L2 cache misses, and TLB misses. The overall execution time is calculated by scoring the events with their penalties and adding the pure CPU costs.

$$T_c(P,B,C) = P * \left(C * w_c + M_{L1,c}\left(\frac{B}{P},C\right) * l_{L2} + M_{L2,c}\left(\frac{B}{P},C\right) * l_{Mem} + M_{TLB,c}\left(\frac{B}{P},C\right) * l_{TLB} \right)$$

with

$$M_{Li,c}(B_p,C) = 2 * |Re|_{Li} + \begin{cases} C * \frac{H_p}{|Li|_{Li}} * \min\left\{1, \frac{|Re|_{Li}}{|Li|_{Li}}\right\}, & \text{if } \min\{H_p, |Re|_{Li}\} \leq |Li|_{Li} \\ C * \min\left\{3, 1 + \log\left(\frac{H_p}{|Li|_{Li}}\right)\right\}, & \text{if } \min\{H_p, |Re|_{Li}\} > |Li|_{Li} \end{cases}$$

and

$$M_{TLB,c}(B_p,C) = 2 * |Re|_{Pg} + \begin{cases} |Re|_{Pg} * \left(\frac{\min\{H_p, |Re|_{Pg}\}}{|TLB|}\right), & \text{if } \min\{H_p, |Re|_{Pg}\} \leq |TLB| \\ C * \left(1 - \frac{|TLB|}{\min\{H_p, |Re|_{Pg}\}}\right), & \text{if } \min\{H_p, |Re|_{Pg}\} > |TLB| \end{cases}$$

$$(\text{if } |Re|_{Pg} > |TLB|) \quad + \begin{cases} C * \left(\frac{H_p}{|L2|_{L2}}\right), & \text{if } H_p \leq |L2|_{L2} \\ C * \min\left\{2, 1 + \log\left(\frac{H_p}{|L2|_{L2}}\right)\right\}, & \text{if } H_p > |L2|_{L2} \end{cases}$$

$|Re|_{Li}$ and $|Cl|_{Li}$ denote the number of cache lines per relation and cluster, respectively, $|Re|_{Pg}$ the number of pages per relation, $|Li|_{Li}$ the total number of cache lines, both for the L1 ($i = 1$) and L2 ($i = 2$) caches, and $|TLB|$ the number of TLB entries. w_c denotes the pure CPU cost per tuple. To calibrate w_c, we reduced the cardinality so that all data fits in L1, and pre-loaded the input relation. Thus, we avoided memory

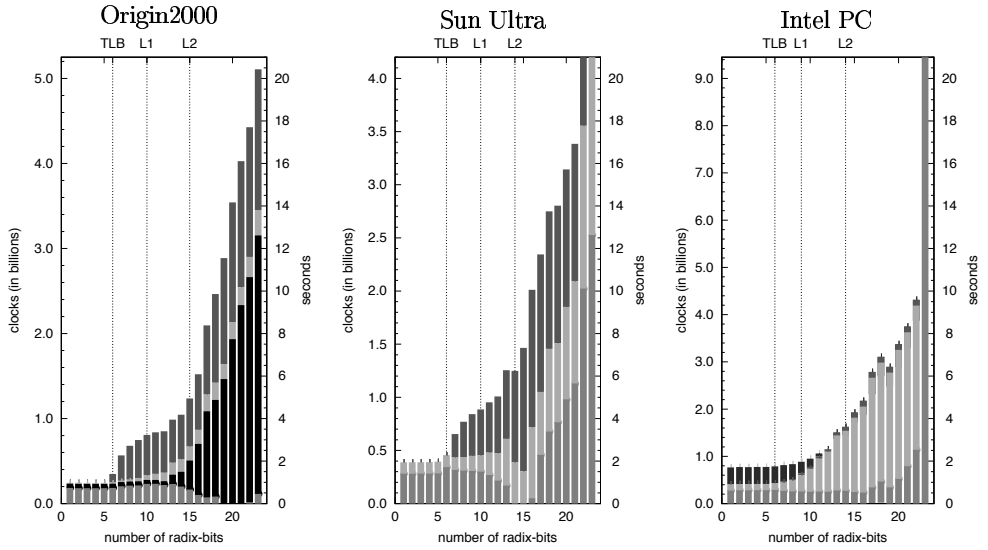

Figure 6.12: Optimized Radix-Cluster ($C = 8$M, 1 pass)

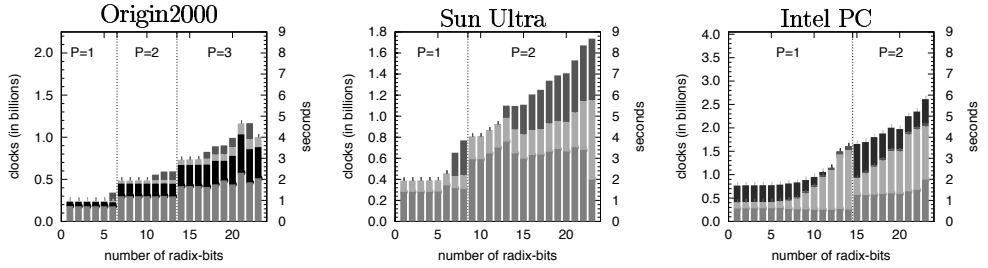

Figure 6.13: Optimized Radix-Cluster ($C = 8$M, best)

access completely. We measured $w_c = 100$ns on the Origin2000, $w_c = 200$ns on the Sun, and $w_c = 180$ns on the PC (including resource stalls).

The first term of $M_{Li,c}$ equals the minimal number of Li misses per pass for fetching the input and storing the output. The second term counts the number of additional Li misses, when the number of distinct Li lines accessed concurrently (i.e. x=min $\{H_p, |Re|_{Li}\}$)[7] either approaches the number of available Li lines ($x \leq |Li|_{Li}$) or even exceeds this ($x > |Li|_{Li}$). A detailed description of this and the following formulas is given in [MBK99]. First, the probability that the requested cluster is not in the cache—due to address conflicts—increases until $H_p = |Li|_{Li}$. Then, the cache capacity is exhausted, and a cache miss for each tuple to be assigned to a cluster is certain. But, with further increasing H_p, the number of cache misses also increases, as now also the cache lines of the input may be replaced before all tuples are processed. Thus, each input cache line has to be loaded more than once. The first two terms of $M_{TLB,c}$ are made up analogously. Additionally, using a similar schema as $M_{Li,c}$, the third term models—for relations that contain more pages than there are TLB entries—

[7]Using min $\{H_p, |Re|_{Li}\}$ instead of simply H_p takes into account, that smaller relations may completely fit in Li, i.e. with $H_p > |Li|_{Li} > |Re|_{Li}$, several (tiny) clusters share one cache line.

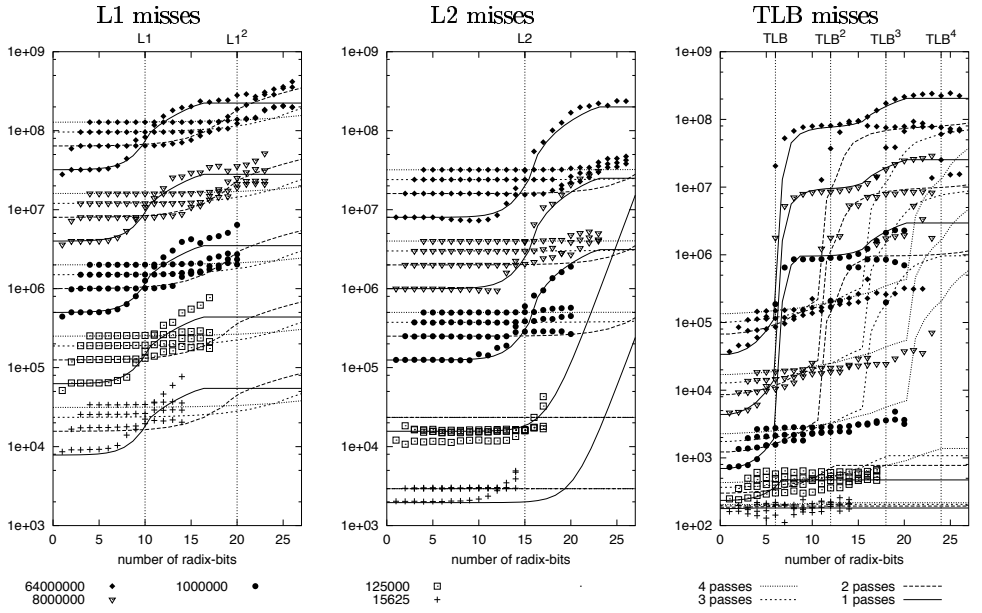

Figure 6.14: Measured (points) and Modeled (lines) Events of Radix-Cluster (Origin2000)

the additional TLB misses that occur when the number of clusters either approaches the number of available L2 lines ($H_p \leq |L2|_{L2}$) or even exceeds this ($H_p > |L2|_{L2}$).

Figure 6.14 compares our model (lines) with the experimental results (points) on the Origin2000 for different cardinalities. The model proves to be very accurate for the number of cache misses (both, L1 and L2) and TLB misses. The predicted elapsed time is also reasonably accurate on all architectures (cf. Figure 6.15). The plots clearly reflect the increase in cache and TLB misses and their impact on the execution time whenever the number of clusters per pass exceeds the respective limits.

Only for very high cardinalities on the Origin2000, it is slightly too low. Here, the complete amount of data to be handled reaches the capacity of a single CPU board. Thus, parts of the allocated memory are likely to be physically located on another board, requiring either remote memory accesses or process migration to the other board. The occurrence of such events is hard to predict, hence, our model does not include the additional costs for them.

The question remaining is how to distribute the number of Radix-Bits over the passes. We conducted another number of experiments, using a fixed number of passes, but varying the number of Radix-Bits per pass. Figure 6.16 depicts the respective results for 4, 8, 12, 16, 20, and 24 Radix-Bits, using 2 passes. The x-axis shows $B + \frac{B_1}{5}$, hence, for each number of Radix-Bits ($B = B_1 + B_2$) there is a short line segment consisting of $B-1$ points. The first (leftmost) point of each segment represents $B_1 = 1, B_2 = B-1$, the last (rightmost) point represents $B_1 = B-1, B_2 = 1$. The results show, that even distribution of Radix-Bits ($B_1 \approx B_2 \approx \frac{B}{2}$) achieves the best performance.

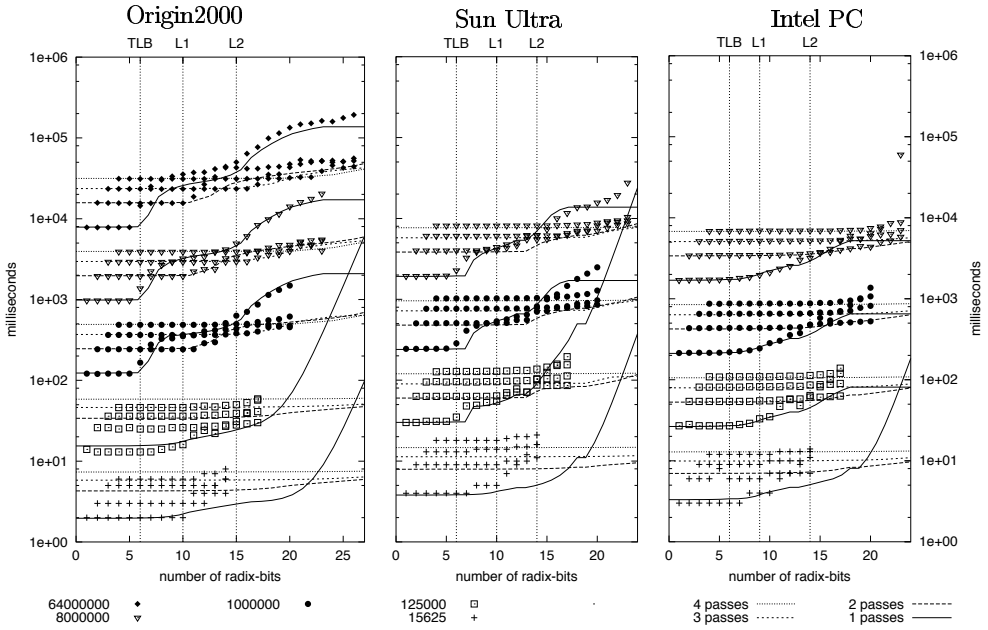

Figure 6.15: Measured (points) and Modeled (lines) Performance of Radix-Cluster

Isolated Join Performance

We now analyze the impact of the number of Radix-Bits on the pure join performance, not including the clustering cost. With 0 Radix-Bits, the join algorithm behaves like a simple non-partitioned Hash-Join.

The Partitioned Hash-Join exhibits increased performance with increasing number of Radix-Bits. Figure 6.17 shows that this behavior is mainly caused by the memory costs. While the CPU cost is almost independent of the number of Radix-Bits, the memory cost decrease with increasing number of Radix-Bits. The performance increase flattens after the point where the entire inner cluster (including its hash table) consists of less pages than there are TLB entries (64). Then, it also fits the L2 cache comfortably. Thereafter, performance increases only slightly until the point that the inner cluster fits the L1 cache. Here, performance reaches its maximum. The fixed overhead by allocation of the hash-table structure causes performance to decrease when the cluster sizes get too small and clusters get very numerous. Again, the PC shows a slightly different behavior. TLB cost does not play any role, but "partial stalls" (i.e. stalls due to dependencies among instructions) are significant with small numbers of Radix-Bits. With increasing numbers of clusters, the partial stalls decrease, but then, resource stalls increase, almost neutralizing the memory optimization.

Like with Radix-Cluster, once the memory access is optimized, the execution of Partitioned Hash-Join is dominated by CPU cost. Hence, we applied the same optimizations as above. We inlined the hash-function calls during hash build and hash probe as well as the compare-function call during hash probe and replaced two `radix` by simple assignments, saving five function calls per iteration. Further, we replaced the modulo division ("%") for calculating the hash index by a bit operation ("&"). Figure 6.18 depicts the original implementation of our Hash-Join routine and the optimizations

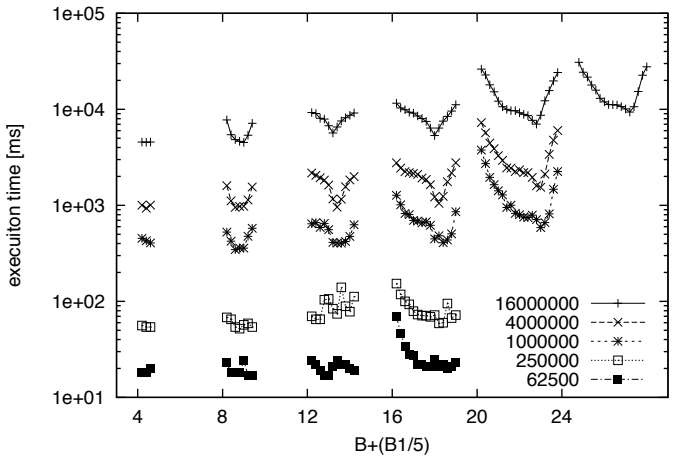

Figure 6.16: Bit-distribution for 2-pass Radix-Cluster

applied.

Figure 6.19 shows the execution time breakdown for the Optimized Partitioned Hash-Join. For the same reasons as with Radix-Cluster, the CPU cost are reduced by almost factor 4 on the Origin and the Sun, and by factor 3 on the PC. The expensive divisions have vanished completely. Additionally, the dependency stalls on the PC have disappeared, but the functional unit stalls remain almost unchanged.

As for the Radix-Cluster, we also provide a cost model for the Partitioned Hash-Join. The model takes the number of Radix-Bits, the cardinality[8], and the (average) repeat rate of the join column values (i.e. the join hit rate) as input.

$$T_h(B,C,r) = C * w_h + M_{L1,h}(B,C,r) * l_{L2} + M_{L2,h}(B,C,r) * l_{Mem} + M_{TLB,h}(B,C,r) * l_{TLB}$$

with

$$M_{Li,h}(B,C,r) = (2+r) * |Re|_{Li} + \begin{cases} C * \frac{||Cl||}{||Li||}, & \text{if } ||Cl|| \leq ||Li|| \\ C * (4+2r) * \left(1 - \frac{||Li||}{||Cl||}\right), & \text{if } ||Cl|| < ||Li|| \end{cases}$$

and

$$M_{TLB,h}(B,C,r) = (2+r) * |Re|_{Pg} + \begin{cases} C * \frac{||Cl||}{||TLB||}, & \text{if } ||Cl|| \leq ||TLB|| \\ C * (4+2r) * \left(1 - \frac{||TLB||}{||Cl||}\right), & \text{if } ||Cl|| > ||TLB|| \end{cases}$$

$|Re|_{Li}$, $|Re|_{Pg}$, and $|TLB|$ are as above. $||Cl||$, $||Li||$, and $||TLB||$ denote (in byte) the cluster size, the sizes of both caches ($i \in \{1,2\}$), and the memory range covered by $|TLB|$ pages, respectively.

w_h represents the pure CPU costs per tuple for building the hash-table, doing the hash lookup and creating the result. We calibrated $w_h = 600$ns on the Origin2000, $w_h = 1100$ns on the Sun, and $w_h = 711$ns on the PC (including resource stalls).

The first term of $M_{Li,h}$ equals the minimal number of Li misses for fetching both operands and storing the result. The second term counts the number of additional Li

[8]For simplicity of presentation, we assume the cardinalities of both input relations to be equal.

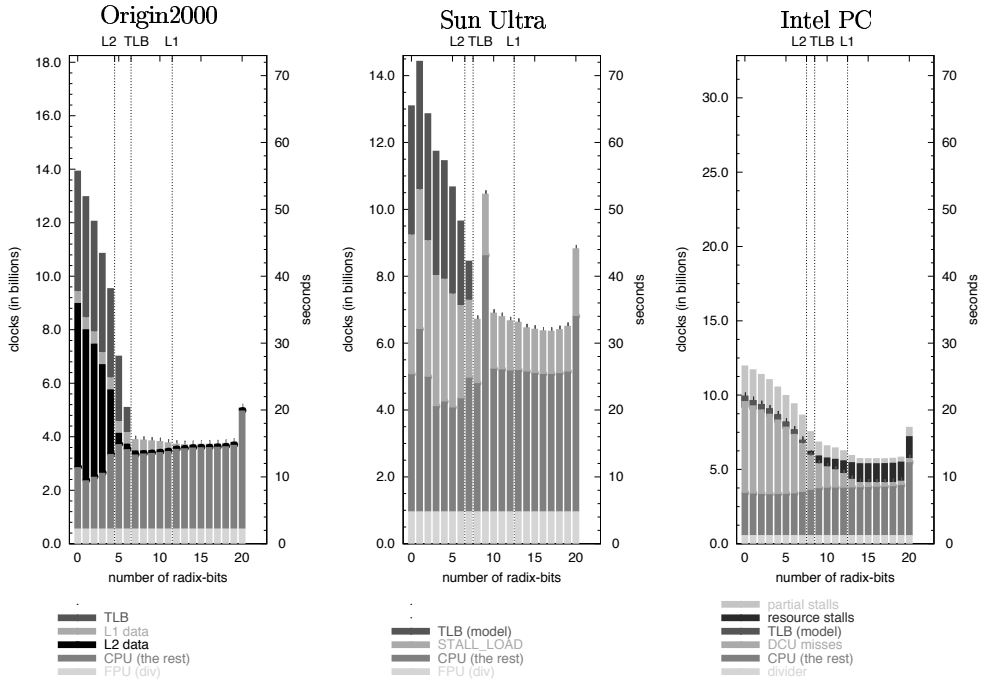

Figure 6.17: Partitioned Hash-Join (C = 8M)

misses, when the cluster size either approaches Li size or even exceeds this. As soon as the clusters get significantly larger than Li, each memory access yields a cache miss due to cache trashing: 4 memory accesses per tuple for accessing the outer relation and the bucket array during hash build and hash probe, and 2 memory access per join hit to access the inner relation and the chain-lists. The number of TLB misses is modeled analogously.

Figures 6.20 and 6.21 confirm the accuracy of our model (lines) for the number of L1, L2, and TLB misses on the Origin2000, and for the elapsed time on all architectures.

Overall Join Performance

After having analyzed the impact of the tuning parameters on the clustering phase and the joining phase separately, we now turn our attention to the combined cluster and join cost. Radix-cluster gets cheaper for less Radix-Bits, whereas Partitioned Hash-Join gets more expensive. Putting together the experimental data we obtained on both cluster- and join-performance, we determine the optimum number of B for relation cardinality.

It turns out that there are three possible strategies, which correspond to the diagonals in Figure 6.21:

phash L2 Partitioned Hash-Join on $B = log_2(C * 12/||L2||)$ clustered bits, so the inner relation plus hash-table fits the L2 cache. This strategy was used in the work of Shatdahl et al. [SKN94] in their Partitioned Hash-Join experiments.

phash TLB Partitioned Hash-Join on $B = log_2(C * 12/||TLB||)$ clustered bits, so

```
hash_join(bun *dst, bun *end                                  /* start and end of result buffer */
 bun *outer, bun *outer_end, bun *inner, bun* inner_end,      /* inner and outer relations */
 int R                                                        /* radix bits */
){
  /* build hash table on inner */
  int pos=0, S=inner_end-inner, H=log2(S), N=2^H, M=(N-1)≪R;
  int next[S], bucket[N] = { -1 }; /* hash bucket array and chain-lists */
  for(bun *cur=inner; cur<inner_end; cur++) {
    int idx = ((*hashFcn)(cur→v2)≫R) % N;          ||int idx = HASH(cur→v2) & M;
    next[pos] = bucket[idx];
    bucket[idx] = pos++;
  }
  /* probe hash table with outer */
  for(bun *cur=outer; cur<outer_end; cur++) {
    int idx = ((*hashFcn)(cur→v2)≫R) % N;          ||int idx = HASH(cur→v2) & M;
    for(int hit=bucket[idx]; hit≥0; hit=next[hit]) {
      if ((*compareFcn)(cur→v2,inner[hit].v2)==0) {||if ((cur→v2 == inner[hit].v2)) {
        memcpy(&dst→v1,&cur→v1, sizeof(int));      ||  dst→v1 = cur→v1;
        memcpy(&dst→v2,&inner[hit].v1, sizeof(int));||  dst→v2 = inner[hit].v1;
        if (++dst≥end) REALLOC(dst,end);
      }
    }
  }
}
```

Figure 6.18: C language hash-join with annotated CPU optimizations (right)

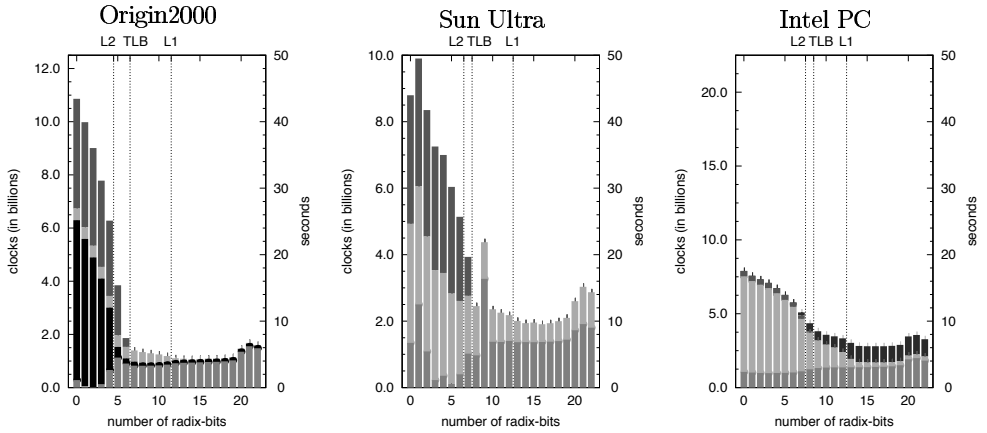

Figure 6.19: Optimized Partitioned Hash-Join (C = 8M)

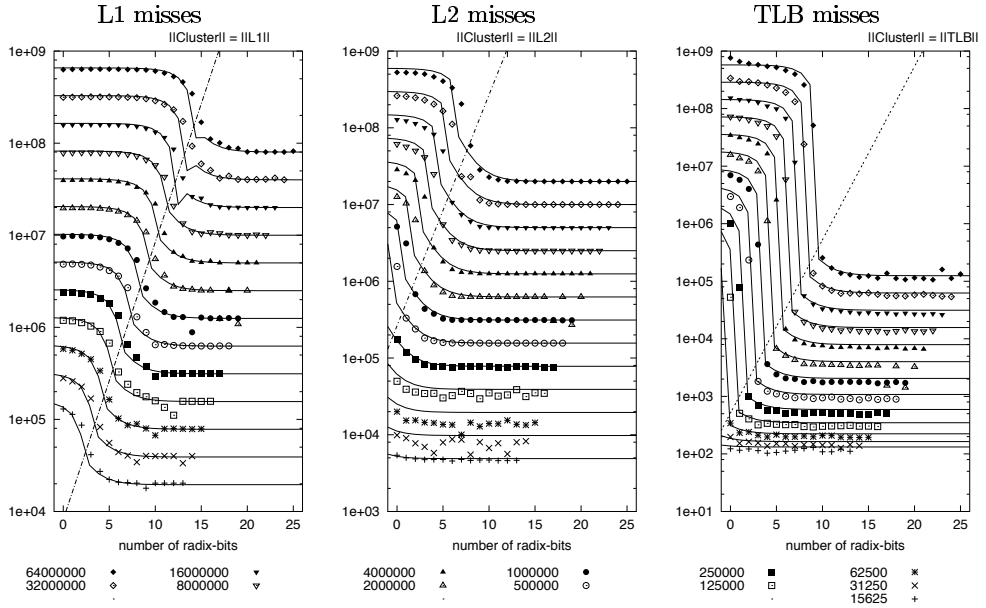

Figure 6.20: Measured (points) and Modeled (lines) Events of Partitioned Hash-Join (Origin2000)

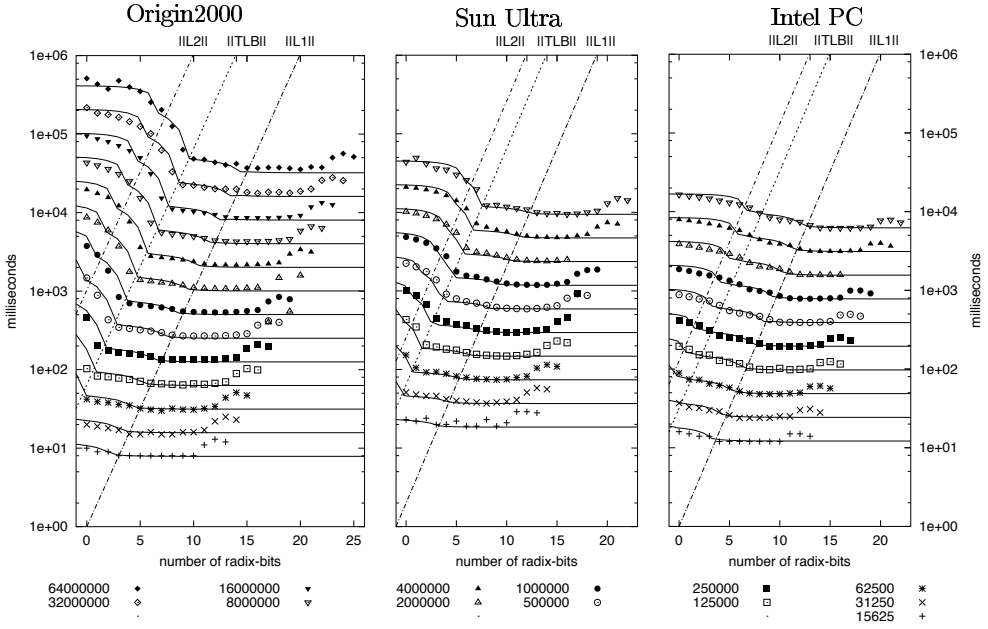

Figure 6.21: Measured (points) and Modeled (lines) Performance of Partitioned Hash-Join

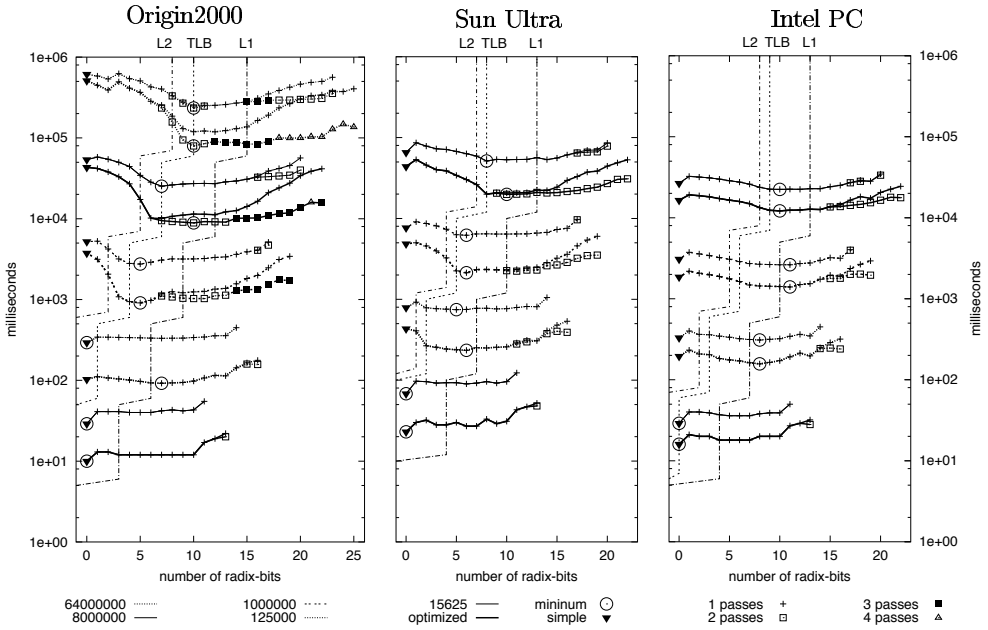

Figure 6.22: Overall Performance

the inner relation plus hash-table spans at most $|TLB|$ pages. Our experiments show a significant improvement of the pure join performance between phash L2 and phash TLB.

phash L1 Partitioned Hash-Join on $B = log_2(C*12/||L1||)$ clustered bits, so the inner relation plus hash-table fits the L1 cache. This algorithm uses more clustered bits than the previous ones, hence it really needs the multi-pass Radix-Cluster algorithm (a straightforward 1-pass cluster would cause cache trashing on this many clusters).

Figure 6.22 shows the overall performance for the original (thin lines) and the CPU-optimized (thick lines) versions of our algorithms, using 1-pass and multi-pass clustering. In most cases, phash TLB is the best strategy, performing significantly better than phash L2. On the Origin2000 and the Sun, the differences between phash TLB and phash L1 are negligible. On the PC, phash L1 performs sightly better than phash TLB. With very small cardinalities, i.e. when the relations do not span more memory pages than there are TLB entries, clustering is not necessary, and the non-partitioned Hash-Join ("simple hash") performs best.

Further, these results show, that CPU and memory optimization support each other and magnify their effects. The gain of CPU optimization for phash TLB is bigger than that for simple hash, and the gain of memory optimization for the CPU-optimized implementation is bigger than that for the non-optimized implementation. E.g., for large relations on the Origin 2000, CPU optimization reduces the execution time of simple hash by approximately 20%, whereas it yields 66% with phash TLB. Analogously, memory optimization achieves a reduction of slightly less than 60% for the original implementation, but more than 80% for the optimized implementation. Combining both optimizations reduces it by almost 90% (a factor 10 of improvement).

The same can be observed for the Sun Ultra, although the absolute gains are somewhat smaller due to the fact that the Ultra CPU is so slow that trading memory for CPU is not as beneficial as on the Origin2000.

The overall effect of our optimizations on the PentiumIII is just over 50%. One cause of this is the low memory latency on the PC, that limits the gains when memory access is optimized. The second cause is the appearance of the "resource-stalls"; which surge in situations where all other stalls are eliminated (and the RISC chips are really steaming). We expect, though, that future PC hardware with highly parallel IA-64 processors and new Rambus memory systems (that offer high bandwidth but high latencies) will show a more RISC-like performance on our algorithms.

6.5 Join Processing With Projections

While the Section 6.4 provides insight in the performance behavior of the various join algorithms, any real-life RDBMS join query goes accompanied by some projection of non-join columns into the result. The precise way in which this is done strongly differs between "traditional" relational query processing and the vertically fragmented query processing in Monet.

In this Section, we investigate optimization of CPU- and memory-resources of equi-join *including* projections, as formulated by the following SQL join-project query:

```
SELECT larger.a1, .., larger.aY, smaller.b1, .., smaller.bZ
FROM   larger, smaller
WHERE  larger.key = smaller.key
```

Without loss of genericity, we assume that the "larger" table has the same number of tuples or more than the "smaller" table.

We now discuss what extra query costs are incurred if projection columns are taken into account and discuss algorithms and query processing strategies for optimizing CPU- and memory-resources. In Section 6.5.1 we do this in a "traditional" relational DBMS setting, while in Section 6.5.2 we do the same for Monet, among other by contributing a new cache-conscious query processing algorithm called *Radix-Decluster* that works in conjunction with the earlier described Radix-Cluster. Finally, in Section 6.5.3 we evaluate the performance of the various query processing strategies.

6.5.1 Cache-Conscious Join: "traditional" Strategy

In "traditional" relational query processing, the simplification of omitting costs of projection is *conceptually* trivial as the projection column values are just a bit of "extra luggage" traveling with the tuples that flow through an operator tree. This does not change anything to the join algorithm. In the case of Hash-Join, it means that the projection columns of the inner relation are included in the hash-table that is built when scanning the full inner relation. Then, the full outer relation is scanned, hash-lookup performed, and a result table is produced that consists of the projection column values of both tables.

Similarly, when using Radix-Cluster followed by Partitioned Hash-Join in a relational query engine, the first scan of the Radix-Cluster accesses the full relation; producing a clustered copy that includes the key column(s) plus all projection columns. This wider intermediate relation is then further clustered in any additional passes of the

Radix-Cluster algorithm. Partitioned Hash-Join on the matching clusters works as described just above.

While the algorithms do not change conceptually, the *performance* of a relational engine will be affected by increased CPU and memory access cost:

- as the input relations for Hash-Join or Radix-Cluster are not BATs, but full relations consisting of tuples that span all columns, scanning them causes (much) more memory access, which in case of queries that only project on a minority of the columns, will lead to inefficient cache-line usage. This point was also made in Section 6.3.2.

- relational operator implementations must manipulate tuples in a generic way, and cannot use the Monet code-expansion techniques described in the previous section (which generates a different routine at compile-time for each data-type processed). The reason why this is impossible is that there are too many possible combinations of tuple constellations to generate routines for (the number of key columns and projection columns, and their types cause a combinatorial explosion). As a consequence, an interpreter mechanism must be used in a relational engine, that examines each tuple and calls a type-specific method/routine for processing it, which typically leads to at least one hard-to-predict CPU branch (function call) for each value in a tuple. This again leads to mis-prediction stalls and memory stalls (due to lack of parallel memory access). On various kinds of query loads in relational DBMS technology, low levels of Instruction Level Parallelism (ILP) have been demonstrated – often barely one instruction per cycle, while modern CPUs are capable of 5 or more – due to such stalls, which themselves often account for 30%-50% of execution time alone [ADHW99, KPH+98, BGB98, TLPZT97].

In order to measure the performance of "traditional" (Partitioned) Hash-Join in a relational context while using Monet as our experimentation vehicle, we created the new `integerX` Monet atomic types in an extension module, for all $X \in \{1, 4, 16, 64, 256\}$. An `integerX` value models a relational tuple that stores X simple integer column values. The implementation of the atom ADT routines for this type try to mimic the behavior of a multi-column relational engine; e.g., copying an `integer8` during Partitioned Hash-Join or Radix-Cluster involves copying 8 integers with `memcpy` from soft-coded record offsets. The reason for this is that at DBMS kernel compile time, a relational engine can make no hard assumptions on the table formats that pass through the algebraic operators. Therefore, value handling is more interpretative and at least involves separate ADT calls for each column value. In the case of Monet, the fact that each operator works on a fixed table type (i.e. 2-column tables) enables it to hard-compile many optimizations such that typically the inner loop of a join algorithms does not contain a single method call (like described in Figures 6.11 and 6.18).

For experimentation convenience, we introduce in the same extension module a conversion command

```
[integer](BAT(any,integerX), int Y) : BAT[any, integerY]
```

that constructs a new BAT with a thinner or wider `integerX` column, either by omitting values, or by padding them. In the case of thinning (i.e. `Y<X`) a read-only BAT,

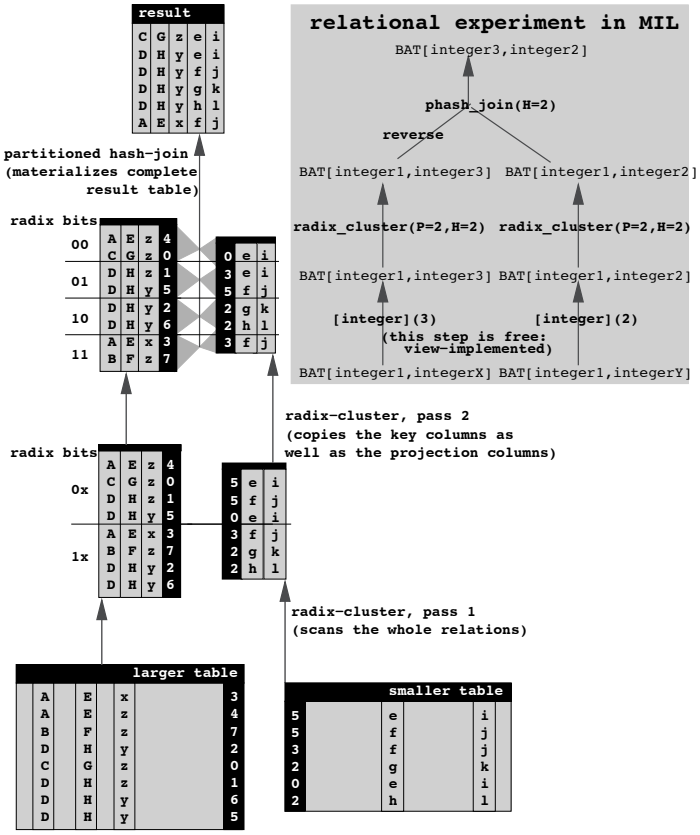

Figure 6.23: Relational Partitioned Hash-Join Strategy

the command optimizes performance by constructing a *view* on the original BAT at zero cost (see Section 5.3.2).

The basic non-partitioned relational Hash-Join strategy would be coded in MIL like this:

```
01  smaller_all := [integer]([integer](smaller_key,1).reverse, 128).reverse;
02  larger_all := [integer]([integer](larger_key,1).reverse, 128).reverse;

03  smaller_project := [integer](smaller_all.reverse, 8).reverse;
04  larger_project := [integer](larger_all.reverse, 8).reverse;

05  res_join := join(larger_project, smaller_project.reverse);
```

In the example case above, we use "smaller" and "larger" tables that each have 128 columns, and the projection widths are Y=Z=8; we emulate relational storage of both tables in Monet by storing them as `BAT[integer128,integer1]`, where the tail columns contain the "key" value and the head contains all other columns.

Notice that the fact that `smaller_project` and `larger_project` are MIL views on the base BATs `smaller_all` and `larger_all`, means that the projection is not materialized. Projection only occurs implicitly when the join in line 05 accesses both views – just like what would happen in a "traditional" relational system performing projection. Furthermore, the copying of each `integer8` (view) value from its storage as `integer128`

is done with 8 `memcpy` calls that fetch values at regular intervals (i.e. at positions 0, 16, 32, 48, 64, 80, 96 and 112). In practice, this means that each `memcpy` causes a memory cache miss.

We can encode the Partitioned Hash-Join in the relational case by modifying the latter part of the script:

```
..
05  cluster_smaller := radix_cluster(smaller_project, P, H);
06  cluster_larger := radix_cluster(larger_project, P, H);
07  res_join := phash_join(cluster_larger, cluster_smaller.reverse, H);
```

Here we use primitives that are introduced in the `radix` extension module, that introduce the Radix-Cluster and Partitioned Hash-Join algorithms as new MIL operators. The first parameter of `radix_cluster` is a BAT that is to be clustered on its head column, its output is a materialized result BAT, and the other parameters are the number of passes and the number of Radix-Bits. As mentioned before, the number of Radix-Bits are divided equally over all passes (with a maximum difference of one due to the Radix-Bits not being an exact multiple of the number of passes). The `phash_join` is the implementation of the Partitioned Hash-Join that assumes inputs with the join columns clustered in `H` Radix-Bits, which is the additional third parameter.

This relational strategy is expected to have a different performance characteristic than the join experiments described in the previous Section, which did not take projection columns into account. First, there will be many more cache misses due to the reasons described above. Even Radix-Cluster cannot avoid those cache misses, as it is inherent to the relational storage format of base data. Second, there will be much more CPU cost, due to the fact that function-call overhead cannot be eliminated. In Monet algorithms, CPU optimizations cause a four-fold performance improvement, as described in Section 6.4.2. Third, the fact that the projection columns are taken as "extra luggage" with the join keys through the Partitioned Hash-Join means that the tuple size becomes larger: instead of clusters with `[key,oid]`, we have `[key,oid,integerX]` tuples in the clusters. As the clusters are tuned to fit into the cache, the clusters can hold less tuples, so we need to create more (and smaller) clusters in the Radix-Cluster phase. This may even cause the Radix-Cluster to make additional passes, and certainly will increase its cost, as we always have observed that to be monotonically increasing with the amount of Radix-Bits.

We evaluate the performance of this strategy in Section 6.5.3.

6.5.2 Cache-Conscious Join: Monet Strategies

As for join processing in Monet with column projections, the situation is also more complex than described in Section 6.4, though for different reasons than in a "traditional" relational query engine. In the case of Monet, a join query is processed by first constructing a join-index `BAT[oid,oid]`. This join-index is `mark`-ed on both sides to form to "pivot" `BAT[void,oid]` with a newly numbered head column (a dense sequence $0..N-1$) and in the tail column the tuple IDs (`oid`-s) from either input relation. Subsequently, the projections are materialized by joining the pivot of the input relation with each projection column-`BAT[void,T]`. This is depicted in Figure 6.24.

This strategy would yield the following MIL statements for our basic join-project SQL query:

```
01
```

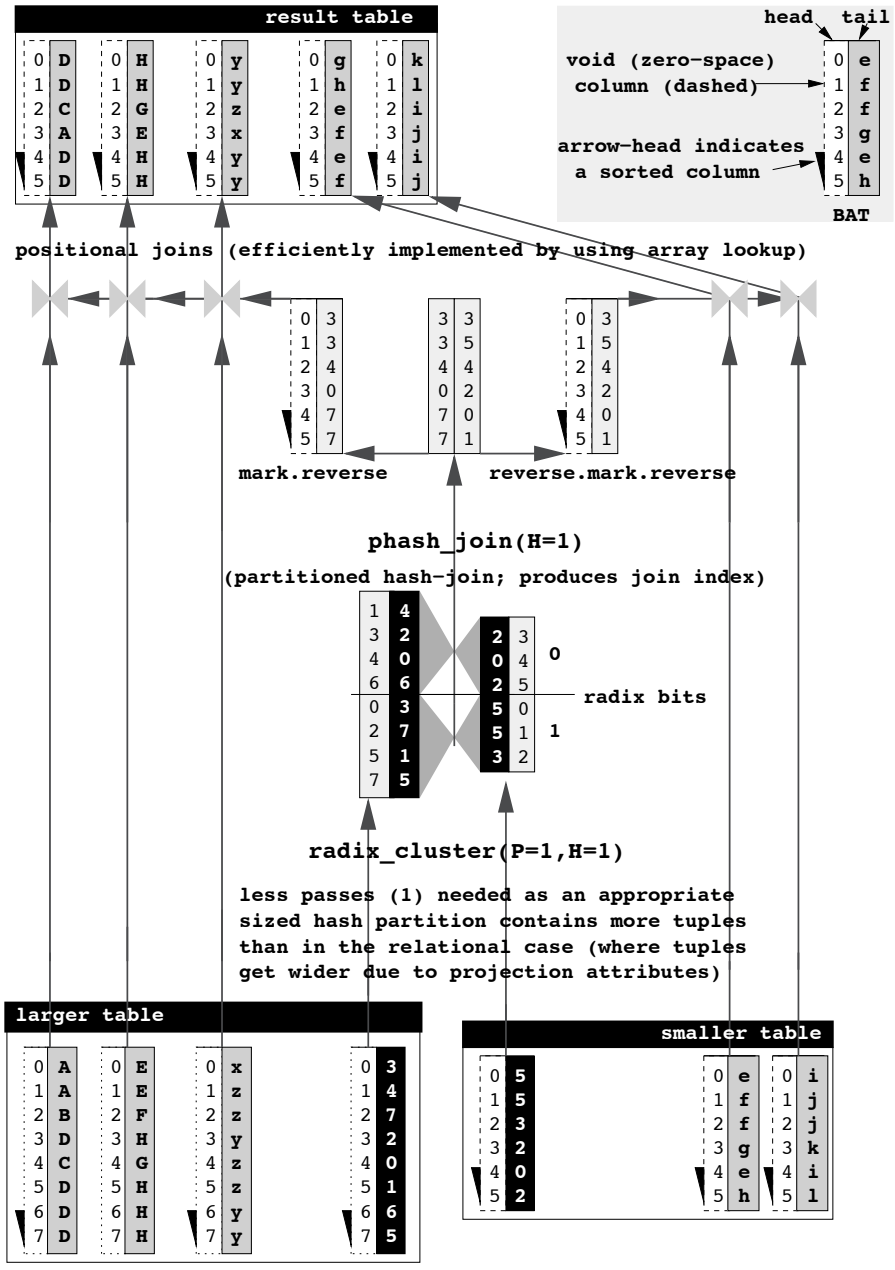

Figure 6.24: Standard Monet Partitioned Hash-Join Strategy

```
02  # the physical join algorithm is either positional-, merge- or Hash-Join.
03  res_join := join(larger_key, smaller_key.reverse);
04  res_larger := res_join.mark.reverse;
05  res_smaller := res_join.reverse.mark.reverse;

    # positional-join projected columns from smaller table into result
A1  res_a1 := join(res_smaller, smaller_a1);
AX  ....
AY  res_aY := join(res_smaller, smaller_aY);

    # positional-join projected columns from larger table into result
B1  res_b1 := join(res_larger, larger_b1);
BX  ....
BZ  res_bZ := join(res_larger, larger_bZ);
```

The join-index is named `res_join`, the pivots `res_larger` and `res_smaller`, the projection columns `smaller_a`$_x$ and `larger_b`$_x$.

A Positional-Join is a highly efficient kind of join found in the Monet system, that occurs when an `oid`-column is joined with a `void` column. It is easy to lookup a value in a `void`-column, as the value you look up already tells its position (a search accelerator like hash-table or B-tree is not necessary, and CPU-cost is very low). The Positional-Join algorithm joins an outer `BAT[any,oid]` with an inner `BAT[void,any]` by scanning over the outer-BAT and performing positional lookup into the inner BAT.

In a typical data warehouse, the join at line 03 would be positional if the "key" columns are foreign keys between tables with a 1-1, 1-N or N-1 relationship (in those cases, one of the key columns would be of type `void`). However, if the columns are a N-M relationship, or if they do not form a foreign key at all, we could use the Partitioned Hash-Join as described earlier in this chapter, as this is a generic join algorithm that optimizes use of memory- and CPU-resources.

The Partitioned Hash-Join uses Radix-Cluster to quickly cluster both the `smaller_key` and `larger_key` BATs into clusters that fit the memory cache, and then repeatedly performs Hash-Join on the corresponding clusters. So instead of the simple MIL script above, we use the following alternative MIL statements to generate `res_join`:

```
00  # first radix cluster both key columns on H bits in P passes
01  cluster_larger := radix_cluster(larger_key, P, H);
02  cluster_smaller := radix_cluster(smaller_key, P, H);

    # partitioned hash join on clusters of H Radix-Bits.
03  res_join := phash_join(cluster_larger, cluster_smaller.reverse, H);
..
```

The latter phase of the query (lines A1-AY,B1-BZ) fetches column values from the projected columns using Positional-Join.

```
..
BX  res_bx := join(res_larger, larger_bx);
..
```

This performs fine with increasing table sizes of the larger table up until one `larger_b`$_x$ column-BAT exceeds the size of the memory cache.

We now turn our attention to what happens if this happens. First, we discuss what happens if a `larger_b`$_x$ BAT[9] does not fit the memory cache. Then, we discuss what happens if even a `smaller_a`$_y$ BAT plus its hash-table does not fit anymore.

[9]These column-BATs may differ in tail-type and hence byte-width, but for simplicity we assume all have approximately the same size.

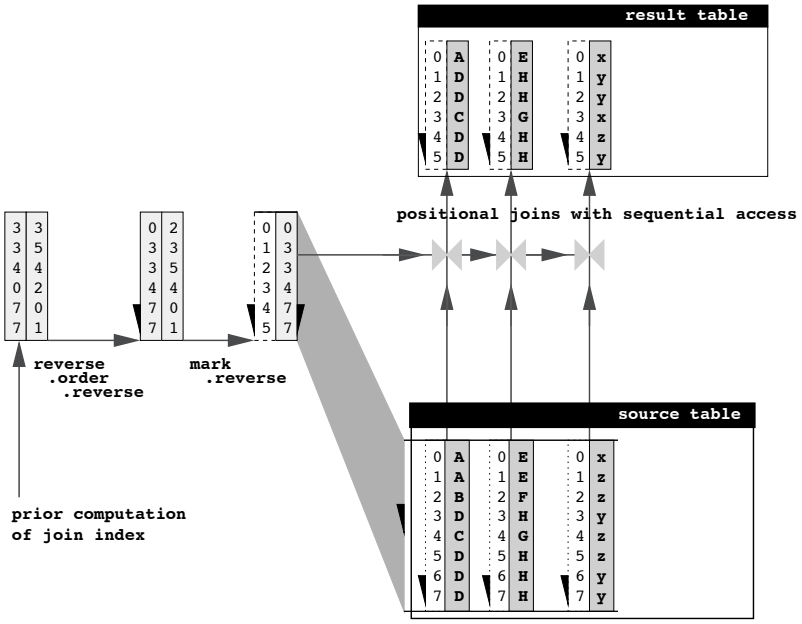

Figure 6.25: Memory-Efficient Projection Joins Using Sorting

Ordering The Join Index to Improve Memory Access of Projections

If the BATs storing the columns of the "larger" table do not fit the memory cache anymore, the Positional-Joins in the last X statements of the MIL script will start to generate cache misses. This is caused by the fact that the `oid`-s in the tail of the `res_larger BAT[void,oid]`-s are not sorted; hence the access to the `larger_b`$_x$ column-BATs is random.

The *sorted* projection strategy solves this problem, by sorting the result of the join first on the `oid`-s that point to the "larger" table (the head column of `res_join`):

```
..
03  res_join := join(larger_key, smaller_key.reverse);
    res_join_ordered := res_join.reverse.order.reverse;

04  res_larger_reordered := res_join_reordered.mark.reverse;
05  res_smaller := res_join_reordered.reverse.mark.reverse;
..
    # positional-join projected columns from larger table into result
B1  res_b1 := join(res_larger_reordered, larger_b1);
BX  ....
BZ  res_bZ := join(res_larger_reordered, larger_bZ);
```

As a result, the `res_larger_reordered` will be a `BAT[oid,oid]` ordered on tail, hence the Positional-Joins on the `larger_b`$_x$ columns will cause a nice sequential access to both `res_larger` (as it is scanned in its role as "outer" join operand) and `larger_b`$_x$. We must, however, take into account that `res_join_reordered` may be a `BAT[oid,oid]` that itself is larger than the memory cache, in which case the sorting operation itself could have caused a great many cache misses itself, and therefore perform badly. Let us therefore shortly discuss the memory access properties of various sorting algorithms.

Monet uses a CPU-optimized Quick-Sort algorithm for sorting large relations. The CPU-optimizations reduce the amount of function calls, by doing all value-comparison

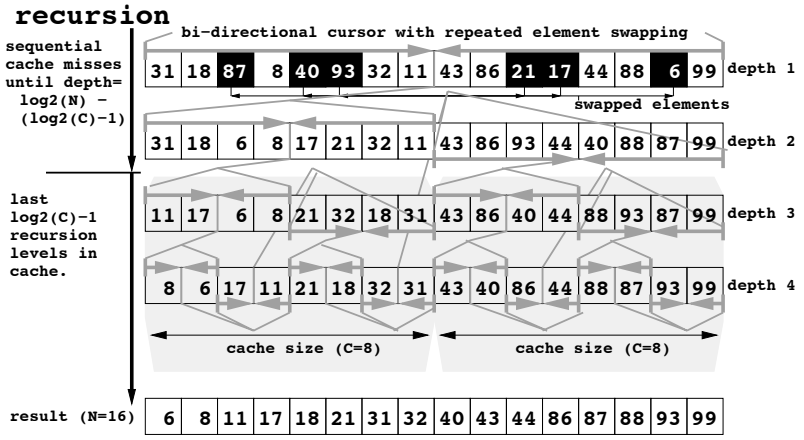

Figure 6.26: The Memory Access Pattern of Quick-Sort

and data movement inline, using C macros. In this sense it differs from the standard unix library call `qsort`, as that routine compares values with a user-provided function, and (often) moves values with `memcpy` (see also Section 5.13).

The memory access pattern of the Monet Quick-Sort consists of one sequential scan per recursion level (walking two cursors simultaneously, one from the start of the BAT forward, as well as another from the end of the BAT backward, until both meet in the middle). This is depicted in Figure 6.26. Quick-Sort is binary recursive and therefore takes $\log_2(ntuples)$ recursion levels to sort a BAT, hence its total memory access consists of $\log_2(ntuples)$ sequential scans. However, since Quick-Sort zooms into ever smaller sub-chunks of the BAT, there will be cache re-use in the deeper recursion levels as soon as such a chunk fits the memory cache, which happens when $sizeof(chunk) = sizeof(BAT/(2^{level}) \leq sizeof(cache)$. Hence, the total memory cost of Quick-Sort is $\log_2(ntuples) - \log_2(sizeof(cache)/sizeof(tuple))$ sequential scans.

This $O(N \log(N))$ complexity – with rather low $\log(N)$ – means that the Monet Quick-Sort implementation behaves quite good both concerning CPU efficiency and memory access pattern. Still, for some simple data types, in particular columns containing `oid`-s, one can further improve the memory access performance by using *Radix-Sort* instead of Quick-Sort.

Radix-Sort is essentially a Radix-Cluster on all bits, hence we do:

```
...
03  res_join := join(larger_key, smaller_key.reverse);
    res_join_ordered := res_join.reverse.radix_cluster(P1, H1).reverse;
...
```

Where P_l is a suitable number of Radix-Cluster passes and H_l is here the total number of "significant bits", where we define the most significant bit in a collection of cardinal integer values as the highest bit set in the binary representation of its largest value. The head column of the `join(larger_key, smaller_key.reverse)` is of type `oid`, and contains the `oid`-s from the matching tuples in the "larger" table. Table-`oid`-s are automatically generated by the `void` columns of Monet, and therefore these integer values are from the range $[0, ..., N-1]$, where N is the number of tuples in the "larger" table. We call such an integer sub-domain a "dense" domain. We hence see that in

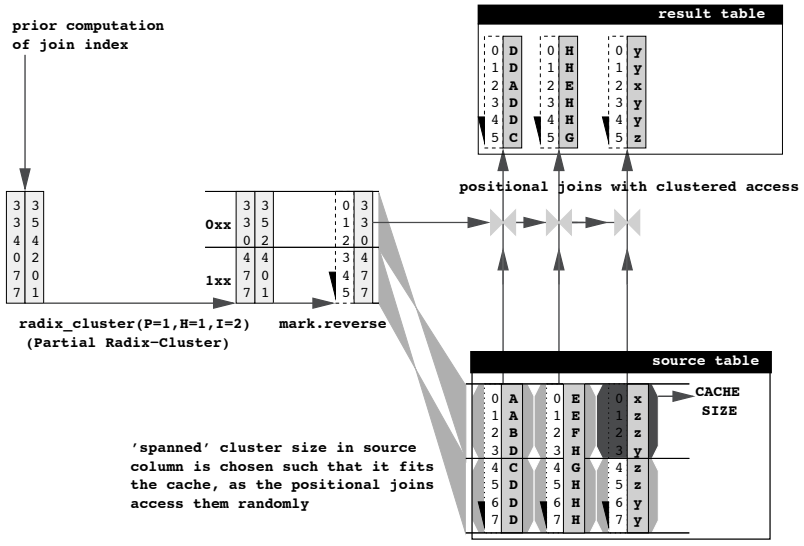

Figure 6.27: Projection Joins Using Partial Radix-Cluster

dense domains, the number of significant bits is minimal (i.e. $H_l = \log_2(N)$ – there are no "spoiled" values), and we do not expect skew in such a column. This motivates our choice to implement Radix-Cluster for the `oid` type by getting Radix-Bits *without* hashing (for all other types, we hash first). Hashing is not necessary due to absence of value-skew on `oid` columns, and absence of hashing allows us to use Radix-Cluster as Radix-Sort.

Going one step further, the *partial-cluster* projection join strategy supplants Radix-Cluster on all significant bits (i.e., Radix-Sort), by a Radix-Cluster on less bits. For this purpose, we added the possibility to indicate to the Radix-Cluster to ignore a certain number of lower bits (by passing the number of Radix-Bits to ignore as an extra last parameter to `radix_cluster`). This in fact breaks off Radix-Sort, leaving the relation unsorted on the lowermost bits. The relation is sorted, though, on the higher-most bits (i.e. partially ordered). This partial ordering means that in each cluster all values fall in a certain disjunct range. When entering a Positional-Join, this means that each cluster will only fetch join values in the column-BAT in a certain range (or cluster). If these "virtual" clusters in the column-`BAT[void,T]` fit in the memory cache, then the Positional-Join will run well (i.e., not thrash the cache). Note that the maximum size for which this is the case also depends on the byte-width of type `T`. The benefit of this "partial-cluster" join strategy (depicted in Figure 6.27) is that it has the potential to optimize memory performance of the column fetching using Positional-Joins just as well as a full sort, but at a clustering cost that is less than the cost of a full sort.

The detailed performance evaluation for using Quick-Sort or (full) Radix-Cluster is found in Section 6.5.3.

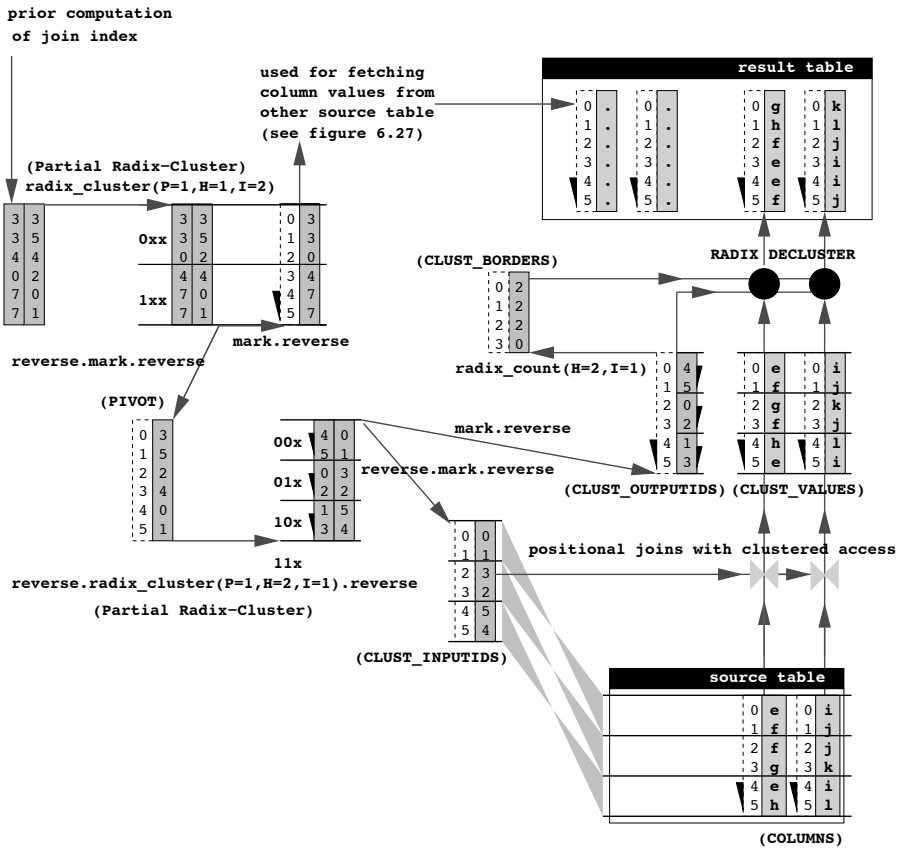

Figure 6.28: Fetching Projectings Efficiently Using Radix-Decluster

The Radix-Decluster Algorithm and its Uses

Let us now returnto the original motivation for discussing sorting, which was to sort the join result (join-index) on the `oid`-s of the "larger" relation, if its column-`BAT[void,T]`-s were too large to fit the cache (which would result in bad memory performance with an unsorted pivot, due to the random access pattern). However, in a generic project-join query, there are not only projections to be done from the "larger" table, but also from the "smaller". Hence, if the column-`BAT[void,T]`-s of the "smaller" table are also larger than the memory cache, the same problem occurs for the positional joins to these column-BATs. It is clear that the join-index cannot simultaneously be sorted on `oid`-s of the "larger" table *and* on those of the "smaller" table. Hence, the sorting of the join-index on the "larger" table must still be done first. One could possibly re-order the pivot for the "smaller" table (which is created by `mark` on the join-index) afterwards:

```
..
05  res_smaller_sorted := res_join.reverse.mark.reverse.radix_cluster(Ps,Hs);
..
```

However, this approach only transfers the problem to later phases of query processing, as follows. The Positional-Joins of `res_smaller_sorted` into the `smaller_a`$_y$ column-

BAT[void,T]-s would run fine, but as a tail-sorted res_smaller_sorted would turn into a BAT[oid,oid] (i.e. it would no longer have a void head column), the result of these Positional-Joins would be of the form BAT[oid,T]. These results would not only take more space than the desired form BAT[void,T], but would also create a problem in further use of the query result, as these res_a_y BATs will not be sorted on head. Join access to them would go to the Hash-Join rather than the Positional-Join, and due to the random access this would pose a memory caching problem as these res_a_y BATs may well be larger than the memory cache.

As a solution, we propose the use of a new memory-conscious join strategy for these projection joins, that is called *cluster-decluster*. This strategy, which is illustrated in Figure 6.28, consists of three phases:

1. **Partial Radix-Cluster:**

   First, the res_smaller is *partially* Radix-Clustered on tail-oid. That is, relation PIVOT is clustered on some number of Radix-Bits, but not on all (significant) Radix-Bits. The number of bits chosen for the practical ordering is the minimal number of bits such that in a Positional-Join the oid-s in each partially sorted cluster point to a memory range in COLUMN that fits the memory cache. For example, if we have a memory cache of 64KB and we assume values to be 4 bytes wide, then a cluster of 16,384 tuples would just fit. If the source table from where the projections come has 10M tuples, we would create $2^{10} = 1024$ clusters to arrive at a mean cluster size of 10,000 (which would be the largest cluster size $< 16{,}384$). Such clusters can be created with a partial Radix-Cluster on the highest significant 10 bits (i.e. bits 24-15, as $\log_2(10\text{M}) = 24$).

2. **Clustered Positional-Join:**

   The purpose of the Radix-Clustering PIVOT in the previous phase is to accelerate the Positional-Join between PIVOT and COLUMN (e.g. res_smaller and smaller_a_x). Because the oid-s in the tail of PIVOT are now partially sorted, each chunk in PIVOT will only randomly access data from one "cluster" in COLUMN. As we have chosen the partial ordering bits in such a way that this randomly accessed region fits the memory cache, during Positional-Join these regions in COLUMN stay memory resident, and Positional-Join will cause much less cache misses than with an unclustered PIVOT. The result of joining the clustered PIVOT with COLUMN is a BAT[oid,T]. The head type is not void because the head type of the clustered PIVOT is not either. One can additionally optimize performance by splitting the clustered PIVOT by column in two BATs, using a mark on both sides of PIVOT, yielding one BAT[void,oid] called CLUST_OUTPUTIDS, that contains the oid-s of the join result table in the tail, and another BAT[void,oid] called CLUST_INPUTIDS, that contains the oid-s of the smaller (input) table in the tail. This latter BAT is then used to perform the positional join into COLUMN. The advantage here is that the resulting BAT[void,T] - which we call CLUST_VALUES – has a void head column. Recall that this join has the special property that each tuple hits exactly once, and that the result tuples appear exactly in the order of the left join operand, which enables the positional join implementation to produce a result with a void head column if its left operand has one.

3. **Radix-Decluster:**

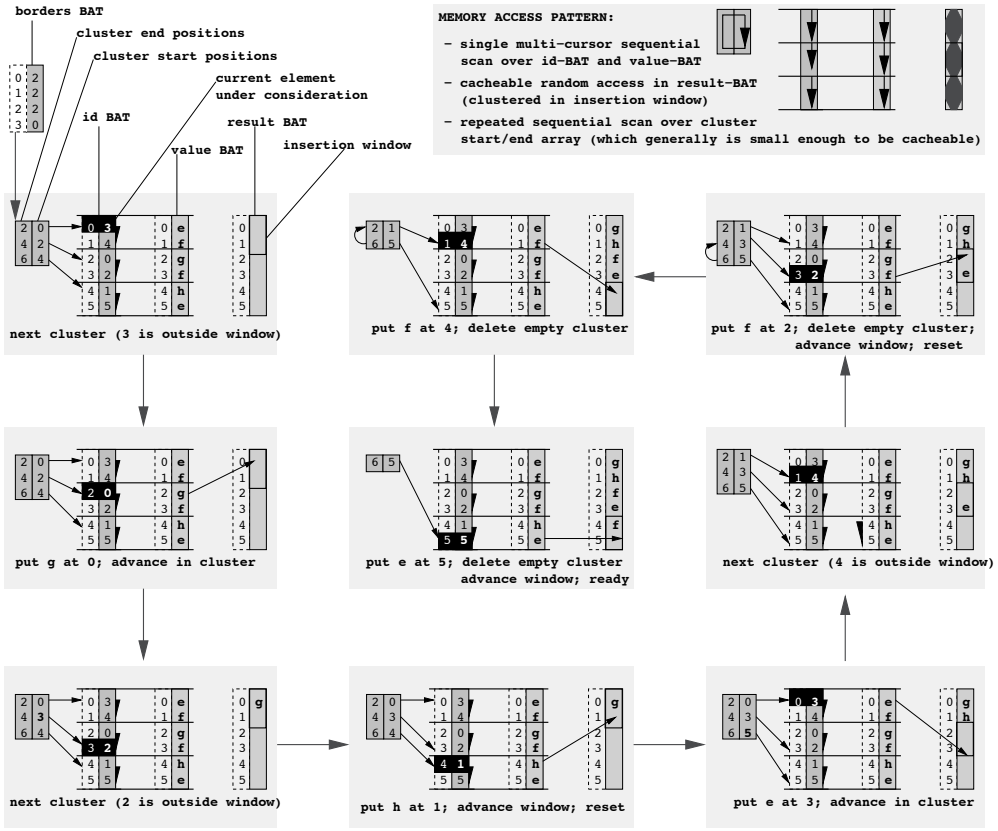

Figure 6.29: The Memory Access Pattern Of Radix-Decluster

In order to complete the operation, we now want to produce the tail values of `CLUST_VALUES` in the order dictated by the tail column of `CLUST_OUTPUTIDS`. We also know that the tail values of `CLUST_OUTPUTIDS` would form a dense sequence $(0, 1, ..N{-}1)$ when sorted (as this column was the result of re-clustering the dense head column of `PIVOT`). This "declustering" can be done efficiently by exploiting the property of the Radix-Cluster algorithm: following a Radix-Cluster on the tail of a BAT that is ordered on head, each cluster in the result will have the head-values still ordered **within** the cluster. So, the tail values within each cluster in `CLUST_OUTPUTIDS` appear in order. Due to the fact that the clusters are sorted, a full sort can be efficiently implemented as a **merge**. This merge operation processes all clusters of `[oid,T]` tuples, holding output-ids and column values, that are stored in the two BATs `CLUST_OUTPUTIDS` and `CLUST_VALUES`. The result of merging all tuples on output-ids order is a `BAT[void,T]` because, when sorted, the output-ids form a dense sequence. Normally, the cost of a merge of N tuples partitioned over H sorted clusters is at least $O(\log_2(H)N)$. However, because of the fact that we know beforehand that the sorted sequence becomes dense, we can bring this cost back to $O(N)$!

The *Radix-Decluster* algorithm, depicted in detail in Figure 6.29, works by keeping open an insertion-window of s `oid`-s $[w, w+1, .., w+s-1]$ during the merge.

Each iteration of the algorithm finds *all* the next s `oid`-s in the clusters and appends them to the result `BAT[void,T]`. This is done by going through all (not yet empty) clusters and inserting from the top of each cluster all elements whose head `oid` fits in the window. Due to the fact that all clusters are sorted on head, we are sure to have found all `oid`-s after having processed all clusters once. Then, the window is shifted s positions and the process repeats. The window size s is preferably much larger than the number of clusters, such that per iteration in each cluster multiple tuples fall into this window. Because these multiple tuples are accessed sequentially in both `CLUST_OUTPUTIDS` and `CLUST_VALUES` from the top of the cluster downward, the cache lines that store these BATs are used fully; giving efficient memory performance. The only restriction on s is that the window size in the result `BAT[void,T]` must fit the memory cache, as it is accessed randomly. Pseudo code of the algorithm is in Figure 6.30 while Figure 6.29 gives a detailed example of how it works.

The cluster-decluster strategy is expressed in the following MIL code:

```
..
    # sub-cluster in Ps passes on Hs significant Radix-Bits, ignoring lowest Hi bits
05  res_smaller_clustered := res_join.reverse.mark.reverse.radix_cluster(Ps, Hs, Hi);

    # positional-join and decluster projected columns from smaller table into result
A0  clust_smaller_borders := res_smaller_clustered.radix_count(Hs, Ri);
    clust_smaller_inputids := res_smaller_clustered.reverse.mark.reverse;
    clust_smaller_outputids := res_smaller_clustered.mark.reverse;

A1  res_a1 := join(clust_smaller_inputids, smaller_a1).
              radix_decluster(clust_smaller_outputids, clust_smaller_borders);
AX   ....
AY  res_aY := join(clust_smaller_inputids, smaller_aY).
              radix_decluster(clust_smaller_outputids, clust_smaller_borders);
..
```

The `radix_count` operator analyzes a (partially) Radix-Clustered BAT and returns the actual sizes of the clusters. It returns a `BAT[void,int]` where for each bit pattern, the tail contains the size of the cluster. These sizes are used in the Radix-Decluster to initialize the `cluster` border structure.

The cluster-decluster join strategy is more expensive than the partial-cluster join strategy. Both strategies feature one initial Radix-Cluster, but the former adds an extra Radix-Decluster operation for each projection column. Hence, it will only be used for getting projection columns from the table with cheaper projections, if cache-conscious projecting is already needed for the join input table with the more expensive projections. Which input table in the join has the more expensive projection phase depends on the number of projection columns in both tables, the data types in these projection columns, and the number of tuples in both input tables. In the MIL script for cache conscious query processing given below, we use partial Radix-Cluster for getting the columns from the "larger" table, and the cluster-decluster strategy for getting the projection columns from the "smaller" table:

```
    # first radix cluster both key columns on H bits
01  cluster_larger := radix_cluster(larger_key, L1,..,Ll);
02  cluster_smaller := radix_cluster(smaller_key, S1,..,Ss);

    # phash join on clusters of H Radix-Bits, followed by Radix-Sort on head column.
03  res_join := phash_join(cluster_larger, cluster_smaller.reverse, H);
    res_join.reverse.radix_cluster(R1,..,Rp).reverse;
```

```
<Type>[]
radix_decluster<Type>(
  int                       cardinality, nclusters,
  Type                      value_bat[cardinality],
  oid                       id_bat[cardinality],
  struct { int start, end } cluster[nclusters])
{
  <Type> result_bat[] = malloc(cardinality*sizeof(<Type>));
  int windowLimit, windowSize = CACHESIZE / 2*sizeof(<Type>);

  for(windowLimit = windowSize; nclusters > 0; windowLimit += windowSize) {
    for(int i=0; i < nclusters; i++) {
      while (id_bat[cluster[i].start] < windowLimit) {
        result_bat[id_bat[cluster[i].start]] = value_bat[cluster[i].start];
        if (++cluster[i].start >= cluster[i].end) {
          cluster[i] = cluster[--nclusters]; // delete empty cluster
          if (i >= nclusters) break;
        }
      } // while more cluster elements in window
    } // while more clusters to merge
  } // while more insertion windows to fill result
  return result_bat;
}
```

Figure 6.30: The Radix-Decluster Algorithm in Pseudo C++

```
04  res_larger_sorted := res_join.mark.reverse;

    # sub-cluster on Rs significant Radix-Bits, ignoring lowest Ri bits
05  res_smaller_clust := res_join.reverse.mark.reverse.radix_cluster(-Ri, Rs);

    # positional-join and decluster projected columns from smaller table into result
A0  borders := res_smaller_clustered.radix_count(Hs, Hi);
A1  res_a1 := join(res_smaller_clustered, smaller_a1).radix_decluster(borders);
AX   ....
AY  res_aY := join(res_smaller_clustered, smaller_aY).radix_decluster(borders);

    # positional-join projected columns from larger table into result
B1  res_b1 := join(res_larger_sorted, larger_b1);
BX  ....
BZ  res_bZ := join(res_larger_sorted, larger_bZ);
```

Radix Accelerator

First we show the overloaded `join` MIL procedure that integrates the radix-cluster and partitioned hash-join algorithm seamlessly in the MIL equi-join using some dynamic tactical query optimization: depending on the actual operand sizes and on the global `CACHE_SIZE` and `CACHE_LINES` variables – which are initialized by the Calibrator module at database startup to the size of the performance-wise most significant memory cache level and number of cache lines – it determines whether or not to use partitioned hash-join, which operand is the inner and outer and the optimal Radix-Bits and passes.

```
PROC join(BAT[void,any::1] left, BAT[any::1,any:2] right) : BAT[oid,any:2] {
  VAR right_batsize := right.info.find("batsize");
  VAR left_batsize := left.info.find("batsize");

  IF (min(right_batsize,left_batsize) > CACHE_SIZE) {
    IF (left_batsize > right_batsize) {
      var nbits := 1 + log2(left_batsize/CACHE_SIZE);
      var npasses := 1 + (nbits-1) / CACHE_LINES;
      RETURN phash_join(right.radix_cluster(npasses, nbits).reverse,
                        left.reverse.radix_cluster(npasses, nbits), nbits).reverse;
    } ELSE {
      VAR nbits := 1 + log2(right_batsize/CACHE_SIZE);
      VAR npasses := 1 + (nbits-1) / CACHE_LINES;
```

```
      RETURN phash_join(left.reverse.radix_cluster(npasses, nbits).reverse,
                        right.radix_cluster(npasses, nbits), nbits);
    }
  }
  RETURN join(left, right);
}
```

In order to automatically apply the cluster-decluster join strategy in MIL queries that can benefit from it, the algorithms have also been wrapped in a MIL *radix search-accelerator*. As discussed in Section 4.4.2, such structures are attached to BATs, may be maintained under updates and can be exploited by algebraic operators to accelerate their execution. While the Monet features built-in direct hashing and T-trees, new search accelerators can be added in extensibility modules.

The `radix_cluster(`X, B`)` operator attaches its clustered result Y to the input `BAT[void, oid]` X and in turn attaches `radix_count(`Y, B`)` to Y by use of a new MIL search accelerator. Also, the `radix_cluster(`X`)` directly returns such an attached Y if found on an X input. The same goes for `radix_count(`Y`)`, when it discovers an attached borders BAT on a clustered input Y.

The second overloaded `join` MIL procedure below makes sure that the cluster-decluster join strategy is applied in Positional-Joins (i.e., those with a `void` head column in the right join operand) whenever the input column is not sorted or clustered, and the target column is larger than the memory cache size.

```
PROC join(BAT[void,oid] left, BAT[void,any::1] right) : BAT[void,any::1] {
  VAR right_info := right.info();
  VAR right_batsize := int(right_info.find("batsize"));

  if (int(right_info.find("tail.sorted")) = 0 and right_batsize > CACHE_SIZE) {
    VAR nbits := 1 + log2(right_batsize/CACHE_SIZE);
    VAR nignore := min(0, (1 + log2(left.count)) - nbits);
    VAR npasses := 1 + (nbits-1) / CACHE_LINES;

    VAR cluster := left.radix_cluster(npasses, nbits, nignore); # only computed first time
    VAR borders := cluster.radix_count(nbits, nignore); # only computed first time
    VAR cluster_values := cluster.reverse.mark.reverse;
    VAR cluster_ids := cluster.mark.reverse;
    RETURN cluster_values.join(right).radix_decluster(cluster_ids, borders);
  }
  return join(left, right);
}
```

6.5.3 Performance Evaluation

In this section, we present experiments done on a single-CPU node of the Origin2000 described in Table 6.1. In these experiments, we executed our example project-join SQL query with the above described Monet and "traditional" relational query processing strategies on relations of equal size $C \in \{125K, 500K, 2M, 8M\}$, consisting of $W \in \{1, 4, 16, 64\}$ all-integer (4-byte) columns, with a join hit rate of 3, and projecting $P \in \{1, 4, 16, 64 | P \leq W\}$ columns from both relations into the result. In all experiments, all processing happens in main-memory (no I/O or page faults).

It should be noted, that in all our experiments, apart from elapsed time, we also measured memory cache misses in both the L1 and L2, as well as branch mis-predictions. In the evaluation of join query performance with projections, we omit the detailed measurement results as well as performance models, similar to those presented earlier in this chapter for the Radix-Cluster and Partitioned Hash-Join algorithms. We do this for reasons of conserving space, and also because performance behaves as expected:

	cardinality C							
	125K		**500K**		**2M**		**8M**	
	reorder	posjoin	reorder	posjoin	reorder	posjoin	reorder	posjoin
unsorted	0	34	0	215	0	2853	0	15928
quick-sort	347	29	1620	118	7456	484	34743	2362
radix-cluster:								
radix-sort	151 19:2	29	769 21:3	118	3158 23:3	484	19460 25:3	2362
4 tuples	144 17:3	27 +43	779 19:3	113 +633	3528 21:4	475 +3432	17913 23:3	2379 +18765
16 tuples	172 15:3	26 +42	671 17:3	116 +428	3009 19:3	486 +2394	16577 21:4	2475 +18217
64 tuples	123 13:1	26 +32	602 15:3	126 +205	2544 17:3	498 +1444	15579 19:4	2533 +12277
1KB	91 11:2	25 +32	515 13:2	124 +189	2337 15:3	588 +771	12967 17:3	2537 +6440
4KB	88 9:2	25 +41	397 11:2	109 +283	1933 13:2	466 +684	11680 15:3	2504 +4230
16KB	84 7:2	26 +31	376 9:2	104 +152	1510 11:2	530 +598	10988 13:3	2788 +3328
64KB	40 5:1	31 +59	381 7:2	123 +136	1534 9:2	490 +600	9466 11:3	2941 +2969
256KB	40 3:1	29 +30	177 5:1	131 +128	1540 7:2	605 +572	7307 9:2	3017 +3001
1MB	43 1:1	32 +29	168 3:1	142 +124	686 5:1	547 +496	7711 7:2	3209 +2740
4MB			178 1:1	243 +138	725 3:1	2075 +497	3416 5:1	9776 +2609
Bits:Passes								

Table 6.3: Positional-Join($3C, C$) Performance (ms) without reordering, with Quick-Sort($3C$) or Radix-Cluster($3C$) (possibly +Radix-Decluster($3C$))

- Quick-Sort indeed exhibits $C(\log_2(C) - \log_2(cachesize))$ cache misses on all memory cache levels,
- Unsorted Positional-Join into a column that is larger than a cache, thrashes that memory cache (degrading with increasing cluster size to the worst-case of one cache miss for each tuple in the outer relation).
- Sorted or Clustered Positional-Join into a column with clusters that fit the cache, exhibits just the amount of misses to read in the column once (i.e. the minimal "compulsory" misses).
- Radix-Decluster with an insertion window size smaller than the cache only exhibits the minimal compulsory misses to read in all input relations once.

Considering the Monet strategy, a first question of interest is which reordering algorithm works best: Quick-Sort or Radix-Sort. Figure 6.31 shows results for various re-ordering algorithms (Quick-Sort, Radix-Sort and Radix-Cluster) and for Positional-Join. The reordering done in our example setting is on the join index previously computed with Partitioned Hash-Join. This `BAT[oid,oid]` has a cardinality of $3C$, due to the join hit-rate of 3. The join costs in this table are for joining this $3C$ join index into a column-`BAT[void,int]` of size C. The reordering costs are for reordering a $3C$ join index. The results show that Quick-Sort is consistently beaten by Radix-Sort on `oid`, which is achieved by Radix-Cluster on all significant bits ($B = \log_2(C)$).

The detailed data in in Table 6.3 show that in general, however, best performance is not obtained with Radix-Sort but with (partial) Radix-Cluster that clusters on a much more coarser cluster-size than 1 tuple (recall Radix-Sort is a Radix-Cluster on all bits). Partial Radix-Cluster makes Positional-Join performance deteriorate a bit, but at the benefit of greatly reduced reordering cost. The fewer Radix-Bits, the better the performance (see also the left graph in Figure 6.32). However, in order to make Positional-Join work well, the clusters should at least fit the L2 cache, and keep improving until they approximate the cost of Positional-Join on a sorted input, when

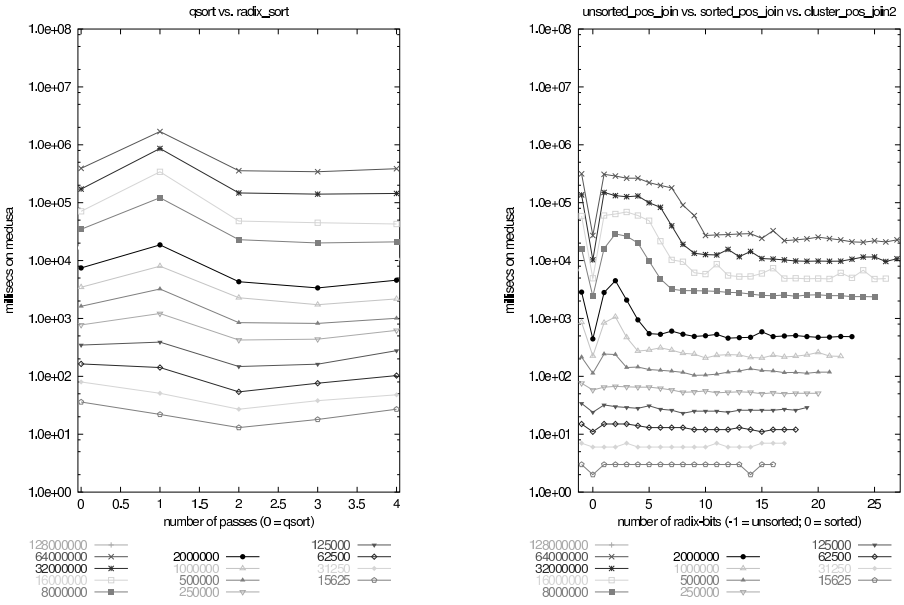

Figure 6.31: Performance of Sorting($3C$) and Positional-Join($3C, C$)

the clusters are so small that they fit the L1 cache. In the case of the Origin2000, a reasonable cluster size seems to be 256KB, which is one sixteenth of the L2 size, but still larger than the 32KB L1. For the larger cardinalities (e.g., 2M and 8M), we see that even if there is only one projection column, Radix-Cluster followed by Positional-Join beats Unsorted Positional-Join, and the performance gain can become a factor 6 if there are many projection columns.[10]

The partial-cluster join strategy can only be used for projecting columns of one of the two join relations efficiently with respect to the memory caches. However, if we need to project columns from both relations, and both relations have so many tuples that their column-BATs exceed the cache size, we need to use the cluster-decluster strategy for doing the projections from the other table. The performance of the Radix-Decluster operation is depicted in the right graph of Figure 6.32. The performance of Radix-Decluster is quite flat, only deteriorating due to memory cache misses when the amount of clusters becomes too large (and the cluster sizes too small). We should note that we found performance of Radix-Decluster to be optimal in this configuration when the insertion window was kept to half of the L2 size (which means that there are still a substantial number of L1 misses).

Getting back again to the detailed numbers in Table 6.3, we should look at the numbers added to the join cost (listed after the plus sign); this is the cost of Radix-Decluster to create the final column result. In the cluster-decluster strategy, each Positional-Join is followed by a `radix_decluster` operation; therefore their costs have to be added. This decreases the performance advantage with respect to unclustered Positional-Join, but one can see that for the larger cardinalities (2M and 8M), Radix-

[10] With larger cardinalities and faster CPUs, the relative penalty of cache thrashing in the Unsorted Positional-Join becomes larger, so the overall gain will improve over time.

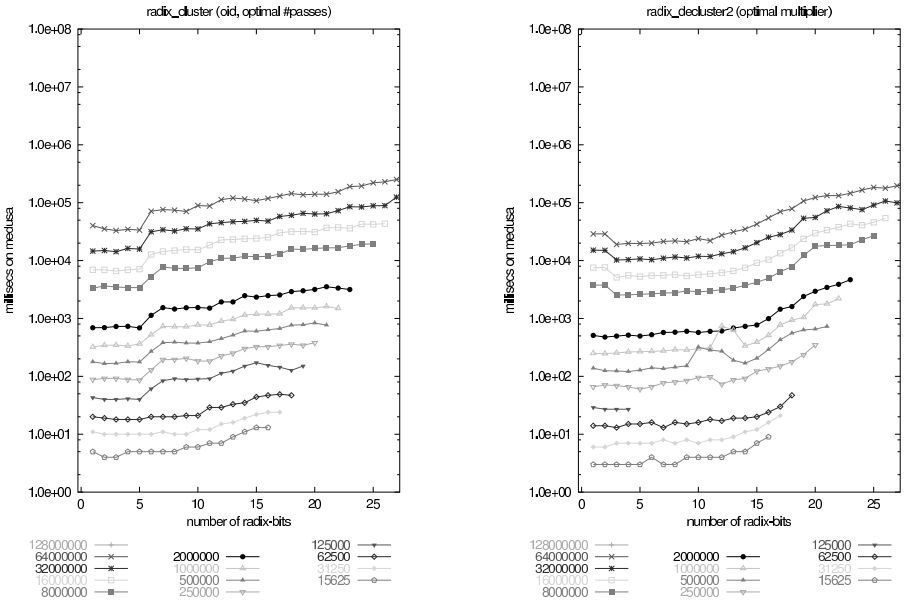

Figure 6.32: Performance of Radix-Cluster($3C$) and Radix-Decluster($3C$)

Decluster still wins.

We now analyze which (cache-conscious) query processing strategy for the projection phase of our generic join query works best in Monet and under which circumstances. As described earlier, we consider tables of up to 256 columns, which is the maximum value of P here. Note that for Monet query performance only P matters, not the actual number of columns in the table W (as they are fragmented vertically in distinct column-BATs). Therefore, a Monet experiment for a certain P holds for all W.

We consider four strategies:

u *Unsorted:* one Positional-Join from the join index into each projected column-BAT.

s *Sorted:* first sort the join index with the best sorting algorithms (i.e. Radix-Sort).

c *partial-Cluster:* first partially cluster the join index. We take the number of Radix-Bits that gives the best result (in this setting, this generally boils down to the 256KB cluster size).

d *cluster-Decluster:* like the clustered experiment, but each Positional-Join is followed by Radix-Decluster.

Figure 6.33 summarizes the performance of the various Monet strategies to process the projection phase of our example SQL join, depending on amount of projection columns P and cardinality C (recall that the join result actually has size $3C$ due to the join hit ratio of 3). For small cardinalities ($C \leq 125K$), all strategies that do any kind of reordering lose to simple unsorted processing of the Positional-Joins, since the column-BATs are so small that they fit the cache anyway. For larger cardinalities, however, the unsorted approach always loses by a big margin (e.g. by more than a factor

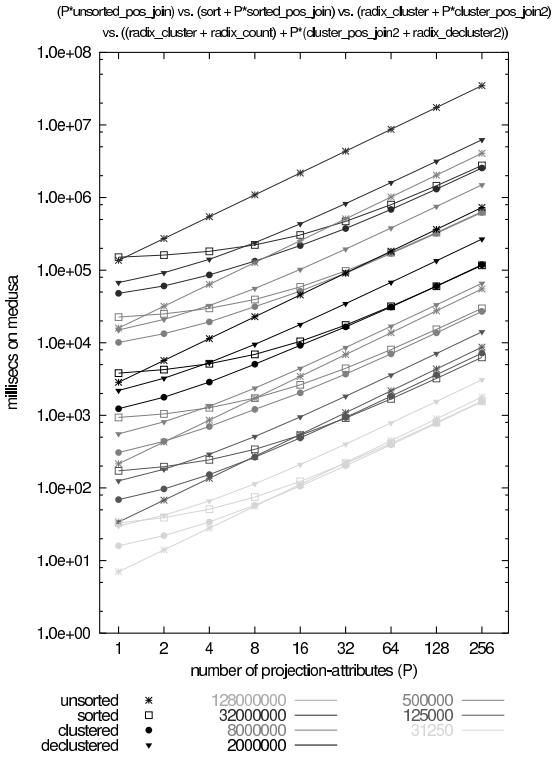

Figure 6.33: Monet Projection Strategies: Unsorted vs Sorted vs Radix-Cluster vs Radix-Decluster

10 at $C = 32M$ and $P = 64$). Like we said before, partial-clustered is always more efficient than sorted processing; the difference becoming larger with more projection columns. Finally, we see that the cluster-decluster strategy always loses from the partial-cluster strategy, but is actually quite competitive, beating unsorted processing by a large margin. One should therefore choose it if one cannot use partial-cluster anymore (i.e. on the second projection table).

As such, we formulate the following rules to arrive at the "optimal" projection strategy in Monet on the Origin2000[11]:

1. if for all relations the largest column-BAT fits the L2 cache, use unsorted processing.

2. if there is only one relation that has projection columns and that does not fit the above rule, use the partial-cluster strategy for the projections from that relation, and use the unsorted strategy for the projections from the other relations.

[11]The rules formulated here are specific to the characteristics of the Origin2000, which has a slow TLB and fast L1 cache. Calibrated hardware characteristics for use in detailed cost models, like those formulated earlier in the Chapter for Radix-Cluster, will provide full performance insight that enables a query optimizer to choose the optimal strategy for any query on any hardware. The development of such detailed performance models for Radix-Decluster and Positional-Join is considered future work here – though it should not be difficult given the consistent performance results we encountered.

P	system	W	C=125K	C=500K	C=2M	C=8M
1	monet	*	168 (u/u)	932 (u/u)	4871 (c/d)	33069 (c/d)
	relational	1	419 → 551	3033 → 2229	14892 → 8833	79173 → 47260
		4	560 → 571	5275 → 2275	16590 → 8965	74948 → 47223
		16	724 → 577	4827 → 2385	18191 → 9919	146674 → 48127
		64	965 → 613	4735 → 2635	30088 → 9911	161291 → 51477
4	monet	*	372 (u/u)	2063 (c/u)	9641 (c/d)	42502 (c/d)
	relational	4	1011 → 1028	5928 → 4459	22882 → 12221	102837 → 81320
		16	1349 → 1060	6602 → 4571	25790 → 14613	169883 → 87721
		64	1395 → 1160	6970 → 5767	46439 → 19611	248099 → 94887
16	monet	*	1133 (c/u)	5982 (c/u)	28509 (c/d)	161900 (c/d)
	relational	16	2753 → 2955	12804 → 12580	53433 → 55849	
		64	3309 → 2989	17098 → 13257	77425 → 58235	
64	monet	*	3960 (c/u)	21294 (c/u)	100461 (c/d)	559464 (c/d)
	relational	64	9992 → 11155	38441 → 46728		

Table 6.4: Join-Project Performance (ms): Monet vs Relational, simple hash-join → fully cache optimized

3. if both relations are big (i.e. their column-BATs do not fit L2) and both have projection columns, use partial-cluster on one and cluster-decluster on the other. One could use the heuristic to use partial-cluster on the relation with most projection columns, or with the smallest product of cardinality and number of projection columns. Notice that in our experimentation setting, where we have equal cardinalities and equal number of projection columns in both join relations, it does not matter which relation is processed with the partial-cluster strategy and which with the cluster-decluster strategy (this setting with equally difficult projections from both relations is actually worst-case for Monet, as we get maximal cluster-decluster cost).

In Table 6.4 we use the "optimal" strategy for Monet, and compare its performance (in milliseconds) with the performance of "traditional" relational query processing, both without and with Partitioned Hash-Join (displayed as *without* → *with*). For the Monet results, we append between parentheses the chosen strategy for projecting columns (displayed as *larger/smaller*).

Considering the optimal overall Monet strategy, we see that in the smaller sizes the **u/u** – both unsorted – strategy wins. At some point, this transitions into **c/u** (here, the declustered strategy is still more expensive than unsorted). At large cardinalities though, the unsorted strategy costs explode due to cache thrashing, and we get **c/d** as a winner. We conclude that the Radix-Decluster operation is instrumental in providing an efficient generic join processing strategy in Monet.

One conclusion to be drawn is that also for relational DBMSs, Partitioned Hash-Join brings substantial advantages. In other words, the results of Section 6.4 not only hold for Monet, where we did our initial experiments, but also for generic RDBMS technology. We observe that the advantage is especially pronounced if the relation sizes are large and the percentage of columns being projected on is low. In those cases, a 3-fold performance improvement can be obtained (e.g. C=8M, W=64, P=1).

Another conclusion to be drawn is that vertical fragmentation as implemented in Monet beats "traditional" query processing by a margin that also is maximized at large cardinalities and low projection percentages, and can reach almost an order of magnitude (e.g., more than a factor 6 at C=2M, W=64, P=1) at our relatively small

table settings.[12]

Even if we compare with cache-optimized relational processing, the Monet advantage is more than 50% (factor 2-3). It is interesting to note that although it is definitively the case that a Monet query plan moves more data around than a relational one, the higher efficiency with which Monet can execute its operations more than compensates for that. There are two reasons why the vertical fragmentation in Monet is instrumental in this: first, it helps reducing cache misses as columns that are not in the projection never need to be scanned and brought into the caches. Secondly, the vertical fragmentation allows MIL to be what it is: an algebraic language with operators that have signatures with little degree of freedom (few parameters, all of them 2-column tables). This small degree of freedom in the algebra gives the operators much knowledge at DBMS kernel compile-time, enabling CPU-optimizations that are otherwise not possible.[13] We have shown in Section 6.4 that such CPU optimizations can attribute for a factor 4 of performance improvement on modern CPUs.

We can argue though, that the advantage of Monet over "traditional" relational query processing is in fact even larger, and will expand in the future, for the following reasons:

- The experimental setting with equally sized join relations and equal numbers of projection columns is worst-case for Monet; as it maximizes Radix-Decluster cost. As Radix-Decluster typically takes at most one third of the query time in our experiments, the gains of having more skewed numbers of projection columns are limited, however.

- The relational implementation tested here uses a Monet infrastructure which we suppose to be more CPU-efficient in a main-memory setting than that of the average RDBMS implementation.

- Hardware advances will worsen the memory access bottleneck and increase the benefit of the extreme CPU-optimizations applied in Monet. At the time of this writing, 2 years after the experiments, the fastest CPU is the Pentium 4 running at 2.2GHz. This is a factor 5 faster than the Pentium III tested here, while memory latency has not improved (it actually became 20% slower in RDRAM). Furthermore, the Pentium 4 has an extremely long pipeline (doubled in length with respect to the Pentium III) making it extremely vulnerable to branch mispredictions (and increasing the benefit of CPU optimizations). This vulnerability is even more present in the Itanium line of server processors, which due to its architecture (EPIC) fully relies on compile-time optimizations. Also, this line of processors exhibits an increasing abundance of parallel hardware resources which is sure to go unused in non-optimized DBMS code.

[12] We actually carried out experiments up to C=64M tuples and W, P=256 columns, but the shortcomings of our relational implementation – i.e. the byte-size of one BAT could not exceed 32-bit integer bounds – limited the relational experiments to those where $P * C < 128M$

[13] One other way to increase the compile-time optimization possibilities is to compile in real-time, e.g. by parsing SQL queries, generating C/C++ – or even assembly – and compile this just-in-time into a dynamically loadable library, which is then loaded and called into for executing the query. Compilation time can be considerable, though, which might be contradictory to ad-hoc query processing as in OLAP, but also could be alleviated by query-plan caching. To our knowledge, no current RDBMS implementations use just-in-time machine code compilation, but we pose that given its increasing benefits on modern hardware this may be a sound technique for future database systems.

6.6 Conclusion

We have discussed trends in modern hardware, showing the importance of good usage of the various cache levels in current computer systems in order to limit access to main memory DRAM chips, which gets exponentially more expensive with the law of Moore. Also, modern super-scalar CPUs have become fragile with respect to efficiency, as they depend on speculative execution to fill their long pipelines and only deliver their advertised performance if program code has much locality and is highly predictable.

In this chapter, we investigate the effects of these phenomena on query processing. As for sequential table access, Figure 6.3 shows that the key issue is to reduce the stride, which is achieved in Monet by vertical fragmentation and enhanced with enumeration types (which makes BUNs, and thus the stride, smaller). As for query processing operators with random access, such as hash-join, we have shown that un-cacheable memory access patterns can lead to almost a magnitude of performance degradation in large equi-joins. We refined partitioned hash-join with a new partitioning algorithm called *Radix-Cluster*, that prevents performance becoming dominated by memory latency (avoiding the memory access bottleneck).

Exhaustive equi-join experiments were conducted on modern SGI, Sun, and PC hardware, and we formulated detailed analytical cost models that explain why and when the Radix-Cluster/Partitioned Hash-Join algorithm makes optimal use of hierarchical memory systems found in modern computer hardware and very accurately predict performance on all three platforms. This modeling work represents an important improvement over previous work on main-memory cost models [LN96, WK90]. Rather than characterizing main-memory performance on the coarse level of a procedure call with "magical" cost factors obtained by profiling, our methodology mimics the memory access pattern of the algorithm to be modeled and then quantifies its cost by counting cache miss events and CPU cycles. We were helped in formulating these models through our usage of hardware event counters present in modern CPUs.

Furthermore, we contributed a *hardware-calibrator* that automatically detects the hardware characteristics of the memory hierarchy, and forms the crucial model parameters for tuning the settings of our Radix-Algorithms. Such a software module should be part of any cache-conscious DBMS, as it allows to tune cache-conscious query processing algorithms independent of the hardware. Monet incorporates the calibrator as an extension module that is run at startup, and we showed with the *Radix-Accelerator*, how the extracted parameters are used to perform dynamic cache-optimized query processing through tactical query optimization in MIL.

Finally, we extended our study from isolated hash-join execution to generic join query processing including projections. Here we compared "traditional" relational query processing (with and without Radix-Algorithms) with various column-oriented query processing strategies in Monet. Here, we contributed another algorithm called *Radix-Decluster*, which in combination with Partial Radix-Cluster and efficient Positional-Join enables fully cache-optimized query processing in Monet. In the overall experiments, we have shown that optimizing equi-joins using the Radix-Algorithms improves performance both in the relational and in the Monet case, while the Monet implementation is consistently significantly faster than the relational approach.

Chapter 7

Monet as RDBMS back-end

The purpose of this chapter is to illustrate how all functionality of MIL and its implementation in Monet can be used to build a full-fledged DBMS. Recall that in our DBMS architecture of Figure 3.2, this entails the design and implementation of a *front-end* system, where Monet serves as *back-end* and the interface between the two is MIL. Rather than describing a completed system[1], we limit ourselves here to a sketch of such a system, yet go into sufficiently deep detail of its implementation – sometimes ad nauseam – until we get at the level of the exact MIL code needed for providing each query processing functionality.

7.1 Introduction

While Monet can be used to manage object-oriented or even graph-like data models [dVW98, WvZF+98], we chose our example case here in the relational field, as this is most common and therefore provides ample reference to other systems. Specifically, we discuss a front-end that provides basic SQL-2 compliant RDBMS functionality. Again, while Monet has been shown to be able to store and query multi-media (audio, video), XML and GIS data types [BQK96, NK97, SKWW00], we limit ourselves here to common business data types. Making the case even more specific, we use a particular database schema and filling in our examples, namely the TPC-H (formerly known as TPC-D) database, as this well-defined DBMS benchmark is targeted to the focal point of the Monet design: analytical query-intensive environments (OLAP, Data Mining).

Functionally, our SQL-2 RDBMS, consisting of Monet as back-end, and the front-end system discussed here, should:

- provide high transaction processing (TP) rates on operational tables of moderate size (such that these tables fit in memory).
- provide high analytical performance (OLAP, Data Mining) on voluminous historical ("data warehousing") data tables.
- provide a mechanism of fine-grained update propagation between the quickly changing operational tables, and the larger and stabler data warehousing images.

[1] Check monetdb.cwi.nl for the ongoing efforts of our group.

The motivation to choose this example setting is twofold. First, it is a showcase to demonstrate not only how analytical queries are processed by a RDBMS based on Monet, but it also shows how updates and transactions can be handled with MIL.

Second, there is strong market relevance for this particular setting: DBMS technology that combines high performance in both transactions and analytical queries[2]. Current DBMS products can often only deliver one of these capabilities at a time. This is one of the reasons why many organizations employ separate *OLTP* and *data warehousing* DBMS environments. Such organizations have built data warehousing systems that are filled periodically by *copy management* tools that periodically extract data from operational systems. Such operational systems are typically highly transaction-oriented, sustain a heavy load of simple update and read requests, and contain the freshest image of the reality that the organization records in its IT infrastructure. The data warehousing system on the other hand, typically deals with data that is a somewhat stale, and is optimized to sustain analytical read-mostly loads consisting of complex queries, that are generated by reporting, OLAP and Data Mining tools. Data volumes in the data warehouse are often much larger than those found in the operational systems, because operational systems tend to record the current state of affairs, whereas data warehouses typically record historical data, very roughly speaking consisting of a (time-)series of snapshots from the operational systems.

While there are many other reasons (e.g. historical, organizational) to have separate OLTP and data warehousing environments [Imm92, Wid95, CD97], one of them certainly is related to performance problems that RDBMS technology has with sustaining both complex query and OLTP loads at the same time. However, we envision that in the near future, organizations will be inclined to integrate the now separate operational and data warehousing environments again. In commercial practice, one can already observe that new applications are often deployed on top of the data warehouse instead of the OLTP systems, because the data model in the data warehouse is richer (it may unite data from various sources, and contains historical information). However, as these new applications also demand (some) fresh data, organizations find themselves inclined to increase the frequency of propagating updates from the OLTP environments to the data warehouse (e.g. from each month to each night, to each hour, etc.). Obtaining high update frequencies might be difficult to achieve performance-wise. Hence the need for a DBMS that can sustain both high-performance TP and analytical query loads.

Additionally, one can identify emerging DBMS applications that explicitly require this combination:

- e-commerce sites that log visitor information to a RDBMS and want to personalize their web pages (e.g. using profiles found by data mining) to the web browsing actions a visitor performs in real-time. This yields a high query and update load to the tables holding the web log.
- telecommunications companies that want to deploy sophisticated billing policies in real-time on their Call Detail Record (CDR) databases. Again, these policies lead to continuous querying to the CDR tables, which are also continuously being updated.
- financial broker risk analysis, that requires highly frequent recalculation of the

[2]Note our assumption that the transaction database fits in main memory.

risk a financial portfolio poses against a highly active database of financial (stock) price information. Up till now, such real-time risk analysis has only been possible on simplified representations of the real data.

The TPC-H Benchmark

We use the data model of the TPC-H benchmark and some of its queries as our running example throughout this chapter, as its schema and queries are well-known and it fits nicely to the query-intensive DBMS area for which Monet was designed (decision support on large data warehouses).

The TPC-D benchmark started out as the generally accepted yardstick for query-intensive loads on a synthetically generated data warehousing environment. However, a "benchmark war" broke out among the commercial RDBMS vendors that initially had helped specify the TPC-D. The center of this dispute was the question which optimization techniques were allowed in TPC-D implementations. In particular, after having implemented *materialized views* in their RDBMS, Oracle was able to remove all costly joins from the TPC-D query execution, which improved their results by more than a factor 100[3]. The outcome of the conflict that followed was that TPC-D got abolished and two new benchmarks were defined: TPC-R (reporting) and TPC-H (ad-hoc querying). The TPC-H benchmark now represents what TPC-D used to represent: decision support environments where users don't know which queries will be executed against a database system; hence, it measures ad-hoc query execution performance. Given this ad-hoc-ness, no prior knowledge of the queries can be built into the physical design of the DBMS.

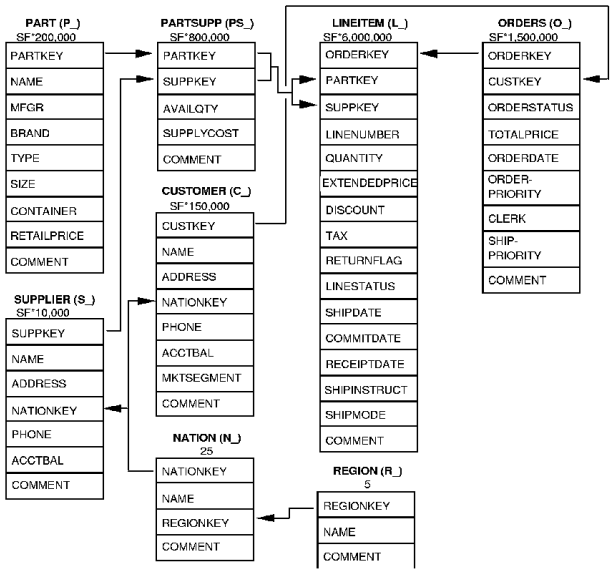

Figure 7.1: TPC-D/H/R Schema

[3]Moreover, in November 1998, Oracle went on to defy its main competitor Microsoft with a "Million Dollar Challenge": $1 million to anyone who could "demonstrate that SQL Server 7.0 is not 100 times slower than the Oracle database when running a standard business query against a large database." Needless to say, Microsoft SQLserver by that time did not implement materialized views.

SF=100						
RDBMS	**hardware**	**CPU**	**#**	**RAM**	**QpH**	**size/SF**
MS SQLserver 7.0	Compaq	PIII Xeon 700MHz	8	4GB	1699.8	13.4GB
MS SQLserver 7.0	IBM	PIII Xeon 700MHz	8	4GB	1454.4	17.0GB
MS SQLserver 7.0	Compaq	PIII Xeon 550MHz	8	4GB	1308.0	12.9GB
MS SQLserver 7.0	HP	PIII Xeon 550MHz	8	4GB	1291.4	15.8GB
MS SQLserver 7.0	NEC	PIII Xeon 550MHz	8	4GB	1194.4	17.0GB
MS SQLserver 7.0	NEC	PIII Xeon 550MHz	4	4GB	800.4	11.6GB
SF=300						
RDBMS	**hardware**	**CPU**	**#**	**RAM**	**QpH**	**size/SF**
DB-2 UDB 7.1	IBM	PIII Xeon 700MHz	64	64GB	7334.4	19.1GB
Informix 8.3 XPS	Compaq	Alpha 16264 733MHz	32	64GB	4951.9	11.1GB
Informix 8.3 XPS	HP	PA-RISC 440MHz	32	32GB	3714.9	10.5GB
Informix 8.3 XPS	HP	PA-RISC 500MHz	8	32GB	1592.3	6.9GB
MS SQLserver 7.0	Compaq	PIII Xeon 700MHz	8	4GB	1402.5	7.4GB

Figure 7.2: TPC-H Results (Y2K)

Figure 7.1 shows the standard schema used in the TPC-H. This schema models an internationally operating company that buys parts from manufacturers and sells them to customers. Customers place orders, that consist of a number of line-items. Each line-item is a certain quantity of some part. Parts are obtained from manufacturers. The main tables in terms of size are the ORDERS and LINEITEMS, which contain 1.5 and 6 million rows respectively. All tables except NATION and REGION can be scaled with a scaling factor (SF); hence with SF=100 the ORDERS and LINEITEMS table contain 150 and 600 million rows respectively. Currently popular scaling-factors among DBMS vendors are SF=30, SF=100 and SF=300. Figure 7.2 shows some current results. These are benchmarks of commonly known RDBMS products, performed on (mostly) SMP machines with large memories and multiple CPUs. The "QpH" (Query-Per-Hour) metric is a combination of the performance on all 22 analysis queries and 2 update ("data refresh") batches.

7.1.1 Road Map

The description of our hypothetical SQL-2 RDBMS front-end is not intended to be complete as we bypass many important issues that are relevant in RDBMS (front-end) design, like query transformation and query optimization. Our description is merely intended to illustrate how MIL and Monet can be used to construct a full-fledged DBMS. Hence, we concentrate on how the RDBMS front-end uses Monet and MIL to fulfill its primary tasks. This involves various aspects of DBMS functionality, discussed in the following sections:

physical data storage In Section 7.2, we discuss how a relational schema can be stored in Monet. In this discussion, we use the schema from the TPC-H benchmark to illustrate how this works in detail, also covering the use of auxiliary indexing structures.

query execution Section 7.3 describes how SELECT-FROM-WHERE queries are executed in Monet, by showing how all major SQL language constructs can be

mapped into the MIL language. We also discuss more complex cases like e.g. multi-column joins. A detailed run of Query-9 from the TPC-H benchmark is used to illustrate how MIL achieves efficient execution of OLAP queries.

transaction management We show in Section 7.4 how Monet can provide a mechanism for handling update queries. Here we demonstrate that while the individual MIL primitives do not have ACID properties, they provide the sufficient functionality to build a fully ACID and high performance SQL update service.

7.2 Physical Storage of Relational Tables

In this section we illustrate how relational table data gets stored in Monet. Figure 7.3 lists an object-oriented data dictionary, where a database consists of a number of tables, that each have a number of columns. Inverted lists may be attached to columns, and join indices may be defined between tables.

This object-oriented structure reflects the internal data structures of the front-end; and might be mapped on BATs so as to store this meta-information in Monet itself.

In the following, we will describe the various kinds of BATs referenced by this data dictionary (i.e. the C-, S-, I-, D- and M-BATs) which are used for all mass-storage, and discuss their role in query processing and the physical resources they claim.

As a concrete example, we study how the TPC-H schema is stored in Monet, search accelerator structures inclusive.

7.2.1 Basic Table Storage in BATs

Relational tables can be stored in Monet using the basic technique described Section 4.3.2, which simply decomposes each table by column and puts each column of type X in a `BAT[void,X]`.

Relevant for Monet is implementation rule number 1.5.6 of TPC-H, which states that vertical fragmentation as a physical design strategy is explicitly forbidden. This rule prevents DBMS vendors from placing table columns that are not accessed by the benchmark queries in a separate fragment; which effectively would make the tables smaller (one could call this *opportunistic* vertical fragmentation). We argue, though, that *full* vertical fragmentation as employed in Monet, where each column is stored in a separate BAT, should be an exception to this rule, since full vertical fragmentation is not an opportunistic strategy. Queries benefit from full vertical fragmentation because only data that is actually needed is accessed; but also pay a price in terms of extra query effort introduced by fragmentation (data that *is* needed by the query is also fragmented and needs to be joined together). Key issue is that the vertical fragmentation is not applied to match the specific access pattern of the benchmark queries, but uniformly across the whole schema. As such, full vertical fragmentation is a physical design strategy that uses no pre-knowledge about the query load, and therefore – in our view – does not violate the faith of the TPC-H benchmark.

The purpose of Figure 7.4 is to provide insight into the storage requirements imposed by the decomposition into BATs. For each column of the TPC-H tables, it shows its column name, type as specified by the TPC-H benchmark rules, the corresponding Monet type, the storage requirements in both the "C-BAT" and "S-BAT" representations, and a detailed description of what values are stored in the columns (taken from

```
class Table {
        relation Set<Column> columns         inverse Column::table;
        relation Set<JoinIndex> dst_indices  inverse JoinIndex::dst;
        relation Set<JoinIndex> src_indices  inverse JoinIndex::src;
        relation Set<UniqueKey> uniqueKeys   inverse UniqueKey::table;
        relation Set<ForeignKey> foreignKeys inverse ForeignKey::table;
        string name;
        BAT h_bat; // holes; oids that are unused
}

class Column {
        relation Table table                 inverse Table::columns;
        relation InvertedList index          inverse InvertedList::idx_column;
        string name;
        integer type;
        BAT c_bat; // main storage BAT[void,type]
}

class UniqueKey {
        relation Table table                 inverse Table::uniqueKeys;
        relation Set<ForeignKeys> refs       inverse ForeignKey:source;
        relation Set<Column> columns;
}

class ForeignKey {
        relation Table table                 inverse Table::foreignKeys;
        relation UniqueKey source            inverse UniqueKey::refs;
        relation Set<Column> columns;
}

class InvertedList {
        relation Column column               inverse Column::index;
        BAT s_bat; // inverted list
        BAT m_bat; // small main-memory summary
        BAT i_bat; // deltas since last sync (inserts)
        BAT d_bat; // deltas since last sync (deletes)
}

class JoinIndex {
        relation Table src                   inverse Table::src_indices;
        relation Table dst                   inverse Table::dst_indices;
        relation Column column; // refers to 'artificial' oid-column
        Expression joinExp;     // can be any boolean expression
}

virtual class Expression                      { integer type;               }
        class ConstExpression    : Expression { string atom;                }
        class ColumnExpression   : Expression { relation Column column;     }
        class OperatorExpression : Expression { string operator;
                                                List<Expression> parameters; }
```

Figure 7.3: Data Dictionary Structure in the SQL Frontend

column	tail	type	C-BAT	S-BAT	description
suppkey	oid	identifier	4	8	unique from [1..SF*10000]
name	str	varchar(25)	20	24	"SupplierXXXXXX"
address	str	char(40)	32	36	random string [25]
nationkey	oid	identifier	4	8	random value [0 .. 24]
phone	str	char(15)	20	24	"II-RRR-PPP-LLLL"
acctbal	dbl	decimal	8	12	random value [-999.99..9,999.99]
comment	str	varchar(101)	68	72	random string, mean length 63
supplier			156	184	SF*10,000 rows
partkey	oid	identifier	4	8	unique from [1..SF*200000]
name	str	varchar(55)	40	44	random string [35]
mfgr	str	char(25)	20	24	"ManufacturerX"
brand	str	char(10)	12	16	"BrandXX"
type	enum1[str]	varchar(25)	1	8	random string from 150 Types
size	enum1[int]	integer	1	8	random value [1..50]
container	enum1[str]	char(10)	1	8	random string from 40 Containers
retailprice	dbl	decimal	8	16	random from [9000.00..19999.99]
comment	str	varchar(23)	28	32	random string, mean length 14
part			115	174	SF*200,000 rows
partkey	oid	identifier	4	8	unique from [1..SF*200000]
suppkey	oid	identifier	4	8	random from [1..SF*10000]
availqty	enum2[int]	integer	2	8	random from [1..9,999]
supplycost	dbl	decimal	8	16	random from [1.00..1000.00]
comment	str	varchar(199)	128	132	random string, mean length 124
partsupp			146	172	SF*800,000 rows
custkey	oid	integer	4	8	unique from [1..SF*150000]
name	str	varchar(25)	20	24	"CustomerXXXXXX"
address	str	varchar(40)	32	36	text string [25]
nationkey	enum1[oid]	identifier	1	8	random value [0..24]
phone	str	char(15)	20	24	"II-RRR-PPP-LLLL"
acctbal	dbl	decimal	8	16	random value [-999.99..9,999.99]
mktsegment	enum1[str]	char(10)	1	8	random string from 5 Segments
comment	str	varchar(117)	80	84	random string, mean lenth 73
customer			166	208	SF*150,000 rows
orderkey	oid	identifier	4	8	unique from [1..SF*6000000]
custkey	oid	identifier	4	8	random value [1..SF*150000]
orderstatus	chr	char(1)	1	8	'F', 'O' or 'P'
totalprice	dbl	decimal	8	16	random from [8100..7559996.22]
orderdate	enum2[date]	date	2	8	random from [start..end-151]
orderpriority	enum1[str]	char(15)	1	8	random string from 5 Priorities
clerk	str	char(15)	16	20	"ClerkXXXXX"
shippriority	enum1[int]	integer	1	8	always 0
comment	str	varchar(79)	56	60	random string, mean length 49
order			93	144	SF*1,500,000 rows
orderkey	oid	identifier	4	8	random from [1..SF*6000000]
partkey	oid	identifier	4	8	random from [1..SF*200000]
suppkey	oid	identifier	4	8	random from [1..SF*10000]
linenumber	enum1[int]	integer	1	8	unique within order from [1..7]
quantity	enum1[int]	integer	1	8	random from [1..50]
extendedprice	dbl	decimal	8	16	random from [9000.00..999999.50]
discount	enum1[dbl]	decimal	1	8	random from [0.00..0.10]
tax	enum1[dbl]	decimal	1	8	random from [0.00..0.08]
returnflag	chr	char(1)	1	8	'R', 'A' or 'N'
linestatus	chr	char(1)	1	8	'O' or 'F'
shipdate	enum2[date]	date	2	8	random from [start+1..end-30]
commitdate	date	date	2	8	random from [start+30..end-61]
receiptdate	enum2[date]	date	2	8	random from [start+1..end-91]
shipinstruct	enum1[str]	char(25)	1	8	random from 4 Instructions
shipmode	enum1[str]	char(10)	1	8	random from 7 Modes
comment	str	varchar(44)	32	36	random string, mean length 27
lineitem			66	164	SF*6,000,000 rows
nationkey	oid	identifier	4	8	unique from [0..24]
name	str	char(25)	12	12	unique string, mean length 7
regionkey	oid	identifier	4	8	random from [0..4]
comment	str	varchar(152)	100	104	random string, mean length 95
nation			120	132	25 rows
regionkey	oid	identifier	4	8	unique from [0..4]
name	str	char(25)	12	16	unique string, mean length 6
comment	str	varchar(152)	100	104	random string, mean length 95
region			116	128	5 rows

Figure 7.4: Details of TPC-H storage in BATs

the TPC-H definition). For now, we focus on the representation in C-BATs, which simply stands for "Column BATs". Storage in S-BATs is discussed in Section 7.2.3 and later. For each TPC-H column described in Figure 7.4 we get a C-BAT named here for convenience "C_<TABLE>_<COLUMN>", e.g. for the first column we get a `BAT[void,oid]` named `C_SUPPLIER_SUPPKEY`.

Each column has a C-BAT (see Figure 7.3), thus we get 61 C-BATs for the 61 columns of the 8 tables in the TPC-H schema. All C-BATs have a `void` head type, indicating a sequence of densely ascending `oid`-s that start at zero (0,1,2,3,...). As explained in Section 5.2 this has two advantages: `void`-s do not take up any space at all, and lookup into `void` columns is positional and therefore highly efficient.

The choice of what tail type to use in each C-BAT is mainly dictated by the requirements the TPC-H rules imposes on column types. An "identifier"must be able to contain values between minus 2 billion and plus 2 billion, hence require at least a 32-bit integer, type (the `int` type in Monet). Similarly detailed requirements hold for the "decimal" and "date" types, hence those must be represented in Monet as `dbl` and `date` (the latter is a user-defined type from the Monet extension module "time", which uses 32-bits integers to store dates as the (possibly negative) number of days since January 1 of the year 1[4]. This `date` type is implemented as an `int` (see Section 5.2.1, "Implementation Types"), hence MIL-selects and joins on dates are executed by the fast int-optimized implementation routines.

Calculating the byte width per tuple in Monet is simple: a `chr` or `bit` value takes one byte, whereas `int` and `date` values take four, and a `dbl` takes 8. Text values of type `str` are variable size, hence are stored in Monet in a separate heap for variable-sized data items. The storage per value is one 4-byte integer byte offset in the BAT plus the space taken up in this extra heap. In this heap, string values are stored as standard UTF-8 strings (byte sequences, terminated by a zero byte). Storage in the heap is aligned to 4-byte addresses, hence one should round up the size of the string (zero-terminator inclusive) to the nearest larger-or-equal multiple of four. To complicate this calculation some more, the string type in Monet eliminates double occurrences in a BAT dynamically up to a certain threshold number of different values. That is, on a string column that contains only the strings "male" and "female", the byte-offsets for all values point in the variable-size heap to the same two values. Hence, on large columns with less than a couple of thousand different values, storage requirements per tuple are just the 4-byte integer offset (as the limited heap-size is amortized among all tuples).

Many columns in the TPC-H tables actually store a sub-range of the domain provided by the required column type. For example LINEITEM.TAX only contains one of the nine values { 0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8} while the implementation rules require to store this column in an 8-byte wide `dbl`. For such columns, we use the "Enumeration Type" functionality of Monet, described in Section 5.2.1. From the outside, the `enum1[dbl]` tail column in the `C_LINEITEM_TAX` BAT behaves as a normal `dbl` column. Internally, it is stored in a `enum1[X]`, which takes just 1 byte per tuple, and also enables a number of processing optimizations (like "Enumeration Views", described in Section 5.3.2). In the entire TPC-H schema, we use 13 `enum1[X]` BATs, and four `enum2[X]` BATs, which, for example, reduce the total storage requirements for the ORDERS and LINEITEMS tables, from 112 to 94 bytes and from 94 to 67 bytes, respectively. These byte-per-tuple storage requirements for each C-BAT are listed in

[4]Give or take a couple of months stashed away by various popes in the middle ages.

the fourth column of Figure 7.4. In the title row per table (the bottom row) of this same fourth column, the total required storage for the table in C-BAT representation is listed. These are obtained by adding the bytes-per-tuple storage requirements for all C-BATs of the table, multiplying by the number of rows in the table (which is parametrized by the TPC-H scaling factor SF), and dividing by 2^{20} to get the total storage required for the table in MB. For example, the `SUPPLIERS` table takes SF*1.5MB (e.g., with SF=1 it takes 1.5MB, with SF=100 it takes 150MB, etc.).

Adding all per-table storage requirements, we see that the C-BAT representation of the TPC-H database in Monet takes SF*0.67GB. This is considerably less than the projected dataset size of the TPC-H benchmark, which is SF*1.0 GB, and certainly less than the storage sizes reported on RDBMS implementations of the TPC-H, which vary between SF*6.9 GB and SF*19.1 GB (see Figure 7.2). This significant difference is mainly caused by the fact that as a `BAT[void,X]` stores just values of one type, storage overheads are reduced to a minimum. Relational tuples have to take into account alignments (which forces byte-padding) and typically reserve some space in order to accommodate updates (in case of VARCHAR values, for example). In Monet, the string storage itself is decomposed into the variable-size atom heap, which does not need to contain any slack-space, and in addition, can perform double-elimination.

It should be said, that the relational storage "bloat" factors of 7-19 observed in the last column of Figure 7.2 also (and especially) includes storage space taken up by indexing structures. Later on, we therefore pay some attention to what index structures might be useful in Monet for the TPC-H database and what additional storage this may take.

7.2.2 Updates in Void Columns

The choice of using the `void`-type in the head of the C-BATs draws complications when updates are done. Monet allows `void`-columns to be appended upon only, as the `void`-ness of the column implies that it always contains a densely ascending range of `oid`-s (0,1,2,3,... etc.). This is not a problem when inserting a new tuple, e.g:

```
insert into SUPPLIER (SUPPKEY, NAME, ADDRESS, NATIONKEY)
values (666, 'Microsoft', 'One Microsoft Way, Redmond (WA)', 17, '01-425-8828080')
```

would become in MIL:

```
C_SUPPLIER_SUPPKEY.insert(37, 666);
C_SUPPLIER_NAME.insert(37, "Microsoft");
C_SUPPLIER_ADDRESS.insert(37, "One Microsoft Way, Redmond (WA)");
C_SUPPLIER_NATIONKEY.insert(37, 17);
C_SUPPLIER_PHONE.insert(37, str(nil));
C_SUPPLIER_ACCTBAL.insert(37, dbl(nil));
C_SUPPLIER_COMMENT.insert(37, str(nil));
```

The above supposes that all `C_SUPPLIER_*` BATs have densely ascending `oid` values in the head columns that end at 36, hence inserting a BUN with value 37 is allowed, and will keep the head column of these BATs `void`. Notice that we keep all BATs of equal length by inserting `nil`-s if values are missing.

Because `oid`-s stored in head columns are system-generated, and system-managed, the RDBMS user will never see them. Therefore, an SQL update will never change these `oid`-s, updates just change the values stored in the tail columns:

```
update SUPPLIER set PHONE = '01-425-8828080' where NAME = 'Microsoft';
```

Becomes in MIL:

```
C_SUPPLIER_PHONE.replace(
    C_SUPPLIER_NAME.select("Microsoft").project("01-425-8828080"));
```

The `replace` uses the `BAT[oid,str]` that resulted from putting the telephone number string in the tail of all BUNs returned by the the `select("Microsoft")`. Because the `C_SUPPLIER_PHONE` has a `void` head column, the `replace` can use positional lookup for finding `[37,nil]`, and changing it into `[37,"01-425-8828080"]`.

In the `select`, we look up the head column `oid` value for the Microsoft tuple in the NAME column-BAT. Depending on the presence of a hash-table accelerator structure, the MIL implementation of `select` will use hash-lookup or sequential scan (see Section 5.3.4).[5] Since the `C_SUPPLIER_NAME` is a `BAT[void,str]`, the relation is sorted on head column, not on tail column, hence the binary search implementation of `select` cannot be used. In the discussion of inverted lists (later on) we will discuss how binary search can be used anyway to accelerate such lookup on huge non-memory resident BATs.

Let us now discuss tuple deletes. How can we maintain a densely ascending `oid` sequence when a tuple from the middle is deleted? The answer is that we cannot, and we must leave a hole (a dummy element) in all `C_SUPPLIER_*` BATs. In order to keep track of these holes, we maintain for each relational table an additional "holes" `BAT[oid,void]` named `H_<TABLE>` (see also Figure 7.3. This H-BAT stores a list of head `oid`-s that have been deleted and whose positions are effectively free in the `C_SUPPLIER_*` BATs.

Concerning what to do with the tail values of these "hole" BUNs, two possibilities arise:

- leave them as is. This approach minimizes BUN delete cost. During query processing, we will have to filter out hole-tuples that have been included in a selection or join predicate explicitly.
- replace all tail values with `nil`. As we will see later, this can later benefit query processing performance, as `nil`-s are automatically excluded from (equi-) joins and most selection predicates.

Both strategies can also be mixed; i.e. those columns that are most often used in selection and join conditions could be nullified, and the others not:

```
delete from SUPPLIER where NAME = 'Microsoft'
```

would become:

```
{ var del := C_SUPPLIER_NAME.select("Microsoft").mark(nil);
  H_SUPPLIER.insert(del);
  C_SUPPLIER_SUPPKEY.replace(del);
  C_SUPPLIER_NATIONKEY.replace(del);
  C_SUPPLIER_NAME.replace(del.project(str(nil));
  C_SUPPLIER_ACCTBAL.replace(del.project(dbl(nil))); }
```

The first statement in this sequence looks up the `oid`-s of any Microsoft tuples, and puts them in the head column of a BAT with nil-`oid`-s in the tail. The second statement

[5] On "small" BATs, that can easily be held in memory, Monet automatically creates a hash-table accelerator on such lookup requests.

registers these oid-s in the hole list for the supplier table. The rest of the statements enforce the nullify policy on the SUPPKEY, NAME, NATIONKEY and ACCTBAL columns. Consequently, in this example the PHONE, ADDRESS and COMMENT follow the leave-as-is deletion policy.

While this scheme allows us to implement relational updates efficiently, and to take full advantage of mass column storage in BATs with void head columns (that take minimal space and offer the possibility of positional lookup of oid values), its downside is that it complicates query execution. Queries get more complicated, because each time a base relation is accessed, holes need to be filtered out explicitly. This filtering, however, can be integrated into normal query processing by including the hole-filter-condition as an implicit selection on each table involved. The general scheme is what corresponds to the leave-as-is deletion policy and is demonstrated as follows:

```
select NAME from SUPPLIER where PHONE = '01-425-8828080'
```

should – if holes would not be an issue – be translated into MIL:

```
select(C_SUPPLIER_PHONE, "01-425-8828080").mirror.join(C_SUPPLIER_NAME).print;
```

However, due to the leave-as-is policy in C_SUPPLIER_PHONE, the Microsoft tuple would now still be found. Holes need to be filtered out, as follows:

```
select(C_SUPPLIER_PHONE, "01-425-8828080").
       mark(nil).diff(H_SUPPLIER).mirror.join(C_SUPPLIER_NAME).print;
```

The cost for this filtering is not large: the mark has a view implementation and is free in MIL (see Section 5.3.4), and the size of the H_SUPPLIER is small, hence the diff uses an efficient hash-based implementation (making sequential pass over the left operand, and performing hash-lookup into its right operand).

This filtering can often be omitted when columns are accessed on which the nil-replacement deletion policy is used:

```
select PHONE from SUPPLIER where NAME = 'Microsoft'
```

becomes in MIL:

```
select(C_SUPPLIER_NAME, "Microsoft").mirror.join(C_SUPPLIER_PHONE).print;
```

as the value "Microsoft" had already been replaced by nil during the tuple delete, the MIL select will not find the Microsoft tuple anyway, hence it need not be filtered out later. This optimization works for almost all selection predicates (except isnil(any):bit) and also for joins.

7.2.3 Column Indexing with Inverted Lists

Storage in separate vertical fragments are called in literature "transposed files" [Bat79] or "projection indices" [O'N87], hence one could already consider the BAT storage scheme described above as an "indexing" scheme.

Still, the basic storage in C-BAT[void,X]-s normally stores the tail column unordered, thus if columns are accessed for a SELECT or JOIN query, the MIL select operator has to use a sequential-scan implementation (see Section 5.3.4). For queries that select only a small percentage of the tuples, this is not an optimal solution. One possible improvement would be to create (and maintain) a hash table on each column. The

built-in hash table structure of Monet implements efficient direct hashing and reduces the complexity of the selection operator to $O(sN)$, where s is the selectivity and N the cardinality. This solution is optimal for in-memory situations. The hash-selection algorithm, however, exhibits a random access pattern to the BAT and the hash table structure. If the relational table is so large that all C-BATs together do not fit into main memory, the usual thing to do in Monet is to *memory map* the disk images of the C-BATs into virtual memory. In such a situation, the hash selection algorithm would causes a significant amount of page faults due to its random access property. Performance can then be improved by ordering the column BATs on tail, so the `select` could use a binary-search algorithm to get to the first matching tuple, and retrieve the other matching tuples in sequential order. Also, the binary search approach on a sorted BAT can also accelerate range-selection, which is impossible with a hash table.

Physical Storage of Inverted Lists

Ordering each column on the tail column would mean that the `oid` sequence in the head columns would be different for all BATs. As a consequence, the head columns would no longer be stored in `void` columns, but as `oid`-s. This both has the disadvantage that storage in `BAT[oid,X]` takes generally twice the space as storage in `BAT[void,T]`, and secondly it would become much more expensive to look up values by `oid`. Lookup in an `oid` column uses a scan, binary-search, or hash-lookup, which – although each is implemented efficiently in Monet – is never as quick as positional lookup in a `void`-column. Therefore, table storage where each column becomes a `BAT[oid,T]` ordered on tail is not a good idea.

The idea of ordering on tail becomes only attractive if it is used *in addition* to the normal decomposition in `BAT[void,T]`. Hence, we can maintain `BAT[oid,T]` copies of these BATs that are ordered on tail, which effectively provides us with *inverted list* search accelerators.

Updating Inverted Lists

This simple approach to inverted lists is bound to cause performance problems if the DBMS does not just sustain a read-only query load, but also has to handle updates. If some tuple is updated, deleted or inserted, costly table rearrangement of the tail-sorted `BAT[oid,T]` would be necessary to keep this BAT ordered on tail. Recall that BATs are dense arrays of values (without holes or slack space), so deleting some BUN in the middle would require moving all subsequent BUNs one position back, to fill up the hole. The average cost of moving half of all tuples on every update is fat too expensive. The default implementation of the MIL `delete` operator actually does not do this, it rather keeps the BAT dense by moving an extreme (either first or last) tuple into the deleted position, sacrificing any sorted properties the BAT holds (the exact BAT update algorithms are discussed in more detail in Section 7.4).

In order to be able to accelerate read-only queries with sorted BATs *and* being able to sustain updates at low cost, we use three BATs, called the S-, I- and D-BAT, to store an inverted list index on a column of a relational table:

S-`BAT[oid,T]` ordered on tail, named `S_<TABLE>_<COLUMN>`. It holds the column values for all tuples of the table at the last *sync-point* in sorted order. The tuples appear in such an `oid` order that the tail values form an ordered sequence. The "S" prefix stands for "sorted", as the S-BAT is a (stale) sorted copy of the C-BAT.

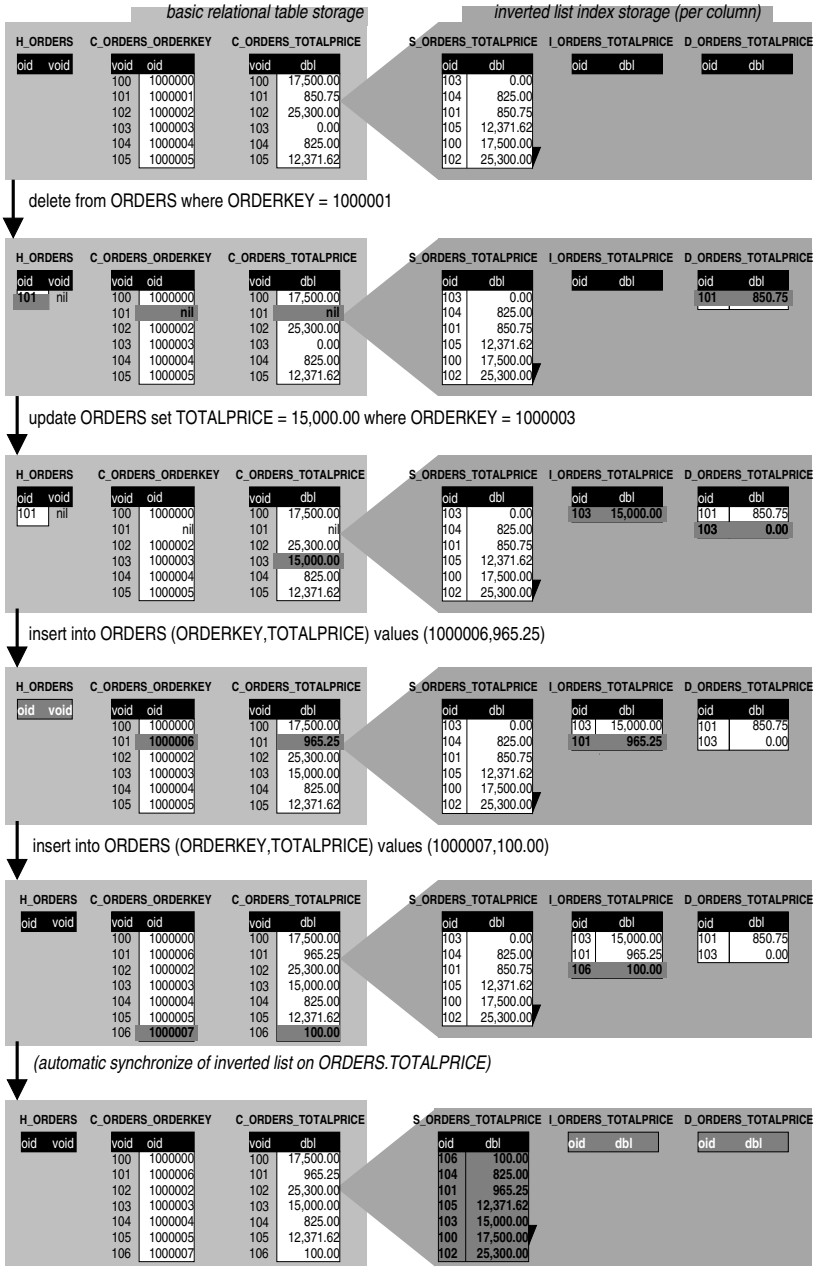

Figure 7.5: Handling Updates in H-, C-, S-, I, and D-BATs

I-BAT[oid,T] named I_<TABLE>_<COLUMN>. It holds the freshest column value for all tuples that have been inserted since the last *sync-point*. The "T" prefix stands for "inserted".

D-BAT[oid,T] named D_<TABLE>_<COLUMN>. It holds the old values for all tuples present at the last *sync-point* that were subsequently deleted. The "D" prefix stands for "deleted".

This can also be seen in Figure 7.3 (the M-BATs mentioned there will be discussed. later in this section).

Note that updates in the C-BAT are represented as a sequence of a delete and an insert, therefore causing an insert in both the D-BAT and the I-BAT.

The update mechanism is illustrated in Figure 7.5. Let us assume that we create an inverted list index on the TOTALPRICE column of the ORDERS table. Hence there will be three additional BAT[oid,dbl]-s:

- one called S_ORDERS_TOTALPRICE that stores all [oid,name] combinations ordered on price (notice the arrowhead drawn at the tail column in Figure 7.5),
- a I_ORDERS_TOTALPRICE holding all new value combinations for tuples that were inserted or ORDERS prices that have been changed.
- a D_ORDERS_TOTALPRICE holding all old value combinations for tuples that were deleted or ORDERS prices that have been changed.

Each time a table column is updated that has an inverted list, the inverted list has to be maintained as well:

insert for each new tuple with oid o that are inserted with column value v, the BUN [o,v] is inserted in the I-BAT.

delete the oid and the old value of the tuple is inserted into the D-BAT.

update is handled as a delete of the old value, followed by an insert of the new value.

The I- and D-BATs grow under tuple updates and inserts. In order to limit the overhead of the I-BAT and in order to effectively optimize query execution, the I- and D-BATs for all columns that have an inverted list index should fit in memory comfortably and should not become significant in size with respect to their S-BATs. Therefore, periodically (or, dynamically, when the size of the I-BAT exceeds a threshold, e.g. a 2% of the size of the S-BAT) we carry out a *sync-point* where all updates are carried through in the S-BAT.

Note that updates could be optimized by checking whether the current value combination [o,v_{old}] is part of the I-BAT (recently inserted), in which case it can be overwritten directly into [o,v_{new}]. This scheme would, however, involve a lookup into the I-BAT whenever an update occurs. If real-time update performance should be optimized, one could choose not to do this optimization, and process all double updates at the sync-point in bulk:

```
X := intersect(D_SUPPLIER_PHONE, I_SUPPLIER_PHONE);
D_SUPPLIER_PHONE.delete(X);
I_SUPPLIER_PHONE.delete(X);
```

At the sync-point, the I- and D-BATs are emptied, and a new S-BAT is created by sorting the C-BAT.

```
S_SUPPLIER_PHONE := C_SUPPLIER_PHONE.order;
D_SUPPLIER_PHONE.delete();
I_SUPPLIER_PHONE.delete();
```

Accelerating Selections with Inverted Lists

The whole purpose of maintaining an inverted list index on a relational table column is to accelerate query performance. The great asset of the inverted list is the S-BAT, which (ignoring updates) contains all column data in sorted order in a `BAT[oid,any]` ordered on tail, hence selection, join and group-by predicates can be executed by implementations that exploit this ordering. Coming back to a previous example, we had a selection on phone number:

```
select NAME from SUPPLIER where PHONE = '01-425-8828080'
```

This query might hurt if the SUPPLIER table were huge and no hash-accelerator were present on the tail of `C_SUPPLIER_PHONE`, as the only applicable MIL implementation of the below `select` would be a sequential scan (see Section 5.3.4):

```
[printf]("%s\n",
         select(C_SUPPLIER_PHONE, "01-425-8828080").
              mirror.join(C_SUPPLIER_NAME));
```

While a sequential scan on a single BAT is much more efficient than a sequential scan on a relational table (as only the data pertaining to one column needs to be scanned), performance can be improved using an inverted list on the PHONE column of SUPPLIER. We modify the `select` as follows:

```
v1 := select(S_SUPPLIER_PHONE, "01-425-8828080");
[printf]("%s\n", v1.mirror.join(C_SUPPLIER_NAME));
```

This `select` accesses a BAT that is sorted in tail, hence can use binary search lookup. This MIL is however not complete: if the I- and D-BATs are non-empty, the `S_SUPPLIER_PHONE` BAT is somewhat stale. Inserted and deleted tuples are taken into account as follows:

```
v1 := select(S_SUPPLIER_PHONE, "01-425-8828080").access(BAT_WRITE);
      v1.insert(select(I_SUPPLIER_PHONE, "01-425-8828080"));
      v1.delete(D_SUPPLIER_PHONE));
[printf]("%s\n", v1.mirror.join(C_SUPPLIER_NAME));
```

We use the `delete` and `insert` instead of `diff` and `sort` as we require bag-semantics.[6] As we assume that the D- and I-BATs are very small (and probably memory resident), the cost of the `delete`, `insert` and second `select` are all negligible.

Accelerating Joins with Inverted Lists

Inverted list indices can also be used to accelerate joins on boolean predicates like "=", "<", ">". Such join operators all have efficient implementations if both join columns are ordered. As the inverted list index provides just that in the S-BAT, it is apt for accelerating a join that, when executed on the standard column BAT, would have to be executed with a nested-loop implementation or would require explicit sorting first:

[6] It might be more elegant to introduce specific read-only MIL bag difference and union operators in the future.

```
join(C_PARTSUPP_SUPPKEY, C_SUPPLIER_SUPPKEY);
```

If we have an inverted list for both SUPPKEY columns, the above MIL can be accelerated to:

```
join(S_PARTSUPP_SUPPKEY, S_SUPPLIER_SUPPKEY);
```

However, the above use of the S-BATs is only legal if both inverted lists are clean: their I- and D-BATs must be empty. One way to ensure this would be to force a sync-point on both inverted lists just before the join on the S-BATs. The heavy cost of executing the join on the C-BATs might justify this.

M-BATs for Inverted List Caching

In the above, we redirect MIL `select` calls from the C-BATs to the S-BATs because the implementation of the `select` uses binary search on BATs that have a sorted tail column (as described in Section 5.3.4, the default implementation uses a sequential scan), and returns a "slice-view" on the S-BAT instead of materializing a new result BAT.

While binary search is obviously more efficient than sequential scan, it still will generate roughly $\log_2(N)$ virtual memory page faults or random disk I/Os to find the desired rows (where N is the number of rows in the table). Common B-Tree implementations minimize the number of random-I/O by using very wide nodes, e.g., such that one node exactly occupies exactly one disk block. Such a tree might have a fanout in the hundreds. If we assume a fanout of 128 (e.g., with a node size of 8*128=1024 bytes), B-Tree search in a two million row table takes $\log_{128}(N)=3$ random I/Os (or page faults) to find the desired row. Simple binary search, on the other hand, takes $\log_2(2000000) = 21$ I/Os. Additionally, the top nodes of a B-tree are often kept in memory. In this example, the two top levels of the B-tree consist of 129 nodes, and 129KB can easily be cached in memory. This reduces the cost of B-Tree search to only one I/O.

Anno 2001, an overhead of twenty random I/Os can represent as much as 200 million idle CPU cycles. Therefore, we want to optimize random I/O cost to S-BATs in Monet, by applying the same technique of caching the "top of the tree" to the inverted list structures. This is done by having a *fourth* BAT, the so-called "memory" M-BAT, in addition to the already mentioned S-, I- and D-BATs for representing an inverted list. The tail column of the M-`BAT[int,`T`]` is constructed by taking a value from the S-`BAT[oid,`T`]` at regular intervals. Therefore, the M-BAT also has a sorted tail column. Its head column contains the row numbers where the values where taken from the S-BAT. This M-BAT is used to restrict the search-space within the S-BAT during selections, with the below scripted MIL procedure:

```
S_SUPPLIER_PHONE.select("01-425-8828080");
```

$\Rightarrow$

```
S_SUPPLIER_PHONE.slice(M_SUPPLIER_PHONE.select(str(nil), "01-425-8828080").max,
                       M_SUPPLIER_PHONE.select("01-425-8828080", str(nil)).min)
                .select("01-425-8828080");
```

This piece of MIL code uses the `slice` MIL operator to take a positional subset of a BAT. As this operator is implemented as a *slice view* (see Section 5.3.2) this is free and subsequent access to the result of the `slice` will cause access to the underlying S-BAT

that is restricted to a (small) subset. If we `load(VM_SEQ)` S-BAT in virtual memory, this in turn means that the I/O (page faults) will only occur for this subset.

These vanilla MIL primitives obtain the same performance behavior as a classical B-Tree with cached top levels: suppose we had constructed the M-BAT by taking a 1-in-128 sample of the S-BAT with 2 million tuples, then the M-BAT holds 16384 tuples (occupying 128KB of memory), and the selected subset will be about 64 tuples in size (occupying 64*8=512 bytes), hence the `select` into the restricted S-BAT just causes one virtual memory page-fault.

7.2.4 Join Index BATs

The join index [Val87] is a binary table storing "surrogate" pairs that represent a precomputed join result. In this context, surrogates might well be represented as `oid`-s. A join index between tables L and R accelerates queries that require a join between L and R, by redirecting the join phase from normal join methods that use the full tables L and R to the much smaller join index representation. Join indices may be used in two directions: from L to R and from R to L. To this purpose, Valduriez advocated the use of duplicated storage of the join index in two B^+ trees: one tree giving fast access on L-`oid` to matching `oid`-s from R, and another tree giving fast access in the other direction (from R-`oid`-s to L-`oid`-s).

In this example of using Monet for relational query processing, we simply consider a join index as an "artificial" data column, that directly refers to `oid`-s of the remote table. For example, take the foreign key relationship from `PARTSUPP.SUPPKEY` to the `SUPPLIER.SUPPKEY` column. Suppose we had an extra column called `=[SUPPKEY$ SUPPLIER_SUPPKEY]`[7] in the `PARTSUPP` table that instead of carrying `SUPPKEY` values carries directly Monet `oid`-s of the matching tuples in the `SUPPLIER` table. Such a join-index-column saves one join step during query processing when we go from `PARTSUPP` tuples to `SUPPLIER` tuples.[8]

In the above example, we would get an additional C-BAT called `C_PARTSUPP_=[SUPPKEY$SUPPLIER_SUPPKEY]`. The additional column could also have been created in the `SUPPLIER` table as `C_SUPPLIER_=[SUPPKEY$PARTSUPP_SUPPKEY]`, but the former is more efficient in Monet as the `SUPPLIER-PARTSUPP` join is N-1, hence we can store the join index column in a `BAT[void,oid]` as a `PARTSUPP` C-BAT.

In order to support efficient updates and join acceleration in both directions, Valduriez proposed storage in two B^+-trees. In the case of our Monet join index representation, access with `PARTSUPP oid`-s to the `C_PARTSUPP_=[SUPPKEY$SUPPLIER_SUPPKEY]` is already fast, because positional lookup can be used into the `BAT[void,oid]`. If this C-BAT is small and always fits in memory, access in the other direction can be made O(1) by using a hash table accelerator on the tail column. If this is not the case (the BAT will not be in memory always), then we can create an inverted list index on the "artificial" `=[SUPPKEY$SUPPLIER_SUPPKEY]` column of `PARTSUPP`, using the exact mechanism as described in the previous section. As explained in Section 7.2.3, this leads to three additional BATs (the S-, I-, and D-BATs) named `S_PARTSUPP_=[SUPPKEY$SUPPLIER_SUPPKEY]`,

[7] Just for the sake of presentation, we name join-index columns with their "mangled" boolean MIL-expression in function notation, where parentheses are substituted by brackets and commas by $.

[8] Notice that object oriented data models provide for such system-managed referencing between tables, by allowing explicitly specified relationships between classes (as opposed to relational systems that have to check referential integrity on table columns).

`I_PARTSUPP_=[SUPPKEY$SUPPLIER_SUPPKEY]`, and `D_PARTSUPP_=[SUPPKEY$SUPPLIER_SUPPKEY]`.

Updating Join Indices

Join indices are typically used to speed up foreign key joins. In those cases, maintenance of the join index can be integrated with referential integrity checking, which alleviates its cost. Let us elaborate with an example. If we added a new supplier, the referential integrity rules that logically follow from the TPC-H schema tell us that there cannot yet be a `PARTSUPP` tuple that matches this previously unseen `SUPPLIER`. Therefore, the join result between `SUPPLIER` and `PARTSUPP` over the `SUPPKEY` column will never alter under inserts of new `SUPPLIER`-s.

Now consider what happens when a new `PARTSUPP` is inserted. Part of the new `PARTSUPP` tuple is a `SUPPKEY` column, which the referential integrity checks for consistency. This check (e.g. by using a `S_SUPPLIER_SUPPKEY.find()` lookup into the inverted list accelerator) must yield a `SUPPLIER oid`. This `oid` can then be used as the value for the "artificial" `=[SUPPKEY$SUPPLIER_SUPPKEY]` column. Therefore, in case of a tuple insert, the referential integrity checks that are required anyway for foreign key values, already deliver the values needed for updating the join index columns.

Similarly, finding out which tuples in the foreign table are affected when an update occurs is already part of the normal referential integrity checking that an RDBMS must perform, and does therefore not count as additional join index maintenance cost. In all, join index updates follow the same (efficient) mechanism as normal C-BAT updates (see Section 7.2.2). If an inverted list is used for accelerating reverse access, the S-, I- and D-BATS for the join index column must also be maintained by the inverted list update mechanism described in Section 7.2.3.

Join Indexing N-M Relations

A special case are N-M foreign key joins. Here, tuples from each side can match zero, one or more times with tuples from the other. Therefore, we cannot use a C-`BAT[void,oid]` for storing the "artificial" join index column in either table. Hence, we use a C-`BAT[oid,oid]` to do this, where the same `oid` can occur 0 or more times in both columns.[9] This C-BAT is kept *sorted* on the head column. This ensures efficient (binary search) access to `oid` values in the head column. An inverted list on the tail column like described above is used to accelerate join access in the other direction (to `oid` values in the tail column). This inverted list index is always present on N-M join indices (it is not optional).

The update protocol for N-M join indices is rather special. Since we want to keep the C-BAT sorted on head, it is treated as an S-BAT: it is not updated directly, but only brought up-to-date occasionally at inverted list sync-points. The I- and D-BATs of the artificial join column are thus used to periodically update *both* the S- and the C-BAT according to the inverted list update mechanism described in Section 7.2.3.

How the Join Index is Used

First, we show a simple example with a join from PARTSUPP to SUPPLIER:

[9] The artificial join-index-column can be interpreted of type `Set<oid>`, as described in Section 4.3.1.

```
select SUPPLIER.NAME
from PARTSUPP, SUPPLIER
where PARTSUPP.SUPPKEY = SUPPLIER.SUPPKEY and
      PARTSUPP.AVAILQTY > 0
```

The above SQL query asks for all supplier names that have a non-empty parts supply listed. The MIL query below can be used if join indices are not present:

```
X := select(C_PARTSUPP_AVAILQTY, 0, int(nil)).reverse;
Y := X.join(C_PARTSUPP_SUPPKEY);
Z := Y.join(C_SUPPLIER_SUPPKEY.reverse);
[printf]("% 15s"\n", Z.join(C_SUPPLIER_NAME));
```

The first and third join are both joins into a `BAT[void,T]`, therefore the positional-join algorithm can be used, which has minimal cost.

The second join probably is the most costly join, because the join tail column of `C_SUPPLIER_SUPPKEY` is not ordered (hence the MIL join implementation must first sort, or create a hash-table or a T-tree explicitly – see Section 5.3.4). However, if we have a join index on the SUPPKEY column of the PARTSUPP and SUPPLIER tables, we can eliminate this join:

```
X := select(C_PARTSUPP_AVAILQTY, 0, int(nil)).reverse;
Y := X.join(C_PARTSUPP_=[SUPPKEY$SUPPLIER_SUPPKEY]);
[printf]("% 15s"\n", Y.join(C_SUPPLIER_NAME));
```

Let us now give an example of using the same join index the other way round:

```
select SUM(PARTSUP.AVAILQTY)
from PARTSUPP, SUPPLIER
where SUPPLIER.SUPPKEY = PARTSUPP.SUPPKEY and
      SUPPLIER.NAME = 'Microsoft'
```

Here, the SQL query asks for the total number of parts available from Microsoft. First, we show the MIL translation without the use of join indices of this same query. This MIL is more sophisticated, as we use the inverted lists on `SUPPLIER_NAME` and `PARTSUPP_SUPPKEY` to improve performance of the `select` and the second `join`, respectively.

```
X := select(S_SUPPLIER_NAME, "Microsoft").reverse;
Y := X.join(C_SUPPLIER_SUPPKEY);
Z := Y.join(S_PARTSUPP_SUPPKEY.reverse);
[printf]("% 9d\n", sum(Z.join(C_PARTSUPP_AVAILQTY)));
```

Notice that the above is only correct if the inverted list on `PARTSUPP_SUPPKEY` is clean. However, if tuples were deleted or inserted since the last sync-point, the third statement should have been:

```
Z := Y.join(S_PARTSUPP_SUPPKEY.reverse).access(BAT_WRITE);
     Z.insert(Y.join(I_PARTSUPP_SUPPKEY.reverse));
     Z.delete(Y.join(D_PARTSUPP_SUPPKEY.reverse));
```

Again, if the join index is used, two joins can be supplanted by one:

```
X := select(S_SUPPLIER_NAME, "Microsoft").reverse;
Y := X.join(S_PARTSUPP_=[SUPPKEY$SUPPLIER_SUPPKEY].reverse);
[printf]("% 9d\n", sum(Y.join(C_PARTSUPP_AVAILQTY)));
```

Since we use the S-BAT of the inverted list on the join index, the same argument as before can be made: if the S-BAT is not clean and updates have been recorded in the I- and D-BATs, the second statement should be corrected to:

```
Y := X.join(S_PARTSUPP_=[SUPPKEY$SUPPLIER_SUPPKEY].reverse).access(BAT_WRITE);
     Y.insert(X.join(I_PARTSUPP_=[SUPPKEY$SUPPLIER_SUPPKEY].reverse));
     Y.delete(X.join(D_PARTSUPP_=[SUPPKEY$SUPPLIER_SUPPKEY].reverse));
```

Given the fact that the D- and I-BATs are kept very small, the additional cost of the additional `delete`, `insert` and `join` operations is low.

7.2.5 Indexing Strategies For OLAP

All RDBMS implementations of the TPC-H listed in Figure 7.2, apply a similar physical design strategy, where the main tables (`ORDERS` and `LINEITEM`) have a *clustered index* on the `SHIPDATE` and `ORDERDATE` columns, respectively. Also, each implementation defines non-clustered indices on all (foreign) keys of each table in the TPC-H.

In the following, we discuss an indexing strategy for the TPC-H database in Monet.

Join Indices

In order to facilitate fast navigation between tables, we create join indices over all foreign key relationships, except the one between `NATION` and `REGION` (with its fixed size of 25 rows this does not make much sense). This leads to the following "artificial" `oid` columns:

- `PARTSUPP.=[PARTKEY$PART_PARTKEY]` an extra column in PARTSUPP (SF * 800.000 rows), taking 4 bytes-per tuple for the C-BAT plus 8 bytes-per-tuple for the S-BAT, hence requires SF*9.6MB.
- `PARTSUPP.=[SUPPKEY$SUPPLIER_SUPPKEY]`, idem, hence another SF*9.6MB.
- `LINEITEM.=[ORDERKEY$ORDERS_ORDERKEY]` an extra column in LINEITEM (SF * 6.000.000 rows), taking 4 bytes-per tuple for the C-BAT plus 8 bytes-per-tuple for the S-BAT, hence requires SF*72MB.
- `LINEITEM_and[=[PARTKEY$PARTSUPP_PARTKEY]$=[SUPPKEY$PARTSUPP_SUPPKEY]]` another SF*72MB. This join index is built on a *composed* key.
- `ORDERS.=[CUSTKEY$CUSTOMER_CUSTKEY]` an extra column in ORDERS (SF * 1.500.000 rows), taking 4 bytes-per tuple for the C-BAT plus 8 bytes-per-tuple for the S-BAT, hence requires SF*18MB.
- `CUSTOMER.=[NATIONKEY$NATION_NATIONKEY]` an extra column in CUSTOMER (SF * 150.000 rows), taking 4 bytes-per tuple for the C-BAT plus 8 bytes-per-tuple for the S-BAT, hence requires SF*1.8MB.

Note that each join index is two-sided, hence consists of the "artificial column" C-BAT in the "left" table (that provides fast access to the "right" table), plus an inverted list index (S-, I, and D-BAT) for fast access from "right" to "left". All included, the storage cost for these join indices is SF*183MB.

Clustered Indices

A clustered index actually re-orders the rows in the base table storage such that the rows appear in the order of the index column. As you can order a base table in only one way, most RDBMS software allows to define one clustered index per table. Anyway, as the TPC-H rules prohibit use of replication of table data in anything else than indices on single columns or (compound) keys, one clustered index is the most TPC-H allows.

The effect of the clustered index on selection queries is best illustrated when one considers a range-selection query on the column with the clustered index on it. This query now selects a consecutive chunk of rows from the base table. This has two benefits. First, such a selection optimizes the amount of I/O, since consecutive rows means that all rows present in the retrieved disk blocks are actually used. Second, instead of random I/O, the RDBMS can use sequential I/O, which is much more efficient. As discussed in Section 3.2.1 this performance advantage of sequential over random I/O is increasing exponentially over time.

The rationale for the RDBMS implementations to create clustered indices on the `SHIPDATE` or `ORDERDATE` columns of the `LINEITEM` and `ORDERS` tables respectively, is that most TPC-H queries have a range- constraint on these columns (or another date column that is strongly correlated to it).

Clustered indexing can also be applied in Monet. In addition to creating an inverted list on the column (e.g. `LINEITEM.SHIPDATE`), this is done by placing the values that are stored in all C-BATs of the `LINEITEM` table such that the `C_LINEITEM_SHIPDATE BAT[void,date]` has an ordered tail column. Range-queries on this column then have the "horizontal" advantages already enjoyed by RDBMSs (all rows in a disk block are used, and sequential I/O) plus the additional "vertical" advantage that only columns in use are retrieved. Monet truly minimizes the amount of I/O required here.

The goal of Monet is to provide ad-hoc query functionality at high performance. We admit that analyzing a fixed set of queries and concluding that two specific clustered indices enhance performance for those queries, does not fit well the ad-hoc philosophy. We believe that the vertical fragmentation and memory/CPU efficiency of Monet do enable ad-hoc querying on any predicate at high performance. However, neither do we want to put ourselves at an unfair disadvantage when comparing the benchmark results of Monet with commercial RDBMS implementations of TPC-H. Therefore, we do create clustered indices on `LINEITEM.SHIPDATE` and `ORDERS.ORDERDATE`. That is, the physical order of the rows stored in the C-BATS follows those respective columns, plus we create inverted lists (S-, I-, and D-BATs) on both columns.

Note that when handling updates to a table with a clustered index in Monet, it is not necessary to immediately reorder the entire table (all C-BATs) if an update breaks the ordering of the index column. The fact that the table is ordered only enhances the access pattern of range-selects on `SHIPDATE` but is not required for database consistency, and breaking the ordering in a small percentage of rows will not influence performance significantly. A full table reorder after updates have broken the ordering, could be done periodically, e.g. only at data warehouse refresh time.

The extra storage costs of these clustered indices are the two inverted lists (one of about SF*7 million values, one of about SF*1 million). The disk space needed for these BATs totals SF*68MB (each S-`BAT[oid,date]` is 8 bytes wide).

Non-Clustered Indices

The question now arises whether additional (non-clustered) indices should be created on additional columns, and if so, on which? Recall that the preferred indexing structure in Monet is the hash table, if the database fits in main memory. Therefore, inverted lists in Monet are relevant only when the database does not fit in main memory. This we may suppose in case of the TPC-H benchmark. We now provide a simple model, that describes the I/O cost in both Monet and a "normal RDBMS" for evaluating a generic select-project query, with or without non-clustered indices:
`select` $\texttt{COL}_1$`,..,`$\texttt{COL}_p$ `from TABLE where` $\texttt{COL}_0$ `= X`

Let f denote the fraction of tuples selected (the selectivity), and n the total number of columns of the table, and C the relation cardinality (the number of tuples). For simplicity, we assume a column-width $w = 4$ bytes for all columns. The query is called a select-project query as it typically consists of two phases. In the *select phase*, we determine which tuples are selected (this involves COL_0), and in the the *project phase*, we fetch the other columns required for the query (COL_1,..,COL_p).

We now discuss the **index-select** execution strategy that one can apply if one has a non-clustered index (e.g. an inverted list) on the selection attribute. As discussed before, the inverted list is a file of `[oid,v]` combinations that tells which tuples have the value `v`. Because this file is ordered on the values, an equi- or range-predicate selects a consecutive range of entries from the inverted list file, where the `oid`-s indicate which tuples are selected. We ignore for simplicity the cost of determining the start and end-point of this consecutive range (which can be optimized, e.g. with the M-BAT technique as described earlier). The I/O cost for reading the selected `oid`-s consists of a sequential read of $fC2w$ bytes. This divided by the sequential bandwidth S gives the number of seconds required: $(2fCw)/S$. The factor 2 is caused by the fact that an inverted list stored pairs of values and object or tuple-ids (e.g. for an `int` column in Monet we get a `BAT[oid,int]` of width 2*4=8 bytes).

After the select-phase, we need to fetch column values for the selected `oid`-s. This can be done in two different **projection** strategies:

random I/O We take the list of `oid`-s from the select phase, and use single block I/O requests (or page faults in virtual memory) to look up these tuples in the file that stores the main table ("normal" RDBMS case) or in the files that store the C-BATs of the needed columns (Monet case). It is good practice to sort the `oid`-s first to avoid requesting a disk block twice for different tuples. The probability for a disk-block to be selected is $1 - (1 - f)^T$, where T stands for tuples per block. In the relational case, this is $T_{rel} = B/(nw)$, and in the case of Monet $T_{monet} = B/w$. The total costs of this phase for a "normal" RDBMS is the number of blocks in the relation Cnw/B times $(1-(1-f)^{B/(nw)})$ times the random access cost per block R, and for Monet we get $p(Cnw(1-(1-f)^{B/w}))R$.

sequential I/O We scan only the projected attributes, hence we get I/O costs of $(pCw)/S$. A "normal' RDBMS must sequentially scan the entire relational table of Cnw bytes, hence its I/O cost becomes $(Cnw)/S$.

For a "normal" RDBMS, the sequential strategy actually would not even use the non-clustered index, as evaluating the selection predicate could just as well be done during the sequential scan. In Monet, however, replacing the index scan by a sequential

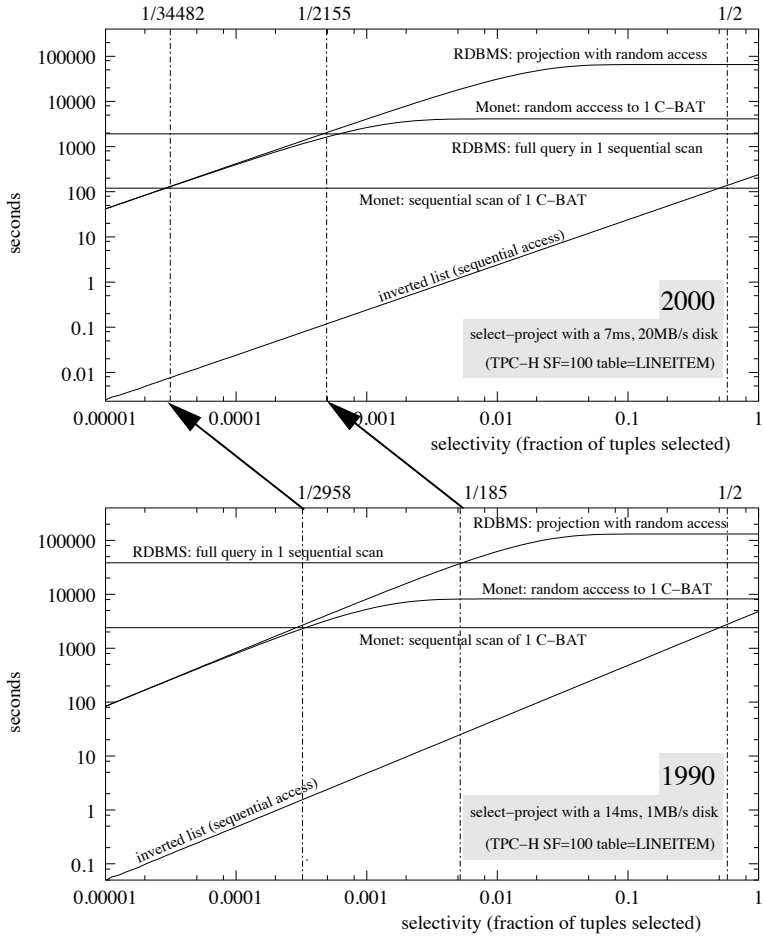

Figure 7.6: When Do Non-Clustered Indices Make Sense?

scan over the C-BAT containing the selection attribute is also a possibility. This requires reading Cw bytes of sequential I/O, which costs $(Cw)/S$ seconds. We call this alternative strategy which does not use any non-clustered index the **full-sequential** strategy.

Figure 7.6 shows the cost predictions given by the model on typical hardware of the year 2000 (up) and 1990 (down). The query modeled has one selection attribute and one projection attribute on the SF=100 LINEITEM table of TPC-H (600 million tuples). The result of the sequential scan execution strategy in a "normal" DBMS is directly given (the highest horizontal line). For obtaining the RDBMS strategy of index-select followed by projection with random access I/O, one should add the corresponding two lines (which is always dominated by random access I/O). That should also be done for both possible index-select strategies in Monet: index-select followed by sequential-scan, and index-select followed by random access. We used only one projection attribute ($p = 1$) in the query, so one can easily deduce how other values of p would perform in the Monet case (simply multiply the Monet sequential

scan or Monet random I/O numbers by p). Finally, the Monet strategy of using sequential scan both for selection and projection is obtained by multiplying the Monet sequential scan line by $p+1$.

These results show that for a "normal" RDBMS, it currently makes sense to use an index for accelerating selections of up to about 1 in 2000 tuples. This already very small selection percentage is rapidly decreasing with the hardware trend that I/O bandwidth follows Moore's law but I/O latency does not, as discussed in Section 3.2.1. A decade ago, when disks had a bandwidth of 1MB/s and a latency of 14ms (as opposed to 20MB/s and 7ms), the break-even point still was at about 1 in 200 (as can be seen in the right plot of Figure 7.6). Any linear rule like "use an index below 1 in 200" or "below 1 in 2000" will soon be outdated as Moore's law continues to hold (not even considering the now widespread usage of RAID devices that increase sequential disk bandwidth even further).

As a conclusion, non-clustered indices are useful for accelerating selections in a "normal" RDBMS only when a handful of tuples are retrieved. A common example of select-queries that only generate such a handful of result tuples, are equi-selects on one (foreign) key value. While that is typical for OLTP loads, in query-intensive areas like OLAP and Data Mining selection percentages are much higher, hence a sequential scan is almost always faster than a query evaluation strategy that uses an index.

In Monet, the same line of reasoning goes for using random access for retrieving projection values, the difference being that the break-even point is even earlier (about 1 in 35000, in this configuration). This is caused by the fact that a disk image of a BAT stores much more tuples per disk block than a non vertically fragmented relation, hence the probability that a disk-page is needed by the random access strategy increases.

Random I/O access in Monet, as little applicable as it is, is technically achieved by memory-mapping the C-BATs and giving the `MADV_RANDOM` virtual memory advise, before accessing them with the MIL `join` operator. As mentioned above, this advise is only wise if very few tuples are selected. Normally, it is better to use sequential file I/O for reading the BAT, or use a memory-mapped BAT with advice `MADV_SEQ` (which is default) and profit from todays high sequential I/O bandwidth.

When comparing in Monet the index-select/sequential-project strategy with the sequential-select/sequential-project strategy we remark the following:

- Since the S-BAT is roughly twice as big as the C-BAT, the amount of sequential scan volume in an index-select is twice that of a sequential-select, if all tuples are selected. Hence, index-select/sequential-project is only better when less than half of all tuples is selected.

- the amount of I/O needed in the index-select/sequential-project is $fC2w + pCw$ bytes versus $(p+1)Cw$ for the sequential-select/sequential-project strategy. The I/O cost reduction when using the index-select is maximal when $f = 0$, and is hence limited to $p/(p+1)$. In other words, for an average query that involves 5 attributes (1 for select, 4 project), we can save at most 20% of I/O cost using inverted lists.

Going back to the issue of storing the TPC-H database in Monet, one "extreme" index design strategy that fits the ad-hoc philosophy would be to create an inverted list on *each* column (as suggested by the inital DSM proposal [CK85]). Hence S-, I-, D- and M-BATs would be created on all columns. As the I-, D-, and M-BATs are

kept very small, the extra disk storage required would come mainly from the S-BATs. Column five of Figure 7.4 shows the number of bytes per row occupied in each S-BAT. In when summing all these totals, we see that the "extreme" indexing strategy would cost only SF*1.33GB in Monet.

However, in the light of our previous analysis and the fact that the selection clauses in OLAP queries like those in TPC-H typically select much more than a handful of tuples, we conclude that all these inverted lists would only slightly enhance performance ($p/(p+1)$) while they do cause extra update and storage cost (conversely said, inverted lists on the key columns only are useful in OLTP settings). When non-clustered indices are not used (apart from those introduced by the join indices), this reduces the total disk storage for TPC-H in Monet to SF*0.92GB, which forms a sharp contrast to the SF*6GB - SF*16GB sizes observed in the commercial RDBMS implementations of TPC-H. One reason for this is that the commercial numbers include file redundancy in RAID devices, which can broadly be estimated to cause a factor 2-3 increase of data sizes. In a comparable Monet implementation, such RAID storage should also be used, but even then Monet is a factor 2-5 more efficient in storing data.

7.3 Query Execution

In the following we outline a simple SQL-to-MIL translator that is the heart of the SQL-speaking Monet front-end. The first step in this translation is a rather literal translation of the SQL query into a *relational algebra tree*, where the tree nodes are (standard) relational operators [Cod70, Dat85].

One of the assumptions here is a query plan that has already undergone *strategical* query optimization (i.e. it has been determined already which relational operators will be used and in which order). As this is a precondition of the translation of the relational query into MIL (which deals with physical execution), this means that strategical optimization has to do without physical details. We think this can be adequately done by taking the abstract measure *tuple flow*: the summation over all nodes in the query graph of the number of tuples that flow through them. In most cases, we expect the optimal physical query graph to be the one that corresponds to a logical query plan with lowest tuple flow. While obviously cases exist where a difference in the physical cost of one algorithm over the other might compensate for the difference in tuple flow, making a 'cheaper' physical algorithm on more tuples faster than a more 'expensive' algorithm on less tuples, this is not probable, and it seems safe to assume that at least the "bad" query plans are avoided this way; which has been stated as the practical goal for RDBMS query optimization [WP00]. As such, one could use any approach of the plethora of query optimization techniques available to minimize this tuple flow, e.g. using heuristic [ABC+76], random-based [GLPK95, GLPK94, PGLK97, Pel97], histogram-based [IP95, PIHS96], or even sampling-based techniques [AGPR99].

In the following discussion, we will not go into query optimization issues, neither considering the order of the relational operators (i.e. strategical query optimization, see Section 5.3.1) nor the choice of the appropriate algorithm for an operator (tactical optimization, see Section 5.3.3), but focus on how the MIL primitives are sufficient to execute SQL query plans efficiently.

7.3.1 Relational Algebra Trees

In this section, we assume that the MIL translator works on a relational query tree and employs a fixed order of evaluation, which is *right-depth-first*[10]. The query tree has been produced by a query optimizer. The task at hand is to generate the corresponding MIL code that executes the query on the database tables stored in persistent C-, S-, I-, D- and M-BATs, as described in the previous Section. The query tree consists of the *basic* nodes **table** (for base tables), and **join**, **select**, **aggregate** and **sort** (the basic relational query processing operators), augmented with the *auxiliary* nodes **scope**, **var** and **relation**:

```
<RELATION> ::= relation( IDENT, scope( { var(IDENT, <EXPR:any>) }*, <TABLE> ) )
    <BASE> ::= relation( IDENT, scope( { var(IDENT, <EXPR:any>) }*, table(IDENT) ) )
   <TABLE> ::= <OPERATOR> | table(IDENT)

<OPERATOR> ::=       join( <EXPR:bit>, <BASE>, <RELATION>)
             |     join1( <EXPR:bit>, <BASE>, scope( { var(IDENT, <EXPR:any>) }*, <TABLE>))
             |    select( <EXPR:bit>, <RELATION>)
             | aggregate( { <EXPR:any> }*, <RELATION>)
             |      sort( { <EXPR:any> }*, <RELATION>)

<EXPR:T> ::= IDENT.IDENT | CONST:T | IDENT(<EXPR:any>,..,<EXPR:any>):T
```

The above is BNF extended with two notations:

- `{X}*`, which means a *list* of zero or more elements `X`.
- typed clauses `Y:Z`, meaning `Y` yielding when evaluated a value of type `Z`.

The latter construct is used to denote expressions of a certain MIL type. As embodied by the last rule, expressions are either column references or literal MIL constants (see Section 4.1), or *simple* MIL-operators; i.e. those `f(..any..):any` that have a simple (non-BAT) return type and parameters. Column reference expressions consist of a relation identifier, a dot and a *column identifier*. Base relations (those **relation** nodes with a **table** grandson, as produced by the `BASE` rule) have as initial allowed set of column identifiers all column names of the persistent table.

Additional column names can be added by the **scope** nodes, which introduce a new column defined by an expression using the **var** clause. Column references in expressions in the query tree may reference columns from the current **relation** as well as from any **relation** that is reachable downward in the tree, without going past any **aggregate** nodes.

Note that columns introduced by **scope** nodes ease the elimination of common sub-expressions. This task, as we indicated before, belongs to the query optimizer.

The basic relational operator nodes have bread-and-butter query processing semantics:

select(*boolExpr,rel*) return a subset of all tuples in *rel* for which evaluating *Expr* gives `true`.

join(*boolExpr,leftRel,rightRel*) returns a new tuple for each combination of tuples from *leftRel* and *rightRel* for which evaluating *boolExpr* yields `true`.

[10] We use linear, non-bushy, join trees to indicate a clear sequential join order. The right-deep order is chosen for notational purposes only

aggregate(*exprList,rel*) returns one tuple for each unique combination of expression results after evaluating the expressions in *exprList* for all tuples in *rel*.

sort(*exprList,rel*) returns all tuples of *rel*, but in lexicographical order of expression results after evaluating the expressions in *exprList* for all tuples in *rel*.

The root of the tree is a **relation**, whose table represents the result of the query. This table is made up by all tuples in the relation, where the tuples consist *only* of those columns mentioned in its **scope** list.

The **join1** node is a special case of the **join** node, which has as additional semantical constraint that the join between the right and left relation (its third and second parameters) couples exactly one right tuple to each left tuple. Certain join conditions with this property can be determined by analyzing foreign key relationships stored in the data dictionary. Each basic node in these relational algebra trees forms a "virtual" relational table with its own set of tuples; however **join1** nodes do not add or remove tuples to/from the relational table of its rightmost parameter, they just may add columns to the table. In the case of MIL query plans, this (rather common) condition allows to save a number of processing steps; and that is why we distinguish between **join1** and **join** nodes.

As an example, we use query 9 of TPC-H, which is among the most costly queries in many RDBMS implementations. Its SQL syntax is as follows:

```
select
      NATION.NAME                                        as NATION,
      extract(YEAR from ORDER.ORDERDATE)                 as YEAR,
      LINEITEM.EXTENDEDPRICE * (1 - LINEITEM.DISCOUNT) -
                PARTSUPP.SUPPLYCOST * LINEITEM.QUANTITY as AMOUNT
from
      PART, SUPPLIER, LINEITEM, PARTSUPP, ORDERS, NATION
where
      LINEITEM.SUPPKEY = PARTSUPP.SUPPKEY and LINEITEM.PARTKEY = PARTSUPP.PARTKEY
      and LINEITEM.ORDERKEY = ORDER.ORDERKEY
      and LINEITEM.SUPPKEY = SUPPLIER.SUPPKEY
      and SUPPLIER.NATIONKEY = NATION.NATIONKEY
      and LINEITEM.PARTKEY = PART.PARTKEY
      and PART.NAME like '%green%'
group by
      NATION, YEAR
order by
      NATION, YEAR
```

Figure 7.7 shows a translation of TPC-H query 9 into relational algebra using a right-deep join plan. In the graphical representation of the same plan depicted by Figure 7.8, the **relation** nodes are painted as grey boxes in the background, with the relational operator nodes they directly encapsulate drawn over them. The tree is annotated with new column identifiers introduced by **scope** nodes at the positions where these occur.

MIL is generated from a relational algebra tree by traversing the basic relational operators in a right-depth-first order and generating MIL for each node. In the following, we will discuss exactly how this code generation works for each kind of node (including the use of join indices – which are mentioned explicitly in Figure 7.8).

7.3.2 Select

The **select** node identifies the tuples that satisfy a boolean expression tree. Let us assume in the following that the tree has a depth of more than one, i.e. that at least

```
relation(R9,
 scope([var(name, R5.name), var(year, R7.year), var(amount, R8.amount)],
  sort([R8.name,R8.year],
   relation(R8,
    scope([var(name, R5.name), var(year, R7.year),
           var(amount, sum(-(*(-(dbl(1),R1.DISCOUNT),R1.EXTENDEDPRICE),
                           *(R3.SUPPLYCOST,R1.QUANTITY))))],
     aggregate([R5.name,R7.year],
      relation(R7,
       scope([],
        join1(=(R4.NATIONKEY,R5.NATIONKEY),
         relation(R5, scope([], table(NATION))),
         scope([],
          join1(=(R3.SUPPKEY,R4.SUPPKEY),
           relation(R4, scope([], table(SUPPLIER))),
           scope([],
            join1(and(=(R2.SUPPKEY,R3.SUPPKEY), =(R2.PARTKEY,R3.PARTKEY)),
             relation(R3, scope([], table(PARTSUPP))),
             scope([var(year,YEAR(R2.ORDERDATE))],
              join1(=(R1.ORDERKEY,R2.ORDERKEY),
               relation(R2, scope([], table(ORDER))),
               scope([],
                join(=(R0.PARTKEY,R1.PARTKEY),
                 relation(R1, scope([], table(LINEITEM))),
                 relation(R6,
                  scope([],
                   select(like(R0.NAME,"%green%"),
                    relation(R0, table(PART))))))))))))))))))))
```

Figure 7.7: Relational Algebra Notation of TPC-H Query 9

the root node is a MIL operator (as the **select** has a boolean expression, it must be a MIL operator that returns the type `bit`).

This *basic expression evaluation algorithm* executes the expression tree left-bottom up, generating for each intermediate node (which is a MIL-operator expression `f(..)`) the multi-join map version of that operator (`[f](..)`).[11] The leaf nodes are either constants, which are generated as-is as parameters to the multi-join maps, or column references. For each column, we have a `BAT[void,any]` available. In the case that the select works on a base relation, these are the C-BATs. In other cases, these BATs are constructed using the basic projection algorithm , which is discussed later on. Each multi-join map yields again a `BAT[void,any]` result, which serves as parameters to multi-join maps higher up the tree.

The `BAT[void,bit]` that results from running the basic expression evaluation algorithm on the root expression, is then turned into an `oid`-list by performing a `select(true)` on it. This produces a subset BAT we call the *selection-BAT*, that has all `oid`-s of the selected tuples in its head column. The selection-BAT is then fed into a `mark.reverse` in order to produce the *pivot* `BAT[void,oid]`. The pivot is a central concept in MIL query processing; its head column contains one `oid` (in densely ascending order) for each tuple of the new relation, where each tail contains the `oid` of the original relation that produced it.

Suppose we have a base table `TAB` with column `COL` and the selection expression `42.0 < foo(TAB.COL,1.0) and TAB.COL ≥ (4*10+2)`, where `foo()` is some arbitrary simple MIL operator. The generated MIL by the basic select code generation is:

```
v1 := [foo](C_TAB_COL,1.0);
```

[11] If all parameters to the operator are constants, the expression counts as a constant which can be evaluated by simply executing the MIL operator – without multi-join map. In this way, constants are eliminated bottom-up.

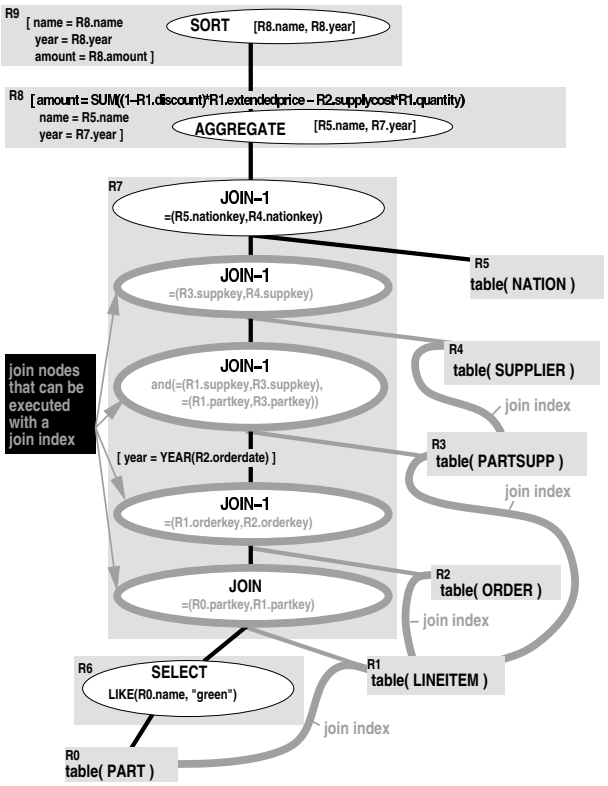

Figure 7.8: Relational Algebra Tree for TPC-H Query 9

```
v2 := [<](42, v1);
v3 := *(4,10);
v4 := +(v3,2);
v5 := [>=](C_TAB_COL, v4);
v6 := [and](v2, v5);
v7 := v6.select(true);                # selection BAT
v8 := v7.mark.reverse;                # pivot BAT
```

An improvement over this approach is to use the built-in `select` range- and equi-select operator, as it directly generates a `BAT[oid,any]` result that can serve as so-called selection-BAT. This code is generated for the root node or for interior nodes when all operators higher in the expression trees are either `or`, `and` or `not`, and the operator is either `=`, $>$, $<$, $\leq$, $\geq$, or is a range condition, that is `and(`$>$`,`$<$`)`, `and(`$\geq$`,`$<$`)`, `and(`$>$`,`$\leq$`)` or `and(`$\geq$`,`$\leq$`)`. In case of an interior node, the upper `and/or/not` nodes generate `intersect(mirror(left),mirror(right))`, `union(mirror(left),mirror(right)` and `diff(mirror(param),mirror(col))`[12]

Taking this modified strategy into account, we would generate the following MIL:

```
v1 := [foo](1.0, C_TAB_COL);
v2 := v1.select(flt(nil), 42.0);
v3 := *(4,10);
v4 := +(v3,2);
```

[12]The second parameter to `diff` is a column from the underlying relation that can be chosen at random. For caching purposes, it is best to re-use the last used table column for this purpose.

```
v5 := C_TAB_COL.select(v4, int(nil));
v6 := intersect(v2.mirror, v5.mirror); # selection BAT
v7 := v6.mark.reverse;                 # pivot BAT
```

One detail that needs to be mentioned for this strategy is *expression normalizing* such that simple comparisons have the column reference as left parameter and the constant as right parameter (this involves turning $<(42, X)$ into $\geq(X, 42)$, etc.). This order of parameters is required when using the MIL `select` operator.

As a final refinement to the select node code generation, the SQL front-end should make use of the inverted list indices when present to enhance performance of range selection expressions on base columns. If we assume that `TAB.COL` has an inverted list and thus a sorted S-BAT and an in memory M-BAT summary plus I- and D-BATs with recent notifications, the following MIL should be generated:

```
v1 := [foo](1.0, C_TAB_COL);
v2 := v1.select(flt(nil), 42.0);
v3 := *(4,10);
v4 := +(v3,2);
v5 := S_TAB_COL.slice(M_TAB_COL.select(int(nil), v4).min, S_TAB_COL.count)
              .select(v4, int(nil)).access(BAT_WRITE);
      v5.insert(I_TAB_COL.select(v4, int(nil)));
      v5.delete(D_TAB_COL);
v6 := intersect(v2.mirror, v5.mirror); # selection BAT
v7 := v6.mark.reverse;                 # pivot BAT
```

As an optimization in those **select** nodes that emit just one MIL `select` from a C- or S-BAT to arrive at the selection column, we have the selected column values already in the tail the select-result. If this column is being referenced in an expression upward in the algebra tree, the basic projection algorithm (described below) does not need to join into the C-BAT holding it, but use the tail column of the select-result instead, with its head column newly `mark`-ed.

For completeness, we discuss what happens if the select expression does not have a MIL-operator as root of the expression tree. One possible case is that the expression is a simple column reference of type boolean. In that case, we can skip the basic expression evaluation phase, and directly select the `true` values to arrive at the selection BAT. The other case is when the selection expression is a constant expression (i.e. one without column references at all) of type `bit`. In that case, we generate the following code for generating the pivot in `v02`:

```
v1 := ..code for computing constant omitted..
if (not(v1) or isnil(v1)) {
  v2 := bat(void,oid); # empty BAT
} else {
  v2 := C_TAB_COL.mirror;
}
```

7.3.3 Basic Projection Algorithm

When expressions are evaluated, column references must be resolved into column-BATs.

For each edge in the tree between two relation nodes, the code generation for the father node computes a so-called pivot `BAT[void,oid]`, which relates `oid`-s in the father relation (head column of pivot) to `oid`-s in the child relation (tail column of pivot). This was already described in the case of the **select** node, and will also be described for the other kinds of nodes (note that the binary **join** node produces two pivots, one for each join relation).

The basic projection algorithm just joins all pivots on the path between the **relation** node where the expression occurs (source) and the destination **relation** node of the column reference, starting at the source downward. As these actions are all of the form `join(BAT[void,oid], BAT[void,oid)`, the efficient Positional-Join algorithm gets used. Even better, by virtue of the *radix-accelerator* described in Section 6.5 such joins are transparently accelerated when necessary by the cluster-decluster cache-optimized join strategy.

The end result of these joins is a combined, "path", pivot `BAT[void,oid]` that relates `oid`-s from the expression-node to the destination node, deeper in the tree. If the column definition is a column of a base table, we can directly join the "path" pivot with its C-BAT in order to obtain the desired projection column. Note also that if the source relation is the direct father of the base relation, we do not need to create the "path" pivot as we can use right away the "single-edge" pivot created by the father.

If the destination, however, is a new column introduced by a **var** expression in the **scope** clause of the destination relation, this same projection algorithm is recursively used to create a projection BAT for that scope column (if the code for that column had not been generated before).

One can see that this basic algorithm can be enhanced by pivot re-use. The most obvious form is to re-use "path" pivots for all projection columns over the same path. A second form is re-using earlier created "path" pivots for creating pivots over a superset of that path. For example, if node `C` refers through node `B` to a column in base relation `A`, it will construct a "path" pivot `C-A` by joining `C-B` and `B-A`. Then, if relation `D` refers through `C` and `B` also to `A`, the pivot `D-A` is constructed by joining `D-C` with the earlier created `C-A`. As we consider right-deep trees only, a simple check for already generated "path" pivots along the path between column reference and definition (and choosing the longest one if there are multiple pivots eligible) provides a good degree of re-use already.

7.3.4 Join

The most common join expression is simple equi-join between two single columns. This is directly supported by the MIL `join` operator. Suppose we have tables `RIGHT` and `LEFT` that both have a column `KEY` and we have a join expression `=(RIGHT.KEY,LEFT.KEY)`. This generates the following MIL:

```
# the basic equi-join
v0 := join(C_LEFT_KEY, C_RIGHT_KEY.reverse);

# dynamically optimized partial cluster
v0 := v0.sql_joincluster(C_LEFT_KEY.count, <ln>, <lw>, C_RIGHT_KEY.count, <rn>, <rw>);

# pivot creation
v1 := v0.mark.reverse;
v2 := v0.reverse.mark.reverse;
```

Here `v1` is the pivot between the join node and the right node, and `v2` the pivot between the join node and the left node.

In Section 6.5.2 we showed that if the number of tuples in one of the join relation is so high that its C-BATs exceed the memory cache size, Positional-Joins with random access generated by the basic projection algorithm, will thrash the memory cache. A solution to this problem for one of the two relations is to partially cluster the join result on the `oid`-s of that relation. This should be done before creating the pivots.

This tactical optimization can be performed by the below `sql_clusterjoin` MIL procedure, that dynamically figures out whether the subsequent computation of column projections can be accelerated by partially clustering the join result. The variables `<ln>`, `<lw>`, `<rn>` and `<rw>` follow from the query plan and can be emitted as constants directly by the SQL front-end.

```
PROC sql_joincluster(
       BAT[oid,oid] joinresult,
       int left_count, left_nproj, left_maxwidth
       int right_count, right_nproj, right_maxwidth) : BAT[oid,oid]
{
  IF ((left_nproj > 0 and left_count*left_maxwidth > CACHE_SIZE/2)
       or (right_nproj > 0 and right_count*right_maxwidth > CACHE_SIZE/2))
  {
    IF (left_count*left_nproj*left_maxwidth > right_count*right_nproj*right_maxwidth) {
      VAR nbits := 1 + log2((left_count*left_maxwidth)/CACHE_SIZE);
      VAR nignore := min(0, (1 + log2(left_count)) - nbits);
      VAR npasses := 1 + (nbits-1) / CACHE_LINES;
      RETURN joinresult.radix_cluster(npasses, nbits, nignore);
    } ELSE {
      VAR nbits := 1 + log2((right_count*right_maxwidth)/CACHE_SIZE);
      VAR nignore := min(0, (1 + log2(right_count)) - nbits);
      VAR npasses := 1 + (nbits-1) / CACHE_LINES;
      RETURN joinresult.reverse.radix_cluster(npasses, nbits, nignore).reverse;
    }
  }
  RETURN joinresult; # no clustering necessary
}
```

The above MIL procedure takes into account the number of *direct* column references that the join node makes to its join parameter relations, *as well as* the sizes of these two relations, and the maximum BUN-width of the C-BATs involved (this can be determined by looking at the column types). Partial clustering is only done if at least one of the relations has a projection column that does not fit half of the cache. Which relation to cluster on is then heuristically selected by comparing the "projection volumes" (the product of size, width and number of projection columns) of both relations. Calling of this `sql_clusterjoin` is done as a post-processing step after any "real" join code discussed in this section, except the exactly-one **join1** case.

Notice that applying the cluster-decluster join strategy on the projections of the other relation is done automatically when needed by virtue of the *radix-accelerator* and therefore does not need explicit code generation.

Multi-Column Equi-Join

The advantage of MIL is that its operators can be implemented for great efficiency, as they have a operator signature with a low degree of freedom. The downside is that this low degree of freedom is achieved by processing data column-at-a-time, which complicates multi-column operations. In the case of multi-column equi-join – this happens e.g. when keys consist of multiple columns – one should combine all columns involved on both sides of the join into numerical (preferably `int`, but also `lng`) columns, and then equi-join those two numerical columns. The best way to do this is to use a *perfect* mapping function `f(...any...):int` that maps each combination of values on a predictable number, and numbers are never assigned to multiple combinations.

In practice, such mapping functions for numerical types yield a numerical type that has a bit-width which is larger than or equal to the sum of the bit-widths of all types of the columns involved, and we use simple bit-shifts and -ors to combine values efficiently

in a non-overlapping way. This mapping strategy can be best executed by a relational algebra tree rewrite. The script below gives an example of a two-column (both `int`) equi-join that is transformed into an equi-join on single-column `lng`-s:

```
join(and(=(LEFT.KEY1,RIGHT.KEY1), =(LEFT.KEY2,RIGHT.KEY2)),
     relation(LEFT,  scope(S1,BASE))
     relation(RIGHT, scope(S2,REL)))

  ⇒

join(=(LEFT.NUM,RIGHT.NUM),
     relation(LEFT, scope(S1+[NUM=xor(<<([lng](t1.k1),32),t1.k2)],BASE))
     relation(RIGHT,scope(S2+[NUM=xor(<<([lng](t2.k1),32),t2.k2)],REL)))
```

When a perfect mapping function does not exist, one can use the MIL operator `hash(any):int` on all non-integer column-BATs using the multi-join map `[hash](b)`, and `[xor](a,b)` all the resulting integers together. This mapping function may be a good hash function, but not a perfect hash function, therefore in the resulting join of integer columns, *false hits* may occur. These then have to be filtered out. Again, we give an example of this strategy as a relational algebra tree rewrite of a two-column equi-join into a single-column equi-join on non-perfectly mapped hash integers, followed by a select to filter out false hits:

```
join(and(=(LEFT.KEY1,RIGHT.KEY1), =(LEFT.KEY2,RIGHT.KEY2)),
     relation(LEFT,  scope(S1,BASE)),
     relation(RIGHT, scope(S1,REL)))

  ⇒

select(and(=(LEFT.KEY1,RIGHT.KEY2), =(LEFT.KEY2,RIGHT.KEY2)),
     relation(TMP, scope(□,
         join(=(LEFT.NUM, RIGHT.NUM),
             relation(LEFT,  scope(S1+[NUM=xor(hash(LEFT.KEY1), hash(LEFT.KEY2))],BASE)),
             relation(RIGHT, scope(S2+[NUM=xor(hash(RIGHT.KEY1),hash(RIGHT.KEY2))],REL))))))
```

A special property of the MIL `hash()` function is that it never sets the highest bit, hence the `xor()` cannot set it either, hence the resulting combined `NUM` values can never by accident form the special `int(nil)` value (which would lead to missed join hits due to the nil-semantics).

Generic Join Expressions

The generic fall-back code generation for join expressions that are not equi-joins is to loop over the right relation and for each tuple use the **select** node code generation to select tuples from the left relation. This is done by generating a scripted MIL procedure on-the-fly that receives as parameters the current right `oid` plus all referenced columns from right relation (and its sons – to obtain column BATs for those the basic projection algorithm is used first). Inside the body of the MIL procedure goes the select code that generates the selection BAT. In this code generation, all references from columns in the right relations (or its sons) are remapped to the MIL procedure parameters.

```
v1 := bat(oid,oid);

proc tmpproc(oid o, any c1, .., any cn) : bat[oid,oid] {
  var selbat := ..use select-node code generation for selecting in left relation ..
  v1.insert(selbat.project(o).reverse);
}
[tmpproc](right_col1.mirror, right_col1, .., right_colN);
undef tmpproc;

v2 := v1.mark.reverse;
v3 := v1.reverse.mark.reverse;
```

A possible refinement is to first pre-process the right relation with an **aggregate** in order to obtain all unique elements. This reduces the size of the outer relation in the above nested-loop join. The normal **aggregate** algorithm would then be extended with code generation for an aggregation pivot. As a post-processing step, the join result would then be joined with the pivot in order to bring back the previously eliminated doubles.

Join Indices

One thing to check, however, before any join code is generated, is whether a *join index* exists for the join expression. This should be looked up by comparing the join expression to those in the data dictionary.[13]

In the simple case of a join between two base relations, the join-index BAT can be used right away as join result `BAT[oid,oid]`. In our right-deep trees, the left relation is always a base table, but the right table may be an inner node. If that is the case, one first needs to create a pivot to the base table that forms the other end of the join index. This base table can be reached through some path in the inner node (this can be done as described in the basic projection algorithm). The join result in those cases is then formed by joining the pivot with the join index.

Notice that depending on how the join index is defined in the data dictionary, one needs to join against the C-BAT or its reverse. In the latter case, the join may be more efficient if one uses the inverted list S-BAT instead, which has an ordered tail column.

Figure 7.8 shows that almost all joins in TPC-H Query 9 are accelerated by join indices. Those join nodes have a gray border, and their expressions are depcted in gray as to indicate that these expressions and joins need not be executed.

Exactly-One Joins

In the case where we join a right relation with a foreign key to a left relation with an unique key on the equality of those keys, we can already deduce from the information in the data dictionary that for each right tuple, we will find exactly one left tuple.

Such joins can in the Monet context be seen as simple projections, as they just serve to add columns to the right relation (they do not remove or introduce new tuples). This in turn means that renumbering and pivot joining is not necessary; therefore `join1` nodes are treated differently than normal `join` nodes.

After computing the join result `BAT[oid,oid]` with the normal methods, it is sorted with `reverse.order.reverse` to make sure the head `oid`-s appear in densely ascending order.[14] This is just done to enhance performance of later processing of this BAT. This join result is not marked, but can be used as a pivot for projecting in columns from the left table using the basic projection algorithm.

[13] Some normalization of join expressions is required to recognize all matching join index expressions easily.

[14] We know that the `oid` sequence must be densely ascending when sorted, because all `oid`-s of the right relation form a `void` column, and we perform joins that hit exactly once only, hence we end up with the same `oid` collection.

Outer Joins

Any **join** or **join1** node can be marked as an "outer-join" in a certain direction, meaning that if a tuple from the source side does not match any from the destination, one extra result is emitted, that has all `nil` values in its projections columns from the destination.

In the case of equi-join, outer-join is supported directly by using the MIL `outerjoin` instead of `join`. Recall that for those `[x,y]` BUNs in its left operand where there are no matching `[y,z]` BUNs in its right operand, the `outerjoin` differs from the `join` in that it emits a `[x,nil]` result BUN. In other words, an "outer join pivot"[15] may contain `nil`-s. Due to the semantics of MIL join (`nil`-s never match), this in turn means that the basic projection algorithm, which should also use `outerjoin` when traversing outer join pivots will generate `nil` values in projection columns for the tuples that did not match.

Join indices are just as applicable for outer joins (as a join index is only a mechanism to re-use join results).

The generic join code for arbitrary (i.e., non-equi) join conditions can support outer-join with just a small addition after the `selbat` computation:

```
if (selbat.count = 0) selbat := bat(oid,oid).insert(o,nil);
```

Finally, in the case of multi-column equi-join with a non-perfect hash function, one needs to modify the "false-hits" filter expression:

```
and(=(LEFT.KEY1,RIGHT.KEY1), =(LEFT.KEY2,RIGHT.KEY2))
```

$\Rightarrow$

```
or(isnil(LEFT.NUM), and(=(LEFT.KEY1,RIGHT.KEY1), =(LEFT.KEY2,RIGHT.KEY2)))
```

As the `NUM` field is computed with `xor()`-s of `hash()` which never yields a `nil`, we know that any `nil` values that do occur are introduced by outer-join projections, and thus should *not* be filtered out (hence the added `or()` condition).

7.3.5 Sort

A **sort** node only makes sense as root of the algebra tree, because it just changes the output order when visualizing the query result. It appears in a relational algebra tree when a SQL query has a `ORDER BY` clause, but also when it has a `GROUP BY` clause but defines any aggregate expressions. In those cases, any additional `ORDER BY` columns that are not in the `GROUP BY`, are appended to the column list of the **sort** node.

In order to sort a relation on one column `BAT[void,any]`, a tail-sorted version of this is produced by `order`. The resulting `BAT[oid,any]` with `oid`-s appearing out of order in the head we call the *global ordering BAT*. This global ordering BAT is `mark.reverse`d to produce the desired pivot `BAT[void,oid]` which relates tuples in the new order to tuples in the old order.

The case of multi-column orderings is an extension of the computation of single-column orderings. After creating the single-column global ordering BAT on the first `ORDER BY` column with the unary `order`, the remaining `ORDER BY` columns are processed iteratively with binary `order` operations, to refine this global ordering. with the previous global ordering BAT as first and the `ORDER BY` column-BAT as second parameter.

[15] Any path pivot that spans at least one outer join is an outer join pivot.

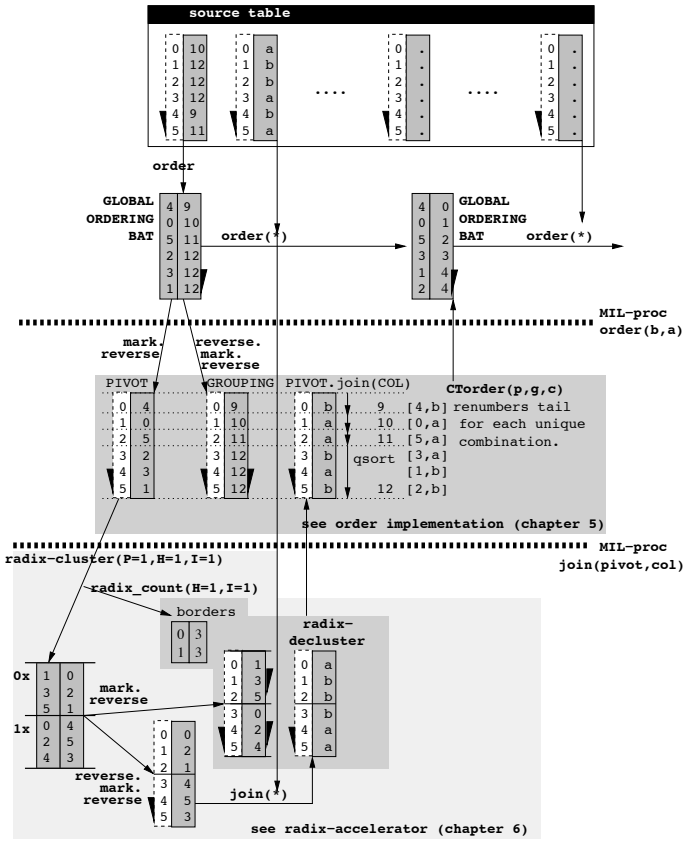

Figure 7.9: Cache-Conscious Multi-Column Sort

As described in Section 5.3.5, the binary `order` is in fact just a MIL-procedure, that marks the global ordering BAT on both sides, creating `PIVOT` and `GROUPING` BATs, that have a new `void` head and contain respectively the tuples `oid` s and the order values in their tail. The additional `ORDER BY` column is then provided with these new dense head `oid`-s, by joining it with the pivot. As such, the `CTorder` can walk sequentially through all its three BAT parameters. Per existing group, it Quick-Sorts the `oid`-s and new `ORDER BY` column values in order to emit the refined global ordering BAT.

These details of the tertiary `CTorder` (see Section 5.3.5) are depicted in Figure 7.9, also including the cluster-decluster implementation of the join between `PIVOT` and the `ORDER BY` column (used automatically thanks to the *radix-accelerator*, see Section 6.5.2). This implementation of multi-column `ORDER BY` thus is decomposed in basic MIL operators that all have cache-friendly access pattern: the possibly costly reordering of each `ORDER BY` is eased by cluster-decluster, whereas the refinement sorting uses the cache-friendly Quick-Sort on sub-chunks of the global ordering BAT (which are much smaller than the total relation and often directly fit the memory cache).

One potential problem in MIL is that most of its operators are set- or bag-defined and thus guarantee nothing about ordering. This is also the case with the multi-join map, of which we use `[printf]` to print result tables. In principle, presenting results

in the correct order is considered a front-end issue. Therefore, for queries with `ORDER BY` we add an additional first index column to print by `[printf]`, which contains the row number. The row number is a unique integer with minimal value 0 – for the tuple that has to come first – and maximal value $N-1$ – for the tuple that has to come last – where N is the number of tuples in the result). In practice, re-sorting the printed result is not necessary, as the MIL implementation of the multi-join map where all BAT-parameters have a `void` column, will output tuples in the order of the densely ascending `oid`-s.

7.3.6 Aggregate

Relational **aggregate** nodes are introduced whenever the SQL query contains a `GROUP BY` clause with aggregate functions in the project list, or if the `SELECT DISTINCT` keyword is used. In the latter case, all SQL projection columns appear both as `GROUP BY` columns and in the projections (in that case there are no "real" aggregates - the algorithm is the same, though).

Aggregates are computed by first creating a so-called grouping-`BAT[void,oid]` where the tail values divide all `oid`-s into a number of disjunct groups. With that grouping-BAT, the MIL pump construct `{f}(EXTENT, GROUPING, COLUMN)` can compute aggregate expressions. `COLUMN` is a column-`BAT[void,any]` holding a column or expression on the child relation, and `EXTENT` is a `BAT[void,any::1]` with the new table `oid`-s in the head and in the tail the unique collection of group values (from the tail of `GROUPING BAT[void,oid]`).

We mentioned that an **aggregate** algebra node cannot be passed by projections to fetch column values deeper in the tree (therefore, it is the only algebra node that does not need to construct a pivot). All available columns are the `scope` expressions which must have an aggregate functor in their root. The only non-aggregate expressions allowed in this **scope** clause are the `GROUP BY` expressions themselves (the first parameter of the **aggregate** node).

The set of `GROUP BY` expressions is available initially as column `BAT[void,any]`s which are produced by the basic projection algorithm. There are two algorithms to compute the grouping-BAT from them. The first algorithm uses a successive execution of the MIL `group` operators on the `GROUP BY` column BATs; first the unary `group` on the first BAT, then binary `group`-s with the result of the previous `group` as first parameter and the next column BAT as second parameter. The final result of this process is the desired `BAT[void,oid]`. The second algorithm uses the code generation for **sort** algebra nodes to compute a *global ordering* `BAT[oid,oid]` from the `GROUP BY` expressions as if they were `ORDER BY` expression. Using `mark`-s, on both sides the earlier described `PIVOT` and `GROUPING` are computed. The latter `BAT[void,any]` is used as grouping-BAT when aggregates are computed with the MIL pump (the second parameter). The column-BATs passed as third parameter to the pump are formed by joining the `PIVOT` with the projected column-BATs. In either case, we often compute a "map-BAT" with `{min}(EXTENT, GROUPING, GROUPING.mirror)` that can be used for joining against the column-BATs holding the `GROUP BY` expressions. This map-`BAT[void,oid]` is necessary whenever the `GROUP BY` columns are part of the projection list (which is typically the case).

The first grouping algorithm uses the MIL `group` operator which is implemented internally using simple and fast hash-grouping. The second algorithm uses the `order`

operator, which is based on repeated chunk-wise Quick-Sort. As a general rule, the first algorithm will be much faster as long as the number of groups is limited (i.e. as long as the hash table fits the memory cache). The second algorithm has a memory access pattern that is cache-friendly, even with a huge amount of groups[16] and therefore outperforms the first in such cases.

A typical case where many different groups are expected is when the **aggregate** node was generated for a `SELECT DISTINCT` SQL query. In other cases, statistics kept in the data dictionary, such as histograms of attribute distributions could be used to estimate the number of different groups for a `GROUP BY` expression in order to choose between the two pivot generation algorithms. Dynamic query optimization is an attractive alternative, though, as it comes as no additional cost in MIL and does not depend on modeling or estimation, but on the actual value distributions. Here, one would pass an extra integer parameter to the MIL `group`-s, that holds maximum memory consumption for the hash-table used while grouping. A good value for this would be like half the size of the most costly cache, as determined by the Calibrator (see Sections 6.3.3). If this version of the `group` encounters too many different groups, it just quits and returns `bat(nil)`[17]. So, after the `group`-s, the generated code would have the form:

```
grouping := group(col1, CACHE_SIZE/2).access(BAT_WRITE);
grouping := :group=(grouping, col2, CACHE_SIZE/2);
..
grouping := :group=(grouping, colN, CACHE_SIZE/2);

if (isnil(grouping)) {
..code generated by sort-based aggregate pivoting ..
} else {
..rest of code generated by group-based aggregate pivoting ..
}
.. processing continues with the projections of the aggregation node ..
```

Here we use the `:f=(x,..)` MIL assignment notation, which is a shorthand for `x := f(x,..)`, because there is a specific implementation of `:group=(b)` that places its results directly in the input-BAT `b`, hence optimizes (cache) memory usage.

Note that *if* the (**sort**-based) pivot generation is used *and* the **relation** node above the **aggregate** node is a **sort** *and* its `ORDER BY` columns are a subset of the `GROUP BY` columns, then the **aggregate** result is already properly sorted, so that no code needs to be generated anymore for the subsequent **sort**.

7.3.7 Generating MIL For TPC-H Query 9

We now do a right-depth-first traversal of the query tree in Figures 7.8 and 7.7, showing which code is generated. In the rightmost operator of the algebra tree there is a simple select that is executed with a multi-join map `[like]`. Thus, for relation `R6`, we compute as pivot `v02`.

```
v00 := [like](C_PART_COLOR, "%green%");
v01 := v00.select(true);
v02 := v01.mark.reverse;
```

[16] This cache-friendliness includes the use of the radix-cluster/decluster join strategy for joining column BATs with the partial ordering, as discussed at the end of the last Subsection.

[17] The binary `group(b,v)` also returns `bat(nil)` if `isnil(b)`

The join between R1=LINEITEM and R0=PART on their PARTKEY columns is accelerated by a join index (simply denoted "R1-R0" here). As the join in R6 is not a base table join (the PART appears below a select), we must join the pivot of relation between relations R7-R0 (v02) with the join index R0-R1. As we come from the side of PART and the join index is stored in LINEITEM, we use the inverted list version of the join index for faster access. Therefore, code for taking into account updates in D- and I-BATs is included.

Given the fact that the rest of the query only projects columns from the LINEITEM side, we can follow a **(c,u)** projection strategy (as mentioned in Section 6.5.3) thus clustering the join result on some H most significant bits.

The join result R6-R1 stored in v03 represents the relation R7. Using mark-s on v03 we obtain the pivot between relations R7-R6 in v04 and the pivot between relations R7-R1 in v05.

```
v03 := v02.join(S_LINEITEM_=[PARTKEY$PARTKEY_PART].reverse).access(BAT_WRITE);
       v02.insert(v02.join(I_LINEITEM_=[PARTKEY$PARTKEY_PART].reverse));
       v02.delete(v02.join(D_LINEITEM_=[PARTKEY$PARTKEY_PART].reverse));

v03 := v03.sql_joincluster(v02.count, 0, 0,
                           S_LINEITEM_=[PARTKEY$PARTKEY_PART].count, 2, 4);

v04 := v03.reverse.mark.reverse;
v05 := v03.mark.reverse;
```

The sql_joincluster in this case partially clusters the join result on LINEITEM, as there are only projections from that side. Notice that R7 in fact references three columns from R1=LINEITEM (to be exact T1.SUPPKEY, T1.PARTKEY, and T1.ORDERKEY), but these expressions do not count as they appear in join-expressions that are handled with join indices. However, each use of a join index counts as one oid column projection, hence we have two column projections of C-BATs with byte-width 4.

What follows now are a four joins with ORDER, PARTSUPP, SUPPLIER, and NATION of which all but the latter can all be handled with join indices. As these are all exactly-one joins, everything happens in the context of column projections inside relation R7.

Column projections are performed by lazy evaluation, which starts when a column is referenced in an expression. This first happens when evaluating the **scope** list of the result of the ORDER join, where the year is computed from the R2.ORDERDATE. Thus, we take the pivot R7-R1 and join it with the join-index between R1=LINEITEM and R2=ORDER. The order mentioned in the **join1** code generation algorithm is free here, as we join into a BAT[void,oid] join-index (it is a C-BAT), hence the result is another BAT[void,oid] (as mentioned before, a void column in the left operand is propagated by the Positional-Join). The basic projection algorithm then continues to produce a column with date tails in v07, and the basic expression algorithm computes a column-BAT[void,int] with extracted years.

```
v06 := v05.join(C_LINEITEM_=[ORDERKEY$ORDER_ORDERKEY]).reverse.order.reverse;
v07 := v06.join(C_ORDER_ORDERDATE);
v08 := [year](v07);
```

We then proceed to execute the three remaining joins of relation R7 in order of appearance. The joins with R3=PARTSUPP and R4=SUPPLIER are accelerated by join indices LINEITEM-PARTSUPP (''R1-R3'') and PARTSUPP-SUPPLIER ("R3-R4"). In order to project the NATIONKEY column from R4 a pivot R7-R4 is constructed by taking R7-t1 and joining through the join indices R1-R3 and R3-R4 in v10. The join with NATION is executed without join index in v12. As this also is an exactly-once join, re-pivoting is not necessary, and all this still happens in the context of projecting columns for relation R7.

```
v09 := v05.join(C_LINEITEM_and[=[PARTKEY$PARTSUPP_PARTKEY],
                               =[SUPPKEY$PARTSUPP_SUPPKEY]]).reverse.order.reverse;
v10 := v09.join(C_PARTSUPP=[SUPPKEY$SUPPLIER_SUPPKEY]).reverse.order.reverse;
v11 := v10.join(C_SUPPLIER_NATIONKEY);
v12 := v11.join(C_NATION_NATIONKEY.reverse);
v13 := v12.join(C_NATION_NAME);
```

What follows are projections for expressions from the aggregate expressions of relation `R8`. Here we use the multi-join map-overwrite `[:f=]` whenever a map result has the same signature as its first parameter and this first parameter is not used further on. The advantage of this overwrite is that less memory is consumed and hot cache lines are better re-used. The disadvantages is that the generated code gets bit cluttered with `access` calls in order to make the BATs updatable.

```
v14 := v05.join(C_LINEITEM_QUANTITY);
v15 := [dbl](v14);                          # enumeration view
v16 := v10.join(C_PARTSUPP_SUPPPLYCOST).access(BAT_WRITE);
v17 := [:*=](v16,15);
v18 := v05.join(C_LINEITEM_DISCOUNT);
v19 := [-](dbl(1), v18);                    # enumeration view
v20 := v05.join(C_LINEITEM_EXTENDEDPRICE).access(BAT_WRITE);
v21 := [:*=](v20,v19);
v22 := [:-=](v21,v17).access(BAT_READ);
```

As we used *enumeration-types* in some C-BATs storing the `LINEITEM` table, the projection joins return a `BAT[void,enum1[dbl]]` in `v14` and `v18`. The single-BAT multi-join maps in `v15` and `v19` then fall into the *enumeration-view* implementation, which is virtually free, because it only processes the enumeration mapping-BAT (see Section 5.3.2).

The evaluation of `R8` continues with generating the "`GROUPING`"-BAT in `v24` with two successive `group`-s. The unique group `oid`-s are collected in `v25`, and `mark`-ed to form the extent of the aggregated table. In `v26`, a map-BAT is constructed that contains for each `oid` of the aggregated table in the head one "example" `oid` of a group member from the original table in the tail. This map-BAT is used to project back the `GROUP BY` columns (`year` and `name`) in `v28` and `v27`. This amount aggregate is finally computed with a MIL pump in `v29`.

```
v23 := group(v13).access(BAT_WRITE);
v24 := :group=(v23, v08);
v25 := v24.reverse.mirror.unique.mark.reverse;
v26 := {min}(v25,v24,v24.mirror);
v27 := v26.join(v13);
v28 := v26.join(v08);
v29 := {sum}(v25,v24,v22);
```

The amount of tuples in `R8` is small, hence the execution cost of the rest of the MIL code is insignificant. What we have here is a two-column `ORDER BY`. Notice that the binary `order` in `v31` is a MIL procedure that uses the tertiary `order` implementation that includes the custer-decluster strategy, as depicted in Figure 7.9. The result is the final global ordering BAT, which is `mark`-ed to create the pivot in `v32`.

The statements `v33-v35` are Positional-Joins that project the final three columns, which are subsequently printed using a multijoin map.

```
v30 := order(v27);
v31 := order(v30, v28);
v32 := v31.mark.reverse;
v33 := v32.join(v26);
v34 := v32.join(v27);
v35 := v32.join(v28);
[printf]("% 9d % 30s % 12d % 5.2f\n", v32.mirror, v33, v34, v35);
```

We now also show the code generated for the alternative **aggregate** pivot creation algorithm, which is sort-based. In the case of the TPC-H data distribution, this algorithm will be more costly than the one showed before, as the number of groups in for a `GROUP BY NATION,COLUMN` is limited (tens of tuples).

Statements `v23-v26` are mostly identical to `v30-v33` from the other query plan. In `v27` the "`GROUPING`"-BAT is computed, in `v28` the extent and in `v29` the map-BAT.

```
v23 := order(v13);
v24 := order(v23, v08);
v25 := v24.mark.reverse;
v26 := v25.join(v22);
v27 := v24.reverse.mark.reverse;
v28 := v24.reverse.mirror.unique.mark.reverse;
v29 := {min}(v28,v27,v25);
v30 := v29.join(v13);
v31 := v29.join(v08);
v32 := {sum}(v31, v27, v26);
[printf]("% 9d % 30s % 12d % 5.2f\n", v30.mirror, v30, v31, v32);
```

As the `ORDER BY` columns of the subsequent **order** node are equal to the `GROUP BY` columns of the **aggregate** node, the result of relation `R8` in BATs `v31-33` is already properly ordered such that no specific **order** code needs to be emitted for `R9`.

7.4 Transaction Management

The simplest transaction system based on MIL would use a global read-counter protected by a short-term lock, and a write-lock. The read-counter serves to count the number of executing read-transactions, where the first concurrent reader acquires the write-lock, and the last to finish releases it. Write-transactions acquire this write-lock, modify the C-, I- and D-BATs according to the scheme in Figure 7.5 and use the global `commit` to atomically commit all changed BATs before releasing the write-lock again.

This simple scheme, however, is inefficient because:

- write-transactions are costly, as saving each modified BAT at least costs two I/Os (one BAT descriptor, one heap).
- there is no concurrency at all between write-transactions, even if they obviously read and modify independent data.
- write-transactions seriously impair the performance of read-queries due to the coarse locking protocol.

In this Section, we outline a more efficient transaction system with ACID properties [GR91] implemented purely in MIL, that uses a two-level locking protocol, exploits virtual memory techniques for Isolation at low resource cost, and achieves Atomicity and Durability with *write ahead logging* [Dav73] and uses BATs to implement *differential files* [SL76]. Its aim is to provide highly concurrent write-transactions in combination with high performance read-only query execution on a nearly up-to-date database version. Figure 7.10 shows the extensions to the data dictionary from Figure 7.3 that are made in order to facilitate this transaction system. In the following, we describe how it provides ACID properties. First, we cover the features needed for consistency and isolation, then those needed for atomicity and durability.

```
class DatabaseVersion {
        relation Set<ReadTable> readTables   inverse ReadTable::version;
        relation Set<WriteTable> writeTables inverse WriteTable::version;
        lock commitLock;
        integer transId;
}

class ReadTable:Table {
        relation DatabaseVersion version     inverse DatabaseVersion::readTables;
}

class WriteTable:Table {
        relation DatabaseVersion version     inverse DatabaseVersion::writeTables;
        lock readLock, writeLock;
        integer nReaders;
        BAT r_bat; // [oid,trans] combinations for read-locked tuples
        BAT w_bat; // [oid,trans] combinations for write-locked tuples
}

class ReadColumn:Column {
        lock refCntLock;
        integer refCnt;
}

class WriteColumn:Column {
        lock columnLock;
        BAT i_bat; // deltas since last sync (inserts)
        BAT d_bat; // deltas since last sync (deletes)
}

class Transaction {
        relation Set<Transaction> waitsFor   inverse Transaction::waitsOn;
        relation Set<Transaction> waitsOn    inverse Transaction::waitsFor;
        lock barrier;
        Set<WriteColumn> modifications; // private not yet committed column deltas
        BAT h_bat; // [table,oid] deleted tuples in this transaction.
        BAT n_bat; // [table,oid] inserted tuples in this transaction
}
```

Figure 7.10: Data Dictionary Extensions for Transaction Management

7.4.1 Consistency and Isolation

Consistency is achieved by a simple two-level locking protocol. Isolation is provided by making sure that a transaction does not make changes to those parts of BATs that may be accessed by other concurrent transactions.

Supporting Multiple Database Versions

Instead of the database being the collection of all tables, another level of abstraction is added by the "database version". The database hence stores a collection of all database versions. Such a database version in principle contains a copy of all tables, columns, keys, indices, etc. The principle motivation behind this is that there is one database version – the one with the highest transaction ID – that is the most fresh "write-version" of the database. All modification queries must run against this database version and follow the transaction mechanism in order to acquire ACID properties. Then, there is one "read-version" that is kept up-to-date with the write-version at regular (small) time intervals. Read-only queries thus have the possibility to bypass all transaction overhead and run with full concurrency (i.e. without any locking) against this slightly older version of the database.

The fact that multiple versions of the same database exist, means that each database

version references its own C-, S-, I-, D-, M- and H-BATs in the Monet back-end. The inverted list structures can in fact be shared between read- and write-version, so only the H- and C-BATs are duplicated between the two database versions. To distinguish between those, we write H_r-BAT and C_r-BAT for those referenced by the read-version and H_w-BAT and C_w-BAT for those referenced by the write-version. Most of the duplication is between the C_r- and C_w-BATs as these are voluminous, but by exploiting OS virtual memory primitives (`mmap` with the `MAP_PRIVATE` flag) the memory and disk consumption of this duplication is reduced to only those virtual memory pages where the C_w-BATs actually differ from their C_r-BAT counterparts.

Once in a while, a new read-version is created by propagating the current write-version to it. This happens in the background on a private copy, so read-queries need never be interrupted. When the old read-version is replaced by the new read-version, the old read-version is scheduled for deletion by setting its transaction ID to `nil`. Its BAT-resources are deleted as soon as the last still running read-queries that use them have finished.

In the data dictionary, the write-version of the database only contains references to "write-tables", whereas the database read-version(s) only refer to "read-tables". A write-table is different from a read-table in that it contains structures for the locking protocol. Similarly, "read-columns" are only referred to by read-tables and "write-columns" are only referred to by write-tables. Write-columns reference an I_w- and D_w-BAT that holds updates to the column that have happened in the write database version since the read-version. In the case of columns with an inverted list, the I_w- and D_w-BATs are supersets of the I- and D-BATs of the inverted list (we use the same virtual memory page sharing technique as in the C_w- and C_r-BATs to minimize resource consumption).

The full BAT data structures of the transaction system are shown in Figure 7.11.

A Simple Locking Protocol

Read-columns add a reference-count and a lock to protect it, to support the garbage collection mechanism for old read-versions. The idea here is that each read-only query increases the reference count of the database colums it is going to use, before starting query execution. When a read-only query finishes using a table column for the last time, it decreases the reference count of its column locks. When a column reference count reaches zero and the database version is to be deleted (it has a `nil` transaction ID), the column is deleted. If it was the last column, the old database version itself is deleted.

Explicit locks on the MIL level are introduced by a simple extension module:

```
MODULE lock;

ATOM lock;

OPERATOR create_lock() : lock;
OPERATOR destroy_lock(lock)
OPERATOR set_lock(lock)
OPERATOR unset_lock(lock)
OPERATOR try_lock(lock) : bit

  ..semaphores etc..

END lock;
```

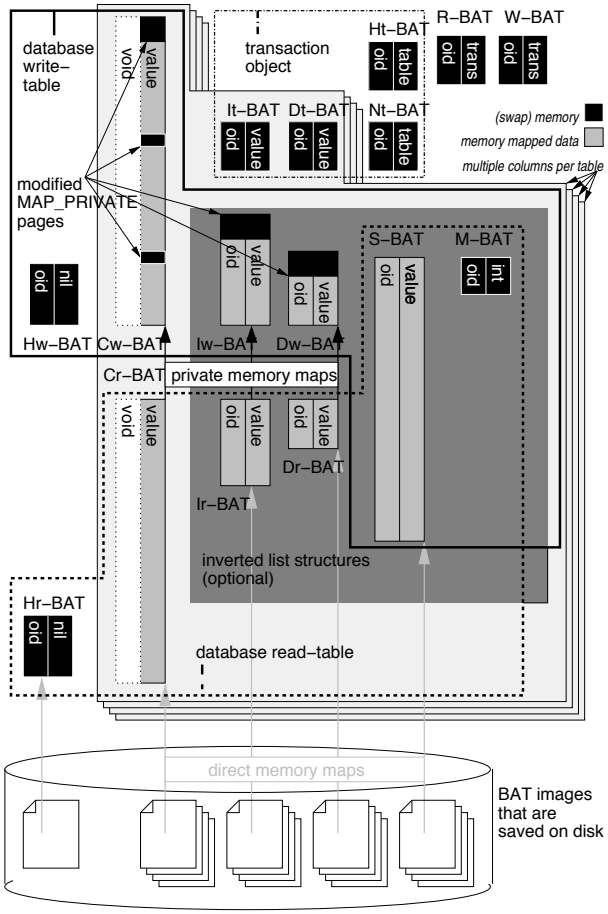

Figure 7.11: BAT Data Structures Of the Transaction System

The idea here is that the data dictionary and other data structures are stored in BATs and MIL variables in the running Monet server, the entire transaction system is programmed out in MIL. We describe here a simple two-level locking protocol: a table-level read-write lock (where multiple readers or a single writer are allowed). For a certain class of queries, i.e., those that have (at least) an equi-select condition on one table key, or (at least) a foreign-key equi-join condition on *each* base relation[18], we lock on the level of tuples (allowing multiple tuple readers or a single writer). Multiple such tuple-level locked transactions can run at the same time. A transaction can run as long as its locks do not conflict with those of already running transactions. Tuple-level read-locks conflict with the table-level write-lock, and tuple-level write-locks conflict with both read- and write-lock on the table level.

To implement this scheme, write-tables have a read/write lock (implemented in Figure 7.10 as a `nReaders` counter protected by a read-lock, and an exclusive write-lock that is set by update transactions or by the first reader (and released by the last reader). For

[18]In such queries, it is easy to establish which tuples participate, hence classical problems like "ghost records" and complex solutions like "predicate-locks" do not play a role.

tuple-level locking, two BATs called the R-BAT and W-BAT store `oid`-s of tuples that are read-locked and write-locked, respectively. Concurrent updates to these BATs are protected by the (short-term) read-lock. Also, there is a `waitsFor/waitsOn` dependency graph between transactions for implementing locking and deadlock detection.

Notice that a transaction consists of one or more queries. Each query can be either a read-only (`SELECT..FROM..WHERE`) or a modification query (i.e. `DELETE`, `UPDATE`, or `INSERT`). The level of transaction locking (tuple or table) is re-evaluated for each newly arriving query in the transaction, but can only *escalate* from tuple- into table-locking (not the other way around). Locks acquired by a query in a transaction are only released after the transaction has finished (i.e. committed or aborted).

A running transaction directly updates the H-, C-, I- and D-BATs as depicted in Figure 7.5. To be precise, it updates the H_t-, C_w-, I_t- and D_t-BATs. The latter two are *private* delta BATs found in a write-column object. Such objects are created on-the-fly for a transaction as it modifies a column. Consequently, when transaction queries use an inverted list and it has modified that column already, it must use the `union(It_TAB_COL, Iw_TAB_COL)` and `union(Dt_TAB_COL, Dw_TAB_COL)` where otherwise the I- and D-BAT had been used. The H_t-BAT is a private version of the H-BAT that holds the `oid`-s of those tuples that have been deleted by the transaction. In fact, all tuple deletes from all tables are recorded in the same H_t-BAT, the tail holding the table name. Similarly, each transaction maintains a N_t-BAT that holds all `oid`-s of new tuples (the tuples the transaction intends to insert).

For handling a query, that is part of some transaction protected with tuple-level locking, we run the query first in "optimistic" mode *without* carrying out any updates, just collecting from it the "read-set" and "write-set" of tuples (i.e. those tuples from base tables that have been read, and those that are about to be deleted or modified). This is called the "read-part" of the query. The BATs holding the resulting read- and write-sets are `intersect`-ed with the R- and W-BATs of their respective write-tables (thus MIL also proves itself useful for implementing the locking protocol). If there are transactions (the "dependents") that already hold conflicting locks, a vertex from the executing transaction to each of those other transactions is created. After creating the vertices, a cycle-detection on the transaction graph is run to detect deadlocks, in which case the new transaction is aborted. A transaction with dependents blocks on its lock, only to be unlocked by the last dependent that finishes executing and removes its locks from the R- and W-BATs and removes its incoming dependency vertices. Once unlocked, the blocked transaction tries again to run the query in optimistic mode, which repeats until this yields no conflicts. If no locking-conflicts arise, the read-set and write-set of the transaction are added to the R- and W-BATs and the second "write-part" of the query is executed which uses MIL update statements to make modifications in the write-version of the database.

In the following, we describe in more detail how this works for the various kinds of modification queries.

DELETE

Starts with a relational algebra tree that represents a query with a result that exactly forms a unique key on the table where deletes must be done. The semantics of such a delete query is that the tuples identified by the retrieved key values are deleted. We add a join to the base table between the base table and this query result, that retrieves

the values of all columns of the tuples that are to be deleted:

```
relation(DELETE, scope([X.KEY], ..XYZ..)

  ⇒

relation(DELETE,
         scope([base_table.COL1, .., base_table.COLn],
               join1(=(base_table.KEY, X.KEY),
                     relation(base_table, scope([], table(TABLE))),
                     scope([X.KEY]), ..XYZ..))))
```

After executing this query in MIL, there will be a `PIVOT` between `DELETE` and `base_table`. Also, for all columns in the **scope** list, there will be a column-`BAT[void,any]` holding all column values that are about to be deleted. These query results are used as follows for performing the deletes:

```
deletes := PIVOT.reverse;
Hw_TABLE.insert(deletes.mark(nil));

# for each column-BAT[void,any] COLi (1 <= i <= n) in DELETE
Dt_TABLE_COLi.insert(deletes.join(C_DELETE_COLi)
Cw_TABLE_COLi.replace(deletes.project(nil)); # is omitted in leave as-is-policy
```

This adds the `oid`-s of all deleted tuples to the H_w-BAT, and for each column inserts all deleted column values to the D_t-BATs. For columns with nil-replacement policy, all tail values of deleted tuples are replaced by `nil` in bulk.

INSERT

We assume an arbitrarily complex relational algebra tree that defines a result table. Its tuples need to be inserted in a base table, where columns are simply matched on name (omitting columns to be filled with NULL values).

```
relation(INSERT, scope([Y.COL1,...,Z.COLn)], ..YZ..)
```

This yields column-`BAT[void,any]`s for each column in the table (for columns omitted in the **scope** of the `INSERT` we use `COL1.project(nil)`).

```
set_lock(LOCK_TABLE);
  inserts := C_INSERT_COL1.mark;
    fills := inserts.join(Hw_TABLE_COLUMN.mark.reverse);
  appends := inserts.mirror.diff(fills.mirror).mark(Cw_TABLE_COL1.count);

  # remove holes that are filled up now
  Hw_TABLE_COLUMN.delete(fills.reverse.mark(nil));
unset_lock(LOCK_TABLE);

# add newly inserted tuples to the Nt-BAT
Nt_TABLE_COLi.insert(col_fills.reverse.project("TABLE"));
Nt_TABLE_COLi.insert(col_appends.reverse.project("TABLE"));

# for each column-BAT[void,any] COLi (1 <= i <= n) in INSERT
col_fills := fills.reverse.join(C_INSERT_COLi);
col_appends := appends.reverse.join(C_INSERT_COLi);

set_lock(LOCK_TABLE_COL);
  Cw_TABLE_COLi.replace(col_fills);
  Cw_TABLE_COLi.insert(col_appends);
unset_lock(LOCK_TABLE_COL);

It_TABLE_COLi.insert(col_fills);
It_TABLE_COLi.insert(col_appends);
```

In the above code, we first couple `oid`-s of the `ins_query` result table with new `oids` in the target base table. Two `BAT[oid,oid]`-s are constructed, one called `fills`, filling in holes in the base table, and the result of the inserts ends up in `appends`, appended at the end of the C-BATs of the base table. The holes are then deleted from the H_w-BAT of the base table, and for each table column the inserts that go into holes are processed with bulk-`replace`-s in C-BATs, and the appends with bulk-`insert`-s. Both changes are also appended to the I_t-BATs.

Notice that inserts use the MIL `set_lock()` and `unset_lock()` primitives to manipulate the locks referred to by the data dictionary: the short-term table lock is required when determining the `oid`-s for the new tuples. For each write-column, the column-lock must be held for serializing the column-BAT inserts (we assume that each transaction inserts into the table columns in the same order).

UPDATE

Like in the case of delete, we assume an arbitrarily complex relational algebra tree that defines a result table that forms a unique key of the base table, but also contains the new values of all columns that are updated. This relational algebra query is rewritten into a join back to the base table.

```
relation(UPDATE, scope([X.KEY, Y.EXPR1, .., Z.EXPRn], ..XYZ..)

  ⇒

relation(UPDATE,
         scope([base_table.COL1, .. base_table.COLn, X.EXPR1, .., Y.EXPRn
               join1(=(base_table.KEY, X.KEY),
                     relation(base_table, scope([], table(TABLE))),
                     scope([X.KEY, Y.EXPR1, .., Z.EXPRn]), ..XYZ..))))
```

This creates a `PIVOT` to the base table, as well as column-BATs that hold for all columns the old values as well as the new ones. This translates in the following MIL:

```
# for each column-BAT[void,any] COLi (1 <= i <= n) in UPDATE
Dt_TABLE_COLi.insert(PIVOT.reverse.join(C_UPPDATE_COLi));
newvals := PIVOT.reverse.join(C_UPDATE_EXPRi);
It_TABLE_COLi.insert(newvals);
Cw_TABLE_COLi.replace(newvals)
```

For all modification queries, the read-set of a modification is formed by taking the tail column of all pivots to the base tables. In addition to that, for all columns that form a foreign key and that are used in the query, those `oid`-s reachable by joining the pivot with the foreign-key column are part of the read-set of the tables to which the foreign keys refer. The write-sets are the head columns of the MIL variables `deletes`, `fills` and `newvals` from the descriptions above.

7.4.2 Atomicity and Durability

Atomicity is achieved by a commit mechanism that either writes all changes made by a transaction to a persistent medium (in case of commit) or none (abort). Durability is provided by a crash-recovery mechanism that ensures that all changes, and only those changes, made by committed transactions are present when the database gets back on-line. A central component in achieving both is a *Write Ahead Logging* (WAL) mechanism implemented in MIL on top of the file-I/O extension module, of which we show the interface:

```
MODULE io;

ATOM fp;

OPERATOR fopen(str filename, str mode) : fp // open a file for read/write
OPERATOR fclose(fp)                         // close an open file
OPERATOR ftrunc(fp)                         // truncates an open file at the current position
OPERATOR ftell(fp) : int                    // returns the current file position
OPERATOR fseek(fp, int pos) : bit           // moves the current file position (may fail)
OPERATOR fflush(fp) : bit                   // flush writes; returns success status
OPERATOR fprintf(fp, str, ..any..) : bit

  ..other standard I/O commands ..

// MIL marshalling
OPERATOR getValue(fp) : any                     // read a MIL value from the stream
OPERATOR putValue(fp, any)                      // write a MIL value to the stream

END io;
```

The module provides basic file I/O with limited MIL value marshalling functionality, as the `getValue()` and `putValue()` manipulate token streams of MIL values. Such marshalling can be done by first outputting the MIL atom type as an 32-bits integer, the byte-length of the value as another 32-bits integer, followed by the bytes that make up the value. As such, the MIL-value can easily be reconstructed when reading bytes from the stream. Simple values as `int` can be marshalled in this way, but also `BAT[any,any]`-s.[19]

The simple WAL structure writes into one sequential file, that contains a marshalled sequence of values that adhere to the following grammar in the same extended BNF we used previously for defining our query algebra:

```
         WAL ::= { <TRANSACTION> }*
 TRANSACTION ::= H:BAT  N:BAT  { <TABLE_DELTA> }*  TRANSACTION_ID:int
 TABLE_DELTA ::= TABLE_NAME:str  { <COLUMN_DELTA> }*
COLUMN_DELTA ::= COLUMN_NAME:str I:BAT D:BAT
```

This states that the WAL is a sequence of transactions, that each consist of a H_t- and N_t-BAT, followed by a number of table-deltas and a transaction-ID. A table-delta consists of a table-name followed by a number of column-deltas. A column-delta in its turn consists of a column name and an I_t- and D_t-BAT. Transactions in the WAL have densely ascending transaction-IDs. The transaction-ID serves as a validator in case the database might crash while writing the WAL. The presence of a correct transaction-ID determines whether the last transaction in the WAL counts as committed or not.

Commit

When a transaction tries to commit, it must carry out the following steps:

1. grab the global database write-version lock.
2. write all its changes to the WAL. This determines whether the commit succeeds. A failed WAL-write must truncate the WAL to its original position before entering the abort sequence. Any of this or the following steps failing should stop the database and enter the crash recovery algorithm.

[19] Marshalling is a bit more complex as that the head and tail types must also be marshalled, and the binary image of the BATs consists of all used areas of all its heaps plus the attached column properties. Recursive BATs are not supported in this marshalling.

3. increment the transaction ID in the database write version.
4. release the global database write-version lock.
5. add all its column-deltas in the I_t- and D_t-BATs to the I_w- and D_w-BATs of the database write-version. These MIL-`insert`-s are done while holding the column-lock of the respective write-columns.
6. add all `oid`-s of tuples it deletes to the respective H_w-BATs. These MIL `Hw_TABLE.insert(Ht.select(tablename))` operations must be protected by holding the short-term table-lock).
7. grab again the global database write-version lock.
8. remove all incoming `waitsOn` vertices in the transaction dependency graph, and wake up any transactions for which that was the last dependency.
9. release the global database write-version lock.

Notice that the H_w-, C_w-, I_w- and D_w-BATs are inserted into by committing transactions, while other queries for transactions that are still ongoing read their stable elements without locking.[20]

Abort

In case of an abort, all modifications of the transaction are undone by:

```
# for each table with inserts or deletes
deletes := Nt.select("TABLE").mark;
inserts := Ht.select("TABLE").mark;

set_lock(LOCK_TABLE);
Hw_TABLE.insert(inserts);
Hw_TABLE.delete(deletes);
unset_lock(LOCK_TABLE);

# for each modified TAB.COL
set_lock(LOCK_TABLE_COL);
Cw_TAB_COL.replace(Dt_TAB_COL);
unset_lock(LOCK_TABLE_COL);
```

The abort then continues with step 7 as in the commit sequence.

Stabilizing a New Read-Version

When creating a new stable version, new BAT disk images for all modified columns need to be made. This is also the time when inverted list indices are reorganized once their I- and D-BATs get too big. Consequently, this process can take some time, so it

[20] In order to make this fully safe, we ensure that no BAT-reallocation in the H_w-, C_w-, I_w- and D_w- takes place by reserving space for a large number of tuples when they are created (which is supported efficiently on the OS level by anonymous virtual memory with reserved pages that are committed on demand), and by triggering any overflow that *does* occur by having all these BATs in `access(BAT_UPDATE)` mode, and testing the result status of all `insert`-s and `replace`-s. In these fail (due to reallocation needs), the transaction lock level is escalated to prevent reads concurrent with a reallocation, and the transaction commit is resumed at the `insert` or `delete` that failed. The MIL procedure that does this is omitted for brevity in the examples.

would be highly undesirable to take either the read- or the write-version off-line while this process lasts.

Taking the read-version off-line is prevented by creating full new disk images independently (in different disk files) from the current read-version. This is supported directly in Monet using its management of persistent BATs with global `commit`. Thanks to the read-column garbage collection mechanism, the switchover can be made instantaneously, without need to wait until all read-queries on the old read-version have finished.

As for the write-version, off-line time is minimized in the following scheme that consists of three phases:

initialization First, we grab the global write-version lock and a table-level write-lock on all modified tables. Then, we record all sizes of the I_w- and D_w-BATs and make a copy of the H_w-BATs into what we will call "H_n-BATs". We then read the current transaction-ID, and release the global lock and all table-level write locks again.

reorganization One by one, we process all modified columns. This starts by making a modifiable copy of the C_r-BAT with `copy(Cr_TAB_COL).access(BAT_WRITE)` into what we call a 'C_n-BAT". As the C_r-BAT is read-only and loaded virtual memory (i.e. `load(VM_NORMAL).access(BAT_READ)`), Monet will use *privately mapped* into virtual memory, meaning that the OS will never write modifications to the disk images, rather create private temporary page copies of the modified or appended pages in the swap file

We then apply all changes in the *transaction subranges* of the I_w- and D_w-BATs to this C_n-BAT by using `Iw_TAB_COL.slice(lo,hi)` and `Dw_TAB_COL.slice(lo,hi)`. The `hi` value is the size recorded in the initialization phase. The `lo` `count(I-BAT)` resp. `count(D-BAT)`for columns with inverted list (0 otherwise), so we get only those column-deltas that were added since the last read-version was stabilized. We then make the C_n-BAT persistent under a new name (by e.g. prefixing all BAT names with the current transaction-ID). If the column has an inverted list, we create an "I_n-BAT" and "D_n-BAT" as `slice(0,hi)` of the I_w- and D_w-BATs and make them persistent under a new name. However, if these I_w- and D_w-BATs are too large, a new S_n-BAT is recreated by sorting the C_n-BAT and making it persistent under a new name. In that case, empty I_n- and D_n-BATs are created. After processing all columns this way, and giving all new BATs mode `access(BAT_READ)`, a consistent new read-version has been stabilized, and it is mapped into virtual memory with `load(VM_NORMAL)`. Lastly, the transaction-ID of the new database version is set, and the transaction-ID of the old read-version is changed into `nil`. To make these changes and all new BAT-images permanent, the BATs holding the data-dictionary are committed using the global `commit`. From that moment on, new read-only queries use the new read-version, whereas the old read-version will be garbage collected as soon as the last read-query executed on it finishes.

synchronization We get back at the write-version and grabbing the table-level write-locks and the global lock again. Further updates are halted after acquiring these locks. In the meantime, possibly some transactions may already have committed since the initialization phase. Thus we change all I_w- and D_w-BATs

`X` into `X.slice(hi, X.count)`, where `hi` was the BAT-size recorded during the initialization phase. Also, all C_w-BATs are destroyed, and recreated from the new C_r-BATs with `copy(Cr_TAB_COL).access(BAT_WRITE)`, again using privately mapped virtual memory. These new C_w-BATs are then brought back up-to-date by reapplying the just modified I_w- and D_w-BATs to them. After processing all columns in this way, the global lock and all table-level locks are released.

Notice that modification queries are only halted during the short initialization and synchronization phases, while they can continue undisturbed during the lengthy reorganization phase.

Crash Recovery

When the database starts up, the database write-version can be reconstructed by starting out loading all H_w-, C_w-, I_w- and D_w-BATs from their H_r, C_r-, I_r- and D_r-BAT disk images with `copy(Cr_TAB_COL).access(BAT_WRITE)` to obtain independent privately mapped copies. For columns that do not have an inverted list, the I_w- and D_w-BATs are created empty.

Then, the WAL is scanned for the range of transactions with IDs larger than the transaction ID of the stable database version. For each such transaction, the table- and column-deltas are read, de-marshalling the H_t-, N_t-, I_t- and D_t-BATs from the bytes in the file, which are then used to re-do all modifications using the normal transaction commit protocol (except, of course, the writing to the WAL). After processing all committed transactions in this way, the write-version of the database is brought online again.

7.5 Conclusion

In this chapter, we have shown that it is possible to build a full-fledged RDBMS on top of Monet as a SQL-2 front-end that uses MIL for *all* its data management needs, including transaction processing. We have also outlined how the implementation in Monet achieves efficient performance for such a system.

Concerning data storage, we described how relational tables can be stored in BATs, in such a way that efficient read- and write-access is possible. In this discussion, we also included the use of two index structures: the inverted list and the join index. Here, we provided a small I/O cost model, showing that for all queries but those that return a handful of tuples (i.e. OLTP queries), the exponential advantage that I/O bandwidth gains over latency causes any non-clustered index structure to be useless regardless of its characteristics – the problem being the projection phase that follows the selection phase in query processing. In contrast, the vertical fragmentation employed in Monet attacks the real problem as it reduces projection costs, by allowing a query to only access those columns that are actually used and compressing these columns with enumeration types.

The practical case used as an example throughout this chapter is the TPC-H benchmark, which nicely fits the Monet focus on query-intensive areas. As for overall storage costs, the disk volume occupied by Monet including indices, tends to roughly equal the nominal database size, which is considerable lower (at least a factor 2-5) than any known RDBMS implementations of TPC-H.

As for query processing, we described *full* relational query translation into MIL, rather than just some canonical examples. This was formalized by defining standard but powerful relational query algebra trees, and providing algorithms for generating (efficient) MIL from them.

Finally, we described a transaction processing scheme built purely with MIL that provides master-slave copies of the data, where the master is the "up-to-date" database image, and the slave is a slightly old version that can be used for complex read-only queries. As Monet will never be an OLTP champion due to its vertical fragmentation, such a mixed usage scenario is relevant as it is conceivable that one uses Monet for strong OLAP performance, but also needs to accommodate updates. While all techniques employed (differential files, master-slave copies, multi-level locking with deadlock detection and write-ahead logging) have been known for decades, the novelty of this scheme lies in the Monet implementation that provides this functionality by relying on shared virtual memory rather than a buffer manager, and cleanly separates transaction functionality from its query algebra, while in other DBMSs these are heavily intertwined. The advantage of the Monet approach is that its query algebra primitives can be optimized more efficiently and analyzed more cleanly (e.g. on their memory cache behavior). The drawback that this places some burden on the MIL user to invoke explicit locking primitives is insignificant as MIL is not intended as an end-user language, but is typically generated by front-end systems.

Chapter 8

Conclusion

8.1 Contributions

Let us recollect the main research questions defined for this thesis in Section 3.1:

- *how to get best performance out of modern CPU and memory hardware on query-intensive DBMS applications?*
- *how to support multiple (complex) data models?*
- *how to provide sufficient extensibility to new domains?*

The contributions of this thesis to answering these research questions are summarized below.

The MIL Language The MIL language and its *BAT algebra* in particular provide a clean and powerful query language that is extensible in all its dimensions. By focusing on column-at-a-time operators, rather than a higher level abstraction of e.g. tables or objects, the language is neutral to various data models. This thesis describes in detail how MIL can be used in a relational DBMS and also gives examples of use in object-oriented query processing. In various collaboration projects, MIL has successfully been used for query processing on networked (bayesian belief networks) [dVW98], topological (triangulated interconnection networks) [WvZF+98] and semi-structured (XML) [SKWW00] data models as well. Note that these data models store data not in blobs or even ADTs [RS97], but decompose their data in BATs such that e.g. neighbor predicates can be tested by MIL set-operators. Thus, there are indeed strong indications that MIL fulfills its goal of being the building block for database systems with widely different data models.

Moreover, the column-at-a-time nature of the BAT algebra yields operators with a low degree of freedom, making MIL versus relational algebra in query processing what RISC is to CISC in CPU technology. This "RISC approach" enables optimization techniques in the MIL implementation for squeezing best performance out of modern super-scalar CPUs, which was also one of the stated goals.

The Monet System The benchmarking of Monet on OO7 [BKK96], Sequoia [BQK96], TPC-D [BWK98] as well as the Drill Down Benchmark [BRK98] provides proof for the feasibility of efficient query processing with MIL.

The Monet system uses a coherent mix of design and implementation ideas (database architecture) that provides useful insight for future database kernel programmers. Techniques like implicit storage in `void` columns, view-implementations and type remappings were effective in eliminating the extra join-overhead that is encountered when relational or object-oriented applications are fully decomposed into the binary table model. Main-memory optimization methods like code expansions were employed throughout the system to make its code as efficient as scientific computation code like matrix multiplication can be, which is otherwise far out of reach for DBMS software [BGB98, KPH+98, TLPZT97, ADHW99].

While this thesis focuses on the core issues concerning Monet, during the course of this research, Monet has successfully been applied in commercial and scientific projects in a wide range of application areas. The author personally participated in the creation of the data mining spin-off company Data Distilleries during the course of this Ph.D. track, and later as full-time overall system architect. Data Distilleries is now a global player in the analytical CRM market, and still uses Monet as back-end in its analytical tool. This tool combines interactive, multi-algorithm, data mining functionality with ad-hoc OLAP. Achieving more than 36-fold speed-up over Oracle8 in data mining loads [BRK98], and running repositories larger than 100GB in daily production at the data centers of the largest Dutch financial institutions, Monet has proven its value in real-life applications.

Monet is also used for scientific applications in the areas of GIS[BK95, BQK96, WvZF+98], XML [SWK+01, SKW01, WSK99], information retrieval [dVB98] and multi-media [NK97, NQK98, NK98, KNW98, dV99, Nes00, dV00, dVWAK00, BdVNK01, dVMNK02]. This has led to a wealth of extension modules that introduce new atomic types, search accelerators and algebraic operators to MIL.

In the near future, our research group will make Monet and an SQL-2 front-end as described in this thesis publicly available in open source.

Run-time Query Optimization Monet defers important physical query optimization decisions to run-time, both enhancing their quality and simplifying the query optimization process. What makes MIL stand out from other database languages is that it is both a logical and an execution language. By providing a direct implementation for this logical algebra language, the Monet system separates query optimization in a *strategical* and a *tactical* phase. Query optimizing systems producing MIL code must transform a high-level query in a sequence of appropriate MIL primitives. This includes determining a good (join-)order, but excludes translation to physical primitives. Choosing an algorithm is performed at run-time inside each MIL operator during the process of tactical query optimization. By using *properties* maintained on relation fragments and full propagation of these properties across operators, Monet conserves maximum information about the data that is being processed, which is also combined with run-time system statistics. By off-loading these activities from the compile-time phase, this approach simplifies the task of query optimizers, and has the potential to take better optimization decisions, as more run-time information is taken into account.

Cache-Conscious Query Processing New Radix-Algorithms that provide the basic building blocks for cache-optimized query processing, were contributed in this thesis. These algorithms trade memory access cost for extra CPU cost. Given the fact that memory latency does not improve (significantly) but CPU power increases exponentially each year, this has become a highly beneficial trade-off in the past decade, and will even be more so in the next. We have shown on hardware from 1999 that the Radix-Algorithms can accelerate equi-join by almost a magnitude (while CPUs from 2002 are three times more powerful yet memory latency has improved nothing). Thus, we conclude that the "battleground" for database query optimization is shifting from I/O to the DRAM layer in the memory hierarchy.

As CPU caches are even more uncontrollable than virtual memory, it is not possible to create an explicit "CPU-cache-buffer-manager" in the DBMS. Rather, one must just rely on the hardware-implemented eviction mechanism (usually LRU). The implicit idea behind Partitioned Hash-Join and also our Radix-Algorithms is to let it work well on a LRU cache. An interesting property of such algorithms is that when it runs well on one cache level, it also runs well on the lower levels (but not the other way around). However, virtual memory is also an LRU cache (fully associative and with large granularity and latency). Thus, when one switches to LRU-cache optimized query processing algorithms with the initial purpose of optimizing CPU cache utilization, the access patterns may become so predictable that a DBMS buffer manager is not strictly needed anymore!

In the case of cache-conscious query processing in Monet, extra performance is gained by the vertical decomposition and data compression (enumeration types) as these optimize cache-line usage. The column-wise query processing is elegantly made cache-conscious by the cluster-decluster join strategy, based on the innovative *Radix-Decluster* algorithm, which represents a significant new research result. The discussion of the SQL-2 front-end showed that by using this strategy, *all* MIL code generated for the relational algebra operators can be cache-optimized. Selections are sequential, equi-joins use the Radix-Partitioned Hash-Join, `ORDER BY`s rely on Quick-Sort and the cluster-decluster join strategy, while `GROUP BY`s make a dynamic choice depending on the cache size between hash-grouping or `ORDER BY`.

Explicit Transaction Management A novel idea tested out in Monet is to construct a query language that in itself does not provide ACID properties, rather provides the building blocks for creating a failsafe transaction system. The advantage of this approach is that queries need only pay for the overheads of transaction management if their application requires transaction functionality. Many query-intensive applications work with bulk-updates or are append-only, so it is a waste to give up precious query performance in e.g. a complicated ARIES algorithm that is intertwined with the buffer manager and random I/O constraints [MHL+92]. While it is easy to point out that absence of transaction management may enhance performance, this thesis also tries to make clear that by providing a carefully defined set of update primitives, MIL *is* capable of supporting sophisticated transaction functionality.

8.2 Future Work

While we consider Monet a largely successful project, there are a many areas were improvements are possible:

Tactical Query Optimization Language Issues Currently, Monet maintains a hard-coded (non-extensible) set of BAT- and column-properties inside the BAT descriptor, whose propagation by the various MIL operators is also hard-coded in their operator implementations. This conflicts with the extensibility requirement (new domains might need new properties) and is error-prone.

Another issue is the fact that some MIL operators still make tactical decisions inside their C implementations. It can be considered just an implementation issue to convert this decision code into proper scripted MIL procedures. However, doing so will expose a large collection of MIL physical operators that clutter the language, such that it is an interesting question whether MIL should, like other query algebras [Güt89], go back to a design with a separate logical and a physical algebra level.

In any case, it would be much more elegant if MIL operator signatures could be extended with predicates that go beyond the types of the parameters and e.g. contain predicates or even rule-expressions on BAT- and column-properties.

Finally, the propagation of such properties would better be described explicitly in propagation rules. It is an open question how powerful such a rule system would be, and whether it would take over the role of MIL procedures as the vehicle for tactical optimization entirely. Also, the interaction of such a rule system with query rewrite systems, such as the MIL squeezer [MPK00] that performs multi-query optimizations, should be studied further.

Pure Column Algebra In the discussion of query processing strategies, we almost exclusively use BATs that have one `void` column, as such BATs can be joined together at trivial cost. Conversely, *not* using such BATs can seriously hinder performance. The question then rises whether a "true" single-column algebra would not even be more elegant. This would only really cause problems for the MIL-join and -order, as these are currently the only operators to produce "real" two-column BATs.

Thus, this column algebra would need to allow an operator to deliver multiple colums as a result. Giving back multiple results is also a functional wish we encountered e.g. in multi-media retrieval. A problem with multiple results is that introducing a tuple or record type should be avoided in order to keep the language a clean column-oriented algebra and thus data model neutral. A possible solution would be to drop the concept of nested functional-style expressions, going back to an assembly-like language where programs consist of sequences of single-operator statements that fill one or multiple variables.

Programmable Virtual Memory On RAID One of our conclusions is that the increasing memory access bottleneck could spell the end of the DBMS buffer manager, handling I/O implicitly via virtual memory. One of the issues that needs investigation is whether current virtual memory implementations can indeed support such an architecture, and in particular whether it can sustain sequential or quasi-sequential LRU cacheable access patterns (such as bi-directional scans of Quick-Sort, or the multi-cursor sequential scan from Radix-Cluster) at sufficiently high bandwidth in order to

not let the query processing operators be I/O limited. A rough rule-of-thumb is that the typical MIL operator uses in the range of 40-100 CPU cycles per tuple, thus a 1GHz CPU consumes 10-25 million tuples per second, which equals 40-100MB/s (assuming 4-byte wide BUNs).

Such bandwidths are attainable by (cheap) RAID devices (or software RAID) but the question is whether the OS implementations of virtual memory are up to this. Our early indications are that performance should be improved, and this could be done using virtual memory reprogramming techniques previously used in OODBMSs [LLOW91, Ple97] by reprogramming catching page faults and scheduling large asynchronous I/O read-ahead operations according to specified access pattern hints.

Parallel Execution As hardware trends continue as now, the introduction of Simultaneous Multi-Threading (SMT) and multi-processing on-a-chip will create a situation that all commodity processor chips contain multiple (logical) CPUs, making the pervasiveness of parallel processing finally a reality. At the same time, it is obvious that highly scalable systems use a shared-nothing architecture, thus the future of parallel processing architectures is hybrid.

At the start of this research, we had hoped to pay more attention to investigating the interaction between column-wise query processing and hybridly parallel machine architectures. This remains an interesting research area to be pursued in the future. In this context, it would be particularly interesting to look at cache-conscious parallel query processing, as the Radix-Algorithms presented in this thesis are designed for uniprocessor execution only.

Just-In-Time Query Compilation Monet can make use of highly optimized implementations of its algebraic operators, as these work column-wise and have a low degree of freedom (i.e. they are "RISCy").

A different road to obtain high CPU efficiency would be to use just-in-time query compilation with real-time code-expansion. Such a system architecture would parse queries and generate highly CPU-efficient Java/C/C++-code, possibly even enriched by pieces of machine-specific assembly (e.g. containing SSE2 instructions). That code is then natively compiled into assembly on-the-fly, and loaded as a shared library, and kept around for re-use later on. The data model to be used would – given our good experiences – use vertically fragmented structures, but it is an open question what query language would best be suited for such a system. The idea might be pursued in MIL, possibly compiling multiple nested MIL operators into one execution operator, but it is possible, that better results would be obtained by doing this on an tuple- or object-algebra as these inherently have more work available per tuple.

Bibliography

[ABC+76] M. Astrahan, M. Blasgen, D. Chamberlin, K. Eswaran, J. Gray, P. Griffiths, W. King, R. Lorie, P. McJones, J. Mehl, G. Putzolu, I. Traiger, B. Wade, and V. Watson. System R: Relational Approach to Database Management. *ACM Trans. on Database Systems*, 1(2):97–137, 1976.

[ABG+86] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proc. of the USENIX Summer Conference*, pages 93–113, Atlanta, GA, USA, May 1986.

[ACM+98] D. August, D. Connors, S. Mahlke, J. Sias, K. Crozier, B. Cheng, P. Eaton, Q. Olaniran, and W. Hwu. Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture. In *Proc. of the Int'l Symp. on Computer Architecture*, pages 227–237, June 1998.

[ADHW99] A. Ailamaki, D. DeWitt, M. Hill, and D. Wood. DBMSs on modern processors: Where does time go? In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 266–277, Edinburgh, Scotland, UK, September 1999.

[Adv99] Advanced Micro Devices. *AMD Athlon Processor Architecture*, 1999. http:// www. amd. com/ us-en/ assets/ content_type/ white_papers_and_tech_docs/ architecture_wp.pdf.

[Adv00] Advanced Micro Devices. *AMD Athlon Processor: x86 Code Optimization Guide*, 2000. http:// www. amd. com/ us-en/ assets/ content_type/ white_papers_and_tech_docs/ 22007.pdf.

[Adv01] Advanced Micro Devices. *QuantiSpeed Architecture*, 2001. http:// www. amd. com/ us-en/ assets/ content_type/ white_papers_and_tech_docs/ QuantiSpeed_whitepaper.pdf.

[AGPR99] S. Acharya, P. Gibbons, V. Poosala, and S. Ramaswamy. Join Synopses for Approximate Query Answering. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 275–286, Philadephia, Pennsylvania, USA, June 1999.

[AP92] A. Analyti and S. Pramanik. Fast Search in Main Memory Databases. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 215–224, San Diego, CA, USA, June 1992.

[AvdBF+92] P. Apers, C. van den Berg, J. Flokstra, P. Grefen, M. Kersten, and A. Wilschut. PRISMA/DB: A Parallel Main Memory Relational DBMS. *IEEE Trans. on Knowledge and Data Eng.*, 4(6):541–554, December 1992.

[Bat79] D. Batory. On Searching Transposed Files. *TODS*, 4(4):531–544, 1979.

[Bat85] D. Batory. Modeling the Storage Architectures of Commercial Database Systems. *TODS*, 10(4):463–528, 1985.

[Bat86] D. Batory. Extensible Cost Models and Query Optimization in Genesis. *IEEE Data Eng. Bull.*, 10(4), December 1986.

[BBC+98] P. Bernstein, M. Brodie, S. Ceri, D. DeWitt, M. Franklin, H. Garcia-Molina, J. Gray, G. Held, J. Hellerstein, H. Jagadish, M. Lesk, D. Maier, J. Naughton, H. Pirahesh, M. Stonebraker, and J. Ullman. The Asilomar Report on Database Research. *ACM SIGMOD Record*, 27(4):78–80, 1998.

[BBG+88] D. Batory, J. Barnett, J. Garza, K. Smith, K. Tsukuda, B. Twichell, and T. Wise. GENESIS: An Extensible Database Management System. *IEEE Trans. on Software Eng.*, 14(11), November 1988.

[BBG+99] J. Baulier, P. Bohannon, S. Gogate, C. Gupta, S. Haldar, S. Joshi, A. Khivesera, Henry F. Korth, P. McIlroy, J. Miller, P. Narayan, M. Nemeth, R. Rastogi, S. Seshadri, A. Silberschatz, S. Sudarshan, M. Wilder, and C. Wei. DataBlitz Storage Manager: Main Memory Database Performance for Critical Applications. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 519–520, Philadephia, Pennsylvania, USA, June 1999.

[BCF+95] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, 1995.

[BD83] H. Boral and D. DeWitt. Database machines: An idea whose time has passed? In H. Leilich and M. Missikoff, editors, *Database Machines*, page 166. Springer-Verlag, Berlin, New York, etc., 1983.

[BDS89] H. Boral, D. DeWitt, and M. Smith. A Single-User Performance Evaluation of the Teradata Database Machine. In *Proc. of the Int'l. Workshop on High Performance Transaction Systems*, volume 359 of *Lecture Notes in Computer Science*, pages 244–276, Berlin, New York, etc., September 1989. Springer-Verlag.

[BdVNK01] P. Bosch, A. de Vries, N. Nes, and M. Kersten. A case for Image Quering through Image Spots. In *Storage and Retrieval for Media Databases 2001*, volume 4315 of *Proceedings of SPIE*, pages 20–30, San Jose, CA, USA, January 2001.

[BGB98] L. Barroso, K. Gharachorloo, and E. Bugnion. Memory System Characterization Of Commercial Workloads. In *Proc. of the Int'l Symp. on Computer Architecture*, Barcelona, Spain, June 1998.

[BHK+86] A. Brown, T. Hirata, A. Koehler, K. Vishwanath, J. Ng, M. Pechulis, M. Sikes, D. Singleton, and J. Veazey. Data Base Management for HP Precision Architecture Computers. *Hewlett-Packard Journal*, 37(12):33–48, 1986.

[BK95] P. Boncz and M. Kersten. Monet: An Impressionist Sketch of an Advanced Database System. In *Proc. Basque International Workshop on Information Technology*, San Sebastian, Spain, July 1995.

[BK99] P. Boncz and M. Kersten. MIL Primitives For Querying a Fragmented World. *The VLDB Journal*, 8(2):101–119, March 1999.

[BKK95] P. Boncz, F. Kwakkel, and M. Kersten. High Performance Support for OO Traversals in Monet. Technical Report CS-R9568, CWI, Amsterdam, The Netherlands, 1995.

[BKK96] P. Boncz, F. Kwakkel, and M. Kersten. High Performance Support for OO Traversals in Monet. In *Proc. of the British National Conference on Databases*, volume 1094 of *Lecture Notes in Computer Science*, pages 152–169, Edinburgh, UK, July 1996.

[BMK99] P. Boncz, S. Manegold, and M. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 54–65, Edinburgh, Scotland, UK, September 1999.

[BMK00] P. Boncz, S. Manegold, and M. Kersten. Optimizing Database Architecture for the New Bottleneck: Memory Access (Extended Paper Published For Best-Of-VLDB'99). *The VLDB Journal*, 9(3):231–246, December 2000.

[BOS91] P. Butterworth, A. Otis, and J. Stein. The Gemstone Object Database Management System. *Communications of the ACM*, 34(10):64–77, 1991.

[BQK96] P. Boncz, W. Quak, and M. Kersten. Monet and its Geographical Extensions: a Novel Approach to High-Performance GIS Processing. In *Proc. of the Intl. Conf. on Extending Database Technology*, pages 147–166, Avignon, France, June 1996.

[BRK98] P. Boncz, T. Rühl, and F. Kwakkel. The Drill Down Benchmark. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 628–632, New York, NY, USA, June 1998.

[BWK98] P. Boncz, A. Wilschut, and M. Kersten. Flattening an Object Algebra to Provide Performance. In *Proc. of the IEEE Int'l. Conf. on Data Engineering*, pages 568–577, Orlando, FL, USA, February 1998.

[BZ98] R. Berrendorf and H. Ziegler. PCL - The Performance Counter Library. Technical Report FZJ-ZAM-IB-9816, ZAM, Forschungzentrum Jülich, Germany, 1998.

[CAB+94] R. Cattell, T. Atwood, D. Barry, J. Dubl, G. Ferran, M. Loomis, and D. Wade. *The Object Database Standard: ODMG*. Morgan Kaufmann, San Mateo, CA, USA, 1994.

[CB74] A. Chamberlin and A. Boyce. SEQUEL: A Structured English Query Language. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, Ann Arbor, Michigan, May 1974.

[CBB+97] R. Cattell, D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, and D. Wade. *The Object Database Standard: ODMG2.0*. Morgan Kaufmann, San Mateo, CA, USA, 1997.

[CCH+98] L. Colby, R. Cole, E. Haslam, N. Jazayeri, G. Johnson, W. McKenna, L. Schumacher, and D. White. Red Brick Vista: Aggregate Computation and Management. In *Proc. of the IEEE Int'l. Conf. on Data Engineering*, pages 174–177, Orlando, FL, USA, February 1998.

[CD85] H. Chou and D. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 121–141, Stockholm, Sweden, August 1985.

[CD87] M. Carey and D. DeWitt. An Overview of the EXODUS Project. *IEEE Data Eng. Bull.*, 10(2):47–54, 1987.

[CD96] M. Carey and D. DeWitt. Of Objects and Databases: A Decade of Turmoil. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 3–14, Mumbai, India, 1996.

[CD97] S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *ACM SIGMOD Record*, 26(1), March 1997.

[CDF+98] M. Carey, D. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. Tan, O. Tsatalos, S. White, and M. Zwilling. Shoring Up Persistent Applications. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 390–401, Minneapolis, MI, USA, June 1998.

[CDH+99] J. Clear, D. Dunn, B. Harvey, M. Heytens, P. Lohman, A. Mehta, M. Melton, H. Richardson, L. Rohrberg, A. Savasere, R. Wehrmeister, and M. Xu. NonStopSQL/MX. In *Proc. of the Int.'l Conference on Knowledge Discovery and Data Mining*, San Diego, CA, USA, August 1999.

[CDN95] M. Carey, D. DeWitt, and J. Naughton. The 007 Benchmark. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 21–21, Washington, DC, USA, May 1995.

[CG94] R. Cole and G. Graefe. Optimization of Dynamic Query Execution Plans. *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 150–160, May 1994.

[Cha98] S. Chaudhuri. Data Mining and Database Systems: Where is the Intersection? *IEEE Data Eng. Bull.*, 21(1):4–8, 1998.

[CK85] G. Copeland and S. Khoshafian. A Decomposition Storage Model. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 268–279, Austin, TX, USA, May 1985.

[Cod70] A. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6), June 1970.

[Coh78] L. Cohen. *Data Base Management Systems*. Q.E.D. Information Sciences, Inc., Wellesley, MA, USA, 1978.

[CRRS93] K. Chew, A. Reddy, T. Romer, and A. Silberschatz. Kernel Support for Recoverable-Persistent Virtual Memory. Technical Report CS-TR-93-06, University of Texas, Austin, TX, USA, 1993.

[CZ96] M. Cherniak and S. Zdonik. Rule Languages and Internal Algebras for Rule-Based Optimizers. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 401–412, Montreal, Canada, June 1996.

[CZ98] M. Cherniak and S. Zdonik. Changing the Rules: Transformations for Rule-Based Optimizers. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 61–72, Seattle, WA, USA, June 1998.

[Dat85] A. Date. *An Introduction to Database Systems (4th Ed.)*. Addison-Wesley, Reading, MA, USA, 1985.

[Dav73] C. Davies. Recovery Semantics for A DB/DC System. In *Proc. ACM 73 Nat. Conf.*, pages 136–141, Atlanta, GA USA, 1973.

[Deu90] O. Deux. The Story of O2. *IEEE Trans. on Knowledge and Data Eng.*, 2(1):91–108, 1990.

[DeW79] D. DeWitt. DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems. *IEEE Trans. on Computers*, 28(6):395, June 1979.

[DG92] D. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[DGS+90] D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H-I. Hsiano, and R. Rasmussen. The Gamma Database Machine Project. *IEEE Trans. on Knowledge and Data Eng.*, 2(1), March 1990.

[Die99] K. Diefendorff. Pentium III = Pentium II + SSE. *Microprocessor Report*, 13(3):6–11, 1999.

[DKL+94] D. DeWitt, N. Kabra, J. Luo, J. Patel, and J. Yu. Client-Server Paradise. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 558–569, Santiago (Chile), September 1994.

[DKO+84] D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood. Implementation Techniques for Main Memory Database Systems. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 1–8, Boston, MA, USA, June 1984.

[DV92] S. Danforth and P. Valduriez. A FAD for Data Intensive Applications. *IEEE Trans. on Knowledge and Data Eng.*, 4(1), February 1992.

[dV99] A. de Vries. *Content and Multimedia Database management Systems.* PhD thesis, Universiteit Twente, 1999.

[dV00] A. de Vries. Challenging Ubiquitous Inverted Files. In *Proceedings of the First DELOS Network of Excellence Workshop on "Information Seeking, Searching and Querying in Digital Libraries"*, volume 01/W001 of *ERCIM Workshop Reports*, pages 71–75, Zrich, Switzerland, December 2000.

[dVB98] A. de Vries and H. Blanken. Database Technology and the Management of Multimedia Data in MIRROR. *Proc. Multimedia Storage and Archiving Systems III*, pages 443–453, November 1998.

[dVMNK02] A. de Vries, N. Mamoulis, N. Nes, and M. Kersten. Efficient image retrieval by exploiting vertical fragmentation. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, Madison, Wisconsin, USA, June 2002.

[dVW98] A. de Vries and A. Wilschut. On the Integration of IR and Databases. In *Proc. IFIP 2.6 Working Conf. on Database Semantics*, Edinburgh, Scotland, UK, July 1998.

[dVWAK00] A. P. de Vries, M. A. Windhouwer, P. M. G. Apers, and M. L. Kersten. Information Access in Multimedia Databases based on Feature Models. *New Generation Computing*, 18(4):323–339, October 2000.

[eaENS89] et al. E. Neuhold and M. Stonebraker. Future Directions in DBMS Research. *ACM SIGMOD Record*, 18(1):17–26, 1989.

[EEL+97] S. Eggers, J. Emer, H. Levy, J. Lo, R. Stamm, and D. Tullsen. Simultaneous Multithreading: A Platform for Next-Generation Processors. *IEEE Micro*, 17(5):12–20, 1997.

[Eic87] M. Eich. A Classification and Comparison of Main Memory Database Recovery Techniques. In *Proc. of the IEEE Int'l. Conf. on Data Engineering*, pages 332–339, Los Angeles, CA, USA, February 1987.

[Eic89] A. Eich. Main Memory Database Research Directions. In *Database Machines. 6th International Workshop*, pages 251–268, Deauville, France, June 1989.

[FGN+95] D. Florescu, J. Gruser, M. Novak, P. Valduriez, and M. Ziane. Design and Implementation of Flora, A Language for Object Algebra. *Information Sciences*, 87(1-3):1–27, 1995.

[FKT86] S. Fushimi, M Kitsuregawa, and H. Tanaka. An Overview of The System Software of A Parallel Relational Database Machine GRACE. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, Kyoto, Japan, August 1986.

[FR97] G. Fahl and T. Risch. Query Processing Over Object Views of Relational Data. *The VLDB Journal*, 6(4):261–281, 1997.

[Fra84] A. Frank. Requirements for Database Systems Suitable to Manage Large Spatial Databases. In *Proc. of the Int'l. Symposium on Spatial Data Handling*, volume 1, pages 38–60, Zurich, Switzerland, 1984.

[GBC+95] D. Greenley, J. Bauman, D. Chang, D. Chen, R. Eltejaein, P. Ferolito, P. Fu, R. Garner, D. Greenhill, H. Grewal, K. Holdbrook, B. Kim, Leslie Kohn, H. Kwan, M. Levitt, G. Maturana, D. Mrazek, C. Narasimhaiah, K. Normoyle, N. Parveen, P. Patel, A. Prabhu, M. Tremblay, M. Wong, L. Yang, K. Yarlagadda, R. Yu, R. Yung, and G. Zyner. UltraSPARC: The Next Generation Superscalar 64-bit SPARC. In *COMPCON Digest of Papers*, pages 442–451, 1995.

[GBLP96] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. In *Proc. of the IEEE Int'l. Conf. on Data Engineering*, pages 152–159, New Orleans, LS, USA, February 1996.

[GD87] G. Graefe and D. DeWitt. The EXODUS Optimizer Generator. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, San Francisco, CA, USA, May 1987.

[Ger95] B. Gerber. Informix online XPS. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, page 463, San Jose (CA), May 1995.

[GGMS97] D. Gluche, T. Grust, C. Mainberger, and M. Scholl. Incremental Updates for Materialized OQL Views. In *Proc. of the Int'l. Conf. on Deductive and Object-Oriented Databases*, pages 52–66, Kyoto, Japan, December 1997.

[GKKS97] T. Grust, J. Kröger, D. Kluge, and M. Scholl. Query Evaluation in CROQUE - Calculus and Algebra Coincide. In *Proc. of the British National Conference on Databases*, pages 84–100, London, UK, July 1997.

[GLPK94] C. Galindo-Legaria, A. Pellenkoft, and M. Kersten. Fast, Randomized Join-Order Selection – Why Use Transformations? In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 85–95, Santiago, Chile, September 1994.

[GLPK95] C. Galindo-Legaria, A. Pellenkoft, and M. Kersten. Uniformly-distributed Random Generation of Join Orders. In *Proc. of the Int'l. Conf. on Database Theory*, pages 280–293, Prague, Czechia, January 1995.

[GMB+81] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, G. Putzolu, and I. Traiger. The Recovery Manager of the System R Database Manager. *ACM Computing Surveys*, 13(2):223–243, 1981.

[GMS92] H. Garcia-Molina and K. Salem. Main Memory Database Systems: An Overview. *IEEE Trans. on Knowledge and Data Eng.*, 4(6):509–516, December 1992.

[GMSvE98] M. Godfrey, T. Mayr, P. Seshadri, and T. von Eicken. Secure and Portable Database Extensibility. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 390–401, Seattle, WA, USA, June 1998.

[GR91] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, USA, 1991.

[Gra93a] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[Gra93b] J. Gray, editor. *The Benchmark Handbook*. Morgan Kaufmann, San Mateo, CA, USA, 1993.

[Gra94] G. Graefe. Volcano, An Extensible and Parallel Dataflow Query Processing System. *IEEE Trans. on Knowledge and Data Eng.*, 6(1), February 1994.

[Gro87] The Tandem Database Group. NonStop SQL, a distributed, high-performance, high-availability implementation of SQL. In *2nd Intern. Workshop on High Performance Transaction Systems*, Lecture Notes in Computer Science, pages 60–104, Asilomar, Pacific Grove, CA, USA, September 1987. Springer-Verlag.

[GT96] G. Goldman and P. Tirumalai. UltraSPARC-II: The Advancement of Ultra-Computing. In *COMPCON Digest of Papers*, pages 417–423, 1996.

[Gut84] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 47–57, Boston, MA, USA, June 1984.

[Güt89] R.H. Güting. Gral: An extensible relational database system for geometric applications. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 33–44, Amsterdam (The Netherlands), August 1989.

[GW89] G. Graefe and K. Ward. Dynamic Query Evaluation Plans. *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 358–366, May 1989.

[HFLP89] L. Haas, J. Freytag, G. Lohman, and H. Pirahesh. Extensible Query Processing in Starburst. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 377–388, Portland, OR, USA, May 1989.

[HKK97] E. Han, G. Karypis, and V. Kumar. Scalable Parallel Data Mining for Association Rules. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 277–288, Tucson, AZ, USA, May 1997.

[HKMT95] M. Holsheimer, M. Kersten, H. Mannila, and H. Toivonen. A Perspective on Databases and Data Mining. In *Proc. of the Int.'l Conference on Knowledge Discovery and Data Mining*, Montreal, Canada, August 1995.

[HRU96] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing Data Cubes Efficiently. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 205–216, Montreal, Canada, June 1996.

[HS81] A. Hong and A. Su. Associative Hardware and Software Techniques for Integrity Control. *ACM Trans. on Database Systems*, 6(3):416–440, 1981.

[HSB+82] L. Haas, P. Selinger, E. Bertino, D. Daniels, B. Lindsay, G. Lohman, Y. Masunaga, C. Mohan, P. Ng, P. Wilms, and R. Yost. R*: A Research Project on Distributed Relational DBMS. *IEEE Data Eng. Bull.*, 5(2), December 1982.

[HSU+01] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium 4 Processor. In *Intel Technology Journal*, http:// developer.intel.com/ technology/ itj/, February 2001.

[Hyp00] Hyperion Corp. Whitepaper. *Large-Scale Data Warehousing Using Hyperion Essbase OLAP Technology*, 2000. http:// www. cis. com. mx/ pdf/ wp_teraplex.pdf.

[IC91] Y. Ioannidis and S. Christodoulakis. On the Propagation of Errors in the Size of Join Results. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 268–277, Denver, CO, USA, May 1991.

[Imm92] A. Immon. *Bulding the Data Warehouse*. John Wiley and Sons, New York, USA, 1992.

[Inf98] Informix Corp. *MetaCube ROLAP User's Guide*, January 1998. http:// www. informix. com/ answers/ english/ docs/ 42metacube/ 5219.pdf.

[Int01] Intel Corp. *Introduction To HyperThreading Technology (document number 250008-002)*, 2001. http:// developer. intel. com/ technology/ download/ 25000802.pdf.

[IP95] Y. Ioannidis and V. Poosala. Balancing Histogram Optimality and Practicality for Query Result Size Estimation. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 233–244, San Jose, CA, USA, May 1995.

[JLR+94] H. Jagadish, D. Lieuwen, R. Rastogi, A. Silberschatz, and S. Sudarshan. Dalí: A High Performance Main Memory Storage Manager. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 48–59, Santiago, Chile, September 1994.

[JSS93] H. Jagadish, A. Silberschatz, and S. Sudarshan. Recovering from Main-Memory Lapses. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 391–404, Dublin, Ireland, August 1993.

[KCJ+87] S. Khoshafian, G. Copeland, T. Jagodis, H. Boral, and P. Valduriez. A Query Processing Strategy for the Decomposed Storage Model. In *Proc. of the IEEE Int'l. Conf. on Data Engineering*, pages 636–643, Los Angeles, CA, USA, February 1987.

[KD98] N. Kabra and D. DeWitt. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 106–117, Seattle, WA, USA, June 1998.

[KdB94] M. Kersten and M. de Boer. Query Optimisation Strategies for Browsing Sessions. In *Proc. of the IEEE Int'l. Conf. on Data Engineering*, pages 478–487, Houston, TX, USA, February 1994.

[Ken95] Kenan Technologies. *An Introduction to OLAP Multidimensional Database Technology*, 1995. http://www.accrue.com/olap/wp_intro_olap.pdf.

[Ker89] M. Kersten. Using Logarithmic Code-Expansion to Speedup Index Access and Maintenance. In *Proc. of the Int'l. Conf. on Foundation on Data Organization and Algorithms*, pages 228–232, Paris, France, October 1989.

[KGBW90] W. Kim, J. Garza, N. Ballou, and D. Woelk. Architecture of the ORION Next-Generation Database System. *IEEE Trans. on Knowledge and Data Eng.*, 2(1):109–124, 1990.

[KK98] J. Karlsson and M. Kersten. Transparent Distribution in a Storage Manager. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1658–1664, Las Vegas, NV, USA, July 1998.

[KMBN01] M. Kersten, S. Manegold, P. Boncz, and N. Nes. Macro- and Micro- Parallelism in a DBMS. In *Proceedings of the European Conference on Parallel Processing (EuroPar)*, Lecture Notes in Computer Science (LNCS), Springer-Verlag, pages 6–15, Manchester, UK, August 2001.

[KNW98] M. L. Kersten, N. Nes, and M. A. Windhouwer. A Feature Database for Multimedia Objects. In *ERCIM Database Research Group Workshop on Metadata for Web Databases*, pages 49–57, Bonn, Germany, May 1998.

[KP99] J. Keshava and V. Pentkovski. Pentium III Processor Implementation Tradeoffs. In *Intel Technology Journal*, http://developer.intel.com/technology/itj/, May 1999.

[KPH+98] K. Keeton, D. Patterson, Y. He, A. Raphael, and W. Baker. Performance Characterization of a quad Pentium Pro SMP using OLTP workloads. In *Proc. of the Int'l Symp. on Computer Architecture*, pages 15–26, Barcelona, Spain, June 1998.

[KPP+97] C. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanović, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick. Scalable Processors in the Billion-Transistor Era: IRAM. *IEEE Computer*, 30(9):75–78, September 1997.

[KPvdB92] M. Kersten, S. Plomp, and C. van den Berg. Object Storage Management in Goblin. In *Proceedings of the International Workshop on Distributed Object Management*, pages 100–116, Edmonton, AL, Canada, August 1992. Morgan Kaufmann.

[KSHK97] M. Kersten, A. Siebes, M. Holsheimer, and F. Kwakkel. Research and Business challenges in Data Mining Technology. In *Proc. Datenbanken in Büro, Technik und Wissenschaft*, pages 1–16, Ulm, Germany, March 1997.

[LC86a] T. Lehman and M. Carey. A Study of Index Structures for Main Memory Database Management Systems. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 294–303, Kyoto, Japan, August 1986.

[LC86b] T. Lehman and M. Carey. Query Processing in Main Memory Database Systems. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 239–250, Washington, DC, USA, May 1986.

[LH99] G. Lauterbach and T. Horel. UltraSPARC-III: designing third generation 64-bit platforms. *IEEE Micro*, 19(3):56–66, 1999.

[LL97] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proc. of the Int'l Symp. on Computer Architecture*, pages 241–251, New York, USA, June 1997. ACM Press.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore Database System. *Communications of the ACM*, 34(10):50–63, 1991.

[LN96] S. Listgarten and M. Neimat. Modelling Costs for a MM-DBMS. In *Proc. of the Int'l. Workshop on Real-Time Databases, Issues and Applications*, pages 72–78, Newport Beach, CA, USA, March 1996.

[LOT94] H. Lu, B. Ooi, and K. Tan. *Query Processing in Parallel Relational Database Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1994.

[LPR99] J. Lindstrom, Tand P. Porkka, and K. Raatikainen. *A Distributed Real-Time Main-Memory Database for Telecommunication*. ClustRa., January 1999. Workshop on Databases in Telecommunications, Edinburgh. http:// eureP817. norbotten. telia. com/ workshop/ slides/ process_cdr.ppt.

[LS98] B. Li and D. Shasha. Free Parallel Data Mining. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 541–543, Seattle, WA, USA, June 1998.

[LVZZ94] R. Lanzelotte, P. Valduriez, M. Zaït, and M. Ziane. Industrial-strength parallel query optimization: issues and lessons. *Information Systems*, 19(4):311–330, 1994.

[MBK99] S. Manegold, P. Boncz, and M. Kersten. Optimizing Main-Memory Join On Modern Hardware. Technical Report INS-R9912, CWI, Amsterdam, The Netherlands, October 1999.

[MBK00] S. Manegold, P. Boncz, and M. Kersten. What happens during a Join? – Dissecting CPU and Memory Optimization Effects. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 339–350, Cairo, Egypt, September 2000.

[MBK02] S. Manegold, P. Boncz, and M. Kersten. Optimizing Main-Memory Join On Modern Hardware. *IEEE Trans. on Knowledge and Data Eng.*, 14(4):709–730, 2002.

[McC95] A. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Technical Committee on Computer Architecture newsletter*, December 1995.

[MHL+92] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *TODS*, 17(1), 1992.

[Mic99] Microstrategy Corp. Whitepaper. *MicroStrategy Intelligence Server Fact Sheet*, 1999. http:// www. microstrategy. com/ files/ Intel_Server_1100.pdf.

[MKW+98] S. McKee, R. Klenke, K. Wright, W. Wulf, M. Salinas, J. Aylor, and A. Batson. Smarter Memory: Improving Bandwidth for Streamed References. *IEEE Computer*, 31(7):54–63, July 1998.

[Mow94] A. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, Computer Science Department, 1994.

[MPK00] S. Manegold, A. Pellenkoft, and M. Kersten. A Multi-Query Optimizer for Monet. In *Proc. of the British National Conference on Databases*, pages 36–51, Exeter, United Kingdom, July 2000.

[NBC+94] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. AlphaSort: A RISC Machine Sort. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 233–242, Minneapolis, MN, USA, May 1994.

[Nes00] N. Nes. *Image Database Management System Design Considerations, Algorithms and Architecture.* PhD thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, December 2000.

[NK97] N. Nes and M. Kersten. Region-based indexing in an image database. In *Proceedings International Conference on Imaging Science, Systems, and Technology*, pages 207–215, Las Vegas, ND, USA, July 1997.

[NK98] N. Nes and M. Kersten. The Acoi Algebra: a Query Algebra for Image Retrieval Systems. In *Proc. of the British National Conference on Databases*, pages 77–88, Cardiff, Wales, UK, July 1998.

[NQK98] N. Nes, W. Quak, and M. Kersten. Metric Indexing to Improve Distance Joins. In B. ter Haar, D. Epema, F. Tonino, and A. Wolters, editors, *Proceedings of the Annual Conference of the Advanced School for Computing and Imaging*, pages 133–139, Lommel, Belgium, 1998.

[NVI01] NVIDIA. *nForce IGP Dynamic Adaptive Speculative Pre-processor (DASP)*, 2001. http:// www. nvidia. com/ docs/ IO/ 14/ ATT/ nForce_DASP_Tech_Brief.pdf.

[OFW99] S. Oberman, G. Favor, and F. Weber. AMD 3DNow! technology: architecture and implementations. *IEEE Micro*, 19(2):37–48, 1999.

[OHUS96] M. Olson, W. Hong, M. Ubell, and M. Stonebraker. Query Processing in a Parallel Object-Relational Database System. *IEEE Data Eng. Bull.*, 19(4):3–10, 1996.

[Oll78] A. Olle. *The CODASYL Approach to Data Base Management.* John Wiley and Sons, New York, USA, 1 edition, 1978.

[O'N87] P. O'Neil. Model 204 Architecture and Performance. In *2nd Intern. Workshop on High Performance Transaction Systems*, Lecture Notes in Computer Science, pages 40–59, Asilomar, Pacific Grove, CA, USA, September 1987. Springer-Verlag.

[OQ97] P. O'Neil and D. Quass. Improved Query Performance with Variant Indexes. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 38–49, Tucson, AZ, USA, May 1997.

[Ora97a] Oracle Corp. *Oracle8 SQL Reference*, April 1997. http://download-west.oracle.com/otndoc/oracle9i/server.901/a90125.pdf.

[Ora97b] Oracle Corp. Whitepaper. *Express Server: Delivering OLAP to the Enterprise*, January 1997. http://www.rockportsoft.com/datasheets/oes_ds.pdf.

[OSS77] E. Ozkarahan, S. Schuster, and K. Sevcik. Performance Evaluation of a Relational Associative Processor. *ACM Trans. on Database Systems*, 2(2):175–195, 1977.

[Pel97] A. Pellenkoft. *Probabilistic and Transformation based Query Optimization.* PhD thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, 1997.

[PGK88] D. Patterson, G. Gibson, and R. Katz. Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 109–116, Chicago, IL, USA, June 1988.

[PGLK97] A. Pellenkoft, C. Galindo-Legaria, and M. Kersten. The Complexity of Transformation-Based Join Enumeration. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 306–315, Athens, Greece, September 1997.

[PIHS96] V. Poosala, Y. Ioannidis, P. Haas, and E. Shekita. Improved Histograms for Selectivity Estimation of Range Predicates. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 294–305, Montreal, Canada, May 1996.

[Ple97] J. Plewe. POET 4.0 und seine Schnittstellen: verschenktes Potential im Query-Bereich. *Objekt Focus, Supplement in Datenbank Focus*, pages 15–19, 1997.

[PWW97] A. Peleg, S. Wilkie, and U. Weiser. Intel MMX for Multimedia PCs. *Communications of the ACM*, 40(1):24–38, 1997.

[RAA+88] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. The CHORUS Distributed Operating System. *Computing Systems*, 1(4):305–370, 1988.

[Ram96] Rambus Technologies, Inc. *Direct Rambus Technology Disclosure*, 1996. http://www.rambus.com/docs/drtechov.pdf.

[RBP+93] R. Rastogi, P. Bohannon, J. Parker, A. Silberschatz, S. Seshadri, and S. Sudarshan. Distributed Multi-Level Recovery in Main-Memory Databases. *Distributed and Parallel Databases*, 6(1):41–71, 1993.

[RS97] M. Roth and P. Schwarz. Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 266–275, Athens, Greece, September 1997.

[RvIG97] E. Riedel, C. van Ingen, and J. Gray. A Study of Windows NT Sequential IO Performance. Technical Report P117, Microsoft Research, Redmond, WA, USA, 1997.

[SA00] H. Sharangpani and K. Arora. Itanium Processor Microarchitecture. *IEEE Micro*, 20(5):24–43, 2000.

[SAB94] H. Steenhagen, P. Apers, and H. Blanken. Optimization of Nested Queries in a Complex Object Model. In *Proc. of the Intl. Conf. on Extending Database Technology*, pages 337–350, Cambridge, UK, March 1994.

[SABdB94] H. Steenhagen, P. Apers, H. Blanken, and R. de By. From Nested-Loop to Join Queries in OODB. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 618–629, Santiago, Chile, September 1994.

[SAC+79] P. Selinger, A. Astrahan, A. Chamberlin, R. Lorie, and A. Price. Access Path Selection in a Relational Database Management System. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 23–34, Boston, MA, USA, May 1979.

[SAH87] M. Stonebraker, J. Anton, and M. Hirohama. Extendability in POSTGRES. *IEEE Data Eng. Bull.*, 10(2):16–23, 1987.

[SCF+86] P. Schwarz, W. Chang, J. Freytag, G. Lohman, J. McPherson, C. Mohan, and H. Pirahesh. Extensibility in the Starburst Database System. In *Proc. Int'l Workshop on Object-Oriented Database Sys.*, Pacific Grove, CA, USA, September 1986.

[Sch87] H. Schek. DASDBS: A Kernel DBMS and Application-Specific Layers. *IEEE Data Eng. Bull.*, 10(2):62–64, 1987.

[SD90] D. Schneider and D. DeWitt. Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 469–480, Brisbane, Australia, August 1990.

[SdBB96] H. Steenhagen, R. de By, and H. Blanken. Translating OSQL Queries into Efficient Set Expressions. In *Proc. of the Intl. Conf. on Extending Database Technology*, pages 183–190, Avignon, France, March 1996.

[Sem97] Sematech. *National Roadmap For Semiconductor Technology: Technology Needs*, 1997. http://www.itrs.net/ntrs/publntrs.nsf.

[Ses98] P. Seshadri. Enhanced Abstract Data Types in Object-Relational Databases. *The VLDB Journal*, 7(3), 1998.

[SFGM93] M. Stonebraker, J. Frew, K. Gardels, and J. Meredith. The Sequoia 2000 Storage Benchmark. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 2–11, Washington (DC), May 1993.

[Sil97] Silicon Graphics, Inc., Mountain View, CA. *Performance Tuning and Optimization for Origin2000 and Onyx2*, January 1997. http://techpubs.sgi.com/library/manuals/3000/007-3430-003/pdf/007-3430-003.pdf.

[SKN94] A. Shatdahl, C. Kant, and J. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 510–512, Santiago, Chile, September 1994.

[SKW01] A. Schmidt, M. Kersten, and M. Windhouwer. Querying XML Documents Made Easy: Nearest Concept Queries. In *Proc. of the IEEE Int'l. Conf. on Data Engineering*, pages 321–329, Heidelberg, Germany, April 2001.

[SKWW00] A. Schmidt, M. Kersten, M. Windhouwer, and F. Waas. Efficient Relational Storage and Retrieval of XML Documents. In *International Workshop on the Web and Databases (In conjunction with ACM SIGMOD)*, pages 47–52, Dallas, TX, USA, May 2000.

[SL76] D. Severance and G. Lohman. Differential Files: Their Application to the Maintenance of Large Databases. *ACM Trans. on Database Systems*, 1(3):256–267, 1976.

[SLR96] P. Seshadri, M. Livny, and R. Ramakrishnan. E-ADTs: Turbo-Charging Complex Data. *IEEE Data Eng. Bull.*, 19(4):11–18, 1996.

[SLVZ95] B. Subramanian, T. Leung, S. Vandenberg, and S. Zdonik. The AQUA Approach to Querying Lists and Trees in Object-Oriented Databases. In *Proc. of the IEEE Int'l. Conf. on Data Engineering*, pages 80–89, Taipei, Taiwan, March 1995.

[SP97] P. Seshadri and M. Paskin. PREDATOR: An OR-DBMS with Enhanced Data Types. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 568–571, Tucson, AZ, USA, May 1997.

[SR86] M. Stonebraker and L. Rowe. The Design of POSTGRES. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, Washington, DC, USA, May 1986.

[SRKC95] S. Shekhar, S. Ravada, V. Kumar, and D. Chubb. Load-Balancing in High Performance GIS: Declustering Polygonal Maps. *Lecture Notes in Computer Science*, 951:196–206, 1995.

[Sto81] M. Stonebraker. Operating System Support for Database Management. *Communications of the ACM*, 24(7):412–418, 1981.

[Sto84] M. Stonebraker. Virtual Memory Transaction Management. *Operation Systems Review*, 18(2):8–16, 1984.

[Sto86] M. Stonebraker. Inclusion of New Types in Relational Database Systems. In *Proc. of the IEEE Int'l. Conf. on Data Engineering*, pages 262–269, Los Angeles, CA, USA, February 1986.

[Sto93] M. Stonebraker. *Readings in Database Systems, 2nd ed.* Morgan Kaufmann, San Mateo, CA, USA, 1993.

[Sun99] Sun Microsystems. *MAJC Archtitecture Tutorial*, 1999. http://www.sun.com/microelectronics/MAJC/documentation/docs/majctutorial.pdf.

[SWK+01] A. Schmidt, F. Waas, M. Kersten, D. Florescu, M. Carey, I. Manolescu, and R. Busse. Why and How to Benchmark XML Databases. *ACM SIGMOD Record*, 3(30), September 2001.

[SWKH76] M. Stonebraker, E. Wong, P. Kreps, and G. Held. The Design and Implementation of INGRES. *ACM Trans. on Database Systems*, 1(3):189–222, 1976.

[Syb96] Sybase Corp. Whitepaper. *Adaptive Server IQ*, July 1996. http://www.sybase.com/content/1008840/iq_wp_l00899.pdf.

[Tan99] Tantau Corp. *Infocharger: How to Analyse 1 Billion CDRs/sec on $200K Hardware*, January 1999. Workshop on Databases in Telecommunications, Edinburgh. http:// eurеP817. norbotten. telia. com/ workshop/ slides/ process_cdr.ppt.

[TDF01] J. Tendler, S. Dodson, and S. Fields. *E-server: Power4 System Architecture*. IBM, 2001. http:// www. ibm. com/ servers/ eserver/ pseries/ hardware/ whitepapers/ power4.pdf.

[Tea95] Shore Team. SHORE: Combining the Best Features of OODBMS and File Systems. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, page 486, San Jose, CA, USA, May 1995.

[Tea99] Times-Ten Team. In-Memory Data Management for Consumer Transactions: The Times-Ten Approach. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 528–529, Philadephia, Pennsylvania, USA, June 1999.

[TLPZT97] P. Trancoso, J. Larriba-Pey, Z. Zhang, and J. Torellas. The Memory Performance of DSS Commericial Workloads in Shared-Memory Multiprocessors. In *Int'l. Symp. on High Performance Computer Architecture*, San Antonio, TX, USA, January 1997.

[Tra95] Transaction Processing Performance Council. *TPC Benchmark D*, 1.2.3 edition, 1995. http:// www. tpc. org/ tpcd/ spec/ tpcd_current.pdf.

[Tra99] Transaction Processing Performance Council. *TPC Benchmark H*, 1.3.0 edition, 1999. http:// www. tpc. org/ tpch/ spec/ h130.pdf.

[Ull89a] A. Ullman. *Principles of Database and Knowledge-base Systems*, volume I. IEEE Computer Society Press, Rockville, Maryland, 1989.

[Ull89b] A. Ullman. *Principles of Database and Knowledge-base Systems*, volume II. IEEE Computer Society Press, Rockville, Maryland, 1989.

[UW97] A. Ullman and J. Widom. *A First Course in Database Systems*. Prentice Hall, Englewood Cliffs, NJ, USA, 1997.

[Val87] P. Valduriez. Join Indices. *ACM Trans. on Database Systems*, 12(2):218–246, June 1987.

[VD91] S. Vandenberg and D. DeWitt. Algebraic Query Processing in EXTRA/EXCESS. *IEEE Data Eng. Bull.*, 14(2):48–52, 1991.

[vdB94] C. van den Berg. *Dynamic query processing in a parallel object-oriented database system*. PhD thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, 1994.

[VKC86] P. Valduriez, S. Khoshafian, and G. Copeland. Implementation Techniques For Complex Objects. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 101–110, Kyoto, Japan, August 1986.

[WD92] S. White and D. DeWitt. A Performance Study of Alternative Object Faulting and Pointer Swizzling Strategies. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 419–431, Vancouver, BC, Canada, August 1992.

[WFA95a] A. Wilschut, J. Flokstra, and P. Apers. Parallel Evaluation of Multi-Join Queries. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 115–126, San Jose, CA, USA, May 1995.

[WFA95b] A. Wilschut, J. Flokstra, and P. Apers. Parallelism in a Main-Memory DBMS: The performance of PRISMA/DB. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, Vancouver, BC, Canada, 1995.

[Wid95] J. Widom. Research Problems in Data Warehousing. In *Proc. of the Int'l Conf. on Information and Knowledge Management*, pages 25–30, Baltimore, Maryland, November 1995.

[Wil91] A. Wilschut. *Parallel Query Execution in a Main-Memory Database System*. PhD thesis, Universiteit Twente, 1991.

[WK90] K. Whang and R. Krishnamurthy. Query Optimization in a Memory-Resident Domain Relational Calculus Database System. *ACM Trans. on Database Systems*, 15(1):67–95, March 1990.

[WO98] V. Wietrzyk and M. Orgun. VERSANT Architecture: Supporting High - Performance Object Databases. In *Proc. of the Int'l. Database Engineering and Applications Symposium (IDEAS)*, pages 141–149, Cardiff, Wales, UK, August 1998.

[WP00] F. Waas and A. Pellenkoft. Join Order Selection – Good Enough Is Easy. In *Proc. of the British National Conference on Databases*, pages 51–67, Exeter, UK, July 2000.

[WSK99] A. Windhouwer, A. Schmidt, and M. Kersten. Acoi: A System for Indexing Multimedia Objects. In *Int. Workshop on Information Integration and Web-based Applications & Services*, Yogyakarta, Indonesia, November 1999.

[WvZF+98] A. Wilschut, R. van Zwol, J. Flokstra, N. Brasa, and W. Quak. Road Collapse in Magnum. In *Proc. of the Int'l Symposium on Advances in Geographic Information Systems*, pages 20–27, Washington, DC, USA, November 1998.

[Yea96] K. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.

[Ze89] A. Zdonik and D. Maier (eds). *Readings in Object-Oriented Databases*. Morgan Kaufmann, San Mateo, CA, USA, 1989.

[ZLTI96] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance Analysis Using the MIPS R10000 Performance Counters. In *Proc. of the Supercomputing '96 Conf.*, Pittsburgh, PA, USA, November 1996.

Samenvatting

Monet: een nieuw database kernsysteem voor analysetoepassingen

In de laatste decennia is de moderne maatschappij steeds verder afhankelijk geworden van informatietechnologie. De hoeveelheid gegevens die wereldwijd wordt opgeslagen in databases groeit exponentieel. Deze toenemende beschikbaarheid van digitale gegevens heeft er, samen met het steeds krachtiger worden van computers, voor gezorgd dat databases op steeds andere manieren gebruikt worden voor een toenemend aantal doelen. Het primaire doel in de beginjaren van databases was vooral *transactieverwerking*: het bijhouden en opzoeken van gegevens (zoals bankrekeningen, adresbestanden, e.d.). Nu organisaties over de jaren heen steeds meer gegevens verzameld hebben, willen ze de groeiende rekenkracht van computers steeds vaker gebruiken om deze gegevens te *analyseren*, bij voorbeeld door het gebruik van *data mining* technieken.

In data mining probeert een computer programma automatisch interessante verbanden te vinden in historische databases (bijvoorbeeld om zo toekomstig klantgedrag te voorspellen, zoals: "wat zijn karakteristieken van verzekerden die een grote kans hebben om schade te veroorzaken met hun auto?"). Data mining werkt als een automatisch zoekproces: vergelijkbaar aan een computerschaakprogramma dat zoekt naar de beste zet, test een data mining programma duizenden hypothesen om zo de beste te vinden. Iedere test van een hypothese wordt beantwoord door een vraag te stellen aan het database systeem (zoals: "hoe is het aantal schaderijders verdeeld over geslacht?"). Omdat de resultaten van data mining altijd door een menselijke expert beoordeeld en gestuurd moeten worden, wordt er van dit soort toepassingen verwacht dat ze zo snel werken dat je er op kunt blijven wachten. Al met al leidt de toepassing data mining dus tot de eis aan een database systeem dat er duizenden vragen op grote historische databestanden beantwoord moeten worden in een zeer korte tijd (seconden of minuten).

Daarbij komt dat de gegevens die in databases worden opgeslagen steeds diverser worden, daar waar in het verleden alleen tabellen met administratieve gegevens werden opgeslagen, zijn dit tegenwoordig hele (XML) documenten, beelden, video's of bijvoorbeeld geografische gegevens, die vaak complex zijn van structuur. Zo bevatten documenten die op het internet te vinden zijn vele verwijzingen naar elkaar en zijn geografische gegevens opgeslagen in topologische netwerken of als kaart structuren.

Op dit moment is het gebruik van *relationele* database systemen de dominante manier om informatiesystemen te realiseren. Dit soort systemen worden als standaardpakket verkocht onder namen als Oracle, IBM DB2 en Microsoft SQLserver. Deze relationele database systemen deden opgang rond 1980. Al met al zijn er op een aantal terreinen veranderingen gaande die de geschiktheid van deze relationele database technologie voor hun toepassingen beïnvloeden:

toepassingen De toepassingen waarvoor databases gebruikt worden zijn sterk aan het veranderen (meer gegevens, complex van structuur en inhoud, waarbij de toepassing zich steeds sterker richt op – intensieve – gegevensanalyse). Data mining is hier een goed voorbeeld van. Daar komt bij dat de huidige generatie relationele database systemen niet ontworpen is voor zware analyses maar voornamelijk voor optimale verwerking van transacties, hetgeen in de tijd dat deze systemen opgang deden de voornaamste toepassing was.

data model Er worden regelmatig veranderingen voorgesteld in de manier waarop database systemen hun data zouden moeten opslaan (het *data model*) en ook de manier waarin je daar kan zoeken (de *query-taal*). Zo is er naast de relationele database systemen nu ook behoefte

aan object-georiënteerde, deductieve, actieve en bijvoorbeeld object-relationele database systemen.

hardware Computer hardware is continu aan verandering onderhevig. De wet van Moore[1] stelt dat hardware van dezelfde prijs iedere twee á drie jaar in kracht verdubbelt. Deze trend geldt voor bijna alle hardwarecomponenten maar gaat niet op voor de toegangstijden tot harde schijven en tot *RAM computergeheugens*, die veel langzamer verbeteren. Dit zorgt er voor dat computerprogramma's die vaak de harde schijf of het RAM geheugen benaderen, niet profiteren van de wet van Moore en dus relatief steeds inefficiënter worden.

Een andere trend is dat processoren (het "hart" van de computer), zoals bijvoorbeeld de Pentium 4, steeds sneller worden maar voor het daadwerkelijk in de buurt komen van hun potentiële kracht steeds afhankelijker zijn van karakteristieken van de programmacode zoals de voorspelbaarheid en onafhankelijkheid van opeenvolgende programma instructies.

Dit alles is relevant voor dit onderzoek omdat de huidige database systemen hun RAM geheugentoegang niet optimaliseren en voornamelijk bestaan uit onvoorspelbare en van elkaar afhankelijke programma instructies. Met andere woorden: de huidige generatie relationele database systemen gaat steeds minder efficiënt om met de computer hardware.

Vraagstelling

Het hier gepresenteerde onderzoek gaat over database architectuur, dat wil zeggen het complex aan ontwerp en implementatie beslissingen dat ten grondslag ligt aan een database systeem. Gezien de hierboven geschetste ontwikkelingen was het idee dat de dominante relationele database systemen wel eens sub-optimaal zouden kunnen zijn voor de huidige hardware en database toepassingen, die immers heel verschillend zijn dan die gangbaar waren rond 1980. Losjes gesteld was de vraag in hoeverre je verbeteringen in de architectuur van relationele databases zou kunnen realiseren als je met een schone lei zou beginnen.

Dit proefschrift concentreert zich op de volgende vragen over de architectuur van database systemen die gericht zijn op gegevens analyse (in tegenstelling tot transactieverwerking):

- *hoe kun je database systemen bouwen zodat ze op een optimale manier met moderne hardware omgaan en dus zo snel en zoveel gegevens als mogelijk is kunnen analyseren?*
- *hoe kan één database systeem geschikt worden gemaakt voor het opslaan en analyseren van gegevens die gemodelleerd zijn in verschillende modelleertalen en waarin gezocht wordt met verschillende query-talen?*
- *hoe de uitbreidbaarheid naar nieuwe toepassingsgebieden – zoals beeld, video, XML en geografische informatie systemen (GIS) – goed te ondersteunen?*

Aanpak

Database architectuur is een zeer breed veld zodat een eenduidig antwoord op deze vragen niet mogelijk is, alleen al omdat de vragen niet exact zijn (het is niet op voorhand bekend wat voor toepassingsgebieden en analyses van belang zijn, welke data modellen ondersteund moeten worden, etc.). Exacte antwoorden zijn in dit proefschrift alleen te vinden daar waar bepaalde vragen heel precies zijn ingekaderd (zoals bijvoorbeeld het hoofdstuk over het optimaliseren van RAM geheugentoegang in de verwerking van *join* queries). De methodologie toegepast in de rest van dit proefschrift is om aan de hand van een concreet prototype systeem, genaamd *Monet*, een weloverwogen mix aan architectuur principes uit te proberen en

[1] Gordon Moore is een van de oprichters van Intel, de grootste fabrikant van PC processors.

vervolgens het systeem te testen in realistische toepassingen of toepassingsscenario's om zo te kijken in hoeverre de gehanteerde architectuur principes werken. De architectuur principes toegepast in Monet zijn de volgende:

kern systeem Monet is alleen de "kern" van een database systeem. Deze kern kan omvat worden door verschillende besturingsmodules die specifieke data modellen en query-talen aanbieden. De uitdaging in Monet was om een coherent stelsel van basis datastructuren en queryprimitieven te vinden waarmee een zo groot mogelijke diversiteit aan datamodellen en query-talen efficiënt ondersteund kan worden.

koloms-gewijze verwerking Koloms-gewijze opslag van gegevens heeft voordelen voor vele analysetoepassingen, omdat die vaak een minderheid van alle kolommen in een tabel gebruiken maar wel veel rijen. Dit heeft tot gevolg dat minder gegevens vanaf de harde schijf naar het computergeheugen en vervolgens van het computergeheugen ook weer naar de processor getransporteerd hoeven te worden. De in dit proefschrift gedefinieerde *MIL* query-taal waarin koloms-gewijze gegevensbewerkingen gespecificeerd kunnen worden bestaat uit algebraïsche operatoren met steeds zeer weinig (een handvol) parameters. Doordat deze operatoren zo'n lage "vrijheidsgraad" (weinig parameters) hebben is het vervolgens mogelijk om optimalisaties in de programmacode van het database systeem door te voeren die leiden tot grote snelheidswinst op moderne processoren. Deze optimalisties zijn erop gericht om gegevens te verwerken met een minimum aan processorinstructies en door de uitvoering van deze instructies zeer voorspelbaar te maken.

cache optimalisatie Het ontwerp van Monet concentreert vooral op het bereiken van efficiënte gegevensverwerking in het RAM geheugen. Normaliter concentreren database systemen zich op de optimalisatie van toegang tot de harde schijf, omdat die wordt gezien als de belangrijkste snelheidsbelemmerende factor. Dit proefschrift stelt echter dat door het relatief steeds langzamer worden van RAM geheugen, dit RAM geheugen de tol van belangrijkste snelheidsbelemmerende factor over heeft genomen van harde schijf toegang. Het optimaliseren van RAM toegang in database systemen komt erop neer dat de algoritmen waarmee de cruciale, snelheidsbepalende query operatoren geïmplementeerd zijn optimaal gebruik moeten maken van de snelle maar kleine "cache" geheugens die tegenwoordig in de processorchips geïtegreerd zijn. Door zo goed als mogelijk van deze cache-geheugentjes gebruik te maken wordt de frequentie waarin de processor gegevens van en naar het RAM geheugen moet transporteren geminimaliseerd. Het mooie is dat door gebruik te maken van de faciliteiten die besturingssystemen (zoals Windows, Linux en Unix) bieden voor het manipuleren van *virtueel geheugen* deze RAM optimalisaties simpelweg een fijnmaziger versie zijn van de optimalisaties voor harde schijf toegang, zodat er twee vliegen in een klap geslagen kunnen worden met deze aanpak.

transacties gescheiden van queries In database systemen is transactie verwerking typisch dicht verweven met query-verwerking en het beheer van de harde schijf en RAM geheugen. Monet verwijdert transactiefunctionaliteit uit het "kern" systeem en brengt het onder in een optionele uitbreidingsmodule. Op deze manier worden databasetoepassingen die geen transactiefunctionaliteit nodig hebben niet vertraagd door transactieprotocollen.

Resultaten

Gedurende het onderzoek, beschreven in dit proefschrift, heeft de database onderzoeksgroep van het CWI het bedrijf Data Distilleries opgericht, dat data mining oplossingen verkoopt (tegenwoordig gespecialiseerd in toepassing van data mining op het terrein van de marketing) waarbij het Monet systeem onderdeel is van hun product. De auteur van dit proefschrift heeft vervolgens enige jaren dit onderzoek stopgezet om zich in te zetten voor dit bedrijf en de bedrijfsmatige toepassing van Monet, wat er toe heeft geleid dat uitgebreide praktijk ervaringen zijn opgedaan voor de toepassing van Monet in data mining omgevingen. Daarnaast wordt Monet gebruikt voor wetenschappelijk onderzoek aan het CWI, de Universiteit van Amsterdam, de Technische Universiteit Twente en Universiteit Utrecht. Verder is Monet getest op een aantal standaard *benchmarks* en is er een serie aan wetenschappelijke artikelen over Monet gepubliceerd.

De behaalde resultaten kunnen als volgt worden samengevat:

- de realisatie van het concrete Monet systeem, dat binnenkort algemeen en vrij ter beschikking zal worden gesteld.
- de formulering van MIL algebra taal, die efficiënte koloms-gewijze query-verwerking mogelijk maakt.
- er is aangetoond dat volledige relationele database functionaliteit (SQL) efficiënt ondersteund kan worden met MIL en meer dan dat, want de Monet kern is ook gebruikt voor het realiseren van object-georienteerde database systemen (ODMG, MOA).
- MIL toont aan dat transactie- en query-verwerking van elkaar gescheiden kunnen worden. De prijs die daarvoor betaald wordt is dat MIL niet strict transactie-veilig is. Dat is echter geen probleem als men beseft dat MIL niet direct door mensen gebruikt wordt maar door aansturende modulen (computer programma's) waarvan afgedwongen kan worden dat die zich aan bepaalde veiligheids regels houden.
- dit proefschrift formuleert nieuwe *radix-algoritmen* die het mogelijk maken om alle snelheidsbepalende query-verwerking op een cache-optimale manier te verrichten. Specifieke nieuwe algoritmen zijn de *radix-cluster* en *radix-decluster*, en deze zijn tot in detail onderzocht door modellen die hun gedrag voorspellen te formuleren en te testen. Deze zeer gedetailleerde modellen die gedrag tot op het niveau van cache-geheugen toegang beschrijven maar toch onafhankelijk zijn van specifieke hardware karakteristieken, zijn een nieuwe stap in de modellering van query-verwerking. Zulk soort modellering is van groot belang om de verwerking van queries te kunnen optimaliseren.
- de over het algemeen positieve ervaring bij de inzet van Monet voor data mining, beeld-databases, video-databases, geografische databases, en XML databases, toont aan dat de Monet kern een groot aantal toepassings gebieden kan ondersteunen.
- verschillende tests hebben uitgewezen dat Monet zeer snel is in het verwerken van data mining en andere zware analyse taken (OLAP) en daarin de tot nu toe gebruikelijke relationele database systemen de baas is.

Hoewel Monet beschouwd kan worden als een grotendeels succesvol project, zijn er nog zeer veel verbeteringen mogelijk, waaronder het uitbreiden van de faciliteiten voor parallelle verwerking, het optimaliseren van virtueel geheugengebruik voor harde schijf systemen met hoge bandbreedte ("RAID") en het introduceren van instantane query compilatie technieken met het oog op het nog verder optimaliseren van programmacode voor verwerking op moderne processoren.