Chapter 11: Emerging “vertical” database systems in support of scientific data

Per Svensson[1], Peter Boncz[2], Milena Ivanova[2], Martin Kersten[2], Niels Nes[2]

[1]Swedish Defence Research Agency, Stockholm, Sweden
[2]Centrum Wiskunde & Informatica (CWI), The National Research Institute for Mathematics and Computer Science, Netherlands

Abstract. This chapter surveys and discusses the evolution of a certain class of database architectures, more recently referred to as “vertical databases”. The topics discussed in this chapter include the evolution of storage structures from the 1970’s till now, data compression techniques, and query processing techniques for single- and multi-variable queries in vertical databases. Next, the chapter covers in detail the architecture and design considerations of a particular (open source) vertical database system, called MonetDB. This is followed by an example of using MonetDB for the SkyServer data, and the query processing improvements it offers.

1. Introduction

Author: Per Svensson

1.1 Basic concepts

Consider a High-Energy Physics experiment, where elementary particles are accelerated to nearly the speed of light and made to collide. These collisions generate a large number of additional particles. For each collision, called an "event", about 1-10 MB of raw data are collected. The rate of these collisions is about 10 per second, corresponding to 100’s of million or a few billion events per year. Such events are also generated by large-scale simulations. After the raw data are collected they undergo a "reconstruction" phase, where each event is analyzed to determine the particles it produced and to extract hundreds of summary properties (such as the total energy of the event, momentum, and number of particles of each type).

To illustrate the concept of vertical *vs.* horizontal organization of data, consider a dataset of a billion events, each having 200 properties, with values labeled V0,1, V0,2, *etc*. Conceptually, the entire collection of summary data can be represented as a table with a billion rows and 200 columns as shown in Figure 11.1(a). A horizontal organization of the table simply means that the physical layout of the data is row-wise, one row following its predecessor, as shown in Figure 11.1(b). Usually the entire table is stored into disk pages or files, each containing multiple rows. A vertical organization means that the layout of the data is column-wise as shown in Figure 11.1(c). Note that the entire column containing a billion values is usually stored in multiple disk pages or multiple files.

Suppose that a user wishes to get the event ID’s that have energy, E, greater than 10 MeV (million electron-volts) and that have number of pions, N_p, between 100 and 200, where “pion” is a specific type of particle. This predicate can be written as: $((E > 10) \wedge (100 < Np < 200))$. It is obvious that in this case searching over the vertically organized data is likely to be faster, since only the data in the two columns for E and N_p have to be brought into memory and searched. In contrast, the horizontal organization will require reading the entire table. Given this simple observation, why were relational database systems typically built with a horizontal organization? As will be discussed next, the majority of database systems were designed for transaction processing where frequent updates of randomly requested rows were expected, which is the reason for choosing the horizontal organization. In this chapter we discuss the class of applications that benefit greatly from a vertical organization, which includes most scientific data applications.

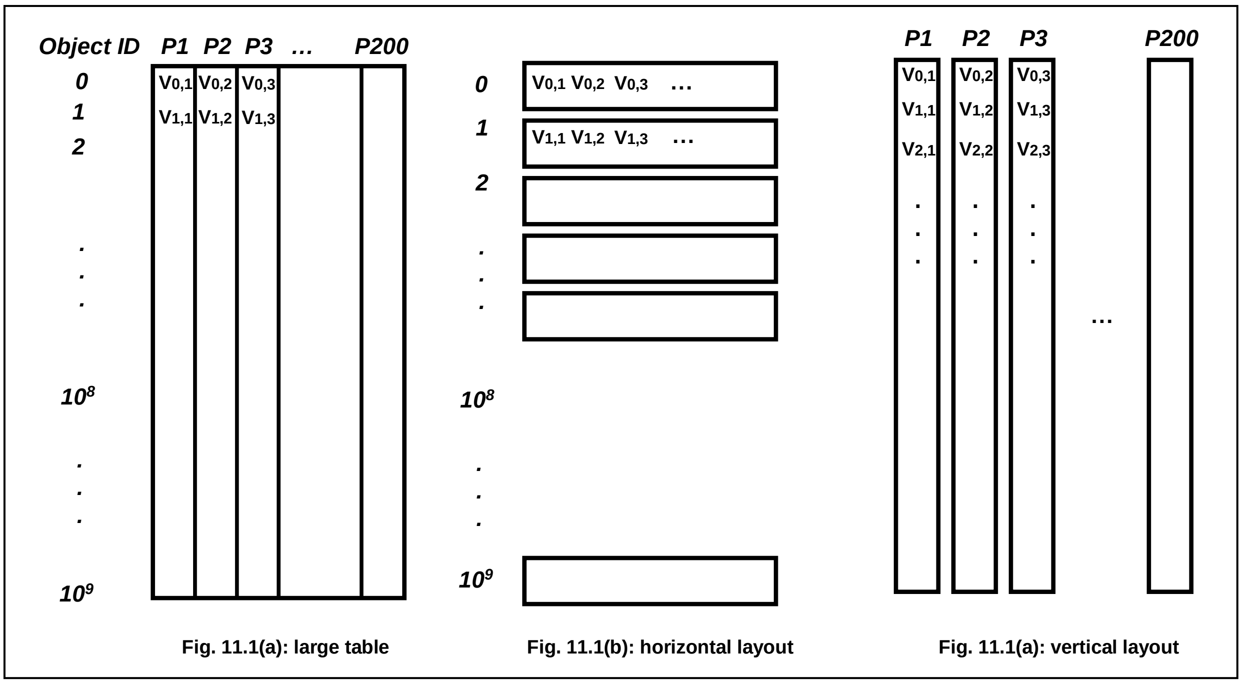

Figure 11.1 Horizontal vs. vertical organization of tabular (relational) data

1.2 *Design rules and user needs*

A recent contribution to the literature on database design for what its authors call *complex analytics* is [MF04], describing the design rationale for *Sybase IQ Multiplex*, a parallel, multi-node, shared-storage vertical database system [Syb08] whose major design goal is to efficiently manage large-scale data warehousing workloads. It is argued in this paper that the consequences of adopting the primary design criterion for transactional, write-oriented databases, is “to minimize the portion of stored data that must be locked for exclusive access and the length of time that locks are held”. Thus, according to [MF04], these consequences have generally led to the following set of design rules for transactional, write-oriented databases:

- Since data are usually accessed and modified one record at a time, data should be stored row-wise to allow each record to be updated by a single write operation. Also, data should be stored in small disk pages to minimize the amount of data transferred between memory and disk and to minimize the part of the disk file that needs to be locked during a transaction.
- Indexes should be restricted to a few attributes to avoid locking entire index tree structures on disk and thereby deny access to whole sets of rows, which might otherwise become necessary when indexes are updated.
- Compression of data is usually not profitable because there is often a mix of different data types and unrelated data values in each row. The CPU time required for compression and decompression will therefore not be recovered by reduced data transfer volume.
- Adding or deleting attributes and indexes is likely to be expensive since all, or a large part of, the data pages used by the parent table may be affected.
- Finally, updates of an attribute according to even a simple predicate are likely to be costly because the entire row must be read and written when a single attribute is to be updated.

However, once the primary criterion for the internal design of a database system becomes the achievement of high performance of complex analytics tasks, this set of rules should be changed as follows:

- Since by storing data column-wise instead of row-wise, it is possible to avoid touching those disk pages of a table which are not at all affected by a query, considerable performance improvements may be achieved. Cache efficiency will be enhanced since commonly accessed columns will tend to stay in the cache.

- Data are likely to be read many more times than they are written or updated, making CPU time "investment" in the creation of efficient storage structures more likely to be profitable. Also, data should be stored in large pages so that a large number of relevant data items can be retrieved in a single read operation, resulting in a high overall "hit ratio". Row-wise storage, on the other hand, tends to disfavor large page sizes since each read operation drags into memory also attributes which are not relevant to the query in question, resulting in a low overall hit ratio.
- By using version management instead of record-wise locking techniques (which becomes possible largely because of the typically much smaller number of concurrent updates), each query would see a consistent database state in which no locks, or very few locks, ever occur. Also, version management is what complex analytics users need to be able to keep track of their many different analysis paths across the database.
- It is possible (although probably often not necessary) to index every attribute since searches greatly dominate over updates and because adding an index to an attribute requires only that attribute to be read, not the entire row.
- Data compression is likely to be profitable because the data belonging to one attribute is highly likely to be homogeneous and even auto-correlated.
- Adding or deleting attributes of a table is likely to be cheap since only relevant data would be accessed. Updates of an attribute are likely to be relatively cheap because no irrelevant attribute values need to be read or written.

Similar observations were made much earlier [Sve79a], but were for a long time considered irrelevant in mainstream database research. In fact, these alternative design principles have only comparatively recently received serious attention. This renewed interest is at least partly due to the fact that today, many very large data bases are actually used for data warehousing, decision support, or business or security intelligence applications, areas where similar characteristics apply as those claimed above. In [Sve79a], the following additional observations were made:

- In scientific data analysis, the number of simultaneous users is typically much smaller than in large-scale commercial applications, but on the other hand the users tend to put more complex, usually unanticipated, queries to the system.
- A more automatic response to complex user requests is required from the system components, since in scientific applications no systems or database specialists are usually available.
- A system should be transportable to many computer models, including medium sized computers, since in scientific data analysis applications many users prefer to work with an in-house computer dedicated to the acquisition and analysis of data.

One of the first systems to be developed based on the above principles was the system *Cantor* [KS83, KS86, Sve88, AS88]. The Cantor project pioneered the analysis and coordinated application of many of the above-mentioned techniques and concepts in relational systems.

1.3 Architectural opportunities

Today, there is a growing interest in what has been called *read-optimized* database systems [SAB+05, HLA06], *i.e.*, systems that are oriented towards *ad hoc* querying of large amounts of data that require little or no updating. Data warehouses represent one class of read-optimized systems, in which bulk loads of new data are periodically carried out, followed by a relatively long period of read-only querying. Early interest in this class of systems came from various statistical and scientific applications, such as epidemiological, pharmacological, and other data analytical studies in medicine [WFW75], as well as intelligence analysis applications [BSS97]. *Transposed files*, as vertical storage schemes were usually called at the time, were used in a number of early non-relational read-optimized database systems. A fairly comprehensive list of such systems was given in the paper [CK85], which asserted that the standard tabular scheme for storage of relations is not necessarily the best, and that transposed files can offer many advantages.

In the field of database technology during the 70's and early 80's, there was little consensus on how to perform experiments, or even on what to measure while performing them. Today's experiments and analyses are usually far better planned and executed, and the accumulated scientific knowledge in database technology is vastly greater. There is now very good evidence that vertical database systems can offer substantial performance advantages, in particular when used in those statistical and analytical kinds of applications for which the concept was originally developed, *cf.* Section 5, below.

An important conceptual step associated with the use of transposed files is that a whole range of new architectural opportunities opens. Below is a partial list of such architectural opportunities. We note, however, that only a subset of these techniques have been widely used in systems currently available:

- *column-wise storage* of data, in place of the conventional row-wise data storage layout used in most relational database management systems (RDBMS), can eliminate unnecessary data access if only a subset of the columns are involved in the query
- *clustering*, in particular *sort ordering*, of attribute values, can speed up search over column data
- various kinds of "light-weight" data compression: *minimum byte size, dictionary encoding, differencing* of attribute value sequences, can reduce the amount of data accessed from disk to memory
- *run-length encoding* (*RLE*) data compression for columns that are ordered, can reduce the amount of data fetched from disk into main memory
- *dynamically optimized sequential combinations* of different compression techniques can reduce processing time
- *B-tree* variants or other indexing techniques *designed to efficiently store and retrieve variable-length data* in columns, a requirement for profitable exploitation of many data compression techniques
- conjunctive search, join, and set algebra *algorithms exploiting the column-wise storage structure and working directly on compressed data*
- *lazy decompression* of data, *i.e.*, data are decompressed only as needed instead of as soon as having been brought into main memory, is required if such algorithms are to be used
- *compressed lists* of tuple id's to represent intermediate and final results in such algorithms
- *vectorized operations* on data streams and the *vectorized dataflow network architecture* paradigm, to reduce call overhead costs and allow efficient query evaluation by interpretation of algebraic expressions rather than by compilation to low-level code
- *specially designed buffering techniques* for storing and accessing metadata and results of simple-transaction-type queries, which are in general not well suited to column storage schemes.

Next, we will discuss several of these approaches, with an emphasis on those techniques which have been claimed in the literature to be of particular importance in high-performance systems.

2. Architectural principles

Author: Per Svensson

The architectural principles discussed in this section were proposed by several groups who have designed different vertical databases over the years. Bringing them together in this way does not mean that these principles can be arbitrarily combined with each other. However, they form a collection of ideas one should probably be aware of when designing or acquiring such systems.

The literature review presented next, shows that most of the advantages of vertical storage in databases for analytical purposes have been known and exploited since the early 80's at least, but recently there is renewed wide-spread market and research interest in the matter. The lack of interest in the past now seems to reverse into what might be construed as a canonical vertical storage architecture, replacing the previous consensus that the "flat file with indexes" approach is always preferable.

2.1 Transposed files and the decomposed storage model

A number of early papers deal with issues related to how to *group*, *cluster*, or *partition* the attributes of a database table. For example, the authors of [NCW+84] state: "Partitioning in database design is the process of assigning a logical object (relation) from the logical schema of the database to several physical objects (files) in a stored database. *Vertical partitioning* subdivides attributes into groups and assigns each group to a physical object." That paper, however, was not concerned with such analytical applications in which *ad hoc* queries are dominant. Instead, it discusses how tables may be partitioned in order to exploit known correlations between attribute hit rates, to obtain better average query performance. This is worthwhile mainly in routinely repeating processes where access patterns which display such correlations dominate and change slowly.

The term [*fully*] *transposed file* was used in early papers, such as [THC79, Bat79, Sve79a, Sve79b], to denote what is today called "vertically fragmented" or "vertically decomposed" data structures [MBK00a], "vertical partitioning" [HLA+06], "column-oriented" data bases [AMF06] or "column store" data bases [SAB+05].

[CK85] is the first published paper on transposed files and related structures that is widely referenced in recent database literature. While the authors of [CK85] note that some early database systems used a fully transposed storage model, for example, RM [LS71], TOD [WFW75], RAPID [THC79], ALDS [BT81], Delta [SKM+84] and [Tan83], in that paper the advantages of a *fully decomposed storage model* (DSM) are described. A DSM is a *"[fully] transposed storage model with surrogates*[1] *included"*. In a DSM each column of a relational table is stored in a separate *binary association table* (BAT), as an array of fixed-size two-field records (TID, value), where TID refers to Tuple ID. According to [KCJ+87], the DSM further assumes that two copies of each binary relation are stored, one clustered (*i.e.*, sorted or hashed with an index) on each of the two attributes (TID, value). The authors of [CK85] conclude that there seems to be a general consensus among the database community that the conventional N-ary Storage Model (NSM) is better. They suggest that the consensus opinion is not well founded and that neither is clearly better until a closer analysis is made.

In [KCJ+87], a parallel query processing strategy for the DSM is presented, called the *pivot algorithm*. The algorithm makes use of the *join index* concept [Val87]. An informal description of a generic select-join-project query is given in the paper, where all selects are assumed to be range restriction operations and all joins are equi-joins. The initial *select phase* executes a select operation for every predicate binding in the query, using the appropriate value-clustered BAT as index. The output of this phase is a collection of temporary index lists, each containing the TIDs of selected tuples from conceptual relation tables. All these operations are done in parallel. During the *pivot phase* of the algorithm, the main *m*-way join operation of the query is executed. A "pivot" TID column is chosen from those attributes which appear in the join expression. The result of this phase is another collection of temporary index lists indicating which tuples in each conceptual relation that satisfy the query. Since a join index clustered on the desired TID exists for all entity-based equi-joins, a full scan can always be avoided. During the *value materialization phase* several independent joins are evaluated, preferably in parallel. The join operands are small binary relations containing only TIDs. The final *composition phase* executes an *m*-way merge join which permits a large degree of parallelism. Its operands are all small binary relations containing only TID lists whose cardinality has been maximally reduced due to the select operations.

The practical conclusions from this work, reported in [VKC86] and cited in [KCJ+87], is (1) that DSM with join indexes provides better retrieval performance than NSM when the number of retrieved

[1] The term *surrogate* is used by some researchers to denote object identifiers, or as here in relational databases, *tuple identifiers* or *TIDs*. We will use the latter term unless we make a direct quotation.

attributes is low or the number of retrieved records is medium to high, while NSM provides better retrieval performance when the number of retrieved attributes is high and the number of retrieved records is low, and (2) that the performance of single attribute modification is the same for both DSM and NSM, while NSM provides better record insert/delete performance.

This is an approach which is similar to those used in MonetDB [MBK00a, MBK00b] and in Cantor [Sve82], with the following main differences: (1) DSM provides two predefined join indices for each attribute, one clustered on each of the two attributes (attribute value, TID), while Cantor and MonetDB both use indices which are created as needed during query evaluation; (2) Cantor stores these indices using Run-Length-Encoding (RLE) compression; MonetDB introduces a novel radix cluster algorithm for hash join; (3) although potentially important, parallelism has not been presented as a key design issue for MonetDB, nor was it one for Cantor; (4) the algorithms used in MonetDB and Cantor were both presented as simple two-way joins, corresponding mainly to the composition phase in the DSM algorithm, which is presented as an m-way join.

2.2 The impact of modern processor architectures

Research has shown that DBMS performance may be strongly affected by “cache misses” [ADH+99] and can be much improved by use of *cache-conscious* data structures, including column-wise storage layouts such as DSM and within-page vertical partitioning techniques [ADH+01]. In [ADH+99] this observation is summarized as follows: “Due to the sophisticated techniques used for hiding I/O latency and the complexity of modern database applications, DBMSs are becoming compute and memory bound”. In [BK99], it is noted that past research on main-memory databases has shown that main-memory execution needs different optimization criteria than those used in I/O-dominated systems.

On the other hand, it was a common goal early on for scientific database management systems (SDBMS) development projects to exploit the superior CPU power of computers used for scientific applications, which in the early 70’s could be orders of magnitude higher than those of processors designed for commercial workloads. The main purpose was to make query evaluation compute and memory bound rather than I/O bound whenever possible.

The MonetDB developers [MBK00a, MBK00b, BZN05] have conducted thorough analyses of the effect of modern computer hardware architectures on data base performance. As advances in CPU speed far outpace advances in dynamic random access (DRAM) latency, the effect of optimal use of the memory caches is becoming ever more important. In [MBK00a] a detailed discussion is presented of the impact of modern computer architectures, in particular with respect to their use of multi-level cache memories to alleviate the continually widening gap between DRAM and CPU speeds that has been a characteristic for computer hardware evolution since the late 70’s. Memory access speed has stayed almost constant (within a factor of 2), while CPU speed has increased by almost a factor of 1000 from 1979 to 1999. Cache memories, which have been introduced on several levels to reduce memory latency, can do so effectively only when the requested data are found in the cache.

The authors of [MBK00a] claim that it is no longer appropriate to think of the main memory of a computer system as “random access” memory, and show that accessing data sequentially also in main memory may provide significant performance advantages. They furthermore show that, unless special care is taken, a database server running even a simple sequential scan on a table may spend 95% of its cycles waiting for memory to be accessed. This memory access bottleneck is even more difficult to avoid in more complex database operations such as sorting, aggregation, and join, which exhibit a random access pattern. The performance advantages of exploiting sequential data access patterns during query processing have thus become progressively more significant as faster processor hardware has become available.

Based on results from a detailed analytical cost model, the authors of [MBK00a] discuss the consequences of this bottleneck for data structures and algorithms to be used in database systems and identify vertical fragmentation as the storage layout that leads to optimal memory cache usage.

A key tool whose utilization the MonetDB developers pioneered in database performance research is the use of detailed access cost models based on input from hardware event counters which are available in modern CPUs. Use of such models has enabled them, among other things, to identify a significant bottleneck in the implementation of the partitioned hash-join and hence to improve it using *perfect hashing*. Another contribution is their creation of a *calibration tool* which allows relevant performance characteristics (cache sizes, cache line sizes, cache miss latencies) of the cache memory system to be extracted from the operating system for use in cost models, in order to predict the performance of, and to automatically tune, memory-conscious query processing algorithms on any standard processor.

It is the experience of the MonetDB developers that virtual-memory advice on modern operating systems can be effectively utilized in a way which makes a *single-level storage software architecture* approach feasible. Thus, the MonetDB database software architecture does not feature secondary storage structures or explicit I/O operations, whereas the underlying "physical" storage architecture is multi-level hierarchical, consisting of CPU registers on the lowest level, two levels of hardware cache memory (L1 and L2), main memory, and virtual swap memory on disk.

A conclusion of these studies is that database algorithms and data structures should be designed and optimized for efficient multi-level memory access from the outset. Careless implementation of the key algorithms can lead to a performance disaster that even faster CPUs will not be able to rescue, whereas careful design can lead to an order of magnitude performance improvement. The authors claim, on very good analytical and experimental grounds, that the vertical decomposition storage feature is in fact the basis of achieving such high performance.

One final achievement of these design studies and subsequent implementation improvements is a data mining benchmark result which is two orders of magnitude better than that of some commercial database products.

2.3 Vectorization and the data-flow execution model

The use of vector operators (vectorization) in interpretive query evaluation aims at distributing the (usually heavy) interpretation overhead over many elementary CPU operations. It is the same basic idea that is exploited in a vectorized CPU, although used in this context primarily to improve the efficiency of a software process. It turns out, however, that it is not straightforward to devise an efficient vectorized query evaluation process. This is a topic discussed at some length by [BZN05]. It would seem that the MonetDB developers are more or less alone in taking advantage of this approach today.

In [KS86], a vectorized interpretation technique for query evaluation was presented, claimed to be analogous to the operation of a vectorized dataflow computer [Gil83]. This architectural approach was chosen to make sense of the transposed file architecture when extended to support general relational queries from the basic access patterns and conjunctive queries previously studied in [Sve79a, Sve79b] and briefly discussed in Section 2.6*, "Querying compressed, fully transposed files"*. It has in fact several key features in common with those of MonetDB/X100 described briefly in Section 4.4, below. [KS86] surveys the methods which were developed for translating the parsed and syntactically optimized expression of a relational query into an execution plan in the form of one or more hierarchies of static dataflow networks. Each network in the execution plan is a *bipartite graph, i.e.,* if in such a network two nodes are connected by an arc, then one is a data buffer node and the other an operator node.

Network generation is followed by an execution phase which proceeds in two stages: (1) when initializing a network hierarchy for evaluation, space is assigned to its buffers from a buffer pool common to all networks; and (2) when evaluating a network, initially all but the upstream boundary buffer nodes are empty. Evaluation proceeds by executing operator nodes in some order until all downstream boundary buffer nodes contain a value (usually a vector value). An operator node may execute whenever none of its inbuffer nodes is empty, and none of its outbuffer nodes is full. In the last system version of Cantor (1991), 31 different vectorized, so-called *stream operators* were available to the dataflow network generator. They are software analogues of the machine instructions of a vectorized dataflow computer. On modern "multi-core" computers as well as on shared-nothing multi-node computer systems, Cantor's vectorized dataflow query evaluation process could quite easily be parallelized.

The authors of [BZN05] argue that database systems usually execute less than one instruction per cycle (IPC), while in scientific computation, such as matrix multiplication, or in multimedia processing, IPCs of 2 or more are not uncommon on modern CPU's. The authors claim that database systems do not need to perform so badly relative to scientific computing workloads. Based on experimental results they conclude that there are interpretation techniques which, if exploited, would allow DBMS compute performance to approach that of scientific computing workloads. A key technique by which this may be achieved is *loop pipelining*, whereby interpretation overhead is distributed over many elementary operations. This technique is central to the vectorized prototype query processor X100, recently designed and evaluated by the MonetDB developers. According to [BZN05], its goal is to:

1. execute high-volume queries at high CPU efficiency,
2. be extensible to other application domains like data mining and multi-media retrieval,
3. scale with the size of the lowest storage hierarchy (disk).

To achieve these goals, X100 must manage bottlenecks throughout the computer architecture:

Disk. The columnBM I/O subsystem of X100 is geared towards efficient sequential data access. To reduce bandwidth requirements, it uses a vertical storage layout that in some cases is enhanced with lightweight data compression.

RAM. Like I/O, RAM access is carried out through explicit memory-to-cache routines which contain platform-specific optimizations. The same vertically partitioned and even compressed disk data layout is used in RAM to save space and bandwidth.

Cache. A Volcano-like [Gra94] execution pipeline with a vectorized processing model is used. Small vertical chunks (*e.g.*, 1000 values) of cache-resident data items, called "vectors", are the unit of operation for X100 execution primitives. The CPU cache is the only place where bandwidth does not matter, and therefore (de)compression happens on the boundary between RAM and cache.

CPU. Vectorized primitives expose to the compiler that processing a tuple is independent of the previous and next tuples. Vectorized primitives for projections (expression calculation) do this easily, but [BZN05] try to achieve the same for other query processing operators as well (*e.g*., aggregation). This allows compilers to produce efficient loop-pipelined code.

2.4 Data compression

There are two obvious ways a DBMS can trade CPU cycles to save disk space and thereby I/O bandwidth, a precious resource [MBK00a]. In this context, we are concerned with I/O bandwidth only, since disk space itself has recently become so cheap that its cost rarely matters at all except perhaps in extreme applications, such as large-scale web search [CDG+06]. First, data may be coded into a more compact form [SAB+05]. For example, if one is storing an attribute that is a US customer's state of residence, the state can be coded directly into six bits, whereas the standard two-character abbreviation requires 16 bits and a variable length character string for the full name of the state requires many more. Second, data values may be packed compactly by storing N values in K*N bits, where K is the smallest byte size that can hold any value in the column (*bit packing*). Of course, more sophisticated schemes may be used which can save even more space while allowing for flexible updates with data items which require more than K bits. It is also possible to use additional techniques, in particular sort order, to save additional I/O bandwidth. Note that these simple data compression techniques are equally applicable to row-wise as to column-wise storage schemes. Therefore, when a query is processed using a column-wise storage scheme, what makes the basic difference with respect to I/O bandwidth is the fact that there is no need to transfer data from irrelevant columns into main memory. As we will see, however, additional, more sophisticated compression techniques may also be exploited for column-wise storage.

Although the authors of [AMF06] state that "it was not until the 90s when researchers began to concentrate on how compression affects database performance", the two early papers [Sve79b, EOS81] both emphasize in different ways that fast data access may be achieved through judicious use of data compression in fully transposed (*aka* vertically partitioned) files. Early tutorials on the subject are [Sev83, Bas85, RV93], however, none of them specifically discuss how to achieve query evaluation speed improvements. Recent papers on the use of data compression in relational databases are [WKH+00, AMF06, RS06]. The most significant advantages are typically obtained when combining data compression with fully transposed file or DSM storage, but there are also proposals to use compression in row-oriented storage schemes.

The paper [AMF06] discusses an extension of the column-store storage and access subsystem of the C-Store system [SAB+05], while also addressing the issue of querying compressed data. The extended column storage exploits the fact that "sorted data is usually quite compressible" and suggests storing columns in multiple sort orders to maximize query performance, rather than to minimize storage space. The authors propose an architecture that allows for direct operation on compressed data while minimizing the complexity of adding new compression algorithms. They state: "Compression in traditional databases is known to improve performance significantly. It reduces the size of the data and improves I/O performance by reducing seek times (the data are stored nearer to each other), reducing transfer times (there is less data to transfer), and increasing buffer hit rate (a larger fraction of the DBMS fits in the buffer pool). For queries that are I/O limited, the CPU overhead of decompression is often compensated for by the I/O improvements."

Compression techniques for row-wise stored data often employ *dictionary schemes* to code attribute values in fewer bits. Sometimes Huffman encoding is used whereby varying symbol frequencies may be exploited to gain a more compact total encoding, at the price of having to use varying length codes. Also, in [AMF06], the idea of *frame of reference* encoding (*FOR*) is considered, where values are expressed as small differences from some "frame of reference" value, such as the minimum, in a block of data. Run length encoding (RLE), where repeats of the same element are expressed as (value, run-length) pairs, is presented as an attractive approach for compressing sorted data in a column-wise store.

An important point made by [AMF06] is that if one wants to exploit different compression techniques depending on local properties of data, it is important to find ways to avoid an associated increase in code complexity, where each combination of compression types used to represent the arguments of a join operation would otherwise require its own piece of code. The authors give several examples showing how, by using compressed blocks as an intermediate representation of data, operators can operate directly on compressed data whenever possible, degenerating to a lazy decompression scheme when not possible. Also, by abstracting general properties of compression techniques and letting operators check these properties, operator code may be shielded from having to know details of the way data are encoded.

The paper [HLA+06] discusses techniques for performance improvements in read-optimized data bases. The authors report how they have studied performance effects of compressing data using three commonly used "light-weight" compression techniques: dictionary, bit packing, and *FOR-delta*, the latter a variation of FOR where the value stored is the difference of a value from the previous one, instead of from the same base value.

In [RS06] data base compression techniques are discussed from a more general perspective, pointing out that certain statistical properties of data in a DBMS, in particular skew, correlation, and lack of tuple order, may be used to achieve additional compression. They present a new compression method based on a mix of column and tuple coding, while employing Huffman coding, lexicographical sorting, and delta coding. The paper provides a deeper performance analysis of its methods than has been customary, making it an important contribution also in this context although it does not specifically focus on column-wise storage. Finally, it briefly discusses how to perform certain operations, specifically *Index Scan, Hash Join, Group By with Aggregation,* and *Sort Merge Join,* directly on compressed data.

In the MonetDB system described in [MBK00a], two space optimizations have been applied that reduce the per-tuple memory in BATs:

Virtual TIDs. Generally, when decomposing a relational table, MonetDB avoids allocating the 4-byte field for the TID since it can be inferred from the data sequence itself. MonetDB's (and DSM's) approach leaves the possibility open for non-virtual TIDs as well, a feature which may be useful, *e.g.*, when performing a hash-lookup.

Byte encodings. Database columns often have low domain cardinality. For such columns, MonetDB uses fixed-size encodings in 1- or 2-byte integer values.

In [KS86], the approach to data compression used in Cantor and its test-bed is described. It presupposes the existence of an efficient way to organize attribute subfiles containing varying length data. The so-called *b-list* structure developed for this purpose is a B-tree variant suitable for storing linear lists which exploits an observation made in [Knu73], and also shows similarity to a structure discussed in [MS77].

When data are read from or written into a b-list node, data values are accessed one block at a time. When a block is written, a compression algorithm is first applied to its data, working as follows: First, from all values in a given sequence, its minimum value is subtracted and stored in a sequence header (*cf.* the FOR technique discussed above). Then, to store an arbitrary-length subsequence of integers compactly, four alternatives are considered:

1. use the smallest possible common byte length for the subsequence and store the byte length in the header (bit packing)
2. use the smallest possible common byte length for the difference subsequence and store the first element of the original subsequence as well as the byte length in the header (FOR-delta)
3. if the sequence is a *run* of equal values, store the value and length of the subsequence in the subsequence header (RLE)
4. if the subsequence is a run of equal differences, store the first element of the subsequence, the common difference, and the subsequence length in the header (delta-RLE)

To combine these alternatives optimally in order to store a given data sequence (of length n) as compactly as possible, a dynamic programming, branch-and-bound algorithm was developed. This algorithm subdivides the sequence into subsequences, each characterized by *storage alternative*, *byte size*, *cardinality*, and *size of header*, so as to represent the entire sequence using as few bits as possible, given the above-mentioned constraints. The algorithm solved this problem in time O(n).

For reading and writing data in b-lists, six access procedures are available, allowing sequential as well as direct-addressed read and write access of data segments. One of the sequential read procedures does not unpack run-compressed data sequences, enabling fast access to such data. This facility is used by the conjunctive query search algorithm as well as by the merge-join algorithm.

2.5 Buffering techniques for accessing metadata

In vertical databases, metadata usually play an important role, not only to support users with information about database contents, but also internally to support parameter and algorithm selection during runtime. In order to exploit the advantages of vertical segmentation to achieve the best possible runtime performance, algorithms for searching and accessing data need to frequently consult the metadatabase (MDB) for information about the current status of database contents. For example, to find the fastest way to search a run-length compressed, fully transposed, ordered file (CFTOF search, see Section 2.6 *"Querying compressed, fully transposed files"*), it is important to be able to quickly access at runtime certain properties of the attributes involved in the query, in particular their sort key position, as well as their cardinality and value range. The latter information is needed to obtain good estimates of the selectivity of search clauses, to be used when determining which attribute access sequence the query should choose.

However, the vertical storage structure is not well suited for the management of metadata, since the access pattern of system-generated, runtime metadata lookup queries and updates is much more "horizontally" clustered than typical user queries. In fact, the access pattern of such queries is quite similar to that of a random sequence of simple-transaction queries being issued during a short time interval. To a first approximation the individual items in such a sequence can be assumed to be uncorrelated, *i.e.*, when the system brings into memory a vertically structured MDB buffer load to access one data item in an MDB attribute, the likelihood that it will access another data item from the same buffer load in the near future is no greater than that of a random hit into the attribute. Therefore, the next time a data item is needed from the same MDB attribute, it is quite likely that a new buffer load will have to be fetched from disk. On the other hand, the likelihood is quite high that the next few MDB accesses will involve other properties associated with the same MDB relation, which should favor horizontal clustering. In the first case, the hit rate for the buffered data will be low and the performance of the system will suffer (unless the entire column fits into the buffer and can therefore be assumed to stay in memory permanently).

A simple solution to this “impedance mismatch” problem could be to store the metadata as a collection of hash tables or linked lists, but if one wants the system to be able to answer general queries which involve metadata relation tables, and perhaps other relation tables as well, a more elaborate solution is needed, possibly one similar to that used in C-Store to manage simple-transaction-type queries, see Section 3.2, below. In Cantor, this issue was handled by designing a separate cache memory for MDB relation tables with a least-recently-used (LRU) replacement regime, structured as a collection of linked lists of metadata records, one for each relation, attribute, and b-list storage structure. This amounts to adopting an NSM architecture for the cache, to be used for internal queries only. When a user query involves a MDB relation, the entire MDB is first “unified”, *i.e.*, all updates made to the cache since the previous unification are written out to the transposed files used to permanently store the columns of MDB tables.

2.6 Querying compressed, fully transposed files

The authors of [SAB+05], while describing the architectural properties of the C-Store system, acknowledge previous work on using compressed data in databases, stating that “Roth and Van Horn [RV93] provide an excellent summary of many of the techniques that have been developed. Our coding schemes are similar to some of these techniques, all of which are derived from a long history of work on the topic in the broader field of computer science… Our observation that it is possible to operate directly on compressed data has been made before [Gra91, WKH+00]”. Indeed, the capability to represent multiple values in a single field to simultaneously apply an operation on all the values at once, was exploited earlier in the algorithm for CFTOF search described in [Sve79b].

Batory [Bat79] showed that search algorithms designed for use with transposed files could outperform commonly used techniques such as the use of inverted files (indexes) in a large proportion of cases. In [Sve79b], theoretical and empirical results were presented, showing that conjunctive queries may be evaluated even more efficiently if the transposed file structure is combined with sorting with respect to the primary key followed by column-wise run-length encoded (RLE) data compression, forming a *compressed, fully transposed ordered file* (CFTOF) organization. The performance of a test-bed system was measured and compared with a commercially available database system and with results from an analytical performance model. The results showed that order-of-magnitude performance gains could indeed be achieved by combining transposed file storage and data compression techniques.

In [AS88] the authors later refined these results by combining interpolation search, sequential search, and binary search into a poly-algorithm which dynamically selects the appropriate method, given known metadata. It is shown how this “modified CFTOF interpolation search” significantly improves search performance over sequential CFTOF search in critical cases. The only situation where inverted file range search retains a clear advantage in a CFTOF-structured data base is in highly selective queries over non-key attributes. To add the complex and costly machinery of updatable inverted files to handle that (in a read-optimized database) fairly uncommon special case seems unwarranted in most analytic DBMS applications.

One feature of RLE-compressed vertical data storage architectures that allows for direct operation on compressed data is that not only can they be exploited in conjunctive query searches, but they can be used profitably also in join and set operations on relations, as well as in duplicate removal operations. An important design feature of Cantor’s search and sort subsystem [KS86], which contains algorithms for internal and external sorting, duplicate tuple detection, conjunctive query search, key lookup, merge-join, set union, set difference, and set intersection, is that all these algorithms are designed to work one (or a small, fixed number of) attribute(s) at a time, to match the transposed file principle. In most of these algorithms, scanning a compressed transposed file is done by using the sequential read interface which retains run-compressed data. Internal sorting is carried out using a modified Quicksort algorithm, which for each (group of) attribute(s) produces a stream of tuple identifiers (TIDs) as output, according to which subsequent attributes are to be permuted.

2.7 Compression-aware optimizations for the equi-join operator

In [AMF06], a discussion of a nested-loop join algorithm capable of operating directly on compressed data is presented, and the paper also contains pseudo-code showing how the join operator may take into account the compression state of the input columns. Combinations of the three cases *uncompressed*, *RLE*, and *bit-vector encoded* data are considered. For example, if one of the input columns is bit-vector encoded and the other is uncompressed, then the resulting column of positions for the uncompressed column can be represented using RLE coding and the resulting column of positions for the bit-vector column can be copied from the appropriate bit-vector for the value that matched the predicate. The authors also present results from several benchmark tests, showing clear, often order-of-magnitude, performance improvements from judicious use of data compression, in particular the use of RLE on ordered data.

In [Sve82], the main equi-join algorithm used in Cantor, based on merging CFTOF-represented relations, was presented by way of a simple example. The paper states that “the search [scanning] phase of the equi-join operation … is so similar to conjunctive query search that it is to be expected that analogous results hold”, but no performance measurement data are given. The end result of the scanning phase is a compressed, ordered list of qualifying TID pairs, *i.e.*, a compressed join index. When a join search is to be done on non-key attributes, one or both factors have to be re-sorted first. In such cases, sorting will dominate in the equi-join process, unless the resulting Cartesian product has much greater cardinality than its factors. However, in situations where no re-sorting of operands is necessary, this algorithm provides a fast way of allowing the equi-join operator to work directly on run-length compressed data.

2.8 Two recent benchmark studies

In the paper [SBC+07], results from a benchmarking study are presented, and performance comparisons are made between commercial implementations based on what these authors call “specialized architectures” and conventional relational databases. The tests involve a range of DBMS applications, including both a standard data warehouse benchmark (TPC-H) and several unconventional ones, namely a text database application, message stream processing, and some computational scientific applications. The “specialized architecture” system used in the data warehouse benchmarks was *Vertica*, a recently released parallel multi-node, shared-nothing, vertical database product [Ver08] designed along the lines of C-Store. It utilizes a DSM data model, data compression, and sorting/indexing. On these examples, Vertica spent between one and two orders of magnitude less time than the comparison system, running in a big and expensive RDBMS installation.

Another database design and benchmarking study using semantic web text data was reported in [AMM+07]. The vertical database used in this study was an extension of C-Store capable of dealing with Semantic Web applications, while the row-store system used for comparison was the open source RDBMS *PostgreSQL* [Pos08], which has been found more efficient when dealing with sparse data than typical commercial database products (in this application, NULL data values are abundant). The authors showed that storing and processing Semantic Web data in RDF format efficiently in a conventional RDBMS requires creative representation of the data in relations. But, more importantly, they showed that RDF data may be most successfully realized by vertically partitioning the data that obey logically a fully decomposed storage model (DSM). The authors demonstrated an average performance advantage for C-Store of at least an order of magnitude over PostgreSQL even when data are structured optimally for the latter system.

2.9 Scalability

Over the last decade, the largest data warehouses have increased from 5 to 100 terabytes, and by 2010, most of today’s data warehouses may be 10 times larger than today. Since there are limits to the performance of any individual processor or disk, all high-performance computers include multiple processors and disks. Accordingly, a high-performance DBMS must take advantage of multiple disks and multiple processors. In the note [DMS08], three approaches to achieving the required scalability are briefly discussed.

In a *shared-memory* computer system, all processors share a single memory and a single set of disks. Distributed locking and commit protocols are not needed, since the lock manager and buffer pool are both stored in the memory system where they can be accessed by all processors. However, since all I/O and memory requests have to be transferred over the same bus that all processors share, the bandwidth of this bus rapidly becomes a bottleneck so there is very limited capacity for a shared-memory system to scale.

In a *shared-disk* architecture, there are a number of independent processor nodes, each with its own memory. Such architectures also have a number of drawbacks that limit scalability. The interconnection network that connects each processor to the shared-disk subsystem can become a bottleneck. Since there is no pool of memory that is shared by the processors, there is no obvious place for the lock table or buffer pool to reside. To set locks, one must either centralize the lock manager on one processor or introduce a distributed locking protocol. Both are likely to become bottlenecks as the system is scaled up.

In a *shared-nothing* approach, each processor has its own set of disks. Every node maintains its own lock table and buffer pool, eliminating the need for complicated locking and consistency mechanisms. Data are "horizontally partitioned" across nodes, such that each node has a subset of the rows (and in vertical databases, maybe also a subset of the columns) from each big table in the database. According to these authors, shared-nothing is generally regarded as the best-scaling architecture (see also [DG92]).

3. Two contemporary systems based on DSM: MonetDB and C-Store

Author: Per Svensson

We give next a brief overview of two recently developed vertical database systems in order to contrast their styles.

3.1 MonetDB

MonetDB [Mon08, Boncz02] uses the DSM storage model. A commonly perceived drawback of the DSM is that queries must spend "tremendous additional time" doing extra joins to recombine fragmented data. This was, for example, explicitly claimed in [ADH+01]. According to the authors of [BK99], for this reason the DSM was for a long time not taken seriously by the database research community. However, as these authors observe (and as was known and exploited long ago [Sve79b, Sve82]), vertical fragments of the same table contain different attribute values from identical tuple sequences, and if the join operator is aware of this, it does not need to spend significant effort on finding matching tuples. MonetDB maintains fragmentation information as *properties* (metadata) on each binary association table and propagates these across operations. The choice of algorithms is typically deferred until run-time, and is done on the basis of such properties.

With respect to query processing algorithms, the MonetDB developers have shown that a novel *radix-cluster algorithm* for hash-join is better than standard bucket-chained alternatives. In a radix cluster algorithm, both relations are first partitioned on hash number into a number of separate clusters which each fit the memory cache, before appropriately selected pairs of clusters are hash-joined together (see [MBK00a, MBK00b] for details). The result of a hash-join is a binary association table that contains the (TID1, TID2) combinations of matching tuples, *i.e.*, a join index. As indicated above, subsequent tuple reconstruction is a cheap operation which does not need to be included in the analysis.

The architectural design and key features of the MonetDB system are presented in section 4, and experience from using it in a large-scale scientific database application is presented in section 5.

3.2 C-Store

C-Store [C-S08; SAB+05] features a two-level store with one writable part and one, typically much larger, read-only part. Both storage levels are column-oriented. The use of this principle, called a *differential file,* in data management was studied in [SL76], although its full realization in a relational database may perhaps have been first achieved in C-Store much later. This way, C-Store attempts to resolve the conflicting requirements for a fast and safe parallel-writable store (WS) and a powerful read-only query processor and storage system (RS). Tuples are periodically moved from WS to RS by a batch update process. Although the storage model of C-Store is more complex than the DSM in order to manage also queries of simple-transaction type efficiently, most of its basic design principles are similar to those of DSM and hence best suited for read-optimized databases.

To store data, C-Store implements *projections*. A C-Store projection is *anchored* on a given relation table, and contains one or more attributes from this table, retaining any duplicate rows. In addition, a projection can contain any number of attributes from other tables as long as there is a complete sequence of foreign key relationships from the anchor table to the table from which an attribute is obtained. Hence, a projection has the same number of rows as its anchor table. The attributes in a projection are stored column-wise, using one separate storage structure per attribute. Tuples in a projection are sorted with respect to the same *sort key, i.e.,* any attribute or sequence of distinct attributes of the projection. Finally, every projection is horizontally partitioned into one or more *segments*, each of which is given a segment identifier value *sid*, based on the sort key of the projection. Hence, each segment of a given projection is associated with a *key range* of the sort key for the projection.

To answer SQL queries in C-Store, there has to exist at least one covering set of projections for every table in the logical schema. In addition, it must be possible to reconstruct complete rows of all tables from the collection of stored segments. To do this, C-Store has to join segments from different projections, which is accomplished using *storage keys* and join indices. Each segment associates every data value of every column with a storage key SK. Values from different columns in the same segment with matching storage keys belong to the same row. Storage keys in RS are implicit, while storage keys in WS are explicitly represented as integers, larger than the largest storage key in RS. Assuming that T1 and T2 are projections that together *cover* the attributes of a table T, an entry in the join index for a given tuple in a segment of T1 contains the segment ID and storage key of the joining tuple in T2. Since all join indices are between projections anchored at the same table, this is always a one-to-one mapping. This strategy contributes to efficient join operations in C-Store.

4. The Architecture and Evolution of MonetDB

Author: Peter Boncz

4.1 Design principles

MonetDB has been introduced in the previous section as a database system that uses vertical decomposition in order to reduce disk I/O. However, the principal motivation to use this storage model was not so much to reduce disk I/O in scientific or business intelligence query loads -- though that is certainly one of its effects. Rather, the MonetDB architecture was based on other considerations given in the original Decomposition Storage Model (DSM) [CK85] paper, namely it focused on data storage layout *and* query algebra, with the purpose of achieving higher CPU efficiency.

At the time the original relational databases appeared, CPU hardware followed an in-order, single-pipeline one-at-a-time design, using a low clock frequency, such that RAM latency took just a few cycles and disk I/O was the strongest performance factor in database performance. In modern multi-GHz computers, however, the cycle cost of a CPU instruction is highly variable, and depends in pipelined CPU designs with multiple instructions per clock, on CPU cache hit ratio, on branch misprediction ratio (see [Ross02]), and on dependencies on other instructions (less dependencies leading to faster execution). In other words, the difference in throughput between “good” and “bad” program code has been increasing significantly with newer CPU designs, and it had become clear that traditional database code turned out to fit mostly in the “bad” basket [ADH+99]. Therefore, MonetDB attempted to follow a query execution strategy radically different from the prevalent tuple-at-a-time pull-based iterator approach (where each operator gets its input by calling the operators of its children in the operator tree), as that can be linked to the “bad” performance characteristics of database code. MonetDB aimed at mimicking the success of scientific computation programs in extracting efficiency from modern CPUs, by expressing its calculations typically in tight loops over arrays, which are well-supported by compiler technology to extract maximum performance from CPUs through techniques as strength reduction (replacing an operation with an equivalent less costly operation), array blocking (grouping subsets of an array to increase cache locality), and loop pipelining (mapping loops into optimized pipeline executions).

4.2 The Binary Association Table Algebra

The distinctive feature of MonetDB thus is the so-called Binary Association Table (BAT) Algebra which offers operations that work only on a handful of BATs. The term Binary Association Table refers to a two-column <surrogate,value> table as proposed in DSM. The left column (often the surrogate of the record identity) is called the "head" column, and the right column "tail". The BAT Algebra is closed on BATs, *i.e.* its operators get BATs (or constants) as parameters, and produce a BAT (or constant) result. Data in execution is always stored in (intermediate) BATs, and even the result of a query is a collection of BATs. Some database systems using vertical fragmentation typically use a relational algebra that adopts the table data model, *i.e.* horizontal records inside the execution engine. In the implementation, this leads to query processing strategies where relational tuples (*i.e.* horizontal structures) are reconstructed early in the plan, typically in the Scan operator. This is not the case in MonetDB; data always remains vertically fragmented.

BAT storage takes the form of two simple memory arrays, one for the head and one for the tail column (variable-width types are split into two arrays, one with offsets, and the other with all concatenated data). MonetDB allows direct access to these entire arrays by the BAT Algebra operators. In case that the relations are large, it uses memory mapped files to store these arrays. This was in line with the philosophy of exploiting hardware features as much as possible. In this case, allowing array lookup as a way to locate tuples in an entire table, in effect means that MonetDB exploits the MMU (memory management unit) hardware in a CPU to offer a very-fast O(1) lookup mechanism by position – where the common case is that the DSM surrogate columns correspond to positions.

As shown in Figure 11.2, MonetDB follows a front-end/back-end architecture, where the front-end is responsible for maintaining the illusion of data stored in some end-user format (i.e. relational tables or objects or XML trees or RDF graphs in SQL, ODMG, XQuery and SPARQL front-ends, respectively). In the MonetDB backend, there is no concept of relational tables (nor concepts of objects), there are only BATs. The front-ends translate end-user queries in (SQL, OQL, XQuery, SPARQL) into BAT Algebra, execute the plan, and use the resulting BATs to present results. A core fragment of the language is presented below:

```
reverse(bat[t1,t2] B) : bat[t2,t1] = [ <B[i].tail,B[i].head> | i < | B| ] "swap columns"
mirror(bat[t1,t2] B) : bat[t1,t1]  = [ <B[i].head,B[i].head> | i < | B| ] "make tail equal to head"
mark(bat[t1,t2] B) : bat[t1,TID] = [ <B[i].head,i> | i < | B|] "number tail"

join(bat[t1,t2] L, bat[t2,t3] R) : bat[t1,t3]  = [ <L[i].head,R[j].tail> |  i < |L|, j < |R}, L[i].tail = R[j].head ] "inner join"
uselect(bat[t1,t2] B, t2 v) : bat[t1,void]      = [ <B[i].head,nil> | i < |B|, B[I].tail = v ] "selection on tail"
 [+](bat[t1,t2] L, bat[t1,t2] R) : bat[t1,t2]   = [ <L[i].head,L[i].tail+R[i].tail> | i < |L|, j < |R|, L[i].head=R[i].head ] "map [op]"
group(bat[t1,t2] L, bat[t1,t2] R) : bat[t1,t2]= [ <L[i].head,unique(L[i].tail,R[j].tail)> | i < |L|, j < |R|, L[i].head=R[i].head ]
                                                                     "groupby"
unique(bat[t1,t2] B) : bat[t1,t2]  = { <B[i].head,B[i].tail> | i < |B| } "duplicate elimination"
{sum}(bat[t1,t2] B) : bat[t1,t2]  = [ <U[i].head,sum(select(reverse(L)),h.head)> | U= unique(mirror(B)) ] "aggr {op}"
```

The BAT-algebra notation used above consists of an operation applied to an operand on the left of the ":", and the result in the right. For example, the "join" operator is applied to left (L) and right (R) BATs, and for entries where tail of L is equal to head of R, it generates a result with columns corresponding to head of L and tail of R.

The reverse(), mirror() and mark() operators all produce a result in which at least one of the input columns appears unchanged. In the MonetDB implementation, these operations have a constant-cost implementation that just manipulate some information in the column descriptor, since the result BAT shares the (large) array data-structures holding the column data with its input BAT. The [op]() and {op}() are second-order operators that take a operator name "op" and construct for it a map-operator (that works on the natural join of all input BATs on head column), and a grouped aggregate function, respectively.

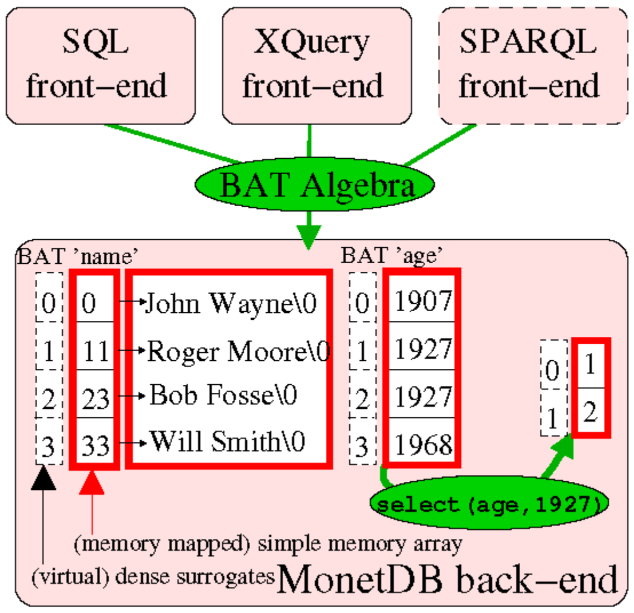

Figure 11.2 MonetDB architecture: the front-end translates queries into

BAT Algebra expressions; the backend executes the BAT plan.

4.3 Efficiency advantages of using the BAT Algebra

The main advantage of the BAT Algebra is its hard-coded semantics, causing all operators to be predicate-less. For comparison, in relational algebra, the Join and Select operators take an arbitrary Boolean column expression that determines which tuples must be joined and selected. The fact that this Boolean expression is arbitrary, and specified at query time only, means that the RDBMS must include some expression interpreter in the critical runtime code-path of the Join and Select operators. Such predicates do not occur in BAT Algebra; therefore we also say it has a "zero degree of freedom". For instance, the hard-coded semantics of join(L,R) is that the predicate is a simple equality between the inner columns of the left BAT, L and right BAT, R and its output are the outer columns for the matching tuples. In case of select, the predicate is equality on the tail column. This absence of freedom allows the implementation of the query algebra to forsake an expression interpreting engine; rather all BAT algebra operations in the implementation map onto array operations. For instance, the expression "select(bat[TID,int] B, int V) : bat[TID,TID] R" in BAT Algebra can be represented at the C level code as something like:

```
for(i=j=0; i<n; i++)
   if (B.tail[i] == V) R.tail[i] = j++;
```

Note that in the "select" BAT statement above:
- the select operator has two parameters B and V and one result R
- B is a BAT with a head-column of type TID, and a tail-column of type "int"
- V is a constant (a single) value of type int
- R is a BAT with both head and tail column of type TID, where the head represents a surrogate

sequence (=0,1,2, …) and the tails contains the qualifying row-IDs of the rows that matched the int=V condition (in the example shown in Figure 11.2, these are rows 1,2).

Such simple loops are amenable to compiler optimization and CPU out-of-order speculation, which lead to high performance. The philosophy behind BAT Algebra can be paraphrased as "the RISC approach to database query languages": by making the algebra simple, the opportunities are created for implementations that execute the common case very fast.

Note that the above code is only correct if the head column of a BAT, B contains a densely ascending TID (tuple identifier, *i.e.* surrogate) sequence starting with 0 (*i.e.* B.head = 0,1,2,…). This happens to be a common case, and MonetDB recognizes this as the "dense" property. Dense TID columns are in fact not stored at all in the implementation, as they are the same as the array index in the column. As many head columns are dense, MonetDB BAT processing often equates to simple array processing. In addition to denseness, MonetDB keeps a series of other run-time properties on columns (uniqueness, sortedness, min/max) that are exploited at run-time under various circumstances. We show below an example of translating an SQL query into expressions in the BAT Algebra, in order to illustrate the advantages of executing the BAT Algebra expressions. The details of the algebra are not important, but rather this example is intended to illustrate the operators used that execute this query. Note that the algebraic expressions represent an execution plan.

Example: the SQL query:

```
SELECT DISTINCT P.firstname,P.lastname, SUM(I.price)
FROM Person P, Item I
WHERE P.id = I.buyer and I.year = 2007
GROUP BY P.Firstname,P.lastname
```

translates into BAT algebra:

```
s := reverse(mark(uselect(Item_year, 2007)))
b := join(s,Item_buyer)
p := join(b,reverse(Person_id))
r := reverse(mark(reverse(p)))
g := group(join(r,Person_firstname), join(r,Person_lastname))
a := {sum}(join(join(reverse(g),r),Item_price)
[print](join(g,Person_firstname), join(g,Person_lastname), a)
```

A potential disadvantage of the DSM model is the large number of joins needed to relate columns, also visible in the above plan, which has 8 join operators. However, note that only a single join operation is a real value-based join (shown in boldface); all other joins are cases where a TID tail-column from a known min/max range is joined into a dense head-column that spans that range. The property detection in MonetDB successfully derives all such joins into a "fetchjoin" algorithm which for each left input fetches a single tail result from the right input using a positional array lookup. Note that fetchjoin is a linear operation at very low CPU cost (a single load instruction). Therefore, MonetDB turns out to perform no expensive additional joins relative to N-ary Storage Model (NSM) execution engines that store tuple records contiguously in disk pages.

The BAT Algebra core process arrays directly and thus foregoes locking and other transaction processing operations. Rather, a separate module with explicit locking primitives and WAL (Write Ahead Log) functionality is offered. Thus, it is up to the various front-ends to ensure that queries do not conflict (referred to as ACID properties), if needed. Note that some front-ends do not perform on-line updates (which is typical in scientific applications and data mining tools) and therefore do not need to use any transaction management. The advantage in MonetDB is that such applications do not suffer any overhead from transaction facilities that they do not use. The SQL and XQuery front-ends both offer full ACID properties, showing that a separation of execution and transaction processing enforcement can indeed be achieved in database architecture.

4.4 Further improvements

The original design of MonetDB had two main weaknesses. First, the reliance on virtual memory for disk storage means that the buffer manager is removed from the system architecture. While removing this layer makes it easier to write efficient data processing algorithms, it means that MonetDB relies on virtual memory advice calls to perform buffering policies. The downside is mainly practical, that is, the implementation of such virtual memory advice can often be incomplete or ineffective, depending on the OS (version). Furthermore, virtual memory prefetching is configured at the OS kernel level, and tuned for different access patterns than those that MonetDB targets. This often leads to I/O prefetch sizes that are too small (and thus, lower bandwidth is achieved). The second main problem in the design is that the BAT Algebra implementation follows a design of full materialization. An algebra operator fully consumes its input BATs, producing a full result BAT. Again, while such loop code is simple and efficient, problems may occur if the result arrays are large. If these are huge, which is often the case with queries on scientific data, output flows via virtual memory to disk, and swapping may happen, deteriorating performance. Both these problems have been fixed in the subsequent MonetDB/X100 system, which introduces a pipelined model operating on small BAT pieces (vectors), and introduces a buffer manager that can perform efficient asynchronous I/O. The use of a buffer manager in MonetDB/X100 also means that compression techniques which work well with vertical storage can be exploited. Furthermore, vertically oriented compressed indexes, such as FastBit (described in Chapter 10) can be exploited as well.

4.5 Assessment of the benefits of vertical organization

The vertical organization of storage in MonetDB led to the achievement of the original goal of high performance and CPU efficiency, and was shown to outpace relational competitors on many query-intensive workloads, especially when data fits into RAM (see case study in the next subsection). Because of the vertical data layout, it was possible to develop a series of architecture-conscious query processing algorithms, such as for instance radix-partitioned hash-joins and radix-cluster/decluster (cache-efficient permutation). Also, pioneering work in architecture-conscious cost modeling and automatic cost calibration were done in this context.

The approach taken by MonetDB of using a front-end/back-end architecture provides practical advantages as well. It is relatively easy to extend with new modules that introduce new BAT Algebra operators. This ease can be attributed to the direct array-interface to data in MonetDB, which basically implies that no API is needed to access data (therefore database extenders do not have to familiarize themselves with a complex API).

The use of a vertical data layout not only on disk, but also throughout query processing turned out to be beneficial, especially when operators access data sequentially. Random data access, even if data fits into RAM, is difficult to make efficient, especially if the accessed region does not fit into the CPU cache. In fact, random access does not exploit all the RAM bandwidth optimally; this is typically only achieved if the CPU detects a sequential pattern and the hardware prefetcher is activated. Therefore main-memory algorithms that have predominantly sequential access tend to outpace random-access algorithms, even if they do more CPU work. Sequential algorithms, in turn, strongly favor vertical storage, as memory accesses are dense regardless of the fact whether a query touches all table columns. Also, sequentially processing densely packed data allows compilers to generate Single Instruction, Multiple Data (SIMD) code, which further accelerates processing on modern machines.

Finally, the idea articulated in the DSM paper [CK85] that DSM could be the physical data model building block that can power many more complex user-levels data models, was validated in the case of MonetDB, where a number of diverse front-ends were built. We describe briefly below the way BATs were used for processing of different front-end data models and their query languages.

- SQL. The relational front-end decomposes tables by column, in BATs with a dense (non-stored)

TID head, and a tail column with values. For each table, a BAT with deleted positions is kept. For each column an additional BAT with insert value is kept. These delta BATs are designed to delay updates to the main columns, and allow a relatively cheap snapshot isolation mechanism (only the delta BATs are copied). MonetDB/SQL also keeps additional BATs for join indices; and value indices are created on-the-fly.

- XQuery. The work in the Pathfinder project [BGV+06] makes it possible to store XML tree structures in relational tables as <pre,post> coordinates, represented in MonetDB as a collection of BATs. In fact, the pre-numbers are densely ascending, hence can be represented as a (non-stored) dense TID column, saving storage space and allowing fast O(1) lookups. Only slight extensions to the BAT Algebra were needed, in particular a series of region-joins called "staircase joins" were added to the system for the purpose of accelerating XPath predicates. MonetDB/XQuery provides comprehensive support for the XQuery language, the XQuery Update facility, and a host of specific extensions.
- Arrays. The Sparse Relational Array Mapping (SRAM) project maps large (scientific) array-based data-sets into MonetDB BATs, and offers a high-level comprehension-based query language [CHZ+08]. This language is subsequently optimized on various levels before being translated into BAT Algebra . Array front-ends are particularly useful in scientific applications.
- SPARQL. The MonetDB team started work in 2008 to offer scalable RDF storage and support for the W3C query language SPARQL to the system.

5. Experience with SkyServer data warehouse using MonetDB

Authors: Milena Ivanova, Martin Kersten, Niels Nes

5.1 Application description and planned experiments

To illustrate the advantages of vertical databases for scientific data management we summarize the experiences from porting the SkyServer application [SGT+02] onto MonetDB. The SkyServer application is a good example of a read-optimized database system with long periods of ad-hoc querying of large data volumes, and periodic bulk-loading of new data. In these settings a column-store architecture offers more efficient data access patterns for disk-bound queries, flexibility in the presence of changing workloads, and reduced storage needs. The MonetDB/SkyServer project [ING+07] started with the purpose of providing an experimentation platform to develop new techniques addressing the challenges posed by scientific data management. Our intent was to examine and demonstrate the maturity of column-store technology by providing the functionality required by this real-world astronomy application. The project shows the advantages of vertical storage architectures for scientific applications in a broader perspective. It goes way beyond micro benchmarks and simulations typically used to examine individual algorithms and techniques. MonetDB/SkyServer allows testing the performance of the entire software stack.

The SkyServer application gives public access to data from the Sloan Digital Sky Survey [SDSS], an astronomy survey with the ambition to map one-quarter of the entire sky in detail. The survey has already collected several terabytes of data. The sky object catalog stored in a relational database reached the volume of 4TB for data release 6 in 2007. The database schema is organized in several sections among which *Photo* and *Spectro* contain the most important photometric and spectroscopic factual data from the survey. The *Photo* section has a structure centered in the *PhotoObjAll* table. The table contains more than 440 columns and more than 270 million rows, which already stresses the capabilities of most DBMSs. A single record in a row-store representation occupies almost 2KB, and the majority of the fields are real numbers representing CCD measurements.

Porting of the SkyServer application to MonetDB was organized in three phases. The goal of the first phase was to develop and enhance MonetDB's features to handle the functionality requirements of the SkyServer application. The target dataset during this phase was the so-called Personal SkyServer, a 1% subset of the archive with a size of 1.5GB, approximately. Since this dataset fits entirely in memory, there were no scalability issues with the main-memory orientation of MonetDB. The large vendor-specific schema (consisting of 91 tables, 51 views, and 203 functions of which 42 are tablevalued) and its extensive use of the SQL persistent storage module functionality required an engineering effort. We had to cast vendor-specific syntax (such as identifiers ‘datetime’ *vs.* ‘timestamp’) in the schema definition into the SQL:2003 standard supported by MonetDB/SQL. We also adapted the application to the column-store architecture and slightly modified the schema reducing data redundancy.

The challenge addressed in the second phase was to scale the application to sizes beyond the main memory limit. The target dataset was a 10% subset of approximately 150GB. The project is currently in its third phase aiming to support the full-sized 4TB database. Some interesting techniques yet to be investigated that may increase system efficiency are: exploring parallel load, interleaving of column I/O with query processing, self-organizing indexing schemes, and exploitation of commonalities in query batches.

5.2 Efficient vertical data access for disk-bound queries

As explained in the introductory section, the major advantage of column-wise storage comes from minimizing the data flow from disk through memory into the CPU caches. Many scientific analytical applications involve examination of an entire table, or a big portion of it, while at the same time spanning just a few attributes at a time. The immediate benefit the column-wise storage brings is that only data columns relevant for processing are fetched from disk.

In contrast, the access pattern in a row-wise storage of wide tables, such as the *PhotoObjAll* table, might require hundreds of columns to be transferred from disk, where many of the columns are irrelevant to the query. This becomes the major performance bottleneck for analytical queries. To illustrate the problem consider the following SQL query searching for moving asteroids (Q15 in [GST+02]).

```
SELECT objID, sqrt(power(rowv,2) + power(colv,2)) as velocity
FROM PhotoObj
WHERE (power(rowv,2) + power(colv,2)) > 50
    and rowv >= 0 and colv >= 0;
```

The execution plan for a row-wise storage organization involves a full table scan which leads to transferring entire records of 440+ columns in order to process the four columns referred to in the query. For the 150GB dataset, the volume transferred is almost 50GB. The execution plan in MonetDB involves scans strictly limited to the columns directly referenced in the query, which amounts to 370MB for the example query above.

The access pattern problem in row-wise storage systems has already been addressed by a variety of techniques, such as indices, materialized views, and replicated tables. For example, if all the columns in a query are indexed, the query can be substantially sped up by scanning the shorter index records instead of touching the wide records of the main table. We illustrate this with the next query example. It extracts celestial objects that are low-z quasar candidates, a property specified through correlations between the objects' magnitudes in different color bands (query SX11 in [GST+02]).

```
SELECT g, run, rerun, camcol, field, objID
FROM Galaxy
WHERE ( (g <= 22)
    and (u - g >= -0.27) and (u - g < 0.71)
    and (g - r >= -0.24) and (g - r < 0.35)
    and (r - i >= -0.27) and (r - i < 0.57)
    and (i - z >= -0.35) and (i - z < 0.70) );
```

The query predicates do not allow efficient index search of the qualifying rows, instead scanning of all the rows is needed. However, a full table scan can be avoided using available indices that contain all the necessary columns. The data volume transferred for the 150GB dataset is 1.8GB, a substantial reduction with respect to the full table scan, but still twice as large as the 850MB transferred in MonetDB for the same query. The reason is that the indices chosen for the query execution contain several additional columns irrelevant for this query.

5.3 Improved performance

In addition to the efficient vertical access pattern, MonetDB employs a number of techniques to provide high performance for analytical applications. Among these are run-time optimization, such as choosing the best algorithm fitting the argument properties, and efficient cache-conscious algorithms exploiting modern computers architecture. To demonstrate the net effect of these techniques on the performance experienced by the end user, we performed a few experiments with the above table- and index-scan queries against both the 1.5GB and 150GB datasets. The elapsed times in seconds are shown in the table below. The performance of the vertical database for index-supported queries is comparable for the small dataset, and 30% better for the large dataset. Queries involving full table scans are sped up by a factor of 5 for the large dataset.

	Table scan 1.5GB	Index scan 1.5GB	Table scan 150GB	Index scan 150GB
Row-store	6.6	0.4	245	24
Column-store	0.4	0.47	53	16

5.4 Reduced redundancy and storage needs

The original SkyServer system utilizes indices and replication to speed up important disk-bound queries. All tables have primary and foreign key constraints supported by B-tree indices, and many tables have covering indices created after careful workload analysis. Replicated tables are also used to speed up some frequent classes of queries. For instance, the *PhotoTag* table is a vertical partition of the *PhotoObjAll* table that stores redundantly its most popular 100+ columns. The *SpecPhotoAll* table stores the most popular of the columns from the pre-computed join of photo and spectrum tables.

In order to support the original queries we replaced the *PhotoTag* and *SpecPhotoAll* tables with views exploiting the advantages of the column-wise storage of MonetDB. This replacement had little impact on the performance of queries that involve those tables because of the column-wise storage organization. However, generating the views was still worthwhile since this saved approximately 10% of the storage needs.

The index support in MonetDB is limited to primary and foreign keys. The system generates indices on-the-fly when columns are touched for the first time. The net effect of reducing data volume is that the storage needs of MonetDB database image decreased by approximately 30%.

5.5 Flexibility

Although secondary access structures in row-wise storage systems improve performance substantially in comparison to full table scans, they exhibit relatively static behaviors with respect to changing workloads. Modern DBMSs come with advanced database design tuning wizards which derive design recommendations using representative workloads. Due to its complexity, the workload analysis is mostly performed off-line and requires data base administrator (DBA) competence to decide on the final database design. When the workload changes, it is probable that the new, unanticipated queries are not supported (or partially supported) by the existing indices which leads to sub-optimal system performance. The typical solution is that the DBA monitors the system functionality and periodically re-runs the workload analysis and modifies the supporting secondary structures.

Recently on-line tuning tools have been proposed [BC07] that take the burden from the DBA, but still incur overhead for monitoring and creation of secondary structures. Dealing with this issue is completely avoided in MonetDB. When the query load changes to incorporate new attributes, the execution plans simply transfer to memory only the new columns of interest. This is achieved without any storage, creation, or monitoring overhead for secondary structures, but simply based on the architectural principles of the column-wise storage systems.

5.6 Use cases where vertical databases may not be appropriate

There are a number of situations where column-wise storage is comparable or slower than row-wise systems. The category of point and range queries is usually efficiently supported in the row-store databases since the available indices enable quick retrieval of qualifying rows. For a small number of qualifying rows, the data transfer is sufficiently efficient and is not perceivable by the end user. For the same query category MonetDB often uses a sequential scan which might be slower than searching with a B-tree index. However, for append-only data, which is the case for scientific data, new types of compressed bit-map indices (described in Chapter 10) require a relatively small space overhead of only 30% of the original data. If this overhead is not prohibitive, then all columns (or columns searched often) can be indexed to provide efficient point and range queries in vertical databases.

Another source of performance overhead in vertical databases are tuple reconstruction joins. Despite their efficient implementation, they may still contribute a substantial cost for queries that request all attributes (referred to as "SELECT *" queries), or queries with a large number of attributes. Here again, using compressed bitmap indices can mitigate this overhead, since joining the results of qualifying tuples from each column can be done by logical operations (AND, OR, NOT) over multiple bitmaps, where each bitmap represents the result of searching the index of each column.

There are some uncommon applications where all (or most) columns are needed in every query. In such cases there is no value to using column-wise organization, and row-wise organization with appropriate indexing (for selecting the desired tuples given predicate conditions) may prove more efficient. Also, row-wise organization may be more appropriate in applications where very few rows are selected, and several columns are involved. An extensive analysis of which organization is best was conducted in [OOW07]. Given a characterization of the query patterns, a formula was developed in order to determine which organization is better. By and large, for applications where a large number of rows is selected, and only a subset of the columns are involved in the query, column-wise organization is superior. Furthermore, in practical experiments described in [OOW07], it was shown that when sequential reads (which are much faster than random read operations) are considered as a possible strategy, column-wise organization is even more favorable because it is much easier to utilize sequential read operations with the vertical data organization.

Although we prefer to reduce data redundancy, in some cases it may prove useful to store derived data when generated, for instance, by expensive computations. For example, the *Neighbors* table groups together pairs of SDSS objects within an a-priori distance bound of 0.5 arc-minutes. Our attempt to replace this table with a view computing the distances showed to be less efficient than accessing the pre-computed table.

5.7 Conclusions and future work

Our experiences with MonetDB/SkyServer application confirm the advantages of column-wise storage systems for scientific applications with analytical disk-bound processing. To improve the performance for point and range queries several techniques for workload-driven self-organization of columns have been developed in MonetDB, such as cracking (continuous physical organization based on access patterns) [IKM07], and adaptive segmentation and replication (splitting columns into segments or replicating segments) [IKN08]. We intend to integrate those techniques in support of the SkyServer application. Since compression has shown to be particularly efficient in combination with column-wise storage [AMF06], we also intend to investigate and utilize appropriate compression schemes for the SkyServer application.

The MonetDB execution engine differs in a fundamental way from state-of-the-art commercial systems. The execution paradigm is based on full materialization of all intermediate results in a query plan. This opens another direction of research exploiting commonalities in query batches by carefully preserving and reusing common intermediate results.

References

[ADH+99] Ailamaki, A., DeWitt, D., Hill, M., Wood, D.A.: DBMSs on a Modern Processor: Where Does Time Go? In *Proc. 25th Int. Conf. on Very Large Databases*, Edinburgh, Scotland (1999).
[ADH+01] Ailamaki, A., DeWitt, D., Hill, M., Skounakis, M.: Weaving Relations for High Performance. In *Proc. of the 27th Int. Conf. on Very Large Databases*, Rome, Italy (2001).
[AMF06] Abadi, D.J., Madden, S., Ferreira, M.C.: Integrating compression and execution in column-oriented database systems. In *Proc. 2006 SIGMOD Conf.*, June 27-29, Chicago. IL, USA. ACM, New York (2006).
[AMM+07] Abadi, D.J., Marcus, A., Madden, S., Hollenbach, K.: Scalable Semantic Web data Management Using Vertical Partitioning. In *Proc. 33rd Int. Conf. on Very Large Databases*, September 23-28, Vienna, Austria (2007).
[AS88] Andersson, M., Svensson, P.: A study of modified interpolation search in compressed, fully transposed, ordered files. In *Proc. 4th Int. Working Conf. on Statistical and Scientific Database Management (SSDBM)*, Rome, Italy, June 21-23. LNCS 339, Springer-Verlag (1988).
[Bas85] Bassiouni, M.A.: Data Compression in Scientific and Statistical Databases. *IEEE Trans. on Software Eng.*, SE-11(10), Oct., 1047-1058 (1985).
[Bat79] Batory, D. S.: On Searching Transposed Files. *ACM Trans. on Database Systems* (TODS), 4(4), 531-544 (1979).
[BC07] Bruno, N., Chaudhuri, S.: An Online Approach to Physical Design Tuning. In *Proc. ICDE*, 826-835, IEEE Computer Society (2007).
[BGV+06] Boncz, P. A., Grust, T., Van Keulen, M., Manegold, S., Rittinger, J., Teubner, J: MonetDB/XQuery: a fast XQuery processor powered by a relational engine. SIGMOD Conference 2006: 479-490
[BK99] Boncz, P.-A., Kersten, M.L.: MIL primitives for querying a fragmented world. *The VLDB Journal* 8(2):101-119 (1999).
[Boncz02] *Monet: A Next-Generation DBMS Kernel for Query-Intensive Applications*. PhD Thesis, Univ. of Amsterdam, The Netherlands, May 2002.
[BSS97] Bergsten, U., Schubert, J., Svensson, P.: Applying data mining and machine learning techniques to submarine intelligence analysis. In *Proc. 3rd Int. Conf. on Knowledge Discovery and Data Mining*. AAAI Press, Menlo Park, CA, USA (1997).
[BT81] Burnett, R., Thomas, J.: Data Management Support for Statistical Data Editing. In *Proc. 1st Lawrence Berkeley Laboratory Workshop on Statistical Database Management* (1981).
[BZN05] Boncz, P., Zukowski, M., Nes, N.: MonetDB/X100: Hyper-pipelining query execution. In *Proc. 2nd Biennial Conference on Innovative Data Systems Research (CIDR)*, VLDB Endowment (2005)
[CDG+06] Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A Distributed Storage System for Structured Data. In *Proc. 7th Symposium on Operating Systems Design and Implementation* (OSDI '06), November 6-8, Seattle, USA (2006).
[CHZ+08] Cornacchia, R., Héman, S., Zukowski, M., de Vries, A.P., Boncz, P.A.: Flexible and efficient IR using array databases. VLDB Journal, 17(1): 151-168 (2008)
[CK85] Copeland, G.P., Khoshafian, S.N.: A Decomposition Storage Model. In *Proc. 1985 SIGMOD Conf.*, ACM, New York (1985).
[C-S08] http://db.csail.mit.edu/projects/cstore/ Accessed 2008-05-22.
[DG92] DeWitt, D., Gray, J.: Parallel Database Systems: The Future of High Performance Database Processing." Comm. ACM, **35**(6), 85-98 (1992).
[DMS08] DeWitt, D., Madden, S., Stonebraker, M.: How to Build a High-Performance Data Warehouse. http://db.csail.mit.edu/madden/high_perf.pdf Accessed 2008-05-22.
[EOS81] Eggers, S.J., Olken, F., Shoshani, A.: A Compression Technique for Large Statistical Databases. In *Proc. 7th Int. Conf. on Very Large Databases* (1981).
[Gil83] Giloi, W.K.: Towards a Taxonomy of Computer Architecture Based on the Machine Data Type View. In *Proc. 10th Ann. Symp. on Computer Architecture*, Stockholm. IEEE Inc., New York (1983).

[GNS07] Gray, J., Nieto-Santisteban, M. A., Szalay, A.S.: The Zones Algorithm for Finding Points-Near-a-Point or Cross-Matching Spatial Datasets CoRR abs/cs/0701171 (2007).
[Gra93] Graefe, G.: Query Evaluation Techniques for Large Databases. *ACM Comp. Surv.* 25(2), 73-170 (1993).
[Gra94] Graefe, G.: Volcano – an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.* 6(1):120-135 (1994).
[GS91] Graefe, G., Shapiro, L.D.: Data Compression and Database Performance. In *Proc. Symp. Appl. Comp.* (1991).
[GST+02] Gray, J., Szalay, A.S., Thakar, A.R. *et al.*: *Data Mining the SDSS SkyServer Database.* Microsoft publication MSR-TR-2002-01, January 2002.
[HLA+06] Harizopoulos, S., Liang, V., Abadi, D.J., Madden, S.: Performance tradeoffs in read-optimized databases. In *Proc. 32nd Int. Conf. on Very Large Databases*, September 12-15, Seoul, Korea (2006).
[IKM07] Idreos, S., Kersten, M.L., Manegold, S.: Database Cracking. In *Proc. 3rd Biennial Conference on Innovative Data Systems Research (CIDR)* 68-78, VLDB Endowment (2007)
[IKN08] Ivanova, M., Kersten, M.L., Nes, N.. Self-Organizing Strategies for a Column-Store Database. *Proc. 11th International Conference on Extending Database Technology*, March 25-30, Nantes, France (2008).
[ING+07] M. Ivanova, N. Nes, R. Goncalves, M. L. Kersten: MonetDB/SQL Meets SkyServer: the Challenges of a Scientific Database. In *Proc. 19th Int. Conf. on Statistical and Scientific Database Management (SSDBM)* (2007).
[KBW85] Khoshafian, S.N., Bates, D.M., deWitt, D.J.: *Efficient support of statistical operations.* IEEE Trans. on Software Eng., SE-11(10), 1058-1070 (1985).
[KCJ+87] Khoshafian, S.N., Copeland, G.P., Jagodis, T., Boral, H., Valduriez, P.: A Query Processing Strategy for the Decomposed Storage Model. In *Proc. ICDE*, 636-643, IEEE Computer Society (1987).
[Knu73] Knuth, D.E.: *The Art of Computer Programming. Vol 3: Sorting and Searching.* Addison-Wesley (1973).
[KS83] Karasalo, I., Svensson, P.: An overview of Cantor – a New System for Data Analysis. In *Proc. 2nd Int. Workshop on Statistical Database Management (SSDBM)* (1983).
[KS86] Karasalo, I., Svensson, P.: The design of Cantor – a new system for data analysis. In *Proc. 3rd International Workshop on Statistical and Scientific Database Management (SSDBM)* (1986).
[LS71] Lorie, R.A., Symonds, A.J: *A Relational Access Method for Interactive Applications. Data Base Systems*, Courant Computer Science Symposia, vol. 6. Prentice-Hall (1971).
[MF04] MacNicol, R., French, B.: Sybase IQ Multiplex – Designed for Analytics. In *Proc. of the 30th Int. Conf. on Very Large Databases*, Toronto, Canada (2004).
[MBK00a] Manegold, S., Boncz, P.A., Kersten, M.L.: Optimizing database architecture for the new bottleneck: memory access. *The VLDB Journal* 9(3), 231-246 (2000).
[MBK00b] Manegold, S., Boncz, P.A., Kersten, M.L.: What happens during a Join? Dissecting CPU and Memory Optimization Effects. In *Proc. of the 26th Int. Conf. on Very Large Databases*, Cairo, Egypt (2000).
[Mon08] http://MonetDB.cwi.nl/ Accessed 2008-05-22.
[MS77] Maruyama, K., Smith, S.E: Analysis of design alternatives for virtual memory indexes. *Comm. of the ACM* 20(4) (1977).
[NCW+84] Navathe, S., Ceri, S., Wiederhold, G., Dou, J.: Vertical partitioning algorithms for database design. *ACM Trans. on Database Systems*, 9(4), 680-710 (1984).
[OOW07] O'Neil, E., O'Neil, P.E., Wu, K.: Bitmap Index Design Choices and Their Performance Implications. *Proc. of IDEAS 2007*, 72-84.
[Pos08] http://www.postgresql.org/ Accessed 2008-05-22.
[Ross02] K. A. Ross, "Conjunctive selection conditions in main memory," ACM SIGMOD 2002.
[RS06] Raman, V., Swart, G.: How to wring a table dry: Entropy compression of relations and querying of compressed relations. In *Proc. 32nd Int. Conf. on Very Large Databases*, September 12-15, Seoul, Korea (2006).
[RV93] Roth, M.A., Van Horn, S.J.: Database Compression. *SIGMOD Record* 22(3) (1993).

[SAB+05] Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E.J., O'Neil, P.E., Rasin, A., Tran, N., Zdonik, S.B.: C-Store: A Column-oriented DBMS. In *Proc. 31st Int. Conf. on Very Large Databases*, Trondheim, Norway, August 30 - September 2 553-564 (2005).
[SBC+07] Stonebraker, M., Bear, C., Cetintemel, U., Cherniack, M., Ge, T., Hachem, N., Harizopoulos, S., Lifter, J., Rogers, J., Zdonik, S.B.: One Size Fits All? – Part 2: Benchmarking Results. In *Proc. 3rd Biennial Conf. on Innovative Data Systems Research (CIDR)*, VLDB Endowment (2007).
[SGT+02] Szalay, A.S., Gray, J., Thakar, A.R. *et al*.: The SDSS SkyServer: Public Access to the Sloan Digital Sky Server Data. In *Proc. 2002 SIGMOD Conf*., 570-581 (2002).
[Sev83] Severance, D.G.: A Practitioner's Guide to Database Compression – A Tutorial. *Inf. Syst*. 8(1):51-62 (1983).
[SKM+84] Shibayama, S., Kakuta, T., Miyazaki, N., Yokota, H., Murakami, K.: A Relational Database Machine with Large Semiconductor Disk and Hardware Relational Algebra Processor, *New Generation Computing* vol. 2 (1984).
[SL76] Severance, D.G., Lohman, G.M.: Differential files: Their application to the maintenance of large databases. *ACM Trans. Database Syst.* 1(3), Sept. 1976, 256-267.
[SDSS] Sloan Digital Sky Survey / SkyServer, http://cas.sdss.org/
[Sve79a] Svensson, P.: *Contributions to the design of efficient relational data base systems. Summary of the author's doctoral thesis*. Report TRITA-NA-7909, Royal Institute of Technology, Stockholm (1979).
[Sve79b] Svensson, P.: On Search Performance for Conjunctive Queries in Compressed, Fully Transposed Ordered Files. In *Proc. 5th Int. Conf. on Very Large Databases*, 155-163 (1979).
[Sve82] Svensson, P.: Highlights of a new system for data analysis. In *Proc. CERN Workshop on Software in High Energy Physics* (invited paper), 4-6 Oct., 119-146 (1982).
[Sve88] Svensson, P.: Database management systems for statistical and scientific applications: are commercially available DBMS good enough? In *Proc. 4th Int. Working Conf. on Statistical and Scientific Database Management (SSDBM),* Rome, Italy, June 21-23. LNCS 339, Springer Verlag (1988).
[SW85] Shoshani, A., Wong, H.K.T.: Statistical and scientific database issues. *IEEE Trans. on Software Eng.*, SE-11(10), Oct., 1040-1047 (1985).
[Syb08] http://www.sybase.com/products/datawarehousing/sybaseiq Accessed 2008-05-22.
[Tan83] Tanaka, Y.: A Data-Stream Database Machine With Large Capacity, in *Advanced Database Machine Architectures*, Hsiao, D.K. (ed .), Prentice-Hall (1983).
[Tei77] Teitel, R.F.: Relational Database Models and Social Science Computing, In *Proc. of Computer Science and Statistics 10th Ann. Symp. on the Interface*, Gaithersburg, Maryland, National Bureau of Standards (1977).
[THC79] Turner, M. J., Hammond, R., Cotton, P.: A DBMS for Large Statistical Databases. In *Proc. 5th Int. Conf. on Very Large Databases*, 319-327 (1979).
[VKC86] Valduriez, P., Khoshafian, S., Copeland, G.: Implementation Techniques of Complex Objects, *Proc. 12th Int. Conf. on Very Large Databases*, Kyoto, Japan (1986).
[Val87] Valduriez, P.: Join Indices. *ACM Trans. on Database Systems* (TODS), 12(2), 218-246 (1987).
[Ver08] http://www.vertica.com/product/relational-database-management-system-overview Accessed 2008-05-22.
[WFW75] Wiederhold, G., Fries, J.F., Weyl, S.: Structured Organization of Clinical Data Bases, In *Proc. of the National Computer Conference*, AFIPS Press (1975).
[WKH+00] Westmann, T., Kossmann, D., Helmer, S., Moerkotte, G.: The implementation and performance of compressed databases. *SIGMOD Record* 29(3), 55-67 (2000).