UNIVERSITY OF AMSTERDAM

UvA-DARE (Digital Academic Repository)

Balancing vectorized query execution with bandwidth-optimized storage

Zukowski, M

Link to publication

Citation for published version (APA):
ukowski, M. (2009). Balancing vectorized query execution with bandwidth-optimized storage

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.

UvA-DARE is a service provided by the library of the University of Amsterdam (http://dare.uva.nl)

Download date: 01 2018

Chapter 3

Databases on modern hardware

Database management systems (DBMSs) provide application developers with a high-level abstraction of data management tasks. They expose generic interfaces that allow accessing and manipulating data, while at the same time providing features like concurrency control, transaction management, failure recovery and consistency checks. Additionally, they hide the hardware details of a used machine, helping applications to run on various platforms.

This chapter discusses the major characteristics of a DBMS, focusing on elements crucial for this thesis. In Section 3.1, we briefly describe the relational data model and relational algebra that are the fundamentals of most existing database engines. Section 3.2 describes the typical architecture of a DBMS, shortly discussing the most important components. Two of these components – the query executor and the storage manager – are of most interest for this thesis, and in this chapter we present different approaches of implementing them, both significantly influencing the research presented in the following chapters of this thesis. First, Section 3.3 discusses the most commonly used implementation approach, based on an *iterator execution model* working on top of an *N-ary storage model*. Then, Section 3.4 presents a completely different approach, found in the already existing MonetDB system, which is using a *fully-materialized algebra* based on the *decomposed (column) storage model*. Both architectures, while having benefits in some areas over the other one, do not make a full use of the possibilities of modern computer hardware. To improve this situation,

multiple optimization techniques have been proposed, as presented in Section 3.5

3.1 Relational model

Since late 1970s, the relational model is the most popular model in database systems. In this model, the data is stored as a set of *N-ary relations*, where each relation is a subset of a Cartesian product of *N domains*. A relation consists of a set of *tuples*, each containing *N attribute values*, one for each *attribute*. Typically, relations are visually represented as *tables*, where tuples are *rows* and attributes are *columns*, as presented in the left-most side of Figure 3.1. Still, the model itself does not impose any particular physical data representation. In particular, the relations by definition are *unordered*.

The operations on relations are defined in *relational algebra*, consisting of a number of *operators*. The basic operators include *projection* (π), *selection* (σ), *aggregation* ($\mathcal{G}$), *Cartesian product* ($\times$) and various types of *join* ($\bowtie$). For example, using relation *People* from Figure 3.1, to compute the age-bonus for all people older than 30, one could use the following relational query:

$$\pi_{Id,Name,Age,Bonus=(Age-30)*50}\ (\sigma_{Age>30}\ (People)) \tag{3.1}$$

Similarly as its underlying model, the algebra does not discuss how particular operations should be performed, but only what is the outcome of a given operator.

While the relational model is the most popular approach in the database world, other solutions exist. For example, object-oriented [Bar96], hierarchical [Bla98] or semi-structured [BGvK+06] databases are all being used in specialized data management tasks. In this thesis we focus on the relational databases and query processing in these systems, but some of the techniques can be applied within other paradigms.

3.1.1 Relational model implementation

When proposed, the relational model was an abstract mathematical concept, without an existing physical implementation. In the second part of 1970s and early 1980s, real-world realizations of this idea have been implemented, e.g. System R [CAB+81] and Ingres [SHWK76]. These systems introduced multiple concepts and designs that often can still be found in the existing relational databases.

Relation

Id	Name	Age
101	Alice	22
102	Ivan	37
104	Peggy	45
105	Victor	25
108	Eve	19
109	Walter	31
112	Trudy	27
113	Bob	29
114	Zoe	42
115	Charlie	35

NSM representation

Page 1

101	Alice	22	102
Ivan		37	104 Peggy
	45	105	Victor
25	108	Eve	19

Page 2

109	Walter	31	112
Trudy		27	113 Bob
	29	114	Zoe
42	115	Charlie	35

DSM representation

Id
101
102
104
105
108
109
112
113
114
115

Name
Alice
Ivan
Peggy
Victor
Eve
Walter
Trudy
Bob
Zoe
Charlie

Age
22
37
45
25
19
31
27
29
42
35

Figure 3.1: Relational table and its representation in the N-ary storage model (NSM) and the decomposed storage model (DSM)

3.1.1.1 Physical relation representation

The most common data representation in relational databases is to keep a relation as a collection of *rows*, each corresponding to a tuple. These rows are typically stored as *records* one after another in one *table* per relation, consisting of disk pages, each storing multiple records. This representation, known as the *N-ary storage model* (NSM), is presented in the central part of Figure 3.1. An alternative representation is the *decomposed storage model* (DSM [CK85]) presented in the right-most part of Figure 3.1. Here, every attribute is stored as a separate area on disk.

3.1.1.2 Query execution plans

To provide the functionality of the relational algebra in a physical world, databases commonly include a set of *physical* operators, roughly corresponding with their logical counterparts. Typically, it is not a one-to-one mapping, as the same logical operator can be implemented in various ways. For example, a logical *join* operator can be executed with a *merge-join* or a *hash-join*, depending on data properties, available resources etc. Various operators are combined into a *query execution plan* – a physical representation of a user query. A good overview of the implementation techniques for the physical query plans is presented in [Gra93].

Within a query plan, typically two execution methods are used [SKS02, Chapter 13.7]: pipelining and materialization. These two approaches are discussed in more detail in Sections 3.3 and 3.4.

```
SELECT   l_returnflag,
         l_linestatus,
         sum(l_quantity) AS sum_qty,
         sum(l_extendedprice) AS sum_base_price,
         sum(l_extendedprice * (1 - l_discount)) AS sum_disc_price,
         sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) AS sum_charge,
         avg(l_quantity) AS avg_qty,
         avg(l_extendedprice) AS avg_price,
         avg(l_discount) AS avg_disc,
         count(*) AS count_order
FROM     lineitem
WHERE    l_shipdate <= date '1998-09-02'
GROUP BY l_returnflag,
         l_linestatus
ORDER BY l_returnflag,
         l_linestatus;
```

Figure 3.2: TPC-H Query 1

3.1.1.3 Query language

DBMSs typically hide the *imperative* nature of the relational algebra by providing some high-level language that is converted into the actual query plan. The de-facto standard for the relational databases is *Structured Query Language* (SQL) [CB74, EKM+04] that expresses the queries in a declarative syntax close to the natural English language. For example, the relational query from Section 3.1 can be expressed with this SQL statement:

```
SELECT Id, Name, Age, (Age - 30) * 50 AS Bonus
FROM   People
WHERE  Age > 30
```

Figure 3.2 presents a more complicated SQL example: Query 1 from the TPC-H benchmark that is often used as an example throughout this thesis.

3.2 DBMS architecture

The components of a typical relational DBMS are presented in Figure 3.3. In fact, the architectures of state-of-the-art DBMS products are significantly more complex and often include dozens of cooperating modules. Still, generally, database consists of the following components:

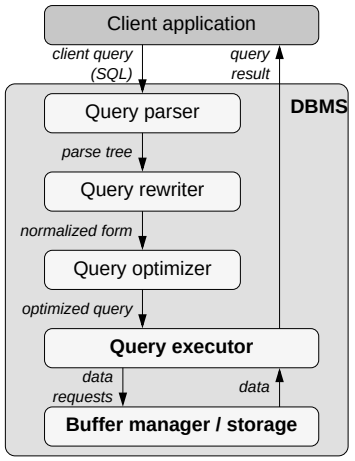

Figure 3.3: A simplified architecture of a DBMS

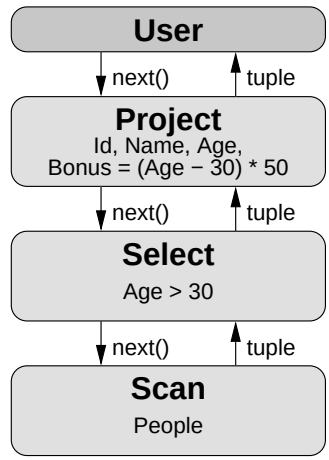

Figure 3.4: An operator tree for a simple SQL query in a tuple-at-a-time execution model

client application – before a query enters a DBMS, it needs to be provided by a client. A query is typically expressed in a high-level query language, e.g. SQL. A client can connect to a DBMS directly, using some DBMS-specific low-level communication protocols, or by exploiting a general-purpose high-level connection infrastructure, such as ODBC [Mic] or JDBC [Suna]. Furthermore, additional components can be used between the actual application and the DBMS, for example specialized utilities for load balancing or query result caching.

query parser – the syntax of the client query is analyzed, and a *parse tree* is built, providing internal representation of a query.

query rewriter – this component checks the parse tree for its semantic correctness (e.g. existence of used table names or proper access rights) and converts it into some *normalized form*. It is typically a tree of logical operations, often close to the relational algebra. This module may perform some additional tasks, for example expansion of user-defined views into the underlying queries.

query optimizer – the major task of this component is to rearrange the query tree in such a way that the expected execution time of the result query is

minimal. It also prepares the physical query plan, with the logical operations (e.g. a generic join) replaced with their physical counterparts (e.g. a hash-join). This module is usually highly complex, and has a tremendous impact on the total query execution time. For example, a wrong order of operations or a bad choice of an operator can result in a computational blow-up at some stage of processing. For such a bad plan, even the fastest query executor cannot process a given plan in satisfying time.

query executor – is the core component of query processing. It accepts a physical query plan, and performs all specified processing steps on data that it receives from the storage layer. The computed results are returned to the client.

buffer manager / storage – takes care of storing data on persistent media, accessing it and buffering it in memory. Typically, it also takes care of handling updates, managing transactions, performing disaster recovery, logging, locking, and more. However, these issues are not the focus of this thesis, and we only concentrate on data storage and access.

Of these components, two are of most importance for this thesis: query executor and storage layer. In the next two sections we discuss two approaches of implementing the query execution layer, based on two different principles: pipelining and materializing [SKS02, Chapter 13.7]. First, in Section 3.3 we analyze a typical query processor using a pipelined iterator interface and working on top of the N-ary tuple storage. Then, in Section 3.4 we discuss the architecture of MonetDB, concentrating on its fully-materialized in-memory query execution model and the use of column-based storage.

3.3 Tuple-at-a-time iterator model

Most database engines internally use the *iterator model* for their query execution layers [Gra94]. In this model, a query plan consists of a set of relational operators, connected in some topology. Typically, it is a tree, but operators can also compose a direct acyclic graph (e.g. in parallel execution plans) or even a graph with cycles [Waa02]. Operators communicate in a "pipeline" manner following the interface based on three major functions: *open()* initializes the operator and its children, *next()* makes operator return the next part of data to the caller, and finally *close()* finishes processing and frees the resources. Typically,

in the next() call, a single tuple is returned, using the NSM-based records. This "pull-based" model is known as *tuple-at-a-time* iterator model.

Figure 3.4 presents an example of an operator tree for the query from Section 3.1. The query execution proceeds as follows. First, the user (client application etc.) asks the top operator (Project) for the next result tuple. Project asks its child (Select) and it in turns asks its child (Scan). Scan retrieves the next tuple from the table, and sends it back to Select. Select checks if the tuple passes its predicate and, if so, sends it back to Project. If not, it asks Scan for the next tuple. Project, for each input tuple, computes an additional column and returns a new tuple to the user. When Scan determines that there are no more tuples in the underlying relation, it sends the end-of-stream identifier throughout the pipeline, which finishes the processing.

To better demonstrate what is happening within an operator, this is a pseudocode for the next() function in the Select operator:

```
Tuple Select::next() {
    while (true) {
        Tuple candidate = child->next();
        if (candidate == EndOfStream)
            return EndOfStream;
        if (condition->check(candidate))
            return candidate;
    }
}
```

3.3.1 Tuple-at-a-time model performance characteristics

The tuple-at-a-time approach is elegant, simple to understand, and relatively easy to implement. Performance-wise the most important characteristics of this model is that for every tuple there are multiple function calls performed. In our example, these include at least multiple next() calls, and calls to evaluate the condition in Select as well as to compute a new attribute value in Project. As a result, the state of each operator, as well as the code used by it, are accessed frequently. These properties result in a set of important performance drawbacks in a number of areas:

CPU instruction cache – if the query plan consists of many different types of operators, their combined instruction-memory footprint can be too large for the CPU I-cache to hold. Since the CPU changes its context between operators every tuple, if the I-cache is not large enough, cache-misses might occur every time a given part of code is accessed.

plan-data cache – each instance of a relational operator consumes some memory to keep its state, necessary for executing the *next()* call. With complex plans (even consisting of very few types of operators), or operators with large state, this data might not fit in the CPU D-cache, resulting in cache-misses.

function call overhead – the communication between operators, as well as many data operations are performed by calling appropriate functions or object methods. Per each operator iteration, multiple such calls are performed. Since a cost of performing a function call, in particular to a dynamically-dispatched (e.g. data-dependent) function, can be in range of tens of CPU cycles, especially when multiple parameters are passed, this overhead can be significant.

tuple manipulation – since tuples are organized as records of attributes, getting a particular value often requires extra steps to determine its position in the record. This record navigation is often repeated for each tuple.

superscalar CPUs utilization – as discussed in Section 2.1.4, modern CPUs have multiple execution units that allow performing multiple operations at the same time. Database engines, performing the same operations for a large number of tuples, seem naturally suited to exploit this feature. Unfortunately, with the tuple-at-a-time approach in each function call only a single operation on a single tuple is performed, not exposing enough work to keep multiple execution units busy. Also, heavy branching and multiple function cause frequent stalls in the pipeline. As a result, typical database code achieves very low instructions-per-cycle (IPC) performance [ADHW99].

compiler optimizations – many compile-time optimizations are impossible with the interpreted tuple-at-a-time approach. For example, due to the dynamic method dispatching, function inlining cannot be applied. Also, processing a single value at a time does not allow application of many performance-critical loop optimizations including loop unrolling, loop pipelining, strength-reduction and automatic SIMDization.

data volume – the commonly used N-ary tuple representation requires all table attributes to be stored in memory and transferred from disk. This might result in a waste of both memory and disk bandwidth, as well as the CPU cache, if a query does not use all attributes. Additionally, records

cumm. time (sec)	**excl. time (sec)**	**calls**	**avg. instr. / call**	avg. IPC	function name
11.9	11.9	846M	6	0.64	ut_fold_ulint_pair
20.4	8.5	0.15M	27K	0.71	ut_fold_binary
26.2	5.8	77M	37	0.85	memcpy
29.3	**3.1**	**23M**	**64**	**0.88**	**Item_sum_sum::update_field**
32.3	3.0	6M	247	0.83	row_search_for_mysql
35.2	**2.9**	**17M**	**79**	**0.70**	**Item_sum_avg::update_field**
37.8	2.6	108M	11	0.60	rec_get_bit_field_1
40.3	2.5	6M	213	0.61	row_sel_store_mysql_rec
42.7	2.4	48M	25	0.52	rec_get_nth_field
45.1	2.4	60	19M	0.69	ha_print_info
47.5	2.4	5.9M	195	1.08	end_update
49.6	2.1	11M	89	0.98	field_conv
51.6	2.0	5.9M	16	0.77	Field_float::val_real
53.4	1.8	5.9M	14	1.07	Item_field::val
54.9	1.5	42M	17	0.51	row_sel_field_store_in_mysql..
56.3	1.4	36M	18	0.76	buf_frame_align
57.6	**1.3**	**17M**	**38**	**0.80**	**Item_func_mul::val**
59.0	1.4	25M	25	0.62	pthread_mutex_unlock
60.2	1.2	206M	2	0.75	hash_get_nth_cell
61.4	1.2	25M	21	0.65	mutex_test_and_set
62.4	1.0	102M	4	0.62	rec_get_1byte_offs_flag
63.4	1.0	53M	9	0.58	rec_1_get_field_start_offs
64.3	0.9	42M	11	0.65	rec_get_nth_field_extern_bit
65.3	**1.0**	**11M**	**38**	**0.80**	**Item_func_minus::val**
65.8	**0.5**	**5.9M**	**38**	**0.80**	**Item_func_plus::val**

Table 3.1: MySQL gprof trace of TPC-H Q1: `+,-,*,SUM,AVG` takes <10%, low IPC of 0.7 (from [BZN05])

representing tuples typically include some meta-data, leading to suboptimal disk usage.

The above properties of the iterator model lead to two major inefficiencies in the traditional database performance. We demonstrate them with an experiment in which TPC-H Query 1 is executed on MySQL. This query scans a single relation consisting of a large number of tuples, performs some simple computations, and finally generates a few aggregate values. The query plan is very simple, and does not include any sophisticated operators such as joins or disk-spilling aggregations. In this situation, one could expect that most of the processing time is spent in data-manipulating functions. Table 3.1, presenting a detailed profiling of the benchmark, shows otherwise. Functions performing the actual operations

on data (in bold) consume less than 10% of total time. This demonstrates the first inefficiency – there is a lot of instructions related to query interpretation and tuple manipulation, causing a high *instructions-per-tuple* ratio. Additionally, the *instructions-per-cycle* factor is significantly lower than achievable on super-scalar processors. This is caused by the inability of the tuple-at-a-time algorithms to exploit multiple processing units, SIMD instructions and many of the crucial compiler optimizations. These two inefficiencies combined result in a very high *cycles-per-tuple* ratio which, even for simple operations, can reach hundreds or thousands of CPU cycles.

The tuple-at-a-time model also brings challenges in the areas of program profiling and optimization. This is caused by the fact that most of the CPU time is spread over a relatively large volume of code, including data processing functions, operator methods, tuple navigation etc. In this situation, it is hard to identify performance bottlenecks and hence introduce significant performance optimizations.

While the pipelined model often suffers in raw processing performance, it has a major benefit over the materializing approach discussed in the following section – scalability. Since in each next() call only a single tuple is passed, as long as there are no large intermediate results inside the query plan, the pipelined model can efficiently process arbitrarily large volumes of data. Maintaining this property is one of the crucial design goals of the new iterator model presented in Section 4.2.

3.4 Column-at-a-time execution in MonetDB

The MonetDB system [Bon02] was designed specifically for analytical data processing. In these scenarios, the query load typically consists of a relatively small number of queries, but the queries are complex and process large amounts of data. To achieve high performance in such scenarios, MonetDB proposed alternative solutions in various layers of the database system.

The crucial difference between MonetDB and traditional systems is in the way data is processed. Instead of using the N-ary tuple model, MonetDB follows the ideas presented in the *decomposition storage model* (DSM) [CK85] and uses an algebra entirely based on *Binary Association Tables* (BATs) [BK99]. This influences the storage layer, query language and the execution layer implementation.

In the storage layer, BATs are simply two-column tables, where *head* and *tail*

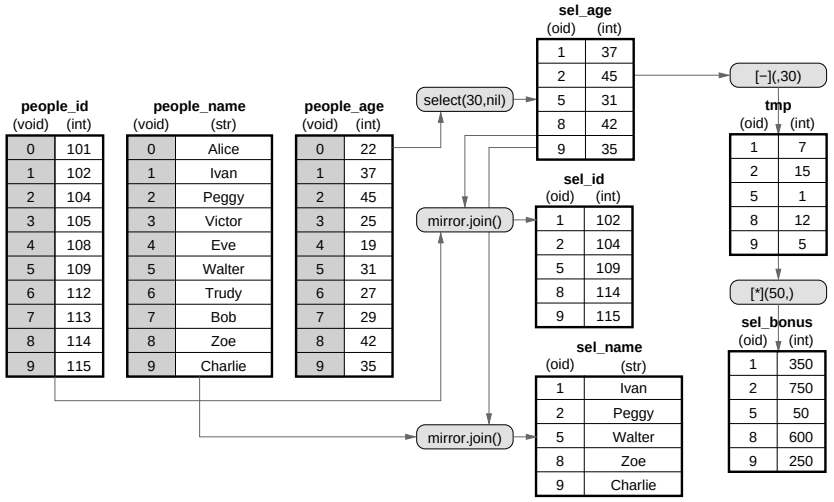

Figure 3.5: Execution of a simple SQL query (see Section 3.1.1.3) in the MonetDB column-at-a-time execution model

columns can contain different data types, as presented in the left-most side of Figure 3.5. A similar data organization has been proposed before in the context of database machines, specifically for vector processors [TKK+88]. Different attributes of the same tuple in an N-ary table are connected by using the value of object-id (*oid*) column, equivalent to the *surrogate* columns in [CK85]. For persistent data, this column typically contains a continuously increasing dense sequence of numbers, and is stored using a special *virtual-oid* (*void*) column type [BK99] that is not physically materialized. As a result, a BAT is often stored using a single column. For fixed-width data this format is equivalent to a simple data array. For variable-width types the storage is separated into two elements: a heap containing the actual data, and a fixed-width array of per-tuple positions in the heap.

The column-based approach in the storage layer has a significant impact on the I/O performance as well as the memory consumption. Since only columns that are actually used by a given query are fetched from disk, the volume of the transferred data becomes a fraction of what a system based on the N-ary storage would use. This is especially important with tables having a large number of columns, as is the case e.g. in data mining applications.

In the processing layer, MonetDB implements its binary algebra using the *column-at-a-time* approach: every operator is executed at once for all the tuples in the input columns, and its output is fully materialized as a set of columns. As a result, the query plan is not a pipeline of operators, but instead a series of sequentially executing statements, consuming and producing columns of data. For example, the execution of the example SQL query from Section 3.1.1.3 in Monet Interpreter Language (MIL) [BK99], is as follows:

```
sel_age   := people_age.select(30, nil);
sel_id    := sel_age.mirror().join(people_age);
sel_name  := sel_age.mirror().join(people_name);
tmp       := [-](sel_age, 30);
sel_bonus := [*](50, tmp);
```

The data flow for this query plan is presented in Figure 3.5. The resulting sel_* BATs constitute the final result. A more complex MIL example is presented in the left-most column of Figure 3.6, which shows the MIL code for the TPC-H Query 1.

The implementation of MonetDB operators is based on a principle of *no degree of freedom*. For every combination of task (e.g. select, sort), input data types (e.g. integer, string) and properties (e.g. sorted, nullable) a single specialized routine is created. Note that this approach would not be feasible in the N-ary model, as the number of possible combinations is too high, but it is maintainable in the binary model. When an operator is called, the version matching the input data types and properties is chosen and executed. Since the operator input is typically stored directly as arrays of values, and the entire input is processed at once, many operations boil down to simple loops over arrays. For example, a simplified code for a routine that selects from a `[void,int]` BAT identifiers of tuples bigger than a given constant and produces an `[oid,void]` result would look as follows:

```
int uselect_bt_void_int_bat_int_const(oid *output, int *input, int value, int size) {
    oid i;
    int j = 0;
    for (i = 0; i < size; i++)
        if (input[i] > value)
            output[j++] = i;
    return j;
}
```

Naturally, it is infeasible to manually implement and maintain all possible combinations of operators. MonetDB uses aggressive macro expansion using the Mx tool [KSvdBB96] that converts operator templates into dozens or even hundreds

MIL statement	SF=1				SF=0.001	
	data volume (MB)	result size (tuples)	time (ms)	band-width (MB/s)	time (usec)	band-width (MB/s)
s0 := select(l_shipdate).mark	45	5.9M	127	352	150	305
s1 := join(s0,l_returnflag)	68	5.9M	134	505	113	608
s2 := join(s0,l_linestatus)	68	5.9M	134	506	113	608
s3 := join(s0,l_extprice)	114	5.9M	235	483	129	887
s4 := join(s0,l_discount)	114	5.9M	233	488	130	881
s5 := join(s0,l_tax)	114	5.9M	232	489	127	901
s6 := join(s0,l_quantity)	68	5.9M	134	507	104	660
s7 := group(s1)	45	5.9M	290	155	324	141
s8 := group(s7,s2)	45	5.9M	329	136	368	124
s9 := unique(s8.mirror)	0	4	0	0	0	0
r0 := [+](1.0,s5)	91	5.9M	206	440	60	1527
r1 := [-](1.0,s4)	91	5.9M	210	432	51	1796
r2 := [*](s3,r1)	137	5.9M	274	498	83	1655
r3 := [*](s12,r0)	137	5.9M	274	499	84	1653
r4 := {sum}(r3,s8,s9)	45	4	165	271	121	378
r5 := {sum}(r2,s8,s9)	45	4	165	271	125	366
r6 := {sum}(s3,s8,s9)	45	4	163	275	128	357
r7 := {sum}(s4,s8,s9)	45	4	163	275	128	357
r8 := {sum}(s6,s8,s9)	22	4	144	151	107	214
r9 := {count}(s7,s8,s9)	22	4	112	196	145	157
TOTAL / average:	1361 (MB)		3724 (ms)	365 (MB/s)	2327 (usec)	584 (MB/s)

Figure 3.6: MIL code and performance profile of the TPC-H Query 1 (see Figure 3.2) running on MonetDB, scale factors 1 and 0.001 (from [BZN05])

of versions differing with input data types, properties etc. While this approach significantly increases the program size, it has highly useful properties influencing the execution performance:

instruction cache – Even though the overall code size is large, the cost of loading a given function is amortized over the entire input of an operator, making it negligible in most cases.

plan-data cache – Since only one operator is executed at any given time (within a single process), its state can make full use of the CPU cache. MonetDB was a platform used for some pioneering work in the area of

cache-conscious databases, and provides a number of cache-conscious algorithms [MBK02, MBNK04].

function call overhead – For most operators, there are no per-tuple function calls happening, making the function call overhead negligible.

tuple manipulation – The data is typically stored as contiguous set of tuples, equivalent to e.g. C arrays. As a result, value access is direct, and does not require any interpretation.

superscalar CPUs utilization – The code inside the MonetDB operators usually has no function calls, has significantly fewer branches, and as a result works much better on modern CPUs. However, expensive main memory accesses often hinder the performance.

compiler optimizations – MonetDB operators code is simple, and many automatic compiler optimization techniques can be applied.

data volume – Since MonetDB uses columnar data representation, only the used columns are being transferred from disk and stored in memory. Additionally, since data is packed in dense arrays, the overhead of record structure present in NSM is avoided, resulting in smaller storage requirements.

Thanks to the above properties, the MonetDB execution model reduces the need of the expensive query plan interpretation and makes the CPU spend most of the time on the actual data processing. With the CPU-efficient operator code, this allows to achieve low cycles-per-tuple cost for large-volume data processing.

The materializing operator model has also benefits in the area of extensibility, profiling and query optimization. Since operators are fully independent, and internally typically perform simple array-processing tasks, new operators can be easily added. Also, query execution time is clearly divided into a sequence of consecutively executing steps. This allows easy profiling and determining possible optimization areas. Finally, before executing each operator, there is more information available than in the tuple-at-a-time model (e.g. the exact relation cardinality), exposing multiple runtime optimization opportunities.

While MonetDB in many areas demonstrates a significant performance improvement over the traditional tuple-at-a-time strategy, it also suffers from a number of problems. The most important one is related to the intermediate result materialization. Even during in-memory processing, writing the results by

each operator can cause high memory traffic, making the operators not CPU-bound, but memory-bound. This can be observed e.g. by comparing the TPC-H Query 1 results for scale factors 1 (1GB) and 0.001 (1MB) presented in Figure 3.6 [1]. For SF=0.001 the data is small enough to fit the intermediate results in the CPU cache, and as a result, the per-operator bandwidth can be even 3 times higher than the memory-bound SF=1 case, and the overall execution is almost two times faster (2327 usec on a 1MB dataset versus 3724 ms on a 1GB dataset). The impact of the result materialization is even more visible on multi-CPU machines, where the memory bandwidth is shared among different CPUs [Zuk02]. Avoiding this problem is one of the crucial goals of the execution model introduced in Section 4.2.

The overhead of the intermediate result materialization is additionally increased in MonetDB by its use of a column-at-a-time algebra. With this approach, tasks involving *multiple* columns often become complicated. For example, an aggregation with multi-attribute keys needs to be decomposed into a number of steps. Similar problems occur in multi-attribute joins or sorting. Binary algebra also enforces *attribute post-projection* [MBNK04] – after an operation, all attributes "carried" through it need to be materialized, as is the case with the `id` and `name` columns in Figure 3.5. In all these cases, additional computational steps causes extra data materialization. These MonetDB problems suggest that, while the columnar storage is good for reducing the I/O cost and for allowing high processing performance, query plans should be expressed using the N-ary approach.

3.4.1 Breaking the column-at-a-time model

The high memory-bandwidth problem of MonetDB has been analyzed in [Zuk02] that focused on the in-memory execution on SMP machines. In such scenarios, the computational benefits of parallelizing queries are often seriously hindered by the relatively poor per-CPU memory bandwidth. In contrast, since cache memories are often dedicated to each CPU, the cost of accessing cache (assuming no cache coherency protocol overheads) does not increase with multiple CPUs.

The imbalance between the main-memory and CPU-cache bandwidth, especially visible in the multi-CPU environments, has led to the idea of *partitioned execution* [Zuk02, Section 4.3.3]. Here, instead of executing a sequence of column-at-a-time statements, columns are broken into smaller *slices*, and the operators execute on them in the pipelined fashion. With the slice size chosen

[1] 2005 results from [BZN05]

such that the intermediate results fit in the CPU cache, in-memory materialization is avoided. As a result, performance is significantly improved, especially during parallel execution. In single-CPU execution, the performance benefits were relatively small, mostly due to the fact that the interpretation mechanisms were implemented in the MIL language, which is relatively inefficient for scripting. With the slice size typically in range of a few hundreds or a few thousands of tuples, the high overhead of MIL interpretation could not be well amortized.

While the performance improvement of partitioned execution was relatively limited, this research has hinted that combining the high performance of column-at-a-time operations with the pipelined execution strategy can both improve the already high performance of MonetDB, as well as allow it to reduce its memory-consumption related problems. This gave a motivation for the research presented in the following chapters of this thesis.

3.5 Architecture-conscious database research

In the previous sections we have discussed two existing execution models that are the base for the research presented in this thesis. This section broadens the picture by presenting the research from the area of *architecture-conscious database systems*. In this field, the performance of database systems on modern hardware is analyzed and improved, both in terms of the modifications to the discussed execution models, as well as new implementations of various data processing tasks.

3.5.1 Analyzing database performance on modern hardware

The detailed analysis of CPU performance on database workloads was first performed by the computer-architecture community [MDO94, CB94, BGB98, LBE+98, KPH+98]. In [MDO94] authors compared a few types of multi-user commercial workloads, including TPC-A [Tra94] and TPC-C [Tra07] benchmarks, simulating OLTP scenarios, with a set of numeric-intensive applications, mostly from the area of scientific computing. One of the observations was that the transaction-processing systems typically use significantly larger instruction footprint, with most instructions executing a relatively small number of times. This is related to the fact that these systems typically do not spend a lot of time in tight loops, as is the case in e.g. numeric processing. This directly impacts the

L1 I-cache performance – the percentage of I-cache misses for TPC-A is a few times higher than for scientific applications. Interestingly, for the L1 D-cache, the results were opposite – multi-user applications suffered from a significantly lower number of misses. For the L2 misses, the situation is slightly different – transaction workloads not only suffer from significant number of I-cache misses, but also D-cache misses are on a par with numeric workloads. Finally, it has been observed that the multi-user workloads, due to their event-based nature, spend a significant amount of time in the kernel space – 40% for TPC-A versus 7% for used non-database workload.

Similar experiments have been presented in [CB94], where the authors additionally analyze the performance of the sort operation, and provide more insight into the exploitation of the superscalar nature of the Alpha AXP CPUs. This paper demonstrates that already in 1994 transaction-processing workloads resulted in significantly higher cycles-per-instruction (CPI), and made inefficient use of the multi-pipeline architecture of the CPUs. Also, the impact of branch mispredictions has been demonstrated to be higher than in most numeric-intensive problems. As a result, it has been shown that transaction-processing programs can spend as little as 20% on actual computation, wasting rest of the time on various stalls, comparing to 30% in sort, and 80% in the Linpack benchmark. These problems have also been identified in [KPH+98], where authors confirm OLTP problems with utilizing out-of-order execution, superscalar issues and branch prediction.

A comparable analysis of the OLTP and decision-support systems (DSS) workloads has been presented in [BGB98, LBE+98, ADHW99]. In [BGB98], the authors demonstrate that the CPI of DSS scenarios is significantly better (factor 4) than in OLTP. Also, DSS systems have better code and data locality, resulting in significantly fewer cache misses. These results are confirmed in [LBE+98], where authors present higher OLTP instructions footprint, and hence an increased number of I-cache misses. Similar conclusions are drawn in [ADHW99], where the authors identify L2 data-cache misses and L1 instruction-cache misses as crucial for performance.

Interestingly, recent analysis of the large-scale OLTP experiments [SK06] presents slightly different conclusions. On the Itanium platform, with the used transaction load, the instruction-cache stalls only contributed to less than 10% of the total time. On the other hand, data-cache misses, particularly expensive L3 misses, consumed ca.60% of total time. These results show that the exact characteristics of the performance highly depends on the used hardware platform, system architecture and query loads.

While this collection of papers is not exhaustive, it demonstrates that the

performance of database systems is far from optimal on modern hardware, especially comparing to numeric-intensive scientific programs. As a result, there is an ongoing activity in the database research community to improve the database architecture by addressing the most important performance problems.

3.5.2 Improving data-cache

Perhaps the most visible inefficiency of classical database algorithms is related to the suboptimal use of the hierarchical memory systems. The impact of the non-uniform access cost has been identified quite early with a pioneering work of Shatdal et al [SKN94]. While in 1994 the difference of the cache-access and memory-access (2-4 cycles versus 15-25 cycles) was an order of magnitude smaller than now, even then cache-conscious algorithms allowed up to 200% performance improvement. Authors proposed a set of techniques that frequently reoccur in the following research, and include: *blocking* – reuses chunks of data that fit in the cache; *partitioning* – a variant of blocking, divides data into cache-sized segments; *key extraction* – reduce the volume of processed data by using only attributes relevant to the current operation; *loop fusion* – combine multiple operations in a single pass over data to reduce memory traffic; *clustering* – rearrange attributes to improve spatial data locality. An early example of using some of these ideas has been presented in the AlphaSort [NBC+95] algorithm, which minimized the number of cache misses. It was achieved by using cache-sized data units with cache-friendly QuickSort [Hoa61], processing {key-prefix, pointer} pairs instead of full records to allow more elements to fit in the cache, and using a cache-friendly replacement-selection tree for merging the runs.

Improving the performance by cache-conscious data reorganization has been proposed in [CHL99], where the authors rearrange elements of pointer-based data structures to increase locality of references. Rao and Ross address similar problem focusing on tree structures [RR99, RR00]. First, in [RR99], they suggest organizing a tree in a read-only array, eliminating the need of storing data pointers. Internal nodes are chosen to fit the cache-line size, optimizing the number of cache-line references, and highly-optimized in-node search routines are used for faster lookup. This work is extended to update-enabled *cache-sensitive* B^+ *Trees* (CSB^+-Trees) [RR00], which modify the B^+-Tree organization such that all the children of a given node are stored contiguously. This reduces the volume of data and hence the number of cache misses.

Vertical data representation in DSM [CK85], has been identified by Boncz et al. [BMK99] to have beneficial impact on cache behavior of applications, due to reduced memory traffic and an increased spatial locality. While this research

has been performed in scope of column-stores, Ailamaki et al. [ADHS01] took this idea further, by proposing a hybrid data storage model, called PAX. In this model, the layout of a disk page is modified to store the same attribute from different tuples contiguously, like in DSM. PAX has the I/O characteristics of NSM, and cache-characteristics of DSM, and has been shown to improve query performance by as much as 48%. The *data-morphing* technique [HP03] identifies the in-memory performance differences between DSM and NSM and generalizes the PAX approach by dynamically dividing a relation into a set of vertical partitions, stored in a PAX-like, mini-page based manner. In all these solutions the model used for storage and for processing is the same. Clotho [SSS+04] decouples these two issues by transparently choosing the storage model most suited for the current workload.

Another approach to optimize cache-memory behavior has been proposed in [ZR03b], where the accesses to nodes in tree-based data structures are buffered, and tree-traversal operations are performed in bulk. This increases the *temporal locality*, as the same cache-lines are used multiple times in one operation. As a result, the throughput of the operations on both cache-conscious and traditional tree-structures is significantly improved, at a cost of an increased response time of single operations.

The early approaches on using the in-memory partitioned hash-join [SKN94] has been extended in work of Boncz et al. [BMK99, MBK00, MBK02], This research found the high-fanout partitioning necessary to split large data volumes into cache-sized sub-relations to result in poor cache and TLB behavior. The proposed multi-pass *radix-cluster* partitioning strategy eliminates this problem, and while performing more work, results in a better overall performance. This work also proposes optimizations to the partitioning code, as well as highly accurate cost-models for choosing the best partitioning method and predicting its performance. This work is further extended in [MBNK04], where authors introduce a cache-friendly *radix-decluster* attribute post-projection algorithm.

Another approach to address the imbalance between the cache and memory latency is to use *data prefetching*. As discussed in Section 2.2.3, modern CPUs provide both implicit and explicit cache prefetching capabilities. Explicit prefetching has been suggested for the general problem of recursive, pointer-chasing, data-structures [LM96, LM99]. In the database community, a series of papers by Chen et al. [CGM01, CGMV02, CAGM04, CAGM05, CAGM07] discusses how these techniques can be applied to database operations. [CGM01] presents how prefetching can be exploited to increase the width of the nodes in a B^+-trees without paying the latency cost of fetching the additional cache-lines, resulting in a reduced tree-depth and hence *lookup* performance boost

of up to 55%. Additionally, the authors discuss how prefetching can improve *range-scanning* performance, where the performance gains can be as high as a factor 8.7. This technique has been further extended to disk-resident *fractal prefetching* $B^+ - Trees$ that use cache-conscious in-page tree organization and processing for good performance, as well as I/O prefetching for improved range scans. [CAGM04, CAGM07] discuss how *group prefetching* and *software-pipelined prefetching* techniques can be applied when performing hash-join operation on a set of tuples. These methods were shown to perform better than cache-partitioning and also be more resistant to interference by other programs. Memory prefetching has also been applied to optimize various data accesses in the *inspector join* algorithm [CAGM05].

It is interesting to see that most of the techniques proposed for improving memory performance by using cache memory have a direct correspondence with similar techniques for improving disk performance by using main memory: multi-pass partitioning is necessary e.g. for external hash-join with a high number of partitions; disk prefetching is commonly applied for scans and index lookups; reducing I/O volume with vertical storage is the major beneficial feature of column stores; and so on. This shows that the problems addressed by the research presented in this section are relatively generic, and only particular hardware constants slightly differ. Since the memory hierarchy continuously becomes more complex, it is possible that in the future similar approaches might need to be applied at different levels of it. This observation gave birth to the area of *cache-oblivious algorithms* [FLPR99, HL07], where the exact characteristics of the hardware are ignored, and algorithms and data structures are designed to maximize the spatial and temporal locality, and hence work well on any cache configurations.

3.5.3 Improving instruction-cache

Instruction cache misses have been identified as a major problem for database performance, especially for OLTP workloads [BGB98, LBE+98, ADHW99]. The main reason for it is poor temporal locality of the accessed instructions – most database systems change position in code very frequently, especially with tuple-at-a-time processing.

Harizopoulos and Ailamaki [HA04] observed that while at a given moment different concurrently running transactions typically use different parts of the program, their overall code paths overlap significantly. To exploit this, they suggest organizing multiple queries needing a particular operator into *execution teams*, and evaluate this operator for all members of the team one after another.

This makes the queries share the instruction-cache, which significantly reduces the number of misses, up to 96.7%. Additionally, better temporal locality of the executed code increases the efficiency of branch predictor, allowing for reduction of mispredicted branches by up to 64%. In the result, this technique has been shown to provide an overall transaction-processing speedup in a live system of up to 31%.

Zhou and Ross [ZR04] observed that the instruction-cache misses are also a significant problem in analytical processing queries, even within a single query. This happens when the instruction footprint of the entire query plan exceeds the CPU I-cache size. To reduce this problem [ZR04] introduced a new *Buffer* operator that saves the tuples coming from the child, and sends then to the parent once the number of tuples reaches a given threshold. This effectively decomposes a query plan into a set of partially-materializing sub-plans, and if the buffering happens in the proper locations in the plan, each of the sub-plans is small enough to fit in the I-cache. As a result, this method can reduce the I-misses by 80% and improve the overall performance by 15%. This approach is especially useful in OLAP queries, where the same operation is performed for large number of tuples in one query.

3.5.4 Exploiting superscalar CPUs

Database systems have not only been identified to work sub-efficiently with cache memories, but also have been shown to poorly utilize the superscalar capabilities of modern CPUs [CB94, ADHW99]. This is especially visible in the *cycles-per-instruction (CPI)* ratio, which for databases is typically in range of 1.2 1.8 for TPC-D benchmark and 2.5 to 4.5 for TPC-C benchmark [ADHW99]. For comparison, modern CPUs can issue a few instructions per cycle, and optimized programs can achieve a CPI of 0.5 or less. The high CPI of database loads can be partially explained by instruction and data stalls, but other factors influence it as well. One of them is the highly branching nature of the traditional tuple-at-a-time model: for every tuple multiple comparisons (e.g. check for data types or computation errors) and various function calls need to be performed. Also, since there is only a single unit of computation at a time (a tuple), there are relatively little *independent* instructions that can be executed in parallel.

Improving the performance by reducing the number of comparisons was proposed in [Aga96], where authors propose the comparison-free *radix-sort* in place of commonly applied, comparison-based *quick-sort* and *merge-sort* algorithms. In [Ros02], Ross analyzed the impact of branch mispredictions when evaluating selection conditions, and presented techniques to replace control-hazards result-

ing from conditional branches with data-hazards that have smaller impact on performance. The reduction of the number of function calls and branches is also one of the effects of the block-oriented processing [PMAJ01].

Another related functionality of modern CPUs are the SIMD instructions. In the database community, there have been relatively little research in this area. In [ZR02], Zhou and Ross apply SIMD instructions to a subset of database operations. The results show that with proper implementation, this method can give speedup close to the number of elements SIMD instructions process, or even higher, if SIMD instructions allow to reduce the number of branches. In [Ros07] SIMD instructions are used to optimize the processing of the hash tables. SIMD potential can also be exploited to improve parallel predicate evaluation, as presented in [JRSS08]. Finally, SIMD-processing is used heavily on many alternative hardware platforms, discussed in Section 3.5.6.

3.5.5 Intra-CPU parallelism

For a long time parallel execution in databases has been exploiting *inter-node parallelism*, using different machines, and *intra-node parallelism*, using multiple CPUs in one machine. Recently, with the increased popularity of the SMT and CMP architectures, *intra-CPU parallelism* becomes an important research area.

The first analysis of the SMT potential for database system was presented in [LBE+98], where the SMT performance was modeled using database logs. The experiments showed that with SMT significantly more instructions are issued every cycle, and memory-access latencies can be tolerated to some extent. As a result, the performance on 8-context SMT processor improved 3-fold for OLTP and 1.5-fold for DSS. [ZCRS05] introduces two SMT-specific methods of executing relational operators. In the *bi-threaded* implementation, the work of an operator is distributed evenly among concurrently executing threads. In the *working-set* version, the "helper thread" knows what data will be needed by the "main thread" and preloads it in advance. Both strategies have been evaluated in row-wise and column-wise record layouts, and shown up to ca. 20% improvement over a standard, SMT-oblivious parallel algorithm.

The potential of CMP architectures for databases and the impact of various CPU-architectural choices have been presented in [HPJ+07]. The authors discussed two types of CMP designs: complex wide-issue, out-of-order, deep-pipelined *fat-camp* (FC) chips, and simple in-order, multi-threaded, simple-pipelined *lean-camp* (LC) chips. The results show that looking at the query response times LC can be worse by up to 70% in DSS loads and up to 12% in OLTP loads. Still, in multi-query saturated scenarios, LC chips can achieve

up to 70% higher overall system throughput. This paper also analyzes how the growing L2 cache sizes increase the L1-hit times, and as a result can have detrimental effect on the overall system performance.

The technical challenges of exploiting CMP machines have been investigated in [CRG07] and [CR07]. In [CRG07] authors propose a parallel buffer structure that enables sharing input and output between the concurrently working operators, providing load balancing at the same time. This work is continued in [CR07], where various parallel aggregation algorithms are proposed for the UltraSPARC T1 chip. The results show that the new architectures require new algorithm designs that make good use of shared L2 cache and exploit available atomic instructions to reduce the need of locking.

3.5.6 Alternative hardware platforms

After the unsuccessful attempts of creating database-specialized hardware in the field of *database machines* (e.g. [AvdBF+92]), research in the high-performance database processing has been focused on exploiting the traditional disk–memory–CPU data flow path and CPU-based processing. However, recent technology developments result in computer architectures where the computational capabilities of a machine are divided between more than one type of devices, resulting in research targeting completely new hardware platforms.

The most visible example of shifting processing to an alternative hardware are *graphics processing units* (GPUs, see Section 2.1.7.3). In [GLW+04] Govindaraju et al. proposed GPU-based implementations of a set of database operations, including predicate evaluation and aggregations. The used GPU processing model is significantly different from traditional CPUs, as it depends heavily on SIMD instructions and cannot efficiently perform random memory accesses. This makes the implementation of various database operations challenging. Still, the presented performance improvement of 2–4 times suggests a huge computational potential of GPUs, as demonstrated in further research on sorting [GGKM06] and joins [HYF+07]. GPUTeraSort [GGKM06] is a GPU-based sorting algorithm that provided the top sorting performance in the PennySort benchmark [NBC+95]. The comparison of CPU-based versus GPU-based join algorithm presented in [HYF+07] shows that also for this operation GPUs can provide order of 2-20 improvement. Combining these standalone techniques into a coherent GPU-based system has been suggested in the GPUQP system [FHL+07]. The rapid improvements in the GPU technology, in terms of both performance as well as programming flexibility (e.g. with the recent NVIDIA

CUDA project [NVI08]), suggests that this approach might become increasingly important in the future of high-performance database systems.

Other alternative hardware architectures have also been used as platforms for database processing. In [GAHF05] a set of relational operators have been evaluated on an Intel IXP2400 network processor with 8 simple low-frequency microengines, each supporting 8 thread contexts, and using an explicitly controlled memory architecture. While exploiting this architecture introduces various implementation challenges, the results show that the high on-chip parallelism allows from 1.9 to over 3 times improvement over a general-purpose CPU. A similar hardware architecture is present in the STI Cell chip [IBM07] that uses 1 general-purpose PPU core and 8 SIMD-specialized SPU cores with a different ISA and no cache. Full utilization of these specialized cores often requires careful algorithm redesign. This has been presented in [Ros07], where Ross demonstrated how a different hash structure results in a 10 times performance improvement. Similarly, in [GBY07] and [GYB07] Gedik et al. presented that properly engineered algorithms on Cell can provide the performance improvement of a factor 4 for sorting (using 16 SPUs) and a factor 8.3 for stream joins, comparing to a dual Intel Xeon machine.

Another hardware area in which database technology is being used are embedded devices. For example, in [BBPV00] the authors analyze the challenges of implementing a database on a smartcard. These devices have significantly different hardware characteristics: slow writes, very little memory etc. As a result, the proposed PicoDBMS system needs to introduce new ideas in the areas of data organization (optimizing for data compactness) and query execution (processing with limited RAM).

3.5.7 Analyzing and improving database I/O performance

While most databases follow the traditional NSM data storage format, there has been a substantial amount of research investigating other solutions. For example, NSM is known to have poor performance for scan-intensive applications, where only a subset of relation attributes is accessed. To improve this, the *decomposition storage model* (DSM) has been proposed [CK85], where all attributes are stored in separate files, and only the used attributes are scanned. This format is also known as *column storage*. Originally, DSM introduced the idea of an extra *surrogate column*, containing key values used to identify tuples in different files. Modern column stores do not require such a column, and use the natural order (possibly involving compression) to reconstruct tuples [SAB+05, ZR03a, BZN05]. However, DSM introduces a need to access multi-

ple disk locations if a single tuple needs to be accessed, which can be a significant problem for sparse lookups and updates – this problem is absent in NSM.

The analysis of NSM and DSM presented in [HLAM06] has shown that DSM can provide a performance advantage for queries reading only a relatively small number of attributes (up to ca. 30% with settings used in the paper), and NSM is better for queries reading a large fraction of the table attributes. These results are somewhat surprising, as, using large enough I/O units (achieved in [HLAM06] with prefetching), the I/O cost of the operations should be proportional to the width of the scanned attributes, making DSM beneficial for all scans accessing a subset of columns. The reason for the low tuple width percentage at which NSM becomes beneficial in [HLAM06] is that both storage models are executing as a data source for an N-ary execution model. For the DSM model, this results in the tuple-reconstruction phase consuming extra CPU time.

The different characteristics of DSM and NSM resulted in a substantial number of approaches that combine different features of these two models. The PAX storage model [ADHS01] and the data morphing technique [HP03] discuss the in-memory properties and both models, and propose hybrid solutions (see Section 3.5.2). The *fractured-mirrors* approach [RDS02] suggests keeping two copies of data on two disks, like in RAID-1 [PGK88], but storing one copy in NSM and another in DSM. This allows using the best model depending on the task, as well as combining both mirrors in a single query for even better performance. The *multi-resolution block storage model* (MBSM) [ZR03a] investigates the physical placement of the DSM-based data. In this approach, the good scan performance of DSM is preserved, while the cost of tuple reconstruction is reduced, since values of different attributes from the same tuple are stored closer on disk than in the naive DSM implementation.

A series of papers discusses the "five-minute rule" [GP87, GG97, Gra07], which says that if an item is accessed roughly every five minutes, it should be main-memory resident for good cost efficiency. Still, the page size at which this break-even interval is applicable continuously increases (1KB in 1987, 8KB in 1997, 64KB in 2007). This trend is caused by the imbalance between the improvements in disk bandwidth and latency.

Another research direction in the I/O optimization area is related to the increasing popularity of solid-state storage devices (see Section 2.3.3). Two major features of these systems are typically analyzed: low random-read latency, and the performance difference between slow general updates and fast 1-to-0 updates. Graefe [Gra07] not only revisits the RAM-disk five-minute rule, but also discusses how low flash latency introduces similar rules for RAM-flash and

flash-disk transfers. Shah et al [SHWG08] investigate a new join algorithm that depends more than traditional approaches on random accesses to disk, efficiently supported by flash. Ross [Ros08] presents a number of algorithms tuned for "write-once-then-erase" nature of flash device. This work is related to the previous research on write-only storage systems [Mai82, RS82], but differs due to the "erase" functionality of flash.

While MEMS devices (see Section 2.3.4) are not yet publicly available, they have been already investigated for database applications. In [YAA03] authors exploit the two-dimensional nature of these devices, and propose a storage model mapping two-dimensional relational tables on these. In related work, Schlosser et al. [SSAG03] demonstrate how the inherent MEMS parallelism can be exploited to provide efficient execution of both row-oriented and column-oriented requests.

3.6 Conclusions

This chapter presented an overview of the database technology, concentrating on the differences between the traditional tuple-at-a-time and the MonetDB column-at-a-time processing model. Both architectures provide a unique set of benefits and problems. Additionally, a set of hardware-related database optimization techniques has been discussed. Combining the best elements from all these areas is the goal of the new approach to the query execution, proposed in the next chapter.