Understanding, modeling, and improving main-memory database performance

Manegold, S.

Citation for published version (APA):
Chapter 2

Preliminaries

Models play a very important role in science and research. Usually, a model is seen as an abstract image or description of a certain part of “the real world” that helps us to analyze and/or better understand this part of the real world. As we saw in Chapter 1, database systems rely on cost models to do efficient and effective query optimization. In this chapter, we first discuss some fundamentals of cost models and informally introduce the terminology we use throughout this thesis. Then, we briefly review the role of database cost models in literature and discuss some approaches in more detail. The last part of this chapter gives a concise overview of our main-memory DBMS prototype Monet, which we use as implementation and validation platform throughout this thesis.

2.1 Cost Models

We indicated in Chapter 1 that different execution plans require different amounts of effort to be evaluated. The objective function for the query optimization problems assigns every execution plan a single non-negative value. This value is commonly referred to as costs in the query optimization business.

2.1.1 Cost Components

In the Introduction, we mentioned already briefly that we consider cost models to be made up of three components: logical costs, algorithmic costs, and physical costs. In the following, we discuss these components in more detail.

2.1.1.1 Logical Costs / Data Volume

The most important cost component is the amount of data that is to be processed. Per operator, we distinguish three data volumes: input (per operand), output, and temporary data. Data volumes are usually measured as cardinality, i.e., number of tuples. Often, other units such as number of I/O blocks, number of memory pages, or
total size in bytes are required. Provided that the respective tuple sizes, page sizes, and block sizes are known, the cardinality can easily be transformed into the other units.

The amount of input data is given as follows: For the leaf nodes of the query graph, i.e., those operations that directly access base tables stored in the database, the input cardinality is given by the cardinality of the base table(s) accessed. For the remaining (inner) nodes of the query graph, the input cardinality is given by the output cardinality of the predecessor(s) in the query graph.

Estimating the output size of database operations — or more generally, their selectivity — is anything else but trivial. For this purpose, DBMSs usually maintain statistic about the data stored in the database. Typical statistics are

- cardinality of each table,
- number of distinct values per column,
- highest / lowest value per column (where applicable).

Logical cost functions use these statistics to estimate output sizes (respectively selectivities) of database operations. The simplest approach is to assume that attribute values are uniformly distributed over the attribute’s domain. Obviously, this assumption virtually never holds for “real-life” data, and hence, estimations based on these assumption will never be accurate. This is especially severe, as the estimation errors compound exponentially throughout the query plan [IC91]. This shows, that more accurate (but compact) statistics on data distributions (of base tables as well as intermediate results) are required to estimate intermediate results sizes.

The importance of statistics management has led to a plethora of approximation techniques, for which [GM99] have coined the general term “data synopses”. Such techniques range from advanced forms of histograms (most notably, \(V\)-optimal histograms including multidimensional variants) [Poo97, GMP97, JKM*98, IP99] over spline synopses [KW99, KW00], sampling [CMN99, HNSS96, GM98], and parametric curve-fitting techniques [SLRD93, CR94] all the way to highly sophisticated methods based on kernel estimators [BKS99] or Wavelets and other transforms [MVW98, VW99, LKC99, CGRS00].

A logical cost model is a prerequisite for the following two cost components. In this work, we do not analyze logical cost models in more detail, but we assume that a logical cost model is available.

2.1.1.2 Algorithmic Costs / Complexity

Logical costs only depend on the data and the query (i.e., the operators’ semantics), but they do not consider the algorithms used to implement the operators’ functionality. Algorithmic costs extend logical costs by taking the properties of the algorithms into account.

A first criterion is the algorithm’s complexity in the classical sense of complexity theory. Most unary operator are in \(O(n)\), like selections, or \(O(n \log n)\), like sorting; \(n\) being the input cardinality. With proper support by access structures like indices
or hash tables, the complexity of selection may drop to $O(\log n)$ or $O(1)$, respectively. Binary operators can be in $O(n)$, like a union of sets that does not eliminate duplicates, or, more often, in $O(n^2)$, as for instance join operators.

More detailed algorithmic cost functions are used to estimate, e.g., the number of I/O operations or the amount of main memory required. Though these functions require some so-called "physical" information like I/O block sizes or memory page sizes, we still consider them algorithmic costs and not physical cost, as these informations are system specific, but not hardware specific. The standard database literature provides a large variety of cost formulas for the most frequently used operators and their algorithms. Usually, these formulas calculate the costs in term of I/O operations as this still is the most common objective function for query optimization in database systems. We refer the interested reader, e.g., to [KS91, EN94, AHV95, GMUW02].

2.1.1.3 Physical Costs / Execution Time

Logical and algorithmic costs alone are not sufficient to do query optimization. For example, consider two algorithms for the same operation, where the first algorithm requires slightly more I/O operations than the second, while the second requires significantly more CPU operations than the first one. Looking only at algorithmic costs, both algorithms are not comparable. Even assuming that I/O operations are more expensive than CPU operations cannot in general answer the question which algorithm is faster. The actual execution time of both algorithms depends on the speed of the underlying hardware. The physical cost model combines the algorithmic cost model with an abstract hardware description to derive the different cost factors in terms of time, and hence the total execution time. A hardware description usually consists of information such as CPU speed, I/O latency, I/O bandwidth, and network bandwidth. The next section discusses physical cost factors on more detail.

2.1.2 Cost Factors

In principle, physical costs are considered to occur in two flavors, temporal and spatial. Temporal costs cover all cost factors that can easily be related to execution time, e.g., by multiplying the number of certain events with their respective cost in terms of some time unit. Spatial costs contain resource consumptions that cannot directly (or not at all) be related to time. In the following, we briefly describe the most prominent cost factors of both categories.

2.1.2.1 Temporal Cost Factors

As indicated above, physical costs are highly related to hardware. Hence, it is only natural that we distinguish different temporal cost factors according to the respective hardware components involved.

**Disk-I/O** This is the cost of searching for, reading, and writing data blocks that reside on secondary storage, mainly on disk. In addition to accessing the database
files themselves, temporary intermediate files that are too large to fit in main memory buffers and hence are stored on disk also need to be accessed. The cost of searching for records in a database file or a temporary file depends on the type of access structures on that file, such as ordering, hashing, and primary or secondary indexes. I/O costs are either simply measured in terms of the number of block-I/O operations, or in terms of the time required to perform these operations. In the latter case, the number of block-I/O operations is multiplied by the time it takes to perform a single block-I/O operation. The time to perform a single block-I/O operation is made up by an initial seek time (I/O latency) and the time to actually transfer the data block (i.e., block size divided by I/O bandwidth). Factors such as whether the file blocks are allocated contiguously on the same disk cylinder or scattered across the disk affect the access cost. In the first case (also called sequential I/O), I/O latency has to be counted only for the first of a sequence of subsequent I/O operations. In the second case (random I/O), seek time has to be counted for each I/O operation, as the disk heads have to be repositioned each time.

**Main-Memory Access** These are the costs for reading data from or writing data to main memory. Such data may be intermediate results or any other temporary data produced/used while performing database operations.

Traditionally, memory access costs were ignored in database systems. The reason for this was, that they were completely overshadowed by the dominating I/O costs in disk-based systems. As opposed to I/O costs, memory access cost were considered uniform, i.e., independent of both the physical locality and the physical order of accesses. This assumption was mainly true on the hardware in the 80’s. Hence, main-memory DBMSs considered memory access costs to be included in the CPU costs.

In this thesis, we demonstrate that due to recent hardware trends, memory access costs have become a highly significant cost factor. Furthermore, we show that memory access on modern hierarchical memory systems depicts similar cost-related characteristics as I/O, i.e., we need to consider both latency and bandwidth, and we need to distinguish between sequential and random access patterns.

**Network Communication** In centralized DBMSs, communication costs cover the costs of shipping the query from the client to the server and the query’s result back to the client. In distributed, federated, and parallel DBMSs, communication costs additionally contain all costs for shipping (sub-)queries and/or (intermediate) results between the different hosts that are involved in evaluating the query.

Also with communication costs, we have a latency component, i.e., a delay to initiate a network connection and package transfer, and a bandwidth component, i.e., the amount of data that can be transfer through the network infrastructure per time.

**CPU Processing** This is the cost of performing operations such as computations on attribute values, evaluating predicates, searching and sorting tuples, and merging tuples for join. CPU costs are measured in either CPU cycles or time. When using CPU cycles, the time may be calculated by simply dividing the number of cycles by
the CPU's clock speed. While allowing limited portability between CPUs of the same kind, but with different clock speeds, portability to different types of CPUs is usually not given. The reason is, that the same basic operations like adding two integers might require different amounts of CPU cycles on different types of CPUs.

Traditionally, CPU costs also cover the costs for accessing the respective data stored in main memory. However, we treat memory access costs separately.

Summarizing, we see that temporal cost are either caused by data access and/or data transfer (I/O, memory access, communication), or by data processing (CPU work).

2.1.2.2 Spatial Cost Factors

Usually, there is only one spatial cost factor considered in database literature: memory size. This cost it the amount of main memory required to store intermediate results or any other temporary data produced/used while performing database operations.

Next to not (directly) being related to execution time, there is another difference between temporal and spatial costs that stems from the way they share the respective resources. A simple example shall demonstrate the differences. Consider to operations or processes each of which consumes 50% of the available resources (i.e., CPU power, I/O-, memory-, and network bandwidth). Further, assume that when run one at a time, both tasks have equal execution time. Running both tasks concurrently on the same system (ideally) results in the same execution time, now consuming all the available resources. In case each individual process consumes 100% of the available resources, the concurrent execution time will be twice the individual execution time. In other words, if the combined resource consumption of concurrent tasks exceed 100%, the execution time extends to accommodate the excess resource requirements. With spatial cost factors, however, such "stretching" is not possible. In case two tasks together would require more than 100% of the available memory, they simply cannot be executed at the same time, but only after another.

2.1.3 Types of (Cost) Models

According to their degree of abstraction, (cost) models can be classified into two classes: analytical models and simulation models.

Analytical Models In some cases, the assumptions made about the real system can be translated into mathematical descriptions of the system under study. Hence, the result is a set of mathematical formulas. We call this an analytical model. The advantage of an analytical model is that evaluation is rather easy and hence fast. However, analytical models are usually not very detailed (and hence not very accurate). In order to translate them into a mathematical description, the assumptions made have to be rather general, yielding a rather high degree of abstraction.
Simulation Models  Simulation models provide a very detailed and hence rather accurate description of the system. They describe the system in terms of (a) simulation experiment(s) (e.g., using event simulation). The high degree of accuracy is charged at the expense of evaluation performance. It usually takes relatively long to evaluate a simulation base model, i.e., to actually perform the simulation experiment(s). It is not uncommon, that the simulation actually takes longer than the execution in the real system would take.

Simulation models are usually used in scenarios where a very detailed analysis as close as possible to the real system is required, but the actual system in not (yet) available. The most prominent example is processor development. The design of new CPUs is evaluated via exhaustive simulation experiments, first, to ensure the correctness and analyze the (expected) performance. The reason is, that producing functional prototypes in an early stage of the development process would be too expensive.

In database query optimization, though it would appreciate the accuracy, simulation models are not feasible, as the evaluation effort is far to high. Query optimization requires that costs of numerous alternatives are evaluated and compared as fast as possible. Hence, only analytical cost models are applicable in this scenario.

2.1.4 Architecture and Evaluation of Database Cost Models

The architecture and evaluation mechanism of database cost models is tightly coupled to the structure of query execution plans. Due to the strong encapsulation offered by relational algebra operators, the cost of each operator, respectively each algorithm, can be described individually. For this purpose, each algorithm is assigned a set of cost functions that calculate the three cost components as described above. Obviously, the physical cost functions depend on the algorithmic cost functions, which in turn depend on the logical cost functions. Algebraic cost functions use the data volume estimations of the logical cost functions as input parameters. Physical cost functions are usually specializations of algorithmic cost functions that are parameterized by the hardware characteristics.

The cost model also defines how the single operator costs within a query have to be combined to calculate the total costs of the query. In traditional sequential DBMSs, the single operators are assumed to have no performance side-effects on each other. Thus, the cost of a QEP is the cumulative cost of the operators in the QEP [SAC+79]. Since every operator in the QEP is the root of a sub-plan, its cost includes the cost of its input operators. Hence, the cost of a QEP is the cost of the topmost operator in the QEP. Likewise, the cardinality of an operator is derived from the cardinalities of its inputs, and the cardinality of the topmost operator represents the cardinality of the query result.

In non-sequential (e.g., distributed or parallel) DBMSs, this subject is much more complicated, as more issues such as scheduling, concurrency, resource contention, and data dependencies have to be considered. For instance, in such environments, more than one operator may be executed at a time, either on disjoint (hardware) resources, or (partly) sharing resources. In the first case, the total cost (in terms of time) is
calculated as the maximum of the costs (execution times) of all operators running concurrently. In the second case, the operators compete for the same resources, and hence mutually influence their performance and costs. More sophisticated cost function and cost models are required here to adequately model this resource contention [LTS90, LST91, SE93, SYT93, LVZ93, ZZBS93, SHV96, SF96, GHK92].

2.2 Logical Cost Models / Estimation

Most DBMSs make certain assumptions on the underlying data in order to perform inexpensive estimations. Christodoulakis studied the implications of various common assumptions on the performance of databases [Chr83, Chr84]. The main set of assumptions studied by him are:

**Uniformity of attribute values:** All possible values of an attribute have the same frequency in the data distribution.

**Attribute Independence:** The data distributions of all attributes in a relation are independent of each other.

**Uniformity of queries:** Queries refer attribute values with equal frequencies.

**Constant number of records per block:** Each block of the file contains the same number of tuples.

**Random placement:** Each record of the file has the same probability to qualify in a query, regardless of its placement among the pages of secondary storage.

He also showed that the expected cost of a query estimated using these assumptions is an upper bound on the actual expected cost. He demonstrated that existing systems using these assumptions tend to utilize expensive query evaluation strategies and that non-uniformity, non-independence, and non-random placement could be exploited in database design in order to reduce the system cost. In addition to providing such extensive motivation for better estimation techniques, his work also pioneered in the usage of several mathematical techniques such as *Schur concavity* [MO79] in database performance evaluation.

The System-R optimizer, assumed that the underlying data is uniform and independent ([SAC+79]). As a result, only the number of tuples and the lowest and highest values in each attribute are stored in the system catalogs, and it is assumed that all possible values between the two extremes occur with the same probability. Hence, very few resources are required to compute, maintain, and use these statistics. In practice, though, these assumptions rarely hold because most data tends to be non-uniform and has dependencies. Hence, the resulting estimates are often inaccurate. This was formally verified in the context of query result size estimation by Ioannidis and Christodoulakis in [IC91]. In their work they proved that the worst case errors incurred by the uniformity assumption propagate exponentially as the number of joins in the query increases. As a result, except for very small queries, errors may become
extremely high, resulting in inaccurate estimates for result sizes and hence for the execution costs.

Several techniques have been proposed in the literature to estimate query result sizes, most of them contained in the extensive survey by Mannino, Chu, and Sager [MCS88]. The broad classes of various estimation techniques are described in the following sections.

2.2.1 Sampling-based Techniques

These techniques compute their estimates by collecting and processing random samples of the data, typically at query optimization time. There has been considerable amount of work done in sampling-based techniques for result size estimation [Ant92, ASW87, CMN99, GGMS96, HNSS96, HS92, HS95, LNS90, SN92, OR86, LS95]. Since these techniques do not rely on any precomputed information about the data, they are not affected by database updates and do not incur storage overheads. Another advantage of these techniques is their probabilistic guarantees on the accuracy of the estimates. Some of the undesirable properties of the sampling-based techniques are: (1) they incur disk I/Os and CPU overheads during query optimization, and (2) the information gathered is not preserved across queries and hence these techniques may incur the costs repetitively. When a quantity needs to be estimated once and with high accuracy in the presence of updates, the sampling technique works very well (e.g., by a query profiler). To overcome point (2), techniques for incremental maintenance of random samples have been developed in recent works [GMP97, GM98].

Another weak point of sampling is that the relations which are to be sampled have to be available. In other words, sampling can only be applied to base table or completely calculated intermediate results. Propagating samples through the operators of a complex query is generally not possible, especially with joins. These problems have been analyzed in detail in [CMN99, AGPR99, GGMS96].

2.2.2 Parametric Techniques

These techniques approximate the actual data distribution by a parameterized mathematical distribution, such as the uniform distribution [SAC+79], multivariate normal distributions or Zipf distributions [Chr83]. The parameters for these distributions are obtained from the actual data distributions, and the accuracy of this approximation depends heavily on the similarity between the actual and parameterized distributions. The main advantage of this approach is the small storage overhead involved and the insignificant run-time costs. On the other hand, real data often does not resemble any simple mathematical distribution and hence such estimations may cause inaccuracies in estimates. Also, since the parameters are precomputed, this approach may incur additional errors if the database is updated significantly. Variants of this approach are the algebraic techniques, where the actual data distribution is approximated by a polynomial function. The coefficients of this function are determined using regression techniques [SLRD93]. A promising algebraic technique was proposed calling for
adaptively approximating the distribution by a six-degree polynomial, whose coefficients are varied dynamically based on query feedback [CR94]. Some of the problems associated with the algebraic techniques are the difficulties in choosing the degree of the polynomial function and uniformly handling result size estimates for operators other than simple selection predicates. On the other hand, the positive results obtained in the work of Wei Sun et al. [SLRD93] on algebraic techniques indicates their potential applicability.

### 2.2.3 Probabilistic Counting Techniques

These techniques have been applied in the contexts of estimating the number of unique values in the result of projecting a relation over a subset of attributes ([GG82, FM85, SDNR96]). The technique for estimating the number of distinct values in a multi-set, proposed by Flajolet and Martin [FM85] makes an estimate during a single pass through the data and uses a small amount of fixed storage. Shukla et al. applied this technique in estimating the size of multidimensional projections (the cube operator) [SDNR96]. Their experiments have shown that these techniques can provide more reliable and accurate estimates than the sampling-based techniques [SDNR96]. The applicability of these techniques to other operators is still an open issue.

### 2.2.4 Non-parametric (Histogram-based) Techniques

These techniques approximate the underlying data distribution using precomputed tabular information (histograms). They are probably the most common techniques used in practice (e.g., they are used in DB2, Informix, Ingres, Microsoft SQL-Server, Oracle, Sybase, Teradata). Since they are precomputed, they may incur errors in estimation if the database is updated and hence require regular re-computation.

Most of the work on histograms is in the context of single operations, primarily selections. Specifically, Piatetsky-Shapiro and Connell dealt with the effect of histograms on reducing the error for selection queries [PSC84]. They studied two classes of histograms: *equi-width* histograms and *equi-depth* (or *equi-height*) histogram [Koo80]. Their main result showed that equi-width histograms have a much higher worst-case and average errors for a variety of selection queries than equi-depth histograms. Muralikrishna and DeWitt [MD88] studied techniques for computing and using multi-dimensional equi-depth histograms. By building histograms on multiple attributes together, their techniques were able to capture dependencies between those attributes. Several other researchers have dealt with "variable-width" histograms for selection queries, where the buckets are chosen based on various criteria [Koo80, KK85, MK88]. Koo's thesis [Koo80] contains extensive information on using histograms inside an optimizer for general queries and the concept of variable-width histograms. The survey by Mannino, Chu, and Sager [MCS88] contains various references to work in the area of statistics on choosing the appropriate number of buckets in a histogram for sufficient error reduction. That work deals primarily with selections as well. Histograms for single-join queries have been minimally studied and then again without emphasis on optimality [Chr83, Koo80, MK88]. Probably the earliest
efforts at studying optimality of histograms for result size estimation of join operators are those of Ioannidis and Christodoulakis [IC93, Ioa93]. They introduce two new classes of histograms, *serial* and *end-biased* histograms, and show that certain types of these classes, so called *V-optimal*(F,F) histograms, are optimal for worst-case errors of tree equality-join and selection queries. Practicality issues in computing the optimal histograms were not addressed in their work.

Some of the limitations of earlier work on histograms are as follows. First, they were mostly restricted to estimating the result sizes of a few operators such as selections and equi-joins. Second, barring the study by Ioannidis and Christodoulakis, most of the earlier work on histograms has been empirical in nature with almost no effort to identify optimal histograms. Finally, the computation techniques for the new classes of histograms proposed by [Ioa93] were too expensive to be of use in practice. Also, being restricted to predicates on single attributes, most of the earlier work on histograms did not deal with the effects of correlation between attributes from the same relation. Some of the work that did consider multiple attributes together [Ioa93] assumes that multi-dimensional histograms can be built in practice, but do not fully explore the practicality issues involved.

The work by Ioannidis and Poosal [IP95] marks probably the first effort to find a compromise between optimality and practicality of histograms. The work is continued and extended in [PIHS96] and [JKM'98].

There is lots of on-going work on and around the use of histograms in database query evaluation. Some recent issues are new techniques how to construct and maintain histograms efficiently [GMP97, CMN98, MVW98, AC99], and the use of histograms for approximate query answering [PGI99].

### 2.3 I/O-based Cost Models

Physical cost functions belong to the core of proprietary code of a database vendor. Their design, accurate tuning, and alignment with all other database components requires high level of expertise and knowledge of both hardware and database components. Hence, the vendors keep their physical cost functions as precious secrets.

Early work on System-R [SAC'79] uses a cost function balancing both factors I/O and CPU using a constant weight, a factor difficult to determine in practice. Moreover, given the discrepancy in I/O and CPU cost granularity, i.e., microseconds versus milliseconds, the former has become the prime factor in choosing a query execution plan.

A subsequent study on System-R* [ML86] identified that in addition to the physical and statistical properties of the input data streams and the computation of selectivity, modeling buffer utilization plays a key role in accurate estimation. This requires using different buffer pool hit ratios depending on the levels of indexes as well as adjusting buffer utilization by taking into account properties of join methods, e.g., a relatively pronounced locality of reference in an index scan for indexed nested loop join [GLSW93].

With I/O being the dominant cost factor, database research has developed various
techniques to reduce the number of I/O operations needed to answer a query. Two of the most prominent of these techniques are in-memory buffers to cache frequently access pages of the database relations and indices to access a fraction of a table, e.g., as requested by a selection predicate, without the need to scan the whole table. While significantly improving database performance, these techniques make cost estimation more complicated, and it becomes more difficult to predict the exact number of I/O operations that will be needed to answer a query. A number of research works has been devoted to analyzing and predicting the impact of indices, buffer pools, and various buffer replacement strategies on the number of I/O operations, e.g., [CD85, SS86, Sac87, ML89, CS89, DYC95].

Numerous further I/O-based cost models appear in a database literature. However, in most cases, the cost models themselves are not the (primary) subject. Rather, they are just presented as necessary tools for query optimization. Though sharing some commonalities, most physical cost models are specific for the respective DBMS, its architecture, algorithms, data structures, and the hardware platform it runs on.

2.4 Main-Memory Cost Models

Relatively little work has been done on modeling of the performance cost of main-memory DBMSs (MM-DBMSs). Early work on the design of database machines provides hints on the interdependencies of algorithms and memory access [Ozk86, Su88, BF89], but this research track has long been abandoned. This can partly be attributed to a lack of need, as use of MM-DBMS techniques have since been restricted to areas like real-time database systems (e.g., telecom switching, financial trading) that required relatively simple queries; say a hash-lookup in a single table.

In recent database literature, mainly the work around two research prototypes, IBM’s office-by-example (OBE) and HP’s Smallbase, has dealt with the issue of query optimization and cost modeling in main-memory environments.

2.4.1 Office-By-Example (OBE)

Whang and Krishnamurthy [WK90] discuss query optimization techniques applied in IBM’s research prototype office-by-example (OBE). OBE is a memory-resident domain relational calculus database system that extends the concepts of query-by-example (QBE). To enable cost-based query optimization, they present a complete cost model. Due to the assumption that the data being processed is resident in main memory, the traditional database cost factor, I/O access, becomes obsolete. Rather, CPU computation cost now becomes dominant. Modeling CPU costs, however, is very difficult as too many parameters, like the software design, the hardware architecture, and even programming styles, may affect the CPU computation costs. A detailed analysis of larger systems to count the CPU cycles is virtually impossible. The solution that Whang and Krishnamurthy propose is to use an approach using both experimental and analytical methods. First, they identify the system’s bottlenecks using an execution analyzer / profiler. Only bottlenecks will be used to model the system’s CPU cost.
To prevent the cost model from drifting frequently due to changes in the program, the bottlenecks are improved as much as possible with reasonable effort. The next step is then to find, by experiments, relative weights of different bottlenecks and to determine their unit costs. Finally, they develop comprehensive cost formulas based on these unit costs.

For OBE, Whang and Krishnamurthy identified the following bottlenecks and measured the respective unit costs:

1. evaluating the expressions involved in predicates (unit cost = $C_1$);
2. comparison operations needed to finally determine the outcome of predicates (unit cost = $C_4$);
3. retrieving a tuple from a (memory-resident) relation (unit cost = $C_3$);
4. unit operation in creating an index (creating an index on a relation of $n$ tuples takes $n \log_2 n$ such unit operations; unit cost = $C_4$);
5. unit operation in the sorting needed to prepare a multi-column index (unit cost = $C_5$).

The most interesting result of their experiments was that evaluating expressions in predicates turned out to be by far the most expensive operation in OBE. While $C_2$ through $C_5$ are of approximately the same order, $C_1$ exceeds them by approximately an order of magnitude.

2.4.2 Smallbase

Listgarten and Neimut [LN96, LN97] classify main-memory cost models into three categories:

**hardware-based** A hardware-based cost model is constructed analogously to traditional I/O-based cost models. Instead of counting I/O operations, now CPU cycles are counted. While conceptually simple, this approach is difficult to implement. In addition to the problems mentioned in [WK90], Listgarten and Neimut point out that hardware policies like cache-replacement or pre-fetching need to be incorporated, which are hard to model. Further, portability would be very limited, as such policies vary between hardware architectures. However, once constructed, a hardware-based model is believed to be accurate and reliable.

**application-based** This second category matches the approach presented in [WK90]: costs are expressed in terms of a system's bottleneck costs. While being simpler to develop than hardware-based cost models, application-based cost models are less general. The bottlenecks found highly depend on the workload used to identify them, and hence may not sufficiently represent the costs of all types of queries. In principle, application-based models are easier to port than hardware-based models, by simply regenerating the profiles. However, this might not only
result in different unit costs, but also in a different set of bottlenecks. In this case, the model itself changes, not just the instantiation, and hence the cost functions of all database operations need to be re-formulated in terms of the new set of bottlenecks.

**engine-based** The third type of cost models is a compromise between detailed but complex hardware-based models and simple but less general application-based models. An engine-based cost model is based on the costs of primitive operations provided by the (MM-)DBMS's execution engine.

Listgarten and Neimat propose a two-step process to construct engine-based cost models. First, the general model is created by identifying the base costs and expressing query processing costs in terms of these building blocks. Second, the model is instantiated by determining the relative values for the base costs, and then verified.

Step one requires detailed knowledge about the internals of the execution engine and is usually to be done by hand. In case of doubt about how detailed the model should be, they propose to make it as detailed as practically feasible. Simplifications or further refinements can be done during verification. For their Smallbase system, Listgarten and Neimat identified the following primitive costs:

- fetching a column or parameter value
- performing arithmetic operations
- performing boolean operations
- evaluating a comparison
- evaluating a like expression
- scanning a table, T-tree index, hash index, temporary table
- creating/destroying a T-tree index, hash index, temporary table
- sorting tuples
- selecting distinct tuples
- performing a join (nested loop join, merge join)

Dependencies of these costs on factors like tables size, data type, etc. are dealt with during the second step.

Step two is automated by developing a cost test program that instantiates and verifies the model. For each unit costs, two queries are provided whose execution costs differ in only that value (plus maybe further cost that are already known). Further, formulas that specify the dependency of each unit cost on the table size have to be specified. The cost test program than finds the respective parameters and verifies the model by running each pair of queries several times with varying table sizes and performing a least square regression on the difference in execution time of the pairing queries.
2.4.3 Remarks

The work described above, as well as other recent work on main-memory query optimization [LC86b, PKK+98], models the main-memory costs of query processing operators on the coarse level of procedure calls, using profiling to obtain some 'magical' constants. As such, these models do not provide insight in individual components that make up query costs, limiting their predictive value.

2.5 Cost Models for (Heterogeneous) Federated and Multi-Database Systems

From a cost modeling perspective, classical sequential, parallel, and distributed database management systems share the property that the whole system is developed by the same vendor. Hence, those developers in charge of the query optimizer have access to all specification details and the source code of the execution engine. Thus, they can exploit this detailed insight when designing cost models for query optimization. However, when it comes to global query optimization in heterogeneous federated and multi-database systems, the picture looks differently. Usually, the individual DBMSs gathered in such systems are off-the-shelf products made by one or more different vendors. This means that no detailed knowledge about these systems is available to the vendor of the federated/multi-DBMS. The participating DBMSs have to be treated as “black boxes”, and thus, new techniques are required to acquire proper cost models for global query optimization. The following sections briefly present some approaches published in recent database literature.

2.5.1 Calibration

Pegasus Du, Krishnamurthy, and Shan [DKS92] propose a calibration-based approach to obtain cost models for the participating DBMSs. The costs of basic operators — such as sequential scan, index scan, index lookup, different join algorithms — are modeled as rather generic formulas. The cost of a sequential scan over a relation $R$ evaluating a predicate $P$, e.g., is given as:

$$C_{seq\_scan}(R, P) = c_0 + c_1 \cdot ||R|| + c_2 \cdot ||R|| \cdot s(P)$$

with

- $||R||$: the number of tuples in relation $R$
- $s(P)$: the selectivity of predicate $P$
- $c_0$: the initialization cost for the select
- $c_1$: the cost to retrieve a single tuple and to evaluate $P$ on it
- $c_2$: the cost to process a result tuple satisfying $P$

The authors assume, that statistical information about the data stored in the participating DBMSs as well as a (global) logical cost model to derive selectivity factors and intermediate result sizes are available.
The coefficients $c_0$, $c_1$, and $c_2$ are assumed to be functions depending on parameters such as data types, tuple sizes, number of clauses in a predicate, etc. (where appropriate). Further, costs are measured in elapsed time, and cover I/O as well as CPU costs.

In order to calibrate the respective coefficients for a given "black box" database system, the authors designed a special synthetic database and a set of queries whose run times are measured. The major problem that arises here, is that the whole calibration process has to be predictable. For instance, calibration does not make sense, if one does not know how the system will execute a given query (e.g., using which algorithm and which index, if any). Further, effects related to data placement, paginating, etc. have to be eliminated. The presented database and query set take care of these issues. Experiments with AllBase, DB2, and Informix show, that the proposed process derives quite accurate cost models in 80% of the cases.

**IRO-DB** Gardarin, Sha, and Tang [GST96] extend the calibration approach of Du, Krishnamurthy, and Shan [DKS92] for the object-oriented federated database system IRO-DB [GFF97]. The major extension required was to introduce a path traversal ("pointer-chasing") operator and the respective cost formula. Further, cost parameters such as object size, collection size, projection size, and fan out needed to be regarded. Also, the calibration database and the query set are extended to meet the requirements of an object-oriented environment.

### 2.5.2 Sampling

Zhu and Larson [ZL94, ZL96, ZL98] propose a query sampling method. The key idea is as follows. It first groups local queries that can be performed on a local DBS in an MDBS into homogeneous classes, based on some information available at the global level in an MDBS such as the characteristics of queries, operand tables and the underlying local DBS. A sample of queries are then drawn from each query class and run against the user local database. The costs of sample queries are used to derive a cost model for each query class by multiple regression analysis. The cost model parameters are kept in the MDBS catalog and utilized during query optimization. To estimate the cost of a local query, the class to which the query belongs is first identified. The corresponding cost model is retrieved from the catalog and used to estimate the cost of the query. Based on the estimated local costs, the global query optimizer chooses a good execution plan for a global query.

### 2.5.3 Cost Vector Database

**HERMES** Adali, Candan, Papakonstantinou, and Subrahmanian [ACPS96] suggest to maintain a cost vector database to record cost information for every query issued to a local DBS. Cost estimation for a new query is based on the costs of similar queries. For each call to a local DBS, the cost vector registers the time to compute the first answer, the time to compute all the answer, the cardinality of the answer, and the type of predicates to which these values correspond to. Summary table are also generated...
off-line to avoid heavy burden on storage. To estimate the costs of a new sub-query, the sub-query is matched against the cost vector database and a kind of regression is applied. The approach is demonstrated as efficient for sources queried with similar sub-queries.

2.6 Main Memory Database Systems

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

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

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

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

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

Well known main memory systems are Smallbase [BHK+86, LN96] developed by HP, the object-oriented AMOS [FR97] system, the parallel MMDBMS PRISMA [AvdBF+92], and Dalí [JLR+94, RBP+98] by Bell Labs. Smallbase and Dalí have been reshaped into commercial products, under the names Times Ten [Tea99] and DataBlitz [BBG+99], respectively. Their main focus is highly efficient support of OLTP DBMS functionality on small or medium-size data sets. Also, all main relational vendors (IBM, Microsoft, Oracle) are offering small-footprint "ultra-light" versions of their DBMS servers for use in mobile computing devices and web PDAs.

### 2.7 Monet

Our research goals as specified in Section 1.4 require a "real" DBMS to conduct empirical analysis and experimental validation. In principal, there are three options:

1. use a commercial main-memory DBMS, e.g., Times-Ten [Tea99], or DataBlitz [BBG+98, BBG+99];
2. use a fully-fledged commercial disk-based DBMS, such as IBM's DB2, Microsoft's SQL-Server, or Oracle;
3. use a main-memory DBMS research prototype with accessible source code, e.g., our own Monet system.

For this work, we consider the third option to be the most practical. Mainly the fact that we know the internal architectural details and the source code helps us to understand how the complex interaction between database software, compilers, operation system, and hardware does work. We can easily play around with compiler switches and add profiling hooks to gain the necessary insight for our modeling plan. Moreover, it is only the ability to modify data structures, algorithms, and coding techniques that enables us to validate the new techniques we propose. In this section, we give a concise introduction to Monet, focusing on the core features that are important in the given context. For an complete description of Monet, the interested reader is referred to [Bon02].

#### 2.7.1 Design

Monet is a main-memory database kernel developed at CWI since 1994 [BK95], and commercially deployed in a Data Mining tool [KSHK97]. It is targeted at achieving high performance on query-intensive workloads, such as created by on-line analytical processing (OLAP) or data mining applications. Monet uses the Decomposed Storage Model (DSM) [CK85], storing each column of a relational table in a separate binary table, called a Binary Association Table (BAT). A BAT is represented in memory as an array of fixed-size two-field records [OID,value], or Binary UNits (BUN). The OIDs in the left column are unique per original relational tuple, i.e., they link all BUNs that
make up an original relational tuple (cf., Figure 2.1). The major advantage of the DSM is that it minimizes I/O and memory access costs for column-wise data access, which occurs frequently in OLAP and data mining workloads [BRK98, BMK99, MBK02]. The BAT data structure is maintained as a dense memory array, without wasted space for unused slots, both in order to speed up data access (e.g., not having to check for free slots) and because all data in the array is used, which optimizes memory cache utilization on sequential access.

Most commercial relational DBMSs were designed in a time when OLTP was the dominant DBMS application, hence their storage structures, buffer management infrastructure, and core query processing algorithms remain optimized toward OLTP. In the architecture of Monet, we took great care that systems facilities that are only needed by OLTP queries do not slow down the performance of query-intensive applications. We shortly discuss two such facilities in more detail: buffer management and lock management.

Buffer management in Monet is done on the coarse level of a BAT (it is entirely loaded or not at all), hence the query operators always have direct access to the entire relation in memory. The first reason for this strategy is to eliminate buffer management as a source of overhead inside the query processing algorithms, which would result if each operator must continuously make calls to the buffer manager asking for more tuples, typically followed by copying of tuple data into the query operator. The second reason is that all-or-nothing I/O is much more efficient nowadays than random I/O (similarly to memory, I/O bandwidth follows Moore’s law [Moo65], I/O latency does not).
In Monet, we chose to implement explicit transaction facilities, which provide the building blocks for ACID transaction systems, instead of implicitly building in transaction management into the buffer management. Monet applications use the explicit locking primitives to implement a transaction protocol. In OLAP and data mining, a simple transaction protocol with a very coarse level of locking is typically sufficient (a read/write lock on the database or table level). We can safely assume that all applications adhere to this, as Monet clients are front-end programs (e.g., an SQL interpreter, or a data mining tool) rather than end-users. The important distinction from other systems is hence that Monet separates lock management from its query services, eliminating all locking overhead inside the query operators.

As a result, a sequential scan over a BAT comes down to a very simple loop over a memory array with fixed-length records, which makes Monet’s query operator implementations look very much like scientific programs doing matrix computations. Such code is highly suitable for optimization by aggressive compiler techniques, and does not suffer from interference with other parts of the system. In other words, Monet’s algorithms are implemented directly on the physical storage structures, i.e., without any intermediate storage management layer. Thus it becomes feasible (1) to understand, how the algorithms, respectively their implementations, interact with the underlying hardware, and (2) to accurately model the algorithms’ cost in detail.

2.7.2 Architecture and Implementation

Monet is designed as a MMDBMS kernel providing core database functionality and acting as a back-end for various front-end applications. The user-interfaces are provided via the front-end applications. Sample applications are among others an SQL front-end, an ODMG front-end, as well as OLAP and data mining tools. For communication between the front-end and the back-end, Monet provides an intermediate query language called Monet Interpreter Language (MIL) [BK99]. The core of MIL is made up by primitives that form a BAT-algebra similar to the relational algebra. Around this core, MIL has been developed as a computational complete procedural language giving full access to Monet’s core functionality. Further characteristics of MIL are that (1) MIL programs are interpreted and (2) MIL operators are evaluated one at a time. The latter means that query evaluation in Monet follows a bulk-processing approach. As opposed to pipelining, this means that all intermediate results are fully materialized. Here, the Decomposed Storage Model offers another advantage, keeping intermediate results very “slim” and hence rather small. The crucial advantage of bulk-processing—next to its simple implementation—is the following. With the BAT-algebra primitives operating on binary tables only and a limited set of atomic data types, such as integer, float, character, string, etc., the number of type-combinations per operator is rather limited. We exploit this to overcome a disadvantage of the interpretation approach. Interpretation of algebra operators usually requires a type-switch in the innermost loops, as the actual type-binding is only known at runtime. Such type switches are typically realized by ADT-like function calls, and hence they are rather expensive. With the limited number of type combinations in MIL, we can provide a separate implementation of each operator for each case. To limit the
coding effort, Monet is written in a macro language, from which C language implementations are generated. Per algorithm, only one template-implementation has to be implemented "by hand". At compile time, the macro language then expands the type-specific variants from these templates. The macros implement a variety of techniques, by virtue of which the inner loops of performance-critical algorithms like join are free of overheads like database ADT calls, data movement, and loop condition management. These techniques were either pioneered by our group (e.g., logarithmic code expansion [Ker89]) or taken from the field of high performance computing [Sil97]. Furthermore, Monet is implemented using aggressive coding techniques for optimizing CPU resource utilization [MBK00b] that go much beyond the usual MMDBMS implementation techniques [DKO+84]. In Chapter 5, we present some examples of these techniques.

2.7.3 Query Optimization

In Monet, we pursue a multi-level query optimization approach [MPK00]. Front-end applications can use strategic optimization, exploiting domain-specific knowledge to pre-optimize queries before sending them to the back-end. The MIL interpreter does some multi-query optimization in a tactical optimization phase. This phase removes common subexpressions of multiple queries, possibly sent by different front-ends, and takes care of introducing parallelism in case the Monet back-ends runs multi-threaded. Finally, the Monet kernel itself performs operational optimization. Operational optimization means that for each operator that is to be evaluated the kernel can choose the proper algorithm and its implementation just before executing it. Here, we exploit the fact that due to the bulk-processing approach, all operands are fully available before the operator has to be executed. Next to their sizes, MIL operators maintain some more properties of the BATs they generate, such as their data types and information about whether attribute values are sorted and/or unique. At runtime, the kernel examines the of the operands as well as the systems current state and applies heuristics to pick the most efficient algorithm and implementation for the given situation. For instance in case of a join, the kernel can currently choose between nested-loop join, hash-join, sort-merge join, and index-lookup join. In Chapter 5, we provide additional cache-conscious join algorithms and show how the cost models developed in Chapter 4 are used to find the most suitable algorithm.