In: Proc. DASFAA'95, Singapore, April 1995, pp. 264-272.

Implementing Dynamic Code Assembly for Client-Based Query Processing

J. Thomas, T. Gerbes, T. Härder, B. Mitschang
Dept. of Computer Science, University of Kaiserslautern
67653 Kaiserslautern, Germany

e-mail: {thomas | gerbes | haerder | mitsch}@informatik.uni-kl.de

Abstract

Advanced database management systems (DBMS) are faced with increasingly complex query-processing tasks due to more powerful data models and query languages. These systems are usually conceived for client/server environments, with most application-oriented processing being done in the main-memory buffer at the client side. For performance reasons, it is indispensable to exploit the buffer contents when optimizing queries. As the buffer contents is only known at run time, advanced DBMS should support run-time optimization and offer concepts guaranteeing flexibility and efficiency of the subsequent phase of code generation. This paper addresses these issues. Our approach, *dynamic code assembly*, is based on a strictly orthogonal specification and implementation of the concepts involved in code generation, allowing to dynamically assemble executables from precompiled pieces of code, thus reducing explicit code generation to a minimum. We discuss our considerations and implementation referring to the KBMS KRISYS, although our ideas are generally applicable to advanced DBMS requiring run-time optimization and consequently flexible and efficient generation of executable code at run time.

1. Introduction

In the area of DBMS, there is a trend towards ever more powerful data models and query languages [8], spawned on one hand by advanced applications demanding more expressive modeling and querying facilities, on the other hand by improvements of available hardware, e.g., faster, possibly parallel processor architectures or growing main memories. These developments call for enhancements to query processing concerning semantic expressiveness and extensibility as well as performance.

Semantic expressiveness has been addressed quite successfully by advanced DBMS, e.g., object-oriented DBMS [2] or KBMS [20]. These systems must support *extensibility* at different levels of query processing [9, 10, 11] to cope with later extensions either of the query language (to shift more application-oriented semantics into DBMS query processing) or of evaluation methods (e.g., improved join algorithms). Moreover, the need for extensibility is in accordance to the general goal of a simple, streamlined design of the query-processing framework.

Proceedings of the Fourth International Conference on
Database Systems for Advanced Applications (DASFAA '95)
Ed. Tok Wang Ling and Yoshifumi Masunaga
Singapore, April 10-13, 1995

As determined by the applications' requirements, non-standard DBMS are usually conceived for client/server architectures [5, 12]. While the server is responsible for general data-management tasks and for precomputing data, application-oriented processing is done in the main-memory buffer at the client. Since the expressive power of the query language should be available not only for loading/unloading the buffer, but also for more sophisticated processing tasks on the buffer contents, *main-memory based query processing* must be supported [6, 14]. A simple navigational buffer interface is no longer satisfactory. Instead, a declarative interface is required to allow *querying* the buffer contents. Additionally, adequate main-memory indices should be supported to cope with large application buffers.

Larger client caches improve locality of client-based processing, and exploiting the buffer contents at run time becomes a crucial performance issue for query processing. This calls for *efficient run-time optimization*. Consequently, the subsequent phase of code generation must be performed in a flexible and efficient manner as well. Our approach to achieving flexibility and efficiency is to dynamically assemble executable code from precompiled code fragments reducing explicit code generation to a minimum. We call this step of query processing *dynamic code assembly*.

We devised and implemented dynamic code assembly for KRISYS, a KBMS developed at the University of Kaiserslautern. It features an object-oriented knowledge model and a set-oriented, declarative query language as its user interface. The language is processed following an algebraic approach comparable to relational query processors [15]. The plan-operator concept for client-based query processing has been designed to be extensible and to allow efficient run-time optimizations [26]. Sect. 1 gives an overview of KRISYS and its components. In Sect. 2, we discuss tasks and requirements of code assembly and execution from a conceptual point of view. Sect. 3 describes how we implemented both steps based on the existing plan-level realization. Opposed to conventional query-processing systems requiring *strict compilation*, we can assemble executables by putting together precompiled functions. As will be discussed in Sect. 4, this approach is as effective as strict compilation, and, although we used Common Lisp as implementation platform [1, 25], it is also feasible for C-like languages, e.g., C++. Sect. 6 discusses related work and future research issues.

1. Overview of KRISYS

1.1 Overall Architecture

KRISYS was conceived to support knowledge processing in client/server environments. While processing at the server is performed in the data model of the server DBMS (*data processing*), client-based processing is carried out in the KRISYS knowledge model (*knowledge processing*). Client and server are loosely coupled by an application buffer in the client, the Working-Memory (WM). Moreover, the client hosts the KOALA Processing System (KPS) featuring the query language KOALA as interface to end users and applications (see [6] for details).

1.2 Knowledge Model

The knowledge model of KRISYS is comparable to object-oriented data models [2, 16]. An object is uniquely identified by a name (i.e., object-identifier), and contains a set of attributes to describe its characteristics: *slots* are used for representing properties of an object and relationships to other objects; *methods* are used for expressing object behavior. Attributes can be further described by *aspects*, defining, e.g., the cardinality of a slot. For object structuring, the abstraction concepts of classification, generalization, association, and aggregation [19] are maintained automatically by the system. For details on the knowledge model of KRISYS see [4, 20].

1.3 Query Language KOALA

Retrieval and modification of knowledge base (KB) contents is supported by KOALA [7, 20], a descriptive, set-oriented language constituting the user interface of KRISYS. KOALA features two powerful operations, ASK to query the KB, and TELL to change the state of the KB. The following ASK statement is taken from an application modeling a restaurant environment.

```
(ask (?guest ?wine ?country)
 (and
  (is-instance ?guest persons)                               ①
  (is-element ?country europe)                               ②
  (equal white (slotvalue color ?wine))
  (is-in ?wine (slotvalue produced-wines ?country))
  (is-in ?wine (slotvalues preferred-wine ?guest))           ③
```

Query variables are indicated by a leading question mark. The query asks for all guests (①) that have a European white wine (②) as preferred wine (③), and returns the names of the guests, wines, and countries, respectively. We will refer to this example query throughout this paper.

Since KRISYS was conceived for a client/server environment, processing of KOALA in general involves both server and client. Data not yet residing in the application buffer at the client must be pre-selected at the server and transported to the client. Subsequently, knowledge processing can be carried out at the client. In the following, the focus will be on client-based query processing.

1.4 Working-Memory (WM)

Knowledge processing takes place in WM. The representational framework of WM directly reflects the characteristics of the knowledge model. This applies to the internal structure of objects, as well as to the relationships among objects, the most important of which are the abstraction concepts, forming abstraction hierarchies. Both types of relationships are materialized in WM using main-memory pointers providing fast access to the required information. Moreover, the WM supports efficient access to and set-oriented processing of arbitrary collections of objects, e.g., intermediate results in knowledge processing or data fetched from the server, by providing so-called *Access Structures* (AS). In their basic form, AS are lists supporting simple traversal operations based on a cursor concept. AS may also be realized as trees or hash tables taking the role of main-memory indices that provide key-value access and ordering facilities. AS can be installed dynamically and temporarily during query processing. Thus, knowledge is organized in WM as shown in Fig. 1.

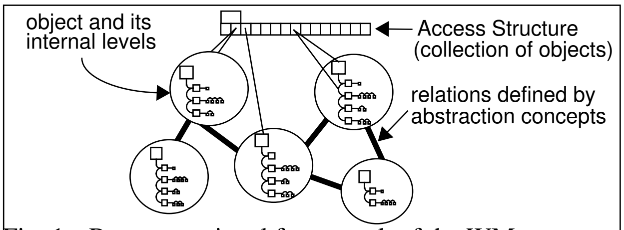


Fig. 1: Representational framework of the WM

1.5 KOALA Processing System (KPS)

KOALA is based on an algebraic processing model that allows conventional (relational) algebraic optimizations to be used to a large extent [22]. Thus, the overall steps of query processing proceed in a similar fashion as the well-known steps of data processing in relational DBMS [15]: first, an algebra graph is generated and optimized; then, a plan-operator graph is constructed; finally, executable code is generated, and the query is evaluated. In the following, we will describe the different representational levels.

Algebra Level

Queries are transformed into algebra graphs and subsequently optimized [23]. On one hand, algebra operators show a functionality that can also be found in conventional relational algebras (e.g., SELECT or JOIN), on the other hand, there are specific operators to handle object structures (e.g., NEST or UNNEST). Without discussing KOALA algebra in detail, we want to give an impression of how a query is translated into an algebraic representation. We refer to the sample query from above. Fig. 2 (left side) shows the corresponding algebra graph [1].

Plan Level

When transforming an algebra graph into a plan-level representation, KPS must exploit the current contents of WM (including storage structures, sort orders, etc.). Data already residing in the application buffer need not be fetched from the server and can be used for optimizing query processing[2]. On the other hand, KPS must ensure that all information required for client-based query

1. The algebraic representation only employs conventional relational operators which is sufficient for the scope of this paper.

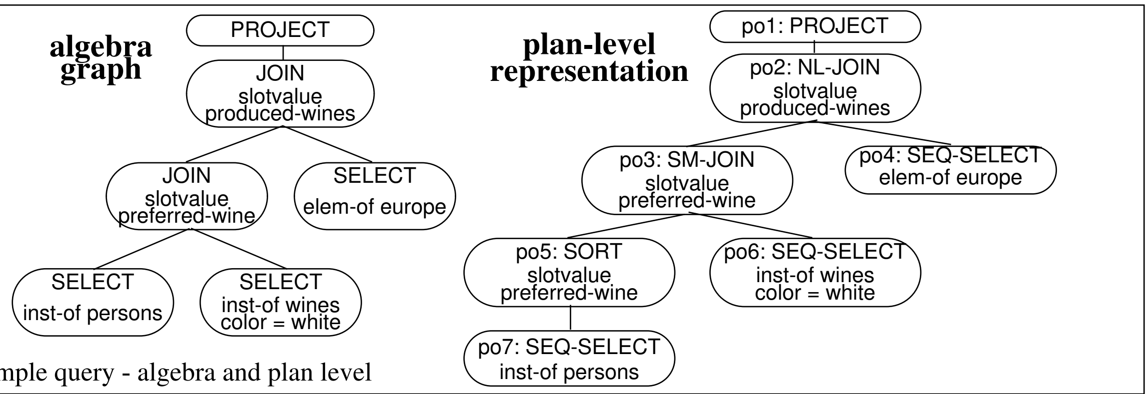


Fig. 2: Example query - algebra and plan level

processing is loaded into WM. This results in queries generally being split into a part to be executed at the server, and a part to be performed at the client [28]. Hence, plan-level manipulations can be completed only at run time.

Fig. 2 (right side) shows the plan-operator graph for our example query. Since the scope of this paper is on client-based processing, we assume that all input data are residing in WM, and that, consequently, all plan operators for our example query are client-based. Moreover, we assume that instances of wines are sorted by their object names. As soon as the instances of persons are selected (operator SEQ-SELECT) and sorted according to their preferred wines (operator SORT), a sort-merge-join can be employed to relate persons to their preferred wines (operator SM-JOIN). Since we do not make any assumption on how the elements of Europe are stored in WM, the second join is realized as a simple nested-loop (operator NL-JOIN).

Our plan-operator approach involves the following concepts (explained in detail in [26]): Plan-operator templates realize a *simple processing paradigm for plan operators*, as well as *extensibility at the plan-operator level*. Base predicates introduce knowledge-model semantics into query processing, e.g., inheritance. Moreover, they guarantee *extensibility of the query language* without affecting existing plan operators (POs). Subgraphs of a PO graph are combined to units of execution, called blocks, which employ logical AS (LAS) for internal data flow, and which rely on AS in WM for exchanging information among each other. AS and LAS ensure *efficient data flow between plan operators*. Moreover, the way in which all these concepts are combined warrants *efficient dynamic query optimization* and the construction of *flexible units of execution*.

These characteristics were achieved by a modular design and realization of the plan level, and, although implemented in LISP, we resorted only to techniques to be found in C-like programming languages as well [17].

Code Assembly and Execution

Units of executable code are assembled by deriving and establishing a block structure for a given PO graph. Blocks are the units of execution in our query-processing approach and introduce a new level of abstraction lying above that of POs. A query is no longer represented as a graph consisting of POs, but as a graph made up of blocks. Hence, executing a query means evaluating the corresponding blocks. Just like POs, blocks accept one or more input streams and produce exactly one output stream. Seen from outside, blocks work in a set-oriented way, yet internally they may operate tuple-wise, depending on the PO(s) contained.

Blocks rely on the PO level, both conceptually and concerning their implementation. Hence, not only generating a plan-level representation for a given query is a task to be performed (at least partially) at run time, but also assembling the code to be executed later on. Flexibility and efficiency of these tasks are important for good performance of overall query processing. We will now address the conceptual and implementational requirements to be met by the steps of code assembly and execution.

2. Code Assembly and Execution - Conceptual Issues

We will describe how blocks are constructed from a given PO graph, and how they can be employed during query evaluation. Subsequently, we will discuss run-time dependencies between block structuring, the plan level, and the system state at run time, thus defining goals to be met by an appropriate realization of code assembly and execution.

2.1 Block Structuring

From the plan-level representation, units of execution (blocks) must be generated that guarantee a query evaluation which is optimal concerning data flow as well as minimal in the amount of intermediate results organized as AS in WM. Intermediate results are required due to either the topology of the PO graph at hand or the processing characteristics of the POs in the graph since POs do not have local buffering facilities [26].

Processing characteristics are associated to each input and output stream of a PO. We distinguish *set-oriented* and *tuple-oriented* streams. The latter may be further classified into streams that are read only once (*single-pass streams*), and those that are accessed several times (*multiple-pass streams*)[3]. For example, the “outer” input of a nested-loop join is single-pass tuple-oriented, while the “inner” input is

2. For efficiency, matching buffer contents to data referenced by a query is an integral part of our approach to query optimization. Integrating matching and optimization is also advocated in [3].

3. This strategy avoids computing input for each pass repeatedly.

multiple-pass tuple-oriented. Only set-oriented and multiple-pass tuple-oriented streams require state information, i.e., facilities for buffering input or output tuples.

Concerning topology of the PO graph, operators whose output stream is referenced more than once must materialize their results, hence need an AS at their output. Special attention must be paid to those leaf POs which perform accesses to the server and supply input for succeeding client-based POs. To exploit set-oriented transfer of data from server to client, results coming in from the server must be buffered in WM. To this end, an AS must be placed after each such client-based PO as buffering facility.

We can now derive a set of rules where to place AS in the PO graph, thus determining block boundaries.

Rule 1: The topmost PO of a query marks the end of (a unit of) execution, and therefore, a block border must be placed directly above it.

Rule 2: Set-oriented operators (that need the output stream to gradually build up their result) must be placed at the end of a block.

Rule 3: Operators whose output is referenced more than once must materialize their results; hence, they must be positioned at the end of a block.

Rule 4: Tuple-oriented, multiple-pass operators need their input temporarily buffered; hence a block border must be placed directly below them.

Rule 5: Operators directly above the borderline to the server (fetching their input from the server) mark the beginning of a block.

These rules guarantee that intermediate results are materialized in AS only if necessary and result in maximum units of pipelined processing. Moreover, placing AS only at those positions in a PO graph allows optimal dataflow. Applying these rules to the PO graph of Fig. 2 results in the block structure shown in Fig. 3.

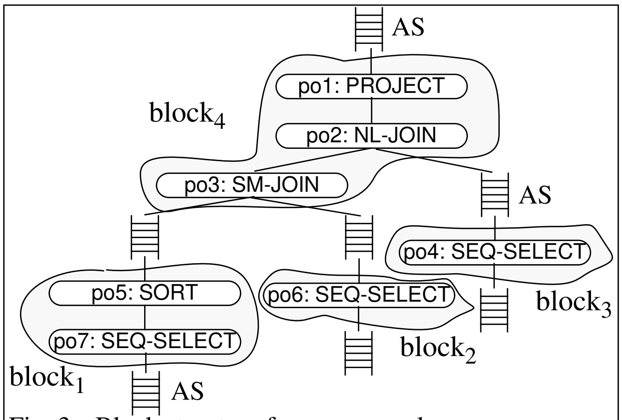


Fig. 3: Block structure for our example query

2.2 Block Execution

A PO graph is transformed into a block-structured graph, i.e., into a representation that abstracts from single POs. Evaluating a query therefore means executing the corresponding blocks. This can be done sequentially, but also in parallel [21], based on a multiprocessor shared-memory architecture [24] and a special plan operator *transmit* [28].

2.3 Run-Time Dependencies

Due to the client/server environment and the need to consider buffer contents, modifications of a PO graph at run time occur quite frequently. These adaptations, in turn, may impact single blocks as well as overall block structuring. To substitute a PO for another with compatible processing characteristics (meaning that the former and the actual PO both require buffering or not) must be possible without having to reconstruct the entire enclosing block. Replacing a PO for another with a different processing characteristics requires block structuring to be reconsidered. This implies reorganizing the internal structure of blocks and redefining block boundaries. Hence, a *flexible modification of internal block structures* as well as a *dynamic redefinition of block boundaries* are important requirements for dynamic code assembly.

We illustrate these considerations referring to the block-structured graph of Fig. 3. Let us assume we replace po3 by a simple nested-loop implementation. Moreover, let po6 constitute the “inner” input of that join. On one hand, po5 becomes superfluous, on the other hand, blocks 1 and 4 can be merged, and the AS in between can be discarded. This is due to the fact that po3 needs to access each tuple of its “outer” input only once. Fig. 4 depicts the resulting block structure for our example query.

As pointed out before, block-structured graphs can be executed sequentially or in parallel. The objectives underlying the rules for constructing blocks guarantee a minimum amount of intermediate results and optimal data flow for sequential evaluation, and, as discussed in [27], also for parallel query execution. The rules introduced above aim at maximum units of pipelined processing. Speaking the other way around, blocks render coarse-grained units of parallelism. Finer-grained parallelism can be achieved by refined block-determination rules. These rules might split up blocks (horizontal parallelism), or split up AS and replicate blocks (vertical parallelism), thus, however, increasing the number of blocks and intermediate results. We leave details to a further publication.

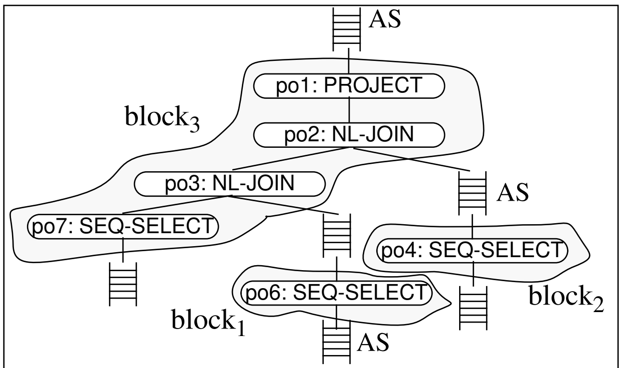


Fig. 4: Modified block structure for our example

3. Code Assembly and Execution - Implementational Issues

In the previous section, we discussed concepts and requirements underlying code assembly and execution. We will now describe their implementation and integration into knowledge processing. We chose Common Lisp [1, 25] for implementation because it supplies a flexible representational basis for object structures and a powerful programming environment for system development.

Code assembly is performed in several steps. We will describe each step together with its input and the resulting data structures, always referring to the block-structured PO graph shown in Fig. 3. Starting from this graph, block borders must be identified. Next, blocks are created, which comprises building up inter-block communication as well as constructing the internal structure of blocks.

3.1 Data Structures and General Remarks

Most figures presented below contain simplified Lisp code emphasizing the important parts of our implementation. To create new structure types (comparable to record types in C++), we use Lisp function *defstruct* requiring the structure's name and fields as parameters. *Defstruct* creates the structure type itself as well as a set of functions to handle that type: Instances of a structure called *example* can be created by *make-example* which also allows to preset some or all fields of the new structure. *Modify-example* stores values in the fields of an already existing structure.

Our data structures are (finite) directed acyclic graphs consisting of nodes and edges. We distinguish *root nodes* (of a graph or subgraph), *inner nodes* and *leaf nodes*. Each node contains a field for the node's unique identifier, possibly several fields for node-specific information, and one field to establish connections, i.e., edges, to child nodes. Since a node can have several children, the latter field is realized as a list. Each entry of that list may be a physical pointer, so a child can be reached directly from a parent node, or the entry may be a symbolic pointer, i.e., a node identifier, so a child node must be looked up via a hash-table, using the node identifier as search key.

For representing POs, we must differentiate between information being specific for all occurrences of the same PO category (e.g., whether the input is processed tuple- or set-oriented) and information holding for every occurrence of a PO in the query under consideration (e.g., the source of its input tuples). This information is stored in structures called PO-CLASS and PO-INSTANCE (Fig. 5).

```
(PO-CLASS
  name
  function
  output-mode
  input-modes)
```

```
(PO-INSTANCE
  name
  PO-CLASS
  parameters
  list-of-input-pos
  no-of-reader-tuple
  no-of-reader-set
  write-characteristics)
```

Fig. 5: Data structures for representing POs

Structures of type PO-CLASS, e.g., SELECT, describe properties common to all its instances. They contain a pointer to the executable code (*function*) and information about the characteristics of input and output streams (zero to two *input-modes* and one *output-mode*). Mode 'tuple' denotes single-pass tuple-oriented access, mode 'set' denotes set-oriented or multiple-pass tuple-oriented access.

The current PO graph is stored in structures of type PO-INSTANCE. Every PO-INSTANCE is identified by field *name* and belongs to a PO-CLASS. Field *parameters* contains information to guide the PO functionality, e.g., base predicates, key functions, or sort predicates. For every input stream, *list-of-input-pos* stores the names of the producing PO-INSTANCEs. Structure PO-INSTANCE is made up of three more fields which we will describe later.

3.2 Generating the PO-Graph

As discussed before, a PO graph is created by generating PO-INSTANCE structures for all POs of the graph and filling their fields with appropriate values.

3.3 Generating Blocks, Communication and Control

Step 1: Identifying Borders

Information where to put borders in the current PO graph is stored in the last three fields of each PO-INSTANCE (*write-characteristics*, *no-of-reader-tuple* and *no-of-reader-set*). To fill these fields, the PO graph is examined recursively following the rules defined in Sect. 2.1, taking the *write-characteristics* of a producing PO and accumulating the characteristics of all consuming POs into *no-of-reader-tuple* and *no-of-reader-set*.

Having derived information on the positions of borders, we can now generate blocks for the given PO graph.

Step 2: Generating Blocks

Each block is stored in a data structure of type BLOCK (Fig. 6 a). Every block has exactly one topmost PO whose name is used as BLOCK identifier (*name*). As blocks are the units of execution, they require a field for the assembled code (*function*) and its *parameters*. The structure of the block graph is represented via the child nodes in *list-of-input-blocks*. The subtree of POs which belongs to each block is not explicitly stored inside the block; it can be extracted dynamically from the original PO graph[4].

A block structure is created for every PO requiring a border above, according to the information collected in the previous step. To this end, only field *name* is filled with the PO name (Fig. 6 b). The other fields are filled in separate steps, described subsequently.

Step 3: Connecting Blocks

The unconnected blocks are linked by inserting symbolic pointers into *list-of-input-blocks* which are collected during a recursive traversal of the PO subgraph of every block. To this end, function *modify-block* is employed (Fig. 6 c).

Step 4: Generating Inter-Block Communication

POs inside a block communicate via LAS (cf. Sect. 1.5). In contrast, blocks read their inputs from and materialize their output in physical AS. For every new block, this step creates a new output AS (Fig. 6 d). By default, the AS chosen is an unsorted list. In the case of a SORT-PO, for example, its parameters are used to create an AS of the considered type (e.g., sorted list or b*tree).

4. With knowledge of the block's topmost PO and the position of the borders.

Step 5: Generating Intra-Block Communication

Before discussing the implementation of this step, we will first recall how POs operate inside a block. Intra-block communication is organized via LAS (Sect. 1.5). If a LAS is requested to produce a new output tuple, it must activate its underlying PO. The PO called, in turn, needs LAS to access its input(s) by invoking function *get-entry-from*. This calling pattern is repeated until the bottom-most operator of a block is reached.

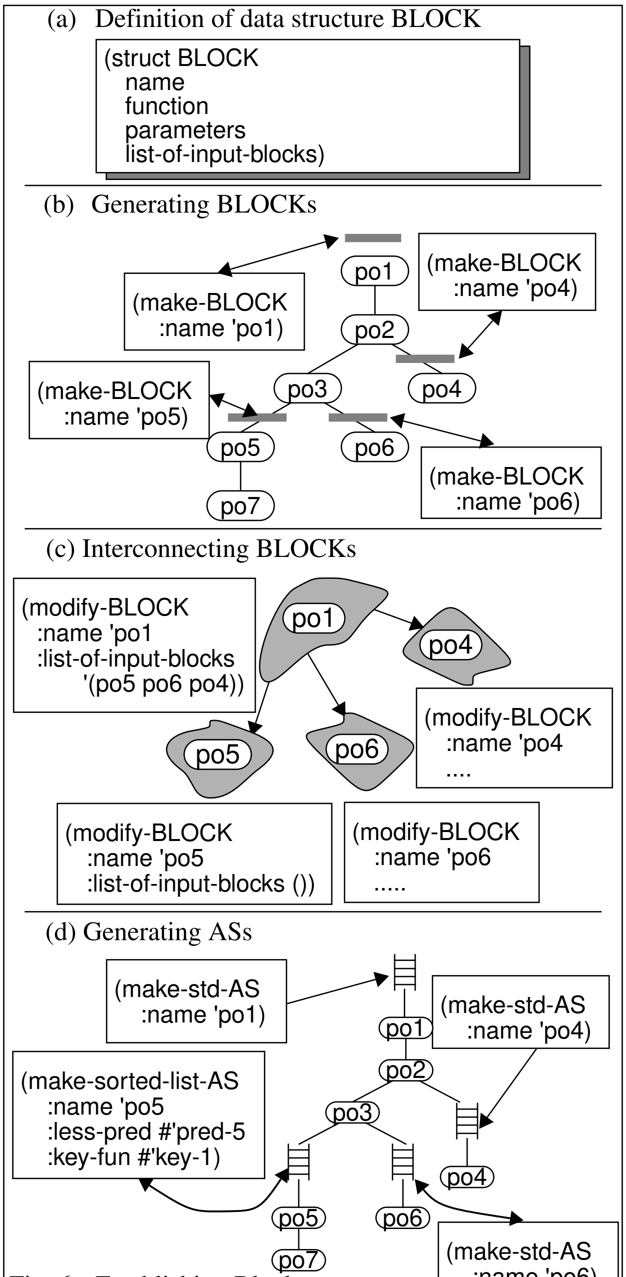


Fig. 6: Establishing Blocks

Upon invocation of *get-entry-from*, the calling PO must supply a *read-mode*. Sequential read modes (first, next, or current) are supported by all POs, so that the input to a sequential PO can be supplied by every other PO or by an AS. On the other hand, there are POs requiring a more sophisticated access to their input which can only be provided by AS, e.g., directly accessing a tuple in a hash table. Such a PO must invoke *get-entry-from* with additional parameters (e.g., a key value for direct access) to be passed to the underlying AS.

To represent intra-block communication, we introduce data structure *LAS*. Firstly, it contains a pointer to the PO to be called, i.e., to the corresponding piece of code (*function*). This PO may again need own *LAS* to access its inputs, which are recursively placed in the parameter list of the consumer *LAS* (cf. Fig. 7). On the other hand, the PO called may be the bottom-most operator of a block and thus receive its input directly from an AS. To this end, we implemented pseudo-PO READ-FROM-AS that connects a leaf PO of a block to its input AS (e.g., LAS-6 in Fig. 7).

Moreover, data structure *LAS* contains a list of parameters which must be supplied to the underlying PO upon activation. We distinguish *static parameters* and *dynamic parameters*. All parameters are tagged by *keywords* preserving independence of the parameter position. They consist of information retrieved from PO-INSTANCEs (e.g., base predicates, key-functions, sort-predicates) and of a *LAS* for every input stream of the PO, produced by function *make-las* (see below). In contrast, *dynamic parameters* must be newly specified for every PO call, e.g., *read-mode* which is filled by an appropriate value each time a PO is activated). The call itself is performed by the *apply* mechanism of Lisp, which automatically handles *keyword* arguments. Its performance and transfer to other programming languages will be discussed in Sect. 4.

To establish the *LAS* for intra-block communication, *make-las* is called for the topmost PO of every block. The function takes the parameters found in PO-INSTANCE and recursively generates *LAS* (function *make-las*) for each input stream to build the list of static parameters. Input streams generated from POs residing in the same block, are accessed by a *LAS* with *function* and *parameters* of the producing PO (dark-shaded boxes in Fig. 7). If, in contrast, the input stream is produced by a PO belonging to a different block, tuples have been materialized in the producer's output AS prior to the execution of that block. Instead of calling the PO, the input can be fetched from this AS by creating a *LAS* with function READ-FROM-AS and appropriate parameters (light-shaded boxes in Fig. 7).

Step 6: Generating Execution Control

To complete code generation, functions to drive the execution of blocks are needed. They are stored in each block together with their parameters, i.e., the name of the output AS and the name of the block's topmost LAS from which to request output. To this end, the output characteristics of a block's topmost PO must be considered, since it defines the output characteristics of the whole block. We distinguish tuple-oriented and set-oriented output characteristics. Tuple-oriented blocks are driven by function *control-for-tuple-oriented-blocks* containing a while loop which simply reads all output tuples of a block and materializes them in an AS. Set-oriented blocks, having a set-oriented PO as topmost operator, are controlled by function *control-for-set-oriented-blocks*. Set-oriented POs operate directly on the output AS and fill the AS on their own. A

SORT-PO, for example, may sort successively by inserting every input tuple at the correct position via a b* tree, or it may firstly insert all tuples in an unsorted list and then sort the whole list in one shot. Hence, *control-for-set-oriented-pos* only initiates set-oriented processing of the block's topmost PO and leaves all other tasks to that operator.

3.4 State of the Implementation

The transformation of a query into an algebraic representation and the subsequent rewrite are already implemented, as well as all constituents of the PO level, including blocks and the functionalities for building them. In addition to different policies for driving single blocks, we can execute block-structured graphs sequentially. We are currently working on concepts for executing blocks in different processing environments ranging from single-processor workstations over multiprocessor architectures to distributed settings. First steps in this direction are the implementation of the *transmit* operator to encapsulate process and processor boundaries between blocks [28] and of a component for parallel scheduling of blocks [27].

4. Validation

4.1 Lisp Implementation

Extensibility and flexibility are major goals of query processing for KRISYS. At the PO level, these characteristics are achieved by a modular design and implementation of its constituents [26]. The steps of code assembly and execution were conceptualized and implemented in such a way that they preserve these advantages. Moreover, the code produced during these steps can be executed almost as efficient as that resulting from conventional (i.e., strictly compilative) query-processing approaches.

4.1.1 Extensibility and Flexibility

Extensibility at the level of code assembly means that new POs can be easily integrated. This can be achieved by adding an appropriate PO-CLASS structure, thereby specifying the new PO's input and output behavior. This information is needed to construct blocks later on. The procedures realizing code assembly need not be modified. Hence, extensibility is guaranteed for code assembly.

Flexibility during code assembly, i.e., being able to react immediately to run-time dependencies, is possible only if the corresponding concepts are of low complexity and can be employed efficiently at run time. Every step of code assembly is executed once and has linear complexity depending on the number of POs in a query. To identify borders, for example, the total number of visits depends on the number of input streams in a PO graph. Since we use only POs with at most two input streams and each visit takes constant time, say t_v, the complexity of that step is bound to 2 * #PO * t_v. Hence, the complexity of overall code assembly is comparatively low.

4.1.2 Efficiency of Produced Code

Concerning efficiency of the code generated, there are two criteria to be considered, the time to build executable code (start-up time) and the efficiency of code execution.

In conventional code generation, e.g., Bubba [29], queries are transformed into C source code which is explicitly compiled as the last step of code generation. Compilation typically takes several seconds of start-up time (cf. Fig. 8, upper part) consuming main memory up to a few megabytes for the compiler, source files and object code. On the other hand, the resulting code is efficient because the compiler can use in-line substitution of functions and optimized argument passing by static parameter blocks.

In our approach, executable code is assembled using fully compiled functions which are combined by data structures containing function pointers. Hence, the resulting code need not be interpreted. This approach, *dynamic code assembly* (DCA), only needs a very short start-up time, since code can be assembled very fast. Measurements show that even for large queries of up to 100 POs, DCA is completed in far less than one second (SUN Sparc SLC, 16 MB RAM). During code execution, however, we do have some overhead due to function calls via *apply* with *keyword* parameters. To our experience, each such call can be weighted as (at most) three function calls with ordinary parameters. This worst-case ratio results from measurements we performed for typical PO calls. Hence, compared to explicit compilation, execution time rises faster as the number of PO calls increases (see Fig. 8, lower part).

Although this overhead is small compared to the efforts for knowledge processing as a whole, it can still be reduced by a more appropriate mechanism for parameter passing. To this end, we will sketch an alternative implementation in C++. By employing C++, we can streamline parameter passing to our needs, yet keep most of the flexibility required for knowledge processing (except for the definition of new PO-CLASSes which may cause modifications of source code and consequently recompilations).

4.2 Transfer from Lisp to C++

Lisp function *apply* together with *keyword* parameters allows convenient and powerful specification of parameters which is overly flexible for DCA. By reducing flexibility, we can eliminate this unnecessary overhead.

In a straightforward fashion, we map each PO-CLASS to a class in C++. Thus we construct a PO hierarchy exploiting inheritance of C++. This hierarchy consists of root class *plan_operator* and virtual base classes *po_with_1_input* and *po_with_2_inputs* for POs with one or two input streams. In the root class, a pure virtual function *po_func* is defined. However, only the interface (the dynamic parameters) of that function is specified, concrete function code is associated to *po_func* only in the derived classes.

From these virtual base classes, concrete classes are derived, one for each PO-CLASS, e.g., a PO for accessing the server (SERVER), SELECT, or JOIN. Moreover, virtual function *po_func* is filled with appropriate function code, and additional fields (for static parameters) are possibly added to the classes. To actually build query graphs, instances of these concrete classes must be generated and put together, as in our Lisp implementation. Concerning efficiency, this approach has a start-up time comparable to that of the Lisp implementation, on the other hand, parameter passing is not as overly flexible as in the

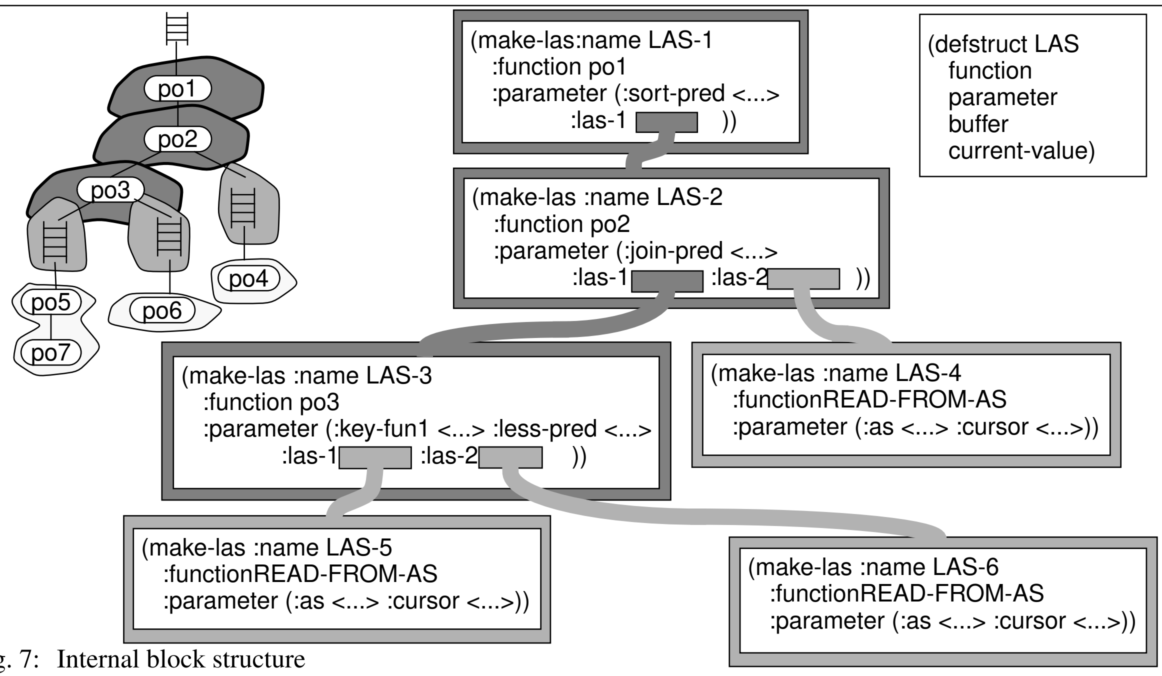


Fig. 7: Internal block structure

case of the *apply*/*keyword* mechanism due to a strict distinguishing between static and dynamic parameters. Compared to pure C, however, C++ still offers some flexibility due to function dispatch at run time. Consequently, the overhead per PO call is less than the *apply*/*keyword* mechanism; thus the break-even point with explicit compilation is shifted accordingly (cf. Fig. 8, upper part).

4.3 Comparison of Implementation Alternatives

In the previous part of Sect. 4, we proved the feasibility of DCA for different implementation environments.It became obvious that choosing a programming platform for DCA is primarily a matter of weighing flexibility against efficiency. The table in Fig. 8, lower part, clearly demonstrates the advantages and disadvantages of each implementation alternative.

The Lisp implementation is not the most efficient one, but it is highly flexible. For a testbed environment, flexibility outweighs rigorous performance considerations. However, in performance-critical scenarios, one would switch to more performance-oriented implementation platforms, e.g., C++, trading flexibility for performance.

5. Conclusion and Related Work

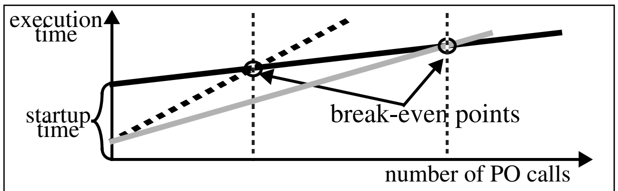
The scope of this paper is client-based query processing in advanced DBMS which are usually conceived for client/server environments. Application-oriented processing is performed mostly in the main-memory buffer at the client. To dynamically exploit the buffer contents for query processing, advanced DBMS should support run-time optimization. This measure must be complemented by a flexible and efficient code-generation phase. *Dynamic code assembly*, our approach to achieve this objective, allows to combine pieces of precompiled code fragments to executable code reducing explicit code generation to a minimum. We exemplified dynamic code assembly referring to the KBMS KRISYS and its implementational platform Common Lisp. Further, we showed the feasibility of our approach for any other C-like language.

Dynamic code assembly is based on a strictly orthogonal specification and implementation of the concepts underlying the plan-operator level. These concepts, which can be found in most (advanced) DBMS, comprise

- knowledge-model semantics (base predicates),
- plan-operator functionality,
- communication between plan operators (LAS),
- main-memory data structures (AS), and
- units of execution (blocks).

This modular partitioning of concepts allows to efficiently handle the tasks involved in code generation at run time. Our approach as well as the mechanisms necessary for its implementation are not restricted to KRISYS but are generally valid for (advanced) DBMS requiring flexible and efficient generation of executable code at run time.

Run-time optimizations may be motivated either by the desire to flexibly adjust execution strategies, as pursued by Volcano [10], or by the desire to dynamically exploit buffer contents, as proposed for ADMS [3]. In Volcano, meta-operator *choose-plan* was introduced that allows to construct an optimized query execution plan from a set of alternative plan fragments prepared by the optimizer. This solution is different from the one presented in this paper, since it allows flexibility only on the level of plan fragments, i.e., on already generated code, and not on the underlying PO level. For efficiency reasons, ADMS integrates matching and query optimization: The query graph is reduced by those parts that match to cached query results and organized by a specific data structure called *logical access-path schema*. In our case, representation of buffered and intermediate (cached) data coincide. Hence, matching and optimization operate on a single representational framework, thus simplifying query-processing implementation. In general, it is difficult to compare our approach to code generation in other DBMS (including Volcano and ADMS), since there is hardly any detailed information on that topic in literature. For relational approaches, mostly System R is mentioned which, however, does not focus on run-time optimization [18]. To our knowledge, there are no OODBMS that offer comparable concepts for optimizing and processing arbitrary queries on the client's buffer contents. ObjectStore [2], for example, provides mainly simple search arguments (path expressions) for navigating the buffer contents. Selecting appropriate indices handling simple search arguments and execution are interleaved. This optimization measure is different to our approach of employing run-time optimization before execution.



	explicit compilation	DCA in Lisp	DCA in C++
notation			
extensibility	low	high	medium
flexibility	low	high	high
start-up time	high	low	low
execution overhead	none	medium	low

Fig. 8: Comparing code-generation strategies

One of the primary goals for future work is to investigate the effects of dynamic code assembly on query optimization in KRISYS. The flexibility of this approach allows to generate a preliminary plan-level representation already at compile time. Hence, instead of generating a plan from scratch, run-time optimization can be restricted to adapting this solution to the current evaluation scenario. Moreover, it might be worthwhile storing plans of previously processed queries for later re-use. This may be advantageous for queries that are posed comparatively often, however, it must consider varying buffer contents.

References

[1] Austin Kyoto Common Lisp, Version 1.605, 1991.

[2] Cattell, R. (ed.): Next Generation Database Systems, Special issue of Communications of the ACM, Vol. 34, No.10, 1991.

[3] Chen, C.M., Roussopoulos, N.: The Implementation and Performance Evaluation of the ADMS Query Optimizer: Integrating Query Result Caching and Matching, in: Advances in Database Technology - EDBT '94, Jarke, M., Bubenko, J. (eds.), LNCS 779, Springer-Verlag, 1994, 323-336.

[4] Deßloch, S.: Semantic Integrity in Advanced Database Management Systems, Doctoral Thesis, Dept. of Computer Science, University of Kaiserslautern, Sept. 1993.

[5] DeWitt, D., Futtersack, P., Maier, D., Velez, F.: A Study of Three Alternative Workstation Server Architectures for Object-Oriented Database Systems, Proc. 16th VLDB Conf., Brisbane, Australia, 1990, 107-121.

[6] Deßloch, S., Härder, T., Leick, F.J., Mattos, N.: KRISYS - a KBMS Supporting the Development and Processing of Advanced Engineering Applications, Bayer, R., Härder, T., Lockemann, P. (eds.): Objektbanken für Experten, Springer 1992.

[7] Deßloch, S., Leick, F.J., Mattos, N.M.: A State-oriented Approach to the Specification of Rules and Queries in KBMS, ZRI-Report 4/90, University of Kaiserslautern, 1990.

[8] Freytag, J., Maier, D., Vossen, G.: Query Processing for Advanced Databases, Morgan Kaufmann, 1993.

[9] Graefe, G, DeWitt, D.: The EXODUS Optimizer Generator, Proc. 1987 ACM SIGMOD Conf., San Francisco, 1987, 160-172.

[10] Graefe, G.: Volcano, an Extensible and Parallel Dataflow Query Processing System, to appear in: IEEE Transactions on Knowledge and Data Engineering, 1993.

[11] Haas, L. Freytag, J., Lohman, G., Pirahesh, H.: Extensible Query Processing in Starburst, in: Proc. ACM SIGMOD Conf., Portland, 1989, 377-388.

[12] Härder, T., Reuter, A.: Database Systems for Non-Standard Applications, Proc. Int. Computing Symposium on Application Systems Development (ed. H.J. Schneider), Nuremberg, Germany, March 1983, Report 13 of the German Chapter of the ACM, Teubner Verlag, Stuttgart, 452-466.

[13] Hong, W., Stonebraker, M.: Optimization of Parallel Query Execution Plans in XPRS, Distributed and Parallel Databases, Vol. 1, 1993, 9-32.

[14] Eich, M. (ed.): IEEE Transactions on Knowledge and Data Engineering, Special Issue on Main-Memory Databases, Vol. 4, No. 6, 1992.

[15] Jarke, M., Koch, J.: Query Optimization in Database Systems, Computing Surveys, Vol. 16, No. 1, June 1984, 111-152.

[16] Kim, W.: Introduction to Object-Oriented Databases, Computer System Series, MIT Press, 1991.

[17] Lippman, S.: C++ Primer, Addison-Wesley, 1989.

[18] Lorie, R., Wade, B.: The compilation of a high level data language, IBM Research Report RJ 2589, 1979.

[19] Mattos, N.M.: Abstraction Concepts: the Basis for Data and Knowledge Modeling, 7th Int. Conf. on Entity-Relationship Approach, Rome, Italy, Nov. 1988, 331-350.

[20] Mattos, N.: An Approach to Knowledge Base Management, in: LNCS 513, Springer-Verlag, 1991.

[21] Pirahesh, H., Mohan, C., Cheng, J., Liu, T.S., Selinger, P.: Parallelism In Relational Database Systems: Architectural Issues And Design Approaches, IBM Research Report RJ 7724, 1990.

[22] Pirahesh, H., Mohan, C.: Evolution of Relational DBMSs toward Object Support: a Practical Viewpoint (invited talk), Proc. GI-Fachtagung "Datenbanksysteme in Büro, Technik und Wissenschaft", Kaiserslautern, March 1991, Informatik-Fachberichte 270, Springer-Verlag, 1991, 16-37.

[23] da Rocha, Rafael: Transformation and Rewrite in the Query-Processing System of KRISYS, Master Thesis (in portuguese), UFRGS, Porto Alegre, May 1992.

[24] Sequent Computer Systems: System Summary, 1990.

[25] Steele Jr., G. et al.: Common Lisp - The Language, 2nd edition, Digital Equipment Corp., 1990.

[26] Thomas, J., Deßloch, S.: A Plan-Operator]Concept for Client-Based Knowledge Processing, Proc. 19th VLDB Conf., August 1993, Dublin, Ireland, 555-565.

[27] Thomas, J., Mitschang. B., Mattos, N.: Parallelism in Client-Based Knowledge Processing - The KRISYS Approach. SFB-Report 25/93, University of Kaiserslautern, 1993.

[28] Thomas, J., Mitschang. B., Mattos, N., Deßloch, S.: Enhancing Knowledge Processing in Client/Server Environments, Proc. 2nd Int. Conf. on Information and Knowledge Management, Nov. 1993, Washington, D.C., USA, 324-334.

[29] Valduriez, P., Danforth, S., Hart, B., Briggs, T., Cochinwala, M.: Compiling FAD, a Database Programming Language, Proc. 2nd Workshop on Database Programming Languages, June 1989, 375-393, 1989.