Balancing vectorized query execution with bandwidth-optimized storage
Zukowski, M

Citation for published version (APA):
Chapter 4

MonetDB/X100 overview

The analysis of database architectures presented in the previous chapter has shown how existing execution models fail to fully exploit the performance potential of modern computers on all layers of the hardware stack: super-scalar CPUs, cache memories, main memory and disk. This motivates the research on a new database architecture presented in this chapter.

MonetDB/X100 is a prototype query processing engine that allows us to investigate new techniques in the field of efficient data processing for data-intensive applications. The techniques introduced in this engine focus on the areas of CPU-efficient execution and bandwidth-optimized storage, corresponding directly with the main thesis of this book, defined in Chapter 1:

With the vectorized execution model database systems can minimize the instructions-per-tuple cost on modern CPUs and achieve high in-memory performance, but bandwidth-optimizing improvements in the storage layer are required to scale this performance to disk-based datasets.

Vectorized execution model. In the execution layer, we try to overcome the inefficiencies of the existing approaches described in Chapter 3. The tuple-at-a-time pipelined model requires expensive query-plan interpretation, resulting in a high instructions-per-tuple number. Additionally, this model is unfriendly for modern CPUs, causing poor utilization of super-scalar CPU features and cache memories, resulting in poor cycles-per-instruction ratios. The bulk-processing model of MonetDB overcomes these problems, and achieves high CPU efficiency. However, this comes at a cost of intermediate result materialization, hindering
the overall performance and limiting scalability. As a result, both models, especially the tuple-at-a-time one, can suffer from a high *cycles-per-tuple cost* during in-memory processing. In Section 4.2 we introduce a new *vectorized in-cache execution model* that combines the best features of the tuple-at-a-time model and bulk processing, providing performance often orders of magnitude higher than the existing solutions.

**Bandwidth improvements.** In the storage layer, systems following the NSM approach result in suboptimal exploitation of the disk bandwidth and available buffer memory. Moreover, even for data-intensive applications, databases often rely heavily on random-access methods that require many disk arms per CPU-core to provide sufficient data delivery rates, leading to installations with hundreds of disks even for small data warehouses \[ZHNB06\]. This approach is unsustainable due to the imbalance in the evolution of hard disk latency and bandwidth, as discussed in Chapter 2 Section 4.3 proposes an alternative approach, where data-intensive tasks are implemented using mostly scan-based methods, and presents ColumnBM – a new storage system designed for this type of processing. While this system reduces the bandwidth requirements by using column-based storage, providing enough data for the highly efficient execution layer is still an important problem, especially since fast improvements in the CPU technology allow more and more queries to be executing at the same time. As a result, further bandwidth-improving optimizations are necessary, and ColumnBM introduces two such techniques: lightweight compression and cooperative scans.

This chapter presents the major features of the proposed architecture, and serves as a foundation for detailed discussion of the key introduced techniques, presented further in Chapters 5, 6 and 7.

### 4.1 MonetDB/X100 architecture

Two major design goals of MonetDB/X100 is to (i) execute high-volume queries at high CPU efficiency, and (ii) scale this efficiency to large, disk-resident data volumes. To achieve these goals, MonetDB/X100 fights performance bottlenecks throughout the entire computer architecture, as visualized in Figure 4.1:

CPU To avoid the overhead present in the tuple-at-a-time model, MonetDB/X100 follows the bulk-processing idea of MonetDB. It performs all operations on data using simple *primitive* functions that operate on multiple values in one call (see Section 4.2.1.3). This reduces the number of function calls,
improving the instructions-per-tuple ratio and increasing the code locality. Additionally, primitives typically consist of simple loops over multiple input values. This exposes multiple compiler-level optimization opportunities, and allows efficient execution on super-scalar CPUs.

Cache MonetDB/X100 avoids the intermediate result materialization overhead of MonetDB by combining the bulk-processing approach with the pipelined iterator model. Instead of single tuples or full columns, the operators exchange data in the form of vectors – small (ca. 100-1000 elements) arrays of input values (see Section 4.2.1.1). These vectors are fully cache-resident, removing the need of expensive memory accesses. Furthermore, relational operators are internally implemented using cache-efficient algorithms.

RAM The buffered disk data is stored in a vertical layout, often compressed, maximizing the amount of useful information that can be kept in memory. Main-memory bandwidth is minimized by only reading relevant attributes and by decompressing the buffered compressed data on the boundary between RAM and cache. Additionally, RAM access is seen in many cases as an input-output operation, and is carried out through explicit memory-to-cache and cache-to-memory routines.

Disk The ColumnBM I/O subsystem of X100 is geared towards efficient sequential data access. To reduce bandwidth requirements, it uses a vertically fragmented data layout, in some cases enhanced with lightweight
data compression (see Chapter 6). Furthermore, it coordinates the scan operators of different queries to maximize data sharing and minimize disk accesses (see Chapter 7). Finally, it minimizes the need for physical data reorganization on updates by using in-memory delta update structures.

The major components of the system, the query execution and storage layers, are discussed in the following sections. Other components typically found in databases, such as the query optimizer or logging and recovery facilities, are not implemented in the current MonetDB/X100 prototype, due to its clear focus on the efficient query execution techniques. These components are in many cases orthogonal to the techniques presented in this thesis, and might become a part of the system in the future.

4.1.1 Query language

MonetDB/X100 uses a simple relational algebra as the query language. The algebra is defined on the physical level of operations, so, for example, different physical types of an aggregation operator are expressed explicitly. A simplified version of the TPC-H query, used as an example in the right-hand side of Figure 4.1, is expressed in SQL as:

```sql
SELECT returnflag, sum(exprice * 1.19) as sum_vat_price
FROM lineitem
WHERE shipdate <= date '1998-09-02'
GROUP BY returnflag
```

This can be translated into the following MonetDB/X100 query:

```sql
HashAggr(
  Project(
    Select(
      Scan([ 'shipdate', 'returnflag', 'exprice' ]),
      <=( shipdate, date('1998-09-02'))),
      [ vat_price = *( exprice, flt('1.19') )],
      [ returnflag ],
      [ sum_vat_price = sum(vat_price) ])
)
```

The MonetDB/X100 version of the full TPC-H Query 1 is presented in Figure 4.2.

Unlike in MIL (MonetDB query language), MonetDB/X100 operators are N-ary, i.e. can have an arbitrary number of input attributes, making the plans highly similar to the ones used by traditional RDBMSs. All the computations, including simple arithmetic, are expressed as functions using prefix notation.
Section 4.1: MonetDB/X100 architecture

![MonetDB/X100 algebra](image)

Figure 4.2: TPC-H Query 1 (see Figure 3.2) in MonetDB/X100 algebra
These functions map directly onto primitives described later. The output attributes are implicitly defined for each operator. For example, Select propagates all the attributes from its child, Project explicitly defines which child attributes are propagated and what new attributes are introduced, and HashAggr returns the input columns used in the group-by list and the results of the aggregate functions.

MonetDB/X100 operators can process two types of input data: dataflows and tables. A dataflow contains tuples delivered in a pipelined fashion. Tables are a special version of dataflows, where the entire input is provided in a single iteration step. This allows operators that e.g. perform random accesses into the data or need to read the same data multiple times. The list of the major operators currently available in MonetDB/X100 is presented in Table 4.1.

4.2 Vectorized in-cache execution model

To provide a high-efficiency query execution engine for data-intensive applications, we propose a new execution model, used in MonetDB/X100, that keeps the benefits of the existing approaches, while avoiding their problems. To maintain scalability, this new model needs to work in a pipelined fashion, avoiding expensive materialization. On the other hand, to achieve high CPU performance, it needs to avoid expensive per-tuple interpretation by processing multiple values at once. This analysis has led to the formulation of an execution model presented in this chapter, based on the concept of vectorized in-cache execution.

4.2.1 Vectorized iterator model

Like in the traditional pipelined model, MonetDB/X100 query plans consist of a tree (or a DAG) of operators. However, instead of single tuples stored in the N-ary format, they operate on vectors of values. Operators provide generic high-level logic, and use primitives – specialized, type-specific functions – for actual data processing. These concepts, visualized in the right-hand side of Figure 4.1 are the basic components of the MonetDB/X100 execution layer.

4.2.1.1 Vectors

Vectors are the basic data manipulation and transfer units in the vectorized execution model. Each vector contains a simple, one-dimensional array of values, similar to the [void, type] BATs found in MonetDB. The number of elements in a vector may vary, and typically ranges from 100 to 1000 tuples.
### Section 4.2: Vectorized in-cache execution model

<table>
<thead>
<tr>
<th>Operator</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic operators</strong></td>
<td></td>
</tr>
<tr>
<td>*Scan</td>
<td>A family of operators providing a dataflow of tuples coming from different sources: ColumnBM storage (ColumnScan), MonetDB BATs (BatScan), simple binary files (RawScan), PAX-formatted [ADHS01] binary files (PaxScan).</td>
</tr>
<tr>
<td>*Save</td>
<td>A family of operators that save the input, fully symmetric to *Scan. Includes ColumnSave (with optional compression), BatSave, RawSave and PaxSave.</td>
</tr>
<tr>
<td>Print</td>
<td>Displays the input values.</td>
</tr>
<tr>
<td><strong>Unary (one input relation) operators</strong></td>
<td></td>
</tr>
<tr>
<td>Select</td>
<td>Returns tuples that match a specified predicate.</td>
</tr>
<tr>
<td>Project</td>
<td>Performs column projection and introduces new columns with values computed from the input columns. Does not perform duplicate elimination.</td>
</tr>
<tr>
<td>HashAggr</td>
<td>Performs aggregation on the input. If the aggregate keys are not provided, global aggregates are computed. If the aggregate functions are not provided, it performs duplicate elimination.</td>
</tr>
<tr>
<td>MergeAggr</td>
<td>Similar to HashAggr, but assumes the input to be clustered by the key values.</td>
</tr>
<tr>
<td>FixedAggr</td>
<td>Similar to MergeAggr, but assumes the input to be clustered in groups of a predefined size.</td>
</tr>
<tr>
<td>TopN</td>
<td>Selects top tuples according to some ordering criteria.</td>
</tr>
<tr>
<td>ExternSort</td>
<td>Performs a partitioned external radix-sort.</td>
</tr>
<tr>
<td><strong>Binary (two input relations) operations</strong></td>
<td></td>
</tr>
<tr>
<td>Hash*</td>
<td>Hash-based in-memory operations. Provide different variants of joins (N-1, N-N, inner, outer etc.) and set operations (union, diff etc).</td>
</tr>
<tr>
<td>Merge*</td>
<td>Similar to Hash*, but assumes both inputs are sorted by the key values.</td>
</tr>
<tr>
<td>FetchJoin*</td>
<td>Joins based on join-indices. Requires a table as the right input.</td>
</tr>
<tr>
<td>CartProd</td>
<td>Produces a Cartesian product of two inputs. Requires a table as the right input.</td>
</tr>
<tr>
<td><strong>Other operations</strong></td>
<td></td>
</tr>
<tr>
<td>Window</td>
<td>Returns tuples from a pre-specified range (counted as the position in the dataflow).</td>
</tr>
<tr>
<td>FlowMat</td>
<td>Materializes an arbitrary dataflow into a table.</td>
</tr>
<tr>
<td>Reuse</td>
<td>Allows multiple consumers to read from the same input data. It adapts to situations when the consumers are not synchronized, and buffers the tuples accordingly.</td>
</tr>
<tr>
<td>Chunk</td>
<td>Creates a dataflow with a new vector size. Additionally condenses the data if the input has a selection vector.</td>
</tr>
<tr>
<td>BEP</td>
<td>Performs Best-Effort Partitioning on the input (see Section 5.4).</td>
</tr>
<tr>
<td>Array</td>
<td>A dataflow generating a sequence of all indices in the N-dimensional space, similar to grid in [BS92].</td>
</tr>
</tbody>
</table>

Table 4.1: Main operators available in the MonetDB/X100 algebra
Vectors are typically transient – during consecutive iterations of a given operator they contain different data. Internally, the pointer to the actual data can be dynamic, allowing e.g. zero-cost access to a large sequential data structure (e.g. a disk page in the buffer pool). For variable-size data types, a vector can have an associated heap structure. This allows temporary storage of e.g. string values.

Vectors are used to pass data between operators, but they are also used heavily inside the operators. For example, the hash-join implementation discussed in Section 5.3 uses vectors to keep the internally computed hash-values of input tuples.

In traditional block-oriented processing, tuples are typically stored in the N-ary form \[PMAJ01\]. The rationale of using vectors holding vertically decomposed data comes from two observations. First, with this solution operations that add or remove columns are trivial and do not need to copy any data. Second, as discussed further in Section 5.2.2, dense data packing in vectors removes the need for tuple navigation, provides better sequential memory access and exposes SIMD opportunities.

### 4.2.1.2 Operators

Operators are the building blocks of the query plan. They operate in a pipelined, pull-driven fashion, as in the Volcano model. The difference is that instead of single tuples, operators exchange data using vectors. A collection of per-attribute vectors represents a set of tuples with values of a given tuple stored at the same position in all vectors. To reduce the need for data copying, it is possible to define that only a subset of the tuples needs to be processed. This is achieved by using an optional selection vector – an array containing offsets of the relevant tuples.

Operators provide generic logic, independent of the input data types, properties etc. To perform the actual work, they use primitive functions, described below. For example, the pseudo code for the `next()` method of the `Select` operator looks like this:

```cpp
int Select::next() {
    while (true) {
        n = child->next(); // see how many tuples are produced by the child
        if (n == EndOfStream)
            return EndOfStream;
        n = condition->evaluate(n);
        if (n > 0) // something was selected
            return n; // return the number of selected tuples
    }
}
```
This logic is very similar to the tuple-at-a-time model (see Section 3.3), but instead of operating on a single tuple, it is vector-based. In this code, condition may be an arbitrary expression that computes the desired predicate. Expressions are components that glue the primitives with their associated input and result vectors. In this case, the result vector of the condition expression is being used as the selection vector of the parent operator. Expressions can be nested – for example, evaluation of condition may trigger the evaluation of its sub-expressions. The leaves in an expression tree can be expressions exported by the child operators, or intermediate expressions produced by a given operator (e.g. discountprice in Figure 4.2).

As mentioned, the operators in MonetDB/X100 are N-ary, allowing an arbitrary number of input attributes. However, the primitives used to perform the operations can only work on a limited combination of input types. As a result, internally, operators need to introduce per-attribute logic. For example, the pseudo code for the next() method of the Project operator is:

```c
int Project::next() {
    n = child->next();
    if (n == EndOfStream)
        return EndOfStream;
    for (i = 0; i < expressions->size(); i++)
        expressions[i]->evaluate(n);
    return n; // return the number of selected tuples
}
```

The operator logic, trivial in case of Project, becomes significantly more complex in operators such as hash-join and merge-join, especially if there are multiple key columns. The problems with providing vectorized versions of such operators, as well as implementation techniques and examples for some typical operations are discussed in Chapter 5.

### 4.2.1.3 Primitives

While operators provide high-level generic logic, primitives are the components that perform all the operations on the actual data. Similar to operators in MIL, primitives are highly specialized for a given task and data type combination. However, the granularity of operation is much smaller. For example, in MIL an operator would perform the entire process of hash-aggregation for the entire column, involving hash number computation, finding a position in a hash table,
updating aggregates etc. In MonetDB/X100, each of these tasks is implemented by one or more primitives, with operator logic gluing them together. As a result, the total number of primitives is even higher than in MonetDB, but they are significantly simpler than the full operators, making the code footprint relatively small. Like MonetDB, MonetDB/X100 relies on the Mx tool \[\text{\cite{KsvdB96}}\] with heavy macro expansion to implement the primitives. This results in a large number of short and efficient routines, usually performing a single loop over the data e.g.:

```c
int map_add_sint_col_sint_val(int n, sint *result, sint *param1, sint *param2) {
    for (int i = 0; i < n; i++)
        result[i] = param1[i] + *param2;
    return n;
}
```

Each primitive is identified with a signature, defining the function of the primitive, its input and output types etc. For example, a primitive adding a signed-integer vector to a constant has a signature `map_add_sint_col_sint_val`, with `col` representing a column of values (vector), and `val` representing a single value (constant). Primitives are organized into groups defining their functionality. For example, “map” primitives produce a single value for each input tuple (e.g. math operations); “select” primitives generate a selection vector according to a predicate; “aggr” primitives take care of aggregation-specific operations etc. This allows determining which primitive needs to be chosen for a specific operation. For example, different primitives that check if float values are greater than a constant will be used if it is used to select tuples in the `Select` operator (`select_gt_flt_col_flt_val`) or if it is used to compute a Boolean value in the `Project` operator (`map_gt_flt_col_flt_val`). Currently, for every signature there is exactly one primitive. However, it is possible to extend this scheme to allow multiple primitive implementations, with one dynamically chosen depending on e.g. a CPU type, similarly as is done in LibOIL \[\text{\cite{Lib}}\]. A choice can also be based on query and data properties, e.g. depending on predicate selectivity, a different `select_*` implementation can be more efficient \[\text{\cite{Ros02}}\].

Another possible extension is to allow compound primitives – primitives that perform complex operations. For example, the computation of the Mahalanobis distance, a performance-critical operation in some multimedia retrieval tasks \[\text{\cite{CvBdV04}}\], can be defined for some cases as follows:

\[
D_M(x, y, \sigma) = \sqrt{\frac{(x - y)^2}{\sigma}}
\]  \hspace{1cm} (4.1)

In MonetDB/X100, this can currently be expressed as an explicit computation:
Section 4.2: Vectorized in-cache execution model

This will construct a tree of expressions, with one primitive for each mathematical operator. In this tree, the intermediate results at all evaluation steps need to be materialized, which introduces a small, yet visible overhead. Another approach is to manually implement a map.mahanalobis primitive containing the entire evaluation, and call it explicitly in the query plan. This can be done either with manual C code or by using a primitive generator based on signature requests, as discussed in [BZN05]. The final option would be to automatically detect often used expression patterns and dynamically compile, link and use the optimized primitives.

MonetDB/X100 primitives, like MonetDB operators, are based on the principle of bulk processing. In one function call, a large set of values is processed, without the need of per-tuple decisions based on data type, input properties etc. This way all the performance benefits of MonetDB discussed in Section 3.4 can be maintained. Additionally, the next section discusses how MonetDB/X100 manages to avoid the in-memory materialization problem that hinders the performance in the original MonetDB model.

The implementation of primitives is a relatively straightforward task for simple operations like projection or selection. However, special care needs to be taken to guarantee their high performance on modern CPUs. The discussion of how this can be achieved even for complex operations is presented in Chapter 5.

### 4.2.2 In-cache execution

MonetDB/X100 primitives are similar to MonetDB operators in many ways: they use highly specialized code, achieve high CPU efficiency, and consume and produce arrays of values. In MonetDB this last property can lead to the materialization of large intermediate results, which results in a significant performance degradation, as discussed in Section 3.4 - the main memory bandwidth is often simply too low to sustain the data hunger of the CPU-efficient primitives. To avoid this problem, MonetDB/X100 exploits the fact that cache-memory bandwidth is significantly higher than that of RAM, and introduces a principle of in-cache execution. The idea is presented in the left-hand side of Figure 4.1: vectors are organized in such a way that their entire memory footprint is small enough to fit in the CPU cache. This allows keeping the materialized results on the CPU die, without the need of expensive writes into main memory.

To demonstrate the importance of keeping the data in the CPU cache, we
analyze the performance of a simple X100 query, equivalent to this SQL statement:

```sql
SELECT l_quantity * l_extendedprice AS netto_value,
       netto_value * l_tax AS tax_value,
       netto_value + tax_value AS total_value
FROM lineitem
```

The simplified plan for this computation is presented in Figure 4.3. It shows that the base columns are read from main memory (storage manager buffers), while the intermediate results stay in the CPU cache. Figure 4.4 demonstrates the speed of the primitives depending on the used vector size. For small vector sizes, the execution time is dominated by the per-vector logic. On the other hand, for large vector sizes, the vectors do not fit in the CPU cache anymore, causing expensive main-memory traffic.

The performance of the individual primitives depends on the location of their input data. As Figure 4.4 shows, the `mul1()` primitive is significantly slower than `add1()` – the reason is that it needs to read a large amount of data from main memory. In this case, with the optimal vector size, it spends 3.5 CPU cycles to add two 4-byte wide values. On a 2.16GHz machine, this results in a memory bandwidth of ca. 4.8 GB/sec. On the other hand, `add1` only spends 0.9 cycles on the equivalent computation (on this platform multiplication and addition execute at the same speed), because it can quickly access all its input data in the CPU cache. The performance of `mul2` lands in between (2.1 cycles), as only one of its inputs is RAM-resident. This experiment demonstrates that even with se-
Section 4.2: Vectorized in-cache execution model

Sequential access keeping the data in-cache can result in a significant performance improvement and explains the performance benefit of MonetDB/X100 over the MonetDB-style processing.

4.2.2.1 Cache interference

Many cache-conscious data processing techniques implicitly assume that during the operation of a given algorithm it has an exclusive ownership of the cache [MBK02]. However, in real-life scenarios, it is often the case that multiple queries are active at the same time in the system, and the execution context of the CPU changes quite frequently. As a result, the cache content useful for one query can be replaced with other data. This problem has been demonstrated e.g. in [CAGM07, Section 8.2.5] where the performance of the hash-join algorithm based on cache-partitioning has been shown to deteriorate significantly (up to 78%) when the cache content is periodically flushed.

To analyze the impact of cache interference on the performance of MonetDB/X100 in-cache processing, two parts of the execution layer need to be considered: the iterator pipeline and the individual operators. The impact on the operators is algorithm-specific and not directly related to the chosen execution model. For example, the cache-partitioned hash-join will suffer from the cache interference problems in all the execution models: tuple-based, column-based, and vector-based. Therefore, we postpone the discussion of cache-conscious operator implementations to Chapter 5 and for now focus on the execution model itself.

An execution context change in the CPU can occur when an application performs a system call, e.g. by requesting an I/O operation. Also, the kernel can decide to preempt the current process, e.g. when an interrupt happens or the quantum of time assigned to this process is finished. In the scenarios MonetDB/X100 is targeted at, the system is typically performing relatively few and large I/Os, hence the interrupts do not occur often. With the typical kernel scheduling frequency in the range of 100 times per second (10ms), it is reasonable to assume that an executing process has an uninterrupted slice of time in the range of 0.1 to 10 ms.

To evaluate the robustness of the vectorized in-cache execution model proposed in MonetDB/X100, we analyze the performance of the TPC-H Query 1 with enforced cache-flushes happening at different intervals. Query 1 is a good candidate to demonstrate the impact on the iterator pipeline itself, as all the operators except for the aggregation are stateless, and the aggregation uses a very small hash-table, as it only computes 4 output values. As a result, only the
vectors containing the data passed between the operators use the CPU cache. In the experiment presented in Figure 4.5, a series of queries is running, and, for part of the queries, a separate program is forcing the cache-flush by touching all the cache lines in a separate cache-sized main-memory area.

As the results show, the slowdown of the queries is in line with the CPU time taken by the flushing program, and not significantly higher. This demonstrates that the vectorized in-cache model is relatively robust to cache interference. One of the reasons the impact of the cache flushing is so marginal, compared to the results from [CAGM07], is that for most operations in this query the data accesses are fully sequential. As a result, if the cache-misses start to occur at the beginning of the vector, the hardware-prefetching mechanisms available in most modern CPUs (see Section 2.2.3) will be triggered and will start loading the remaining part of the data. Additionally, even with a 0.1ms time slice, tens or hundreds of primitives can execute, exploiting the exclusive access to the cache during this period.

4.2.2.2 Vector size and allocation

For high execution efficiency, it is important that the vector sizes in the vectorized execution model are on the one hand as large as possible, to minimize the interpretation overhead, and on the other hand not too large, to fit in the CPU cache. Two aspects of this issue need to be analyzed: how many vectors need to be allocated, and the size of each vector.
In the existing MonetDB/X100 implementation, every primitive uses its input vectors and introduces a new result vector, with vector sizes fixed in the entire query plan. The only optimization currently applied is that if one of the primitive input vectors has only one parent, and shares the same datatype with the result vector, it can be reused. This is similar to the register allocation problem, found in compiler optimization (see e.g. [HKMW66, AC76]). It is possible to see physical vectors as vector registers and then map logical vectors used by the primitives to them. Such an optimization can reduce the number of vectors within a given query.

An interesting point in vector allocation optimization is that different vectors can have different datatype widths. As such, for optimal results, the classical register allocation scheme needs to be extended to be able to e.g. reuse a 64-bit wide vector space as two 32-bit vectors. Another possible optimization exploits the fact that query graphs can often be decomposed into a collection of pipeline-able sub-graphs (separated by e.g. blocking operators). Then, vector allocation for each sub-graph can be performed separately.

Once the number of vectors for a given query sub-graph is known, we can find the vector size by dividing the available D-cache size by the number of vectors. Typically, not the entire cache can be used, as there is space overhead related to operator state, especially for blocking operators with a potentially large state (e.g. hash aggregation). Another problem is related to the trade-offs between targeting different CPU cache levels. While, as presented earlier, L1 can provide much better performance than L2, it can require significantly smaller vectors. The choice should be based on the complexity of the query: for very simple queries L1 should be targeted, and for queries with more vectors, it should be L2.

### 4.2.3 Execution layer performance

To evaluate the performance of the proposed execution model, this section demonstrates the performance of TPC-H Query 1 on a 1GB database running on top of MonetDB/X100 with a varying vector size. The results, presented in Figure 4.6 follow the trend already observed in Figure 4.4: the speed improves very quickly with an increasing vector size, until it reaches the optimum where the vectors still fit in the CPU cache and the interpretation overhead is maximally amortized. Then, once the vectors exceed the cache size, the speed deteriorates again. For comparison, the same figure presents the results for two “traditional” tuple-at-a-time systems (MySQL and a commercial one), MonetDB column-at-a-time processing, and hand-written optimized code. Interest-
Figure 4.6: TPC-H Query 1 performance in X100 with varying vector size (in tuples, horizontal axis, 2005 results from [BZN05])

Interestingly, MonetDB/X100 results with vector size of 1 match almost exactly the performance of the tuple-at-a-time systems. Similarly, with very large vector sizes, MonetDB/X100 execution boils down to the column-at-a-time model, but achieves performance almost two times better than MonetDB. That difference is caused by the cost of after-selection projection of tuple attributes in Query 1 on MonetDB, visible in a series of join operations in Figure 3.6. In MonetDB/X100 this projection is avoided thanks to the selection vector. For a fair comparison, Figure 4.6 additionally includes the MonetDB results without the selection statement – the computation cost for the extra tuples is negligible, as the predicate selects almost all tuples. MonetDB/X100 performance almost exactly coincides with the performance of this approach.

Table 4.2 presents a detailed analysis of Query 1 performance with MonetDB/X100, showing per-primitive and per-operator properties. Since in Query 1 the aggregation produces only 4 tuples, post-aggregation processing time is negligible and is ignored. The results show that the primitives achieve very good per-tuple performance, spending only a few CPU cycles on each operation, and deliver extremely high data throughput (for primitive bandwidth, we count both
### 4.3 Bandwidth-optimized storage

Performance improvements achieved with the vectorized query execution model result in a demand for very high data delivery rates from persistent storage. In memory, MonetDB/X100 can process hundreds of megabytes, or even over a gigabyte, of columnar data on a single CPU core per second, leading to data consumption of 0.1 to 1 bytes per CPU cycle. Providing such data bandwidths, even for a single core system, requires relatively expensive solutions, and scaling for multi-core systems is even more challenging. Additionally, Figures 2.1 and 2.8 demonstrate that CPU performance improves significantly more rapidly than disk bandwidth and latency, suggesting that efficient data delivery will continue.

<table>
<thead>
<tr>
<th>input count</th>
<th>total MB</th>
<th>time (us)</th>
<th>BW MB/s</th>
<th>avg. cycles</th>
<th>X100 primitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.0M</td>
<td>68.7</td>
<td>7294</td>
<td>9418</td>
<td>3.1</td>
<td>select_&lt;date,col,date_val</td>
</tr>
<tr>
<td>5.9M</td>
<td>135.4</td>
<td>11349</td>
<td>11930</td>
<td>4.9</td>
<td>map_-slng_val,slng.col</td>
</tr>
<tr>
<td>5.9M</td>
<td>135.4</td>
<td>12112</td>
<td>11178</td>
<td>5.3</td>
<td>map_*slng,col,slng.col</td>
</tr>
<tr>
<td>5.9M</td>
<td>135.4</td>
<td>11537</td>
<td>11736</td>
<td>5.0</td>
<td>map+_slng_val,slng.col</td>
</tr>
<tr>
<td>5.9M</td>
<td>135.4</td>
<td>9219</td>
<td>14687</td>
<td>4.0</td>
<td>map_*slng,col,slng.col</td>
</tr>
<tr>
<td>5.9M</td>
<td>95.9</td>
<td>6138</td>
<td>15623</td>
<td>2.7</td>
<td>map_uidx,uchr,col</td>
</tr>
<tr>
<td>5.9M</td>
<td>141.1</td>
<td>8508</td>
<td>16584</td>
<td>3.7</td>
<td>map_directgrp_uidx,col,uchr,col,uint_val</td>
</tr>
<tr>
<td>5.9M</td>
<td>135.4</td>
<td>12025</td>
<td>11259</td>
<td>5.2</td>
<td>aggr_count_uidx,col</td>
</tr>
<tr>
<td>5.9M</td>
<td>158.0</td>
<td>15741</td>
<td>10037</td>
<td>6.9</td>
<td>aggr_sum_sint,col_uidx,col</td>
</tr>
<tr>
<td>5.9M</td>
<td>180.6</td>
<td>16588</td>
<td>10887</td>
<td>7.2</td>
<td>aggr_sum_slng,col_uidx,col</td>
</tr>
<tr>
<td>5.9M</td>
<td>180.6</td>
<td>16313</td>
<td>10936</td>
<td>7.2</td>
<td>aggr_sum_slng,col_uidx,col</td>
</tr>
<tr>
<td>5.9M</td>
<td>180.6</td>
<td>16420</td>
<td>10970</td>
<td>7.2</td>
<td>aggr_sum_slng,col_uidx,col</td>
</tr>
<tr>
<td>5.9M</td>
<td>180.6</td>
<td>16575</td>
<td>10875</td>
<td>7.2</td>
<td>aggr_sum_slng,col_uidx,col</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X100 operator</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2363</td>
</tr>
<tr>
<td>6.0M</td>
<td>8244</td>
</tr>
<tr>
<td>5.9M</td>
<td>46709</td>
</tr>
<tr>
<td>5.9M</td>
<td>112699</td>
</tr>
</tbody>
</table>

Table 4.2: TPC-H Query 1 performance trace with MonetDB/X100 (Core2 2.4GHz, SF=1, vector size 1024 tuples, post-aggregation processing ignored)
to be a problem in the future. As a result, software approaches attempting to reduce the data demand from the physical storage become increasingly important.

We address the problem of providing high data bandwidth to the execution layer by introducing a new storage layer, called ColumnBM. It is optimized to fully exploit the available disk bandwidth, and proposes a number of techniques for reducing data volumes that need to be shipped from disk. This section presents the main design choices of this layer, while Chapters 6 and 7 discuss in more details two major optimization techniques ColumnBM introduces: lightweight compression and cooperative scans.

### 4.3.1 Scan-based processing

Database systems are addicted to random disk I/O, caused by non-clustered index lookups, and hardware trends are pushing this model to the limit of sustainability. Foreign-key joins, as well as selection predicates executed using unclustered indices, both may yield large streams of row-IDs for looking up records in a target table. If this target table is large and the accesses are scattered, the needed disk pages will have to be fetched using random disk I/O. To optimize performance, industry-strength RDBMSs make good use of asynchronous I/O to farm out batches of requests over multiple disk drives, both to achieve I/O parallelism between the drives, and to let each disk handle multiple I/Os in a single arm movement (amortizing some access latency).

Massive numbers of unclustered disk accesses occur frequently in benchmarks like TPC-H, and it is not uncommon now to see benchmark configurations that use hundreds or thousands of disks. For example, Table 4.3 shows that four representative TPC-H submissions of even the smallest 100GB data size used an average of 150 disks with total storage capacity of 3.8 terabyte. All these disks are less than 10% full, and the main reason for their high number is to get more disk-arms, allowing for a higher throughput of random I/O requests.

<table>
<thead>
<tr>
<th>#</th>
<th>CPU</th>
<th>RAM</th>
<th>totsize</th>
<th>cost</th>
<th>throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Xeon 3.0GHz dual-core</td>
<td>64GB</td>
<td>4.4TB</td>
<td>47%</td>
<td>19497 10404</td>
</tr>
<tr>
<td>2</td>
<td>Opteron 2GHz</td>
<td>48GB</td>
<td>6.0TB</td>
<td>80%</td>
<td>12941 11531</td>
</tr>
<tr>
<td>4</td>
<td>Xeon 3.0GHz dual-core</td>
<td>32GB</td>
<td>3.2TB</td>
<td>67%</td>
<td>11423 6768</td>
</tr>
<tr>
<td>2</td>
<td>Power5 1.65GHz dual-core</td>
<td>32GB</td>
<td>1.6TB</td>
<td>65%</td>
<td>8415 4802</td>
</tr>
</tbody>
</table>

Table 4.3: Official 2006 TPC-H 100GB results
Note from Table 4.3 that a high number of disks seems especially crucial in the concurrent (5 stream) query scenario. The underlying hardware trends of the past decades, namely sustained exponential growth of CPU power as well as a much faster improvement of disk-bandwidth than of I/O latency, are expected to continue. Thus, to keep each next CPU generation with more cores busy, the number of disks will need to be doubled to achieve system balance.

This exponential trend is clearly unsustainable, and one can argue that in the real world (i.e. outside manufacturer benchmarking projects) it is already no longer being sustained, and database servers are often configured with fewer disks than optimal. The main reason for this is cost, both in terms of absolute value of large I/O subsystems (nowadays taking more than two thirds of TPC-H benchmark systems cost, see Table 4.3), but also maintenance costs. In a multi-thousand disk configuration, multiple disks are expected to break each day [SG07], which implies the need for full-time attendance by a human system administrator.

We argue that the only way to avoid efficiency problems caused by random I/O is to rely more on sequential scans of tables or clustered indices, which depend on disk bandwidth rather than latency. Achieving scan-mostly data plans is possible thanks to a number of techniques:

1. storing relations redundantly in *multiple orders* [AMDM07], such that more query patterns can use a clustered access path. To avoid the cost of updating multiple such tables, updates in such systems are buffered in RAM in differential lists and are dynamically merged with persistent data.

2. exploiting *correlated orderings*. In MonetDB/X100, a min- and max-value is kept for each column per large disk block (see Section 4.3.3). Such meta-data, similar to “small materialized aggregates” [Moe98] and also found e.g. in the Netezza system as “zonemaps” [Net], allows avoiding reading unneeded blocks during an (index) scan, even if the data is not ordered on that column, but on a correlated column. For example, in the `lineitem` table in the TPC-H schema it allows avoiding I/O for almost all non-relevant tuples in range selections on any date column in the TPC-H schema, as dates in the fact tables of a data warehouse tend to be highly correlated. This technique can sometimes result in a scan-plan that requires a set of non-contiguous table ranges.

3. using *multi-table clustering* or *materialized views*, to exploit index range-scans even over foreign-key joins.
4. exploiting large RAMs to fully buffer small (compressed) relations, e.g. the dimensions of a star schema.

5. reducing scan I/O volume by offering column storage (DSM) as an option [ZHNB06, SAB+05] to avoid reading unused columns. The same remark as made in (1) on handling updates applies here.

6. using data compression, where the reduced I/O cost due to size reduction outweighs the CPU cost of decompression. It has been shown that with column storage, (de-)compression becomes less CPU intensive and achieves better compression ratios (Chapter 6, [AMP06]).

After applying (1-2), data warehousing queries use (clustered index) scans for their I/O, typically selecting ranges from the fact tables. Other table I/O can be reduced using (3-4) and I/O bandwidth can be optimized to its minimum cost by (5-6).

### 4.3.2 ColumnBM storage format

ColumnBM stores data in fixed-size blocks. To reduce the cost of moving the disk head when performing multiple scan operations at the same time (which happens in DSM even with a single active scan operator), the blocks from the same column are grouped into large chunks. The size of a chunk is large enough to amortize the disk latency, as presented in Figure 2.9. When performing a scan, if the system detects multiple blocks from a single chunk are useful for a given query, they are all read in one operation. Similarly, when saving the data, the writing process for a block can be delayed to possibly save multiple blocks in one go.

In the storage layer, most database systems follow either NSM or DSM as their data organization format, as discussed in Section 3.1.1. A choice of the storage model has an impact not only on the I/O performance, but also on the query processing efficiency. If the storage format does not match the internal execution layer format, an extra translation phase is necessary. This reduces the performance, as discussed in Section 3.5.7 for the [HLAM06] results.

The PAX storage format [ADHS01], where data is organized in columns, like in DSM, but all attributes are grouped in a single I/O block, like in NSM, has demonstrated that using different data representations on disk and in memory can be beneficial. ColumnBM applies this idea by keeping in-block data in the columnar format, matching the MonetDB/X100 execution layer. However,
it follows and extends the PAX approach by allowing arbitrary vertical fragmentation of a table on disk, similar to the data-morphing technique \cite{HP03} for in-memory data. In this approach, a table is decomposed into non-overlapping groups of attributes, with each group stored as PAX disk blocks. As Figure 4.7 demonstrates, both DSM and NSM can be expressed as special cases, allowing having a uniform infrastructure for different storage layouts.

The flexible partitioning in ColumnBM allows application-specific data organization. For large scans involving a subset of columns, DSM can be applied. For tasks that read relatively small numbers of tuples and use most of the columns, NSM is a better choice. An example of such an application is inverted-list processing in information retrieval \cite{HZdVB06}, where in each query a subset of a three-column table (term-doc-score) is scanned for each term. Since many terms have a relatively small number of occurrences, the scan times for these are dominated by the random-access cost. Arbitrary partitioning is applicable if the access pattern of an application is well known – then attributes often accessed together can be grouped on disk.

Having both PAX and DSM facilities also allows applying the fractured mirrors \cite{RDS02} storage strategy. In this approach, the data is stored in two copies: one NSM (PAX in MonetDB/X100) and one in DSM, distributed among multiple disks. This allows both efficient scan-based processing, using the DSM copy, and fast random-access lookups, using the NSM/PAX copy. This functionality, useful for query execution, is also very important for the update strategies currently being developed for ColumnBM, as discussed in Section 4.3.4.

### 4.3.3 Index structures

To allow efficient range lookups on various attributes, databases typically apply non-clustered B-tree indices. However, if a given index does not cover all the query attributes, extra random accesses to the base table are necessary. This makes this approach useful only if a very small percentage of tuples is selected. To increase the applicability of scan-based approaches, we allow extra indices to carry an arbitrary collection of base table columns, allowing satisfying more queries. This is similar to the projections proposed in \cite{SAB+05}, but ColumnBM always keeps one full copy of a table with all the columns sorted in the same order.

To reduce the volume of scanned data even in the absence of ordering on a range-predicate attribute, ColumnBM provides per-block value-range information, which is a simple case of small materialized aggregates \cite{Moe98}. This information is especially useful with correlated columns, as is the case e.g. in
different date attributes in the “lineitem” table of the TPC-H benchmark. Then, if a table is sorted on one date attribute, and a query has a range predicate on a different date column, per-block statistics can be used to filter-out many disk blocks.

Another index structure used in ColumnBM are join-indices [Val87], which provide direct access to the matching tuple in the parent relation for each tuple in the child table in a foreign-key relationship. This can significantly reduce the join cost when the inner relation is a rarely-updated, memory-resident table, as is often the case with the dimension tables in typical star or snowflake data warehousing schemas.

4.3.4 Updates

ColumnBM provides a novel approach to handling updates which allows combining high performance of scan-based queries with high update throughput. This functionality is the topic of ongoing research [HNZB08], and this section provides only an overview of the design goals and features.

4.3.4.1 Delta-based updates

Most database systems perform updates by directly modifying related disk pages. A stream of such updates results in a high load of random disk accesses in NSM storage, which is even higher in DSM, where multiple columns
need to be accessed. Large disk blocks and in-block compression, often used in analytics-oriented systems, make in-place updates even more expensive. To overcome this problem, MonetDB/X100 uses in-memory deltas, similar to differential files [SL76]. In this approach, updates are buffered in a delta structure, which is transparently merged during query processing. These structures are periodically merged into persistent storage, but the updates are visible even before this process occurs.

Multiple delta structures can be stacked to allow transaction isolation. For example, Figure 4.8 shows a simple two-level approach to implement the read-committed isolation level. Here, the active transaction reads previous updates from a global read-delta, but writes all changes to its private write-delta, keeping them invisible to other users. Finally, when the transaction commits, the write-delta is merged into the read-delta, making the changes visible to other transactions.

### 4.3.4.2 Positional delta trees

In the previous delta-based approaches [SL76, RDS02], updates in the delta structures are stored in the same order as the underlying table, and the update merging is performed by looking at the values of the key attributes. This process requires all the key attributes of a table to be scanned, even if a query does not need to process all of them, potentially reducing the performance advantage provided by DSM. Additionally, finding a position where the deltas need to be applied in the persistent data stream requires a relatively expensive value-based search, similar to a single phase in merge-sort.

An alternative to this value-based approach is a newly introduced Positional Delta Tree (PDT) structure [HNZB08, Section II-C]. PDTs store updates ordered by their position in the table, dynamically adjusting these positions with incoming updates. Having positional access allows the merging process to be performed even if not all key attributes are fetched from the disk, allowing better performance in column stores. Additionally, merging becomes faster since PDTs provide the direct location in the data stream where the updates need to be applied.

PDTs introduce one challenge not present in the value-based approaches. When a tuple is inserted, its exact position needs to be found in the current image of the data, which might consist of both persistent storage and previous updates stored in a PDT. With DSM, if the table key consists of multiple columns, this may require multiple disk accesses for each tuple. To avoid this extra cost, ColumnBM update facilities exploit an optional fractured mirrors
layout discussed in Section 4.3.2. In this approach, an extra NSM/PAX copy of the table is stored, allowing finding the tuple position with just one I/O access. This optimization is especially important when multiple indices or table replicas are present. In such scenarios, any update to the base table needs to be propagated to all the auxiliary structures. This requires getting all the key attributes of these structures from the base table, which, again, may cause multiple I/Os in the plain DSM storage scheme, and can be avoided using the NSM/PAX copy.

4.4 Conclusions

This chapter presented the overview of the MonetDB/X100 system architecture, focusing on the execution and storage layers. In the execution layer a new vectorized in-cache execution model was proposed. By minimizing the interpretation overhead found in most database systems and efficiently exploiting modern CPU features, it provides execution performance often orders of magnitude higher than existing solutions. This model is further investigated in Chapter 5.

The high performance of the execution layer requires new approaches in the storage layer that would scale this performance to disk-resident datasets. The proposed ColumnBM system achieves this by introducing a number of strategies for analytical queries: vertical partitioning, scan-optimized storage and index structures, as well as efficient updates. Two major optimizations in this component, lightweight compression and cooperative scans are discussed in more detail in Chapters 6 and 7.