in: Proc. 19th Int. Conf. on Very Large Database Systems,Dublin, August 1993, pp. 555-566.

A Plan-Operator Concept for Client-Based Knowledge Processing

J. Thomas, S. Deßloch
University of Kaiserslautern, CS Department
P.O.Box 3049, 6750 Kaiserslautern, Germany
e-mail: {thomas | dessloch}@informatik.uni-kl.de

Abstract

In knowledge processing frameworks for advanced DBMS (such as OODBMS or KBMS) suitable for client-server architectures, the efficient realization of client-based, main-memory query processing represents a promising and important step towards an effective support of application processing. In this paper we present a plan-operator concept developed along these lines as part of an algebraic query-processing framework for the KBMS KRISYS [24]. We motivate the main goals (support of extensibility, client-based query processing, and dynamic query optimization) as well as the resulting design objectives followed and give a detailed description of the specification and implementation of the resulting plan-operator concept. It can be characterized as highly modular and orthogonal w.r.t. the overall functionality and allows a flexible and extensive utilization of precompiled code fragments. Additionally, we sketch the advantages of the concept w.r.t. our design objectives and point out that the presented ideas are not limited to KRISYS but also apply in a more general setting.

1. Introduction

Effective support of non-standard applications such as engineering and design, multi-media, or office automation not only requires efficiently managing large amounts of data, but also calls for a representational framework with sufficient expressive power to adequately model these applications. To this end, Object-Oriented Database Systems (OODBMS) [2] [20] [25] [28] as well as Knowledge Base Management Systems (KBMS) [3] [24] [30] feature an object-oriented knowledge model and an associated query language.

In analogy to conventional database systems (DBMS), both types of systems can employ an algebraic approach for query processing [5] [31] [33], furthermore called *knowledge processing*. In this approach, a query is firstly mapped into an algebraic representation, which is then successively rewritten and transformed into an execution plan consisting of plan operators, before it is actually performed.

Proceedings of the 19th VLDB Conference,
Dublin, Ireland, 1993.

An adequate concept for query processing, and for plan execution in particular, is subject to a number of requirements originating from both the properties inherent in the underlying data model or query language and the processing characteristics of the (potential) applications.

(1) Extensibility:

To satisfy specific needs of advanced applications, *extensibility* must be supported at different levels [11] [12] [13]: To cope with later extensions either of the query language (e.g., due to new requirements arising from advanced applications) or of evaluation methods (such as improved join algorithms), a plan-operator concept must exhibit flexibility and follow a modular design that clearly separates the overall plan-execution functionality into orthogonal parts. This requirement is also emphasized by the general goal of a simple, streamlined design of the execution framework.

(2) Client-Based Knowledge Processing:

Non-standard applications are often long-running activities which exhibit a sufficiently high degree of locality of reference. Measurements [23] strongly advocate for the maintenance of a main-memory application buffer that supports locality of processing, the accumulation of updates/changes, and an enhanced representation of objects for efficient application processing (e.g., through the materialization of object references). Such an approach is especially required in the scope of client/server-environments (also called workstation/server environments) to minimize communication traffic between different system components located on separate machines [7] [14] [18]. Since the expressive power of the query language should be available not only for loading/unloading the buffer (i.e., check-out/check-in operations), but also for knowledge processing tasks within the application (in the scope of the buffer), *main-memory based query processing* has to be supported in an efficient manner [8]. Besides the need to cope with the enhanced object representation in the buffer, the execution of plan operators in this scope must avoid the introduction of redundancies within the buffer, both for efficiency and for consistency reasons. Additionally, the use of main-memory indices should be supported as a means to cope with application buffers of increased size.

(3) Dynamic Query Optimization

Query optimization in conventional DBS can usually be done at compile time. This is in some cases not guaranteed in the scope of object-oriented query languages [27]. For example, if the names of classes or collections (i.e., sets of objects independent from the extensions of classes) are allowed as parameters of methods (and queries contained), optimization must be carried out at run-time. Additionally, run-time dispatch of methods must be performed, if overriding of methods occurs. Also, if we rely on the main-memory based query-processing approach described above, it is often more efficient to defer some of the optimization tasks. For example, if the knowledge about main-memory indices would have to be compiled into the code of methods that contain embedded queries, a flexible and dynamic creation of such indices could not be supported, since they could not be exploited by methods or would require extensive recompilations. For the same reason, a main-memory index that needs to be built up and maintained only for a specific application to support its individual processing characteristics could either not be exploited in the processing of methods shared with other applications, or such methods would have to be individually recompiled for specific applications. For these reasons, an adequate plan-operator concept has to support an efficient execution of plan operators also in case the *execution plan is completely known only at run-time*. Hence, there is a need for dynamic query optimization. This means that either an efficient interpretation or a fast compilation of execution plans needs to be provided, which should to a large extent rely on (pieces of) precompiled code that can later on be combined in an effective and efficient manner.

The goal of this paper is to present the specification and implementation of a plan-operator concept that meets these requirements. It has been realized in the framework of KRISYS [24], a Knowledge Base Management System (KBMS) developed at the University of Kaiserslautern. Sect. 2 gives an overview of KRISYS, i.e., of its knowledge model, its query language and its processing model. In Sect. 3, we will outline the conceptual and operational requirements for the plan operators, and subsequently, in Sect. 4, we will discuss their implementation. In Sect. 5, we will validate the realization w.r.t. the initial requirements, and we will show that it is feasible not only for the implementational environment of KRISYS, but also for a more general setting. Besides a resume of the main ideas of the paper, Sect. 6 also gives an overview over current and future development activities.

2. An Overview of KRISYS

2.1 Knowledge Model and Query Language

The knowledge model of KRISYS is comparable to object-oriented data models [4] [19]. An object is uniquely identified by a name (i.e., object-identifier), and contains a set of attributes to describe its characteristics. Attributes can be of two kinds: *slots* are used for representing properties of an object and relationships to other objects; *methods* are used for expressing object behavior. Moreover, attributes can be further described by *aspects*, defining, e.g., the cardinality of a slot. For object structuring, our knowledge model supports the abstraction concepts of classification, generalization, association, and aggregation [22] [26]. The special semantics of these relationships is guaranteed by the system (e.g., inheritance along the classification and generalization relationships). Fig. 2.1 shows an example hierarchy and the overall structure of one of the objects contained.

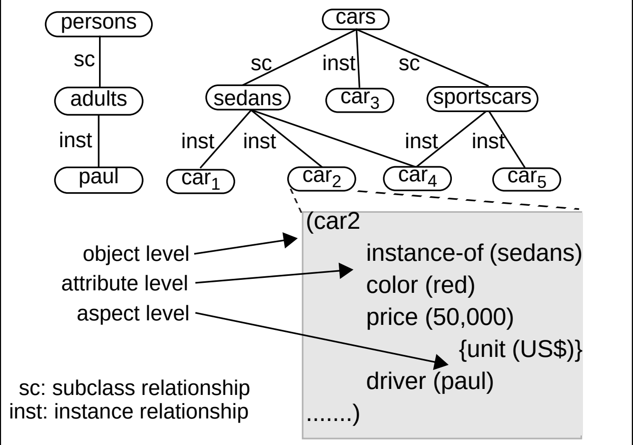


Fig. 2.1: Example hierarchies and an object structure

In contrast to DBMS, KRISYS does not distinguish between schema-information and instance-information - both is represented using the concept of objects (a similar approach is taken in [20]). Objects can therefore represent instances, classes, sets, elements, aggregates, etc.

In addition to the above described concepts, the knowledge model of KRISYS provides various other features not usually found in object-oriented models, as, e.g., integrity constraints and rules. For the scope of this paper, an in-depth discussion of these concepts is not necessary (see [24] for details on rules and [6] for details on integrity constraints).

Retrieval and modification of knowledge base (KB) contents is supported by KOALA [9] [24], a descriptive, set-oriented language constituting the user and application 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 retrieves instances of red sportscars and the persons being the drivers of the cars (i.e., referenced via slot 'driver').

```
(ASK ((?driver SLOTS age) (?car SLOTS price))
     (AND (IS-INSTANCE ?driver persons)
          (IS-INSTANCE ?car sportscars)
          (EQUAL (SLOTVALUE color ?car) red)
          (EQUAL ?driver (SLOTVALUE driver ?car)))))
```

Symbols with a leading question mark are query variables, similar to table variables in SQL, to denote the information to be retrieved. These variables may also appear in the projection clause (the first clause in the ASK statement). In our example, the projection clause states that (in addition to the object names) slot age for persons and slot price for cars is to be included in the result of the query. We will come back to this example query in subsequent parts of this paper.

2.2 Knowledge Processing in KRISYS - the Main Ideas

KRISYS was conceived to support knowledge processing in a client/server environment, which can be seen as the dominating hardware environment for complex, non-standard DB applications. In such environments, applications run on dedicated clients having access to a central server component responsible for an integrated and effective management of information. For the purpose of this paper, it is sufficient to take a more general view on the KRISYS architecture, shown in Fig. 2.2. (see [8] for more details).

The overall architecture is motivated by the following primary goals, necessary to achieve efficient knowledge processing in client/server environments.

- Application-oriented processing at the client side:

To exploit the processing capabilities of the client components and keep the server component from being overloaded, application-oriented processing must be performed at the client side. The server component may concentrate on the effective and efficient management of data [16].

- Loose coupling:

For efficiency and reliability, a loose coupling of client and server components that reduces communication efforts and dependencies between both sides must be achieved [14].

The first measure taken in order to fulfil these requirements was the introduction of a system-controlled application buffer (called working-memory, WM) at the client side. It allows to exploit the applications' locality of reference. By storing currently referenced knowledge, the WM serves to minimize the number of calls to the server DBMS. Thus, if the application only refers to information already residing in WM, no calls to the DBMS have to be issued at all.

The second measure taken is motivated by the expressiveness of the modeling concepts provided by KRISYS. As already mentioned, the knowledge model offers various concepts suitable for modeling application-oriented processing tasks. Consequently, the system components involved in the execution of these constructs must be located at the client side - otherwise the server would be overloaded with application-oriented processing tasks. This holds also for the processing of KOALA statements, the fundamental operational basis of the above mentioned concepts, and for significant parts of the knowledge model (the semantics of the abstraction concepts) which are directly required for this task. Thus, only a part of the processing related to the evaluation of a query, we denote it *data processing*, is performed by the server DBMS (the DBMS kernel PRIMA [15] is employed for this task). The remaining, more complex tasks are carried out by the knowledge-processing component itself on the client side. (A detailed discussion can be found in [6] [8] [24].)

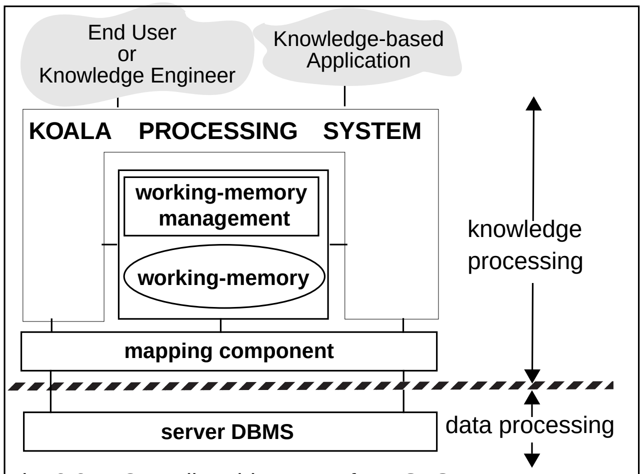


Fig. 2.2: Overall architecture of KRISYS

Knowledge processing in KRISYS thus involves both the server DBMS and the system components of the client. If information referenced by a query must be fetched from the server, the mapping component is invoked. Its main purpose is to conceal details of how knowledge is actually mapped to the primitives of the underlying DBMS[1] to make the upper system components independent of it. Queries can be posed to the mapping component in a meta language supporting a subset of the functionality of the knowledge model. The mapping component then translates such queries into a (set of) queries formulated in the query language of the server DBMS, since the knowledge model of KRISYS cannot be mapped directly to the data model of the server component. Results coming in from the server DBMS consequently have to be transformed into the representation of the knowledge model. Thus, except for the mapping component itself, all processing in the client refers directly to the knowledge model, and not to the actual representation in the DBMS.

2.3 Working-Memory Management and Representation

The above described architectural decisions clearly indicate, that a purely enumerative description of the buffer contents (based on object-identifiers) as well as server requests based

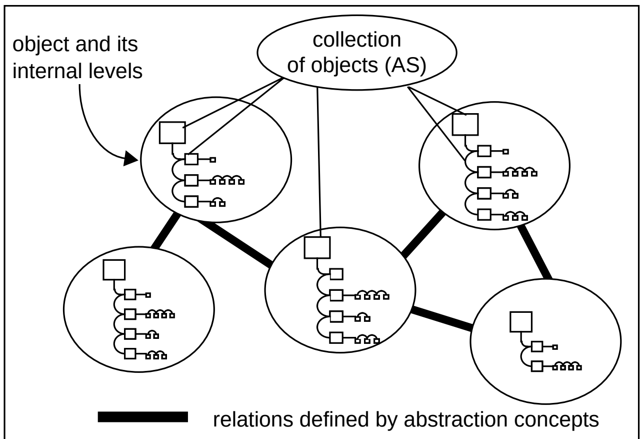
1. E.g., the relation(s) actually employed for representing an object class, indices defined over those relations, etc.

on object names are not sufficient for our tasks. While a navigational style of processing (i.e., following object references via object names) would be directly supported by such an 'object-faulting' approach, the processing of associative queries apparent in methods or inherent in the processing of rules and integrity constraints could not exploit the buffer contents in a satisfactory way. Consider our example query presented above, where red sportscars are involved. Even if a previous query had already caused the installation of all red sportscars in WM, this fact could not be exploited because it is not known to the system. To evaluate the query predicate 'color = red', all sportscars (or at least those not already present in WM) would be unnecessarily transferred to WM.

For this reason, KRISYS must be able to relate the knowledge required by a given KOALA query to the objects already stored in WM. This is achieved by defining the WM contents descriptively, similar to the way KOALA allows to specify collections of objects (e.g., the WM contains 'all red sportscars'). Thus, the problem described above can be dealt with by performing subsumption tests, comparing the predicates of the query with the ones describing the buffer. The results of these tests, which must be performed at run-time, are then used for deciding which parts of a query are directly performed in WM, and which ones have to be delegated for execution to the server DBMS. The component of KRISYS which is responsible for maintaining the buffer description and performing the tests is called *working-memory manager* [29]. Discussing the mechanisms incorporated in this component is beyond the scope of this paper. We leave this issue to a further publication.

The representational framework of the WM (i.e., how objects are actually stored) directly reflects the characteristics of the knowledge model of KRISYS. This model defines three types of relationships for objects. Firstly, there are the relationships within an object, consisting of links from the object name to its attributes and their descriptions (cf. Fig. 2.1). Secondly, there are the relationships among objects. The most important of these relationships are the abstraction concepts, forming abstraction hierarchies which are frequently traversed during knowledge processing. Both types of relationships are materialized in WM using main-memory pointers, thereby providing fast access to the required information. Moreover, we need to support efficient access to and set-oriented processing of arbitrary collections of objects, e.g., intermediate results in query processing. This is provided by the concept of so-called *access structures* (AS). In their basic form, AS are organized as lists of objects and comprise operations for traversing an AS based on a cursor concept. However, AS may also be organized as trees or hash tables, thereby taking the role of main-memory indices. (Of course, additional functionality for associative access is provided in this case.) Main-memory indices (or AS in general) may be introduced dynamically and temporarily (e.g., in the scope of a single query) in order to speed up query processing. Thus, knowledge residing in WM is organized as shown in Fig. 2.3[2].

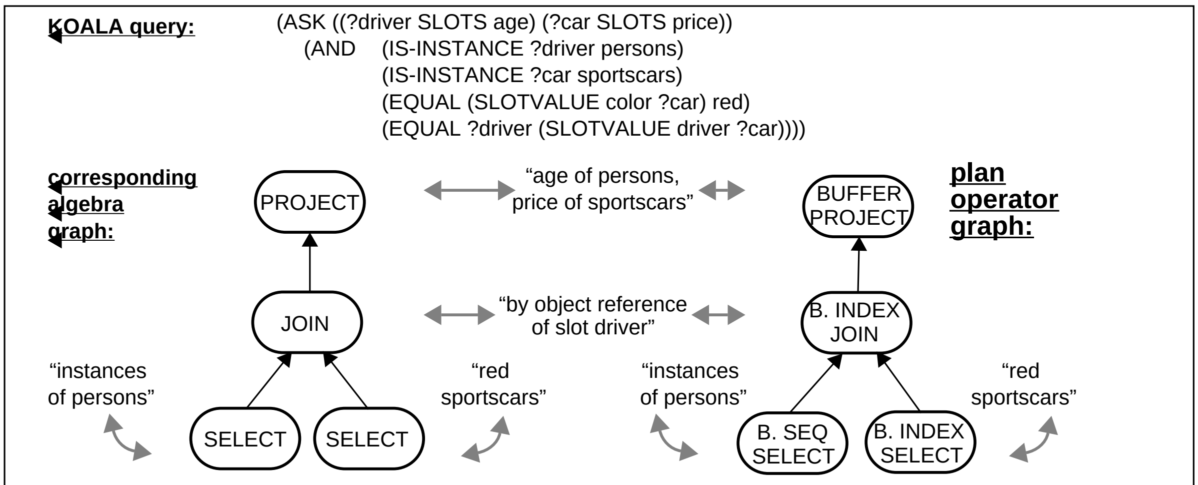
The central idea of the data structures chosen for the representation of these three types of relationships is to avoid redundancy whenever possible. Consequently, no copies of objects are created during query processing as intermediate or final results. Instead, pointer structures are maintained to provide the access, thereby drastically reducing space overhead for temporary processing results and providing a consistent basis for update operations, which do not have to worry about the possible existence of replicated objects[3].


object and its internal levels
collection of objects (AS)
relations defined by abstraction concepts
Fig. 2.3: The organization of knowledge in WM

2.4 KOALA Processing System

Knowledge processing is performed by the KOALA Processing System (KPS). KPS realizes an algebraic processing model that allows conventional algebraic optimizations to be used to a large extent [27]. Thus, the overall steps of knowledge processing proceed in a similar fashion as the well-known steps of data processing in relational DBMS [13]: first, an algebra graph representing the query is generated and subsequently optimized, i.e., rewritten; then, a plan-operator graph is constructed; finally, executable code is generated, and the query is actually evaluated. Queries are transformed into an algebra graph that supports the operations inherent to the knowledge model. On one hand, it consists of algebra operators showing a functionality that can also be found in conventional relational algebras (e.g., SELECT or JOIN). On the other hand there are operators specific for knowledge processing in KRISYS (e.g., for unnesting/nesting of object structures). Without discussing the KOALA algebra in detail, we want to give an impression of how a query is translated into an algebra representation in

2. In addition to the depicted organizational structures, a global hash-table allows the localization of objects via object names.
3. Redundant information occurs for example in generalization hierarchies, where subclasses may share attributes with their superclasses. Representing such attributes only once for all classes involved is an important goal of the data structures for the WM.


KOALA query:

```
(ASK ((?driver SLOTS age) (?car SLOTS price))
  (AND   (IS-INSTANCE ?driver persons)
         (IS-INSTANCE ?car sportscars)
         (EQUAL (SLOTVALUE color ?car) red)
         (EQUAL ?driver (SLOTVALUE driver ?car))))
```

Fig. 2.4: An example query and the corresponding algebra representation

Fig. 2.4 using the sample query already introduced asking for "all persons driving a red sportscar" (as for example Paul).

The algebra (and plan-operator) graph of Fig. 2.4 also contains predicates to further "guide" the application of the algebra operators. These predicates, which we call *base predicates*, define the actual semantics of KOALA, i.e., the predicates to evaluate while executing the algebra operations. The base predicates must be mapped to the representational framework of the underlying knowledge model. For example, to fulfill the predicate "instances of persons", an object must be either a direct instance of *persons* or a direct instance of a subclass of *persons* (e.g., *adults*). These relationships can be verified by the (chain of) references defined by the slots *instance-of* and *subclass-of* of the object.

At compile time, the algebra graph is optimized according to algebraic and non-algebraic criteria. When transforming an algebra graph into a plan-operator graph, the KPS must exploit the architectural characteristics of KRISYS. (For this reason, most of the transformations can be done only at run time.) It first has to communicate with the working-memory manager to identify the information that is already installed in WM. The remaining knowledge must be fetched from the server, to which end specific plan operators (called DBS-operators) are used to interact with the mapping component and install the results of the delegated subquery in WM. All subsequent tasks of query evaluation are then performed in WM. Since the scope of this paper is determined by client-based query processing (i.e., execution of plan-operators on the buffer), we have not introduced DBS-based plan-operators in our sample query. In the following, we assume that (supersets) of the information required for the selections in our query are already contained in WM.

Summing up, the optimization efforts must take into account client processing as well as server processing. If all objects referenced by a query are already installed in WM, it is clearly more efficient to evaluate the query in the client. In this case, KPS exploits existing main-memory indices and chooses appropriate plan-operators (in our example, we have assumed the existence of an index on the 'color'-attribute for sportscars). If, however, a subset of the required objects must be fetched from the server, the costs for performing the query in either client or server component become relevant. The costs related to the client are dominated by the efforts necessary to fetch the missing objects and to propagate the changes back to the server DBMS; those related to the server component are heavily influenced by the existence of database indices. Such information is maintained by the mapping component and supplied to the KPS.

Changes of objects are accumulated in WM and propagated to the server DBMS as well as to all contexts affected by the changes at appropriate points of time [29]. This problem is even more complex if multiple clients operate simultaneously on a KB. Since this is a problem sphere apart from the one under discussion, we do not consider concurrency in this paper.

In the next section, we will take a closer look at the conceptual and implementational requirements of plan operators for knowledge processing in KRISYS.

3. Requirements for the Plan-Operator Level

Since the realization of knowledge processing in KRISYS is a prototypical one, we aim at a maximum degree of freedom, concerning both later extensions of the conceptual design and the actual use of the concepts. This division into conceptual and operational issues is also reflected in the requirements for the plan-operator level of KRISYS which we will present subsequently. For the considerations following, we will concentrate only on the operators to be performed on the contents of the WM. Those plan operators hosting tasks to be delegated to the server DBMS basically consist of calls to the mapping component and are therefore much less complex than those for the WM.

3.1 Conceptual Requirements

A Simple Processing Paradigm for Plan Operators

A simple yet highly flexible processing of plan operators is the basis for effective knowledge processing. This can be achieved by an easy realization of the flow of both data and control through the plan operators. To this end, we see plan operators as functional units accepting one or more input streams and producing exactly one result stream. The input is read and processed until it is exhausted. Thus, the plan operators follow an open-next-close paradigm to be found in many query-processing systems, e.g., [12].

Extensibility at the Plan-Operator Level

As already pointed out in the introduction, *extensibility* must be supported at different levels to meet the specific requirements of advanced applications. Extensibility, in the context of plan operators, means providing new evaluation strategies for algebra operators by specifying new plan operators, as, e.g., new join algorithms. On one hand, this requires a well-defined structure (“template”) for the plan operators which can be used for embedding the desired new functionality. On the other hand, new plan operators must also be integrated into non-algebraic optimization. This can be achieved by providing heuristics describing the applicability of the new operators. As optimization is a problem apart from the one under consideration, we will not discuss it in this paper.

Extensibility of the Query Language

Extensibility of the query language is important to ensure a maximum and optimally adjusted support of the applications by KRISYS and its query-processing facilities. If the applications can tailor the interface to their special needs, the amount of processing to be done by the applications themselves (expressed in a programming language with an interface to KRISYS) is minimized. This means that - based on the representational framework of the knowledge model and the semantics of ASK and TELL - it should be possible to replace or augment the predicates and functions applicable, yet affecting the realizations of the algebra level, the plan level or the code generation as little as possible. Thus, extensibility may for example comprise enhancing the semantics of inheritance or including new, system-defined relationships and their semantics into the user interface of KRISYS. Unlike EXODUS or Volcano [11] [12], we do not want to generate a new optimizer for each dialect that comes into existence by extending KOALA, nor do we aim at extensibility of the data-management facilities, as done in Starburst [13].

3.2 Operational Requirements

Efficiency of Dynamic Query Optimization

Knowledge processing in KRISYS requires dynamic query optimization, since many related decisions can only be taken at run time. A maximum freedom of choice can be achieved by an interpretative approach to query evaluation. However, generating executable code should require as little compilation effort as possible, calling for a compilative solution where any modification of a plan-operator graph inevitably means recompilation. Although both requirements seem to contradict each other, they are equally important for a good system performance, since they facilitate plan construction and code generation, thus reducing the time required for both.

Flexible Units of Execution

Each plan operator can be seen as a separate unit of execution. This is, however, not very efficient, since some kind of execution control is required for each plan operator. Plan operators may be combined in a single unit of execution according to their processing characteristics. We distinguish tuple-oriented plan operators and set-oriented ones[4]. A sequence of tuple-oriented plan operators can be seen as a pipeline that is activated for each input tuple. Thus, a uniform execution control can be defined for this operator sequence. Additional criteria may be applied to construct units of execution, e.g. to prepare different forms of parallelism [32].

Efficient Data Flow between Plan Operators

The WM is the operational platform on which knowledge processing is performed. Using up as little space as possible for this task is a prerequisite for an effective use of the WM. This can only be achieved if intermediate results that must be materialized during the evaluation of a query can be kept small. In certain scenarios, however, a materialization of intermediate results is desirable. This is the case each time several plan operators have the same predecessor. As this plan operator (like any other plan operator) produces a single output stream, the subsequent consumers can only share it, if it is physically existing in WM and distinct access paths (e.g., cursors) can be maintained, one for each reader. Another example where intermediate results must be materialized might be the input to the join operator. If the join is realized via a nested loop, it must access the inner input several times. Hence it is advantageous to have the tuples physically represented in WM.

On the other hand, for a sequence of tuple-oriented plan operators, it is desirable not to materialize any intermediate results but just the inputs to the first plan operator and the output of the last plan operator of the subgraph.

4. If a plan operator can work on each input tuple separately, we call it *tuple-oriented*. A plan operator works *set-oriented*, if it can exert its functionality in a reasonable fashion only if the complete set of input tuples is available (as for example a sort operator).

4. Implementational Issues

4.1 Basic Structure of Plan Operators

Subsequent plan operators in a plan-operator graph are in a producer/consumer relationship. To get new input, a plan operator issues calls to its predecessors. Those, in turn, reply by producing a new output tuple. The consumer should not have to care about how its next input is generated, nor should the producers have to worry about how their output will be consumed. An important objective for our implementation thus is that plan operators must work independently of their position. The location of a plan operator within a plan operator graph must not matter, nor may the preceding/succeeding plan operators have an impact on its processing behaviour. Moreover, to minimize the number of control calls between plan operators, plan operators ask for new input until the next valid output (w.r.t. some qualification condtion) can be generated, or the end of the input is reached.

The resulting generic processing scheme for obtaining a new output of a plan operator (e.g., a selection) is shown in Fig. 4.1. It comprises the communication with the preceding/ succeeding plan operators ("get", "write") and the actual functionality of the plan operator itself ("process" and qualification conditions). Both tasks are embedded into a control structure guiding their application (while-loop). As soon as a plan operator needs input from its predecessors, it calls them to produce their next output tuples, and execution control is passed on to the corresponding plan operators. Together with their answers, they return execution control back to the calling plan operator. There, the input is processed, and a (possibly existing) qualification condition is applied to the result. The process outlined above is repeated until the next valid output tuple has been produced or the input is exhausted.

```
plan-operator(input_sources, output, base_predicates)
    while {not end of input_sources} do
        input ← get({new_input});
        result ← process(input);
        if {qualification conditions apply to result}
                output(result, output_destination); stop
    end
```

Fig. 4.1: Processing scheme for a plan operator

The qualification condition is stated in terms of base predicates with which the plan operator is supplied upon activation. Moreover, the interface includes the sources from where to get the input and the destination where to write the output to. For both purposes we can employ the concept of AS introduced in Sect. 2. Reconsidering the plan-operator graph of Fig. 2.4, we can imagine each arc between two plan operators to be replaced by an AS. The AS serves as output destination for the preceding plan operator and as input resource for the succeeding one. Yet, this approach has a disadvantage that advocates for its refinement. As pointed out in Sect. 3, we distinguish tuple-oriented and set-oriented plan operators. Sequences of tuple-oriented operators can be regarded as pipelines that are activated separately for each input tuple. From a logical point of view, there is no need to materialize any intermediate results inside the pipelines. Exactly this, however, is done by employing AS as communication channels between the operators of the pipeline. Each intermediate result would have to be installed in an AS of the WM. To avoid this, we introduce the concept of *blocks*.

4.2 Blocks

Blocks are units of execution comprising a number of plan operators with the same processing characteristics. Just like plan operators, 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 plan operator(s) contained. Blocks communicate among each other via WM, employing access structures for input and output. By defining the appropriate number of cursors over the same AS, several blocks can use an AS as common input source. The state of a block is defined by the states of the AS at its beginning and at its end (i.e., the position of their cursors). Inside blocks, intermediate results are kept temporarily. To this end, we need an internal data structure for blocks and a mechanism to generate it from or to transform it into an AS. In addition, the control flow between plan operators of a block must be realized. Both topics are addressed in the following.

4.2.1 Communication within Blocks

One important aspect to the communication within a block concerns the data structures by which knowledge is exchanged among the plan operators contained. A data structure suitable for the representational framework of KRISYS must support the relevant functionalities of the knowledge model, i.e., all the operations defined on a single object, as for example the insertion/deletion of an attribute. Plan operators do not need operations over collections of objects, since they map such functionalities to sequences of operations on single objects. Moreover, the data structure should be easy to generate from the representational framework of the WM, the AS. For these reasons, we decided to use as internal data format a representation similar to the one of AS. To distinguish both representations but to indicate their similarity, we named the internal data format *logical access-structures* (LAS), opposed to the "physical" AS of the WM. Structurally, LAS correspond to their "physical" counterparts, yet, only very few plan operators require the full functionality available on the "physical" AS. Most plan operators process their input sequentially and thus must be able to access the *next* tuple, the *actual* tuple and possibly the *first* of the input (e.g., in the inner loop of a join). Moreover, for plan operators that are intended to exploit a sorted input, it is desirable being able to directly access the next tuple having either the same value as the one last considered or having a different value. Therefore, the access modes *next-with-same-value* and *next-with-new-value* are necessary as well. This protocol must also apply to those plan operators that access an AS (remember the implementational objective that plan operators be independent from their actual position within a graph). Therefore we introduced two "pseudo" plan-operators *read* and *write*, realizing exactly this interface either for the AS to be read or for the one to be written and thus mask the existence of physical AS below or above them. Hence, a block consists of *read* operators supplying its input, the plan-operator subgraph to be executed in the scope of the block and exactly one *write* operator for the output of the block. All "real" plan operators within a block access their relevant information via the open-next-close interface of the LAS (cf. Fig. 4.2 (a)).

Plan operators that work in a set-oriented fashion (e.g., sort), might also be content with this simple open-next-close protocol, which, however, may restrict the flexibility of their implementation. Moreover, since certain operations on entire AS (i.e., sets of tuples) are already available as part of the functionality of AS (e.g., duplicate elimination), we decided to have the set-oriented operators work directly on AS and the corresponding interface. For this reason, the *read* and *write* functions are only sketched in Fig. 4.2 (b), showing the basic structure of a set-oriented plan operator.

Employing the concepts described so far, the plan-operator graph shown in Fig. 2.4 is transformed into a plan-level representation as depicted in Fig. 4.3. We assume the plan to be partitioned into two blocks. Both blocks communicate via the (physical) access sequence AS_3.

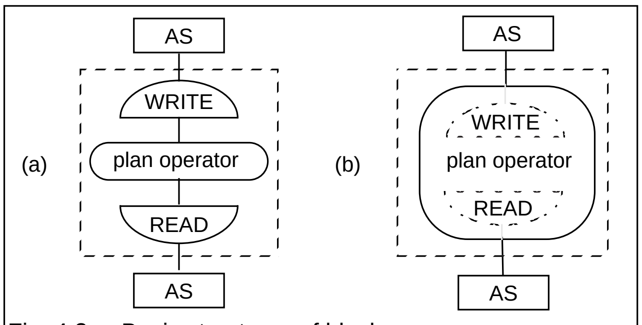


Fig. 4.2: Basic structures of blocks

Besides for the transmission of data, we employ LAS also for actually organizing the control flow among the plan operators of a block. In addition to the name of the underlying plan operator and its parameters, a LAS thus also possesses the entries *actual-value* and *buffer*. The former is used to store the value last computed by the underlying subgraph. It is necessary to avoid repeated activations of this subgraph. The latter entry (*buffer*) is indispensable, since several output tuples may result from a single invocation of a subgraph (e.g., a single invocation of an index select may return a set of tuples matching the selection), yet only one result tuple can be processed at a time. As long as the buffer is not yet empty, the entry *actual_value* is refilled with tuples of the buffer, before issuing a new call to the plan operator below. By doing so, the LAS also tells the underlying plan operator which tuple it currently needs, e.g., the *actual*, the *next* or the *first*. The plan operator in turn has to provide its input, before being able to start working. For this purpose, it addresses the corresponding LAS with their respective access modes via the function *get-entry-from*, that realizes the open-next-close interface described above. In the next section we will consider the processing of blocks in more detail.

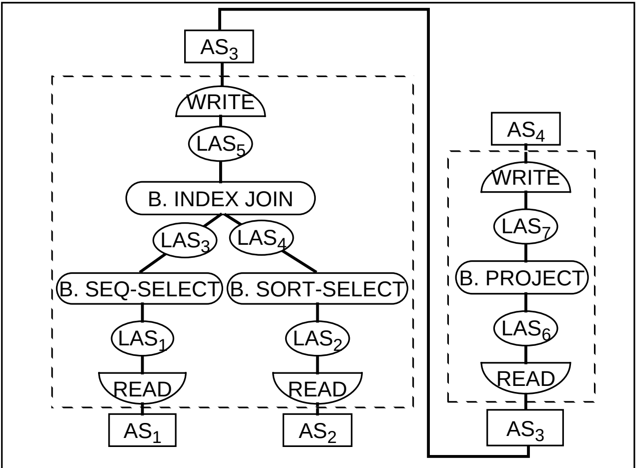


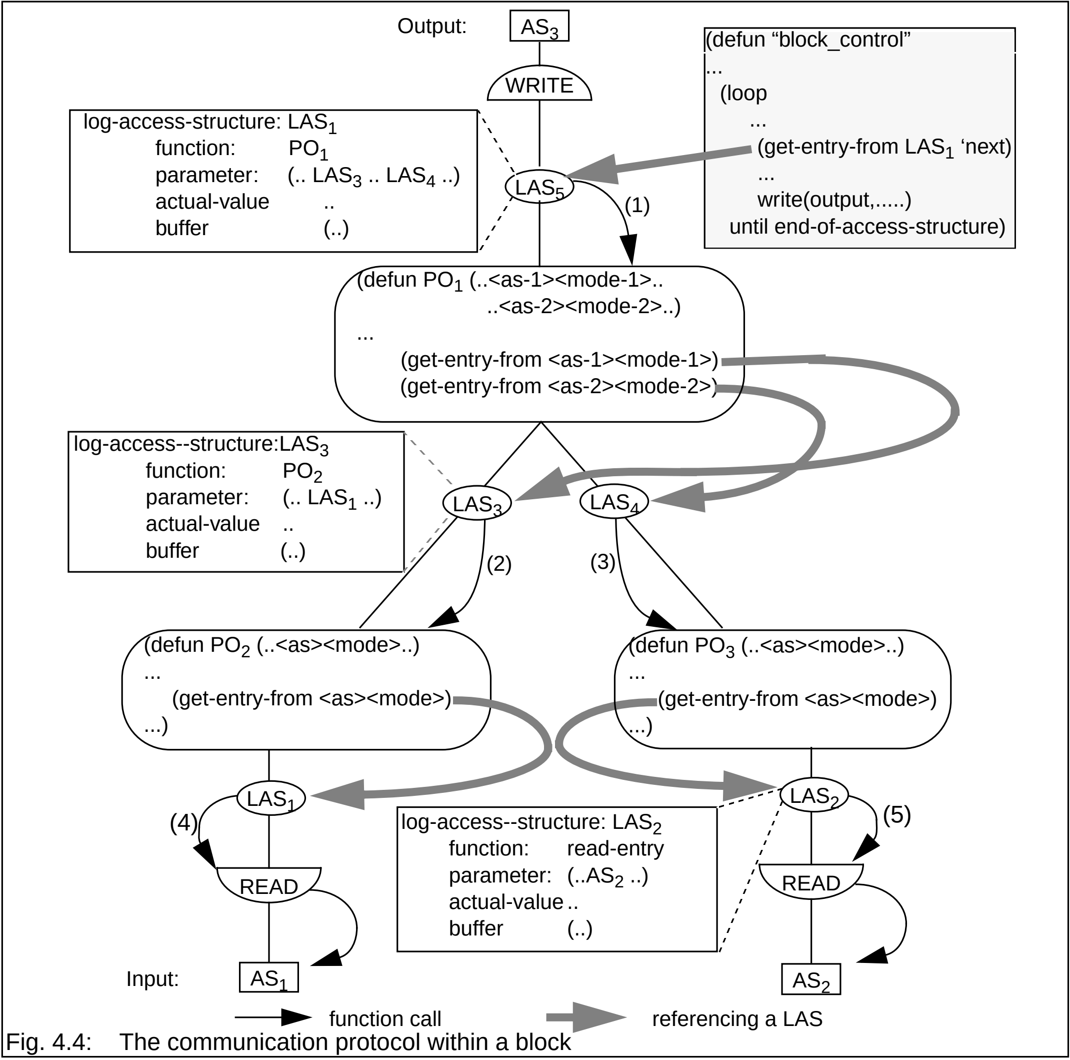
Fig. 4.3: Modified plan-operator graph of Fig. 2.4

4.2.2 Processing of Blocks

In the following, we will discuss the way in which a block is actually processed. For this purpose, we take a more detailed look at the left block of Fig. 4.3. It is depicted in Fig. 4.4 which also shows the basic structure of LAS and of plan operators. We abbreviated the names of the plan operators by PO_i.

Since a block is supposed to work in a set-oriented way when seen from outside, a block control is needed that repeatedly activates the subgraph contained in the block, until its input AS have been processed entirely. In Fig. 4.4, this block control is represented as a simple loop, yet other, more complex evaluation strategies may be desirable[5]. Before a block can actually start executing, some initialization measures must be taken. They relate, e.g., to the way in which the input AS are used by the *read* operators of the block. If an input AS is referenced exclusively by one block, the manipulations to be performed might be carried out "in place", i.e., directly modifying the input AS. If this access

5. This issue is currently being investigated; we will discuss it briefly in the outlook.


Output:
AS3
WRITE
(defun "block_control"
...
(loop
...
(get-entry-from LAS1 'next)
...
write(output,.....)
until end-of-access-structure)
log-access-structure: LAS1
function: PO1
parameter: (.. LAS3 .. LAS4 ..)
actual-value ..
buffer (..)
LAS5
(1)
(defun PO1 (..<as-1><mode-1>..
..<as-2><mode-2>..)
...
(get-entry-from <as-1><mode-1>)
(get-entry-from <as-2><mode-2>)
log-access--structure:LAS3
function: PO2
parameter: (.. LAS1 ..)
actual-value ..
buffer (..)
LAS3
LAS4
(2)
(3)
(defun PO2 (..<as><mode>..)
...
(get-entry-from <as><mode>)
...)
(defun PO3 (..<as><mode>..)
...
(get-entry-from <as><mode>)
...)
LAS1
(4)
READ
log-access--structure: LAS2
function: read-entry
parameter: (..AS2 ..)
actual-value ..
buffer (..)
LAS2
(5)
READ
Input:
AS1
AS2
function call
referencing a LAS
Fig. 4.4: The communication protocol within a block

method either is not suitable or impossible due to several blocks reading this AS, the *read* functions have to copy each input tuple before passing it on to their succeeding plan operators. Hence, initializing the *read* functions of a block is a prerequisite for correct processing. Moreover, the cursors defined over the input AS are set to the beginning of these AS. A third initialization measure pertains to the LAS within a block and invalidates the entries *actual_value* and *buffer* for all LAS. Now the block can start processing.

By activating LAS_5, the block control issues calls to the plan-operator graph inside. LAS_5, in turn, looks up the name and the parameters of the plan operator it must invoke to answer the request coming from above. It has to call PO_1 ((1)) which has as input LAS_3 and LAS_4. With this information, PO_1 can call its left and right input LAS ((2), (3)) referring again to the only interface function *get-entry-from*. Thus, the activation is propagated down the plan-operator graph, until the *read* operators are reached ((4), (5)). They access their underlying physical AS according to the decisions taken in the initialization phase. If the *read* has received its input, control starts flowing back up to the root of the plan-operator graph. As soon as the next valid result has been produced, it is inserted into the output AS by calling the *write* operator.

5. Validating the Implementation

The overall plan-operator approach involves the concepts shown in Fig. 5.1: the plan-operator templates, the (physical) AS, the base predicates and the LAS. In the following, we will validate the goals put up in Sect. 3 by pointing out those (combinations of) concepts that guarantee their fulfilment.

5.1 Validation of Goals

To meet the first objective ("A Simple Processing Paradigm for Plan Operators"), we chose an open-next-close protocol that is reflected not only in the processing strategy of the plan-operator templates but that is also an important part of the functionality of the LAS. By introducing the LAS, we were able to completely isolate the plan-operator templates from the communication paths to be installed at run time. This information is supplied via parameters, just like the base predicates. In addition, the simple realization of the plan operators is guaranteed by resorting to the functionality of AS.

By fulfilling goal 1, defining new plan operators basically involves creating a template with the appropriate interface and specifying the desired functionality which to a large extent can be built from already existing functions and predicates. Hence, also the second conceptual goal, the extensibility at the plan-operator level, could be met. As already pointed out, this does not pertain to information on the relevant characteristics of the new plan operator for optimization, i.e., its applicability, benefits, and disadvantages.

As outlined in Sect. 2, the actual features of KOALA are independent from the basic operations of the knowledge model of KRISYS, which are represented by the plan operators. By introducing the base predicates as a concept orthogonal to all other concepts of our plan-operator approach, extending the query language only affects the base predicates themselves, facilitating this task as far as possible.

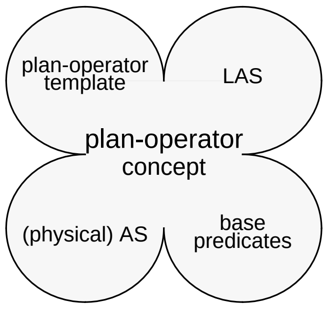


Fig. 5.1: The constituents of our plan-operator concept

Our implementation also meets the requirements of the fourth goal that calls for efficiency of dynamic query optimization. On one hand, a plan operator consists of functionalities independent of the query actually at hand, that therefore remain invariant (plan-operator templates and the required functionality of AS). On the other hand, there are those constituents that either depend on the context of a query (the base predicates) or that can only be known at run time (the LAS). By supplying the variable parts via parameters, the invariant parts can be precompiled without taking the risk of a later recompilation. Moreover, the base predicates can be supplied in a compiled form since they are assumed to remain unchanged in the course of a query evaluation. The only constituents that have to be generated dynamically at run time are the LAS. Since they are generally not very complex, it may be less costly not to compile them but rather to interpret them when executing the query. This optimization decision is, however, left completely optional by our approach. Substituting a plan operator for a new one only involves changing the respective piece of code and modifying the LAS calling this plan operator. Thus, replacing the BUFFER-INDEX-JOIN in the left block of Fig. 4.3 for a nested-loop realization just requires to insert the appropriate function name for the symbolic name PO_1 (cf. Fig. 4.4).

In Sect. 4.2.1 we introduced blocks as the units of execution communicating among each other via AS. Internal processing, however, is kept independent from this representational framework with the help of transformation functions (*read*, *write*) and is based upon separate data structures, the LAS. In addition, a facility is needed to control the execution of a block. Establishing a block therefore comprises positioning the *read*/*write* operators, generating LAS for the plan-operator graph contained and constructing the block control. Thus, the blocks of Fig. 4.3 can be integrated by

- discarding AS_3, the corresponding *read* and *write*, and LAS_5,
- changing the parameter *entry* for B.SEQ-SELECT in LAS_7 to the new input LAS_6, and
- creating a single block control (e.g., by choosing the one of the former block1).

The outcome of this process is shown in Fig. 5.2. Building a block is a comparatively easy task. This holds for the necessary overall modifications of the plan-operator graph, but also when considering the amount of code that might have to be (re)compiled due to the changes. Usually, only the LAS must be generated and may be subject to (run time) compilation. Hence the concept of blocks allows flexible units of execution within a plan-operator graph (cf. objective 5). Moreover, since no intermediate results of blocks must be permanently installed in WM, also the sixth goal (efficient realization of data flow between plan operators) is met.

5.2 A Technical Perspective

In this section, we will judge the implementation of our plan-operator concept from a technical point of view, i.e., concerning its efficiency and feasibility, both with respect to our programming environment [1] and in comparison to other programming platforms as for example C-like languages.

Although LISP is a language that can be interpreted, it is desirable to use precompiled code whenever possible. There are four “packages” of functions necessary to realize plan-operator graphs: functions realizing the plan operators themselves, base predicates, LAS, and the block control. The base predicates can be precompiled, as they do not have to be modified at run time. It must, however, be possible to supply the plan operators with variable base predicates at run time. Yet, due to our implementation, this does not imply that the plan operators cannot be precompiled. Since the base predicates are part of the parameters of a plan operator, they can be “inserted” into their program code in this way. Here, we exploited a characteristic feature of LISP. The LAS consist of the mere data structures and of functions defined on this representation. While the latter remain constant and hence can be precompiled, the data structures can only be established at run time, which basically means to define the correct number of structures (in other programming languages they are called records) with different names. At last, the block control must be built. It contains all the constructs mentioned before and for this reason also cannot be precompiled.

In summary, most of the code making up a block need not be modified any more at run time, only the data structures and the block control must be built. These parts, however, make up only a fraction of the overall code of a block, hence the run-time effort spent is in fact very small. If the block control is as simple as in Fig. 4.4, it might not even be worthwhile compiling the LISP code for the block, yet, if the control facilities become more complex, a compilation may be desirable. This freedom of decision is another big advantage of LISP.

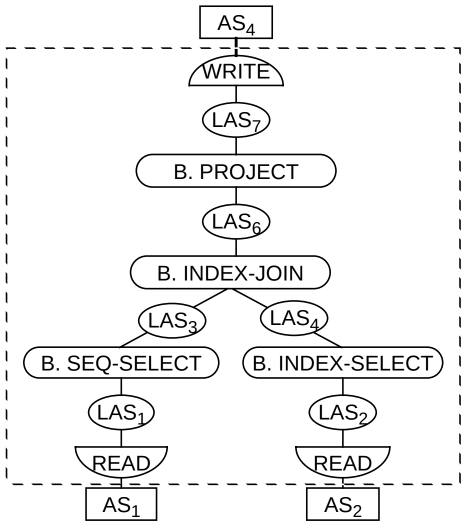


Fig. 5.2: A plan-operator graph in a single block

The design and realization of our plan-operator concept are not restricted to the environment in which KRISYS has been developed. Except for the possibility to interpret the variable parts of a block at run time, the same implementational mechanisms can be exploited using other programming platforms, as for example C.

6. Summary and Outlook

In this paper we presented a plan-operator approach for client-based knowledge processing which we discussed in the framework of the KBMS KRISYS. The design of this approach is motivated by a number of conceptual and operational requirements stemming from the processing characteristics of KRISYS. The fundamental objective is to employ a *simple processing paradigm* for plan operators to guarantee their easy realization and use. Based on this, we also aimed at *extensibility* both of the query language and at the plan-operator level, concerning internal evaluation strategies. Since knowledge processing in KRISYS is largely based upon decisions to be taken only at run time, a *minimization of run-time effort* (e.g., code generation and compilation) is indispensable. Moreover, only an *optimal exploitation of the buffer contents* ensures effective knowledge processing.

We partitioned these objectives into four orthogonal implementational concepts which we realized in a modular fashion. Firstly, the general execution logic of plan operators is supplied by *templates* that can be filled with the specific functionality of the operators. We implemented the communication between plan operators by *logical access-structures*. They support both the flow of control and data and render plan operators independent of their position in a plan-operator graph. The basic functionality of the plan operators is covered by the *access structures* making up the third implementational concept. Finally, the specific semantics of the query language is captured by a set of *base predicates*.

The objectives of our approach, although presented in the framework of the KBMS KRISYS, also apply to other systems intended for non-standard applications. The locality of reference typical for those applications requires facilities for client-based knowledge processing, most important of which an application buffer with a suitable representational framework. To exploit this locality of reference and due to concepts such as methods whose extension can often be known only at run time, KBMS as well as OODBMS need dynamic query optimization. Moreover, to meet specific needs of advanced applications, extensibility has to be supported at different levels. Given the architecture of KRISYS and the different representational frameworks in client and server, our approach to knowledge processing can be placed somewhere in between OODBMS pursuing mere server processing, leaving the maintenance of the client buffer to the application programs [10], and OODBMS performing main-memory query processing, having, however, the same data model in client and server [21]

Finally, it is important to point out that our approach not only fits the special characteristics of KRISYS and of the implementational environment (LISP), but that it is feasible in other query-processing scenarios as well.

The current state of the implementation is such that the transformation of a query into an algebraic representation and the subsequent rewrite are already realized. Moreover, most of the plan-operator level has been implemented, including the LAS. Although we can already provide a simple block control, we are currently working on a more flexible design for this concept. One of the primary goals in this development process is to make blocks powerful enough to cope with different processing environments ranging from single-processor workstations over multiprocessor architectures to distributed settings. This involves the development of a processing scheme for entire plan-operator graphs, especially for inter-block communication.

Acknowledgments

We would like to thank T. Härder and B. Mitschang for the helpful comments on the revision of this paper.

References

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

[2] Atkinson, M., et.al.: The Object-Oriented Database Manifesto, Proc. 1st Int. Conf. on Deductive and Object-Oriented Databases, 1989, Kyoto, Japan, 40-57.

[3] Brodie, M.L., Mylopoulos, J. (eds.): On Knowledge Base Management Systems (Integrating Artificial Intelligence and Database Technologies), Topics in Information Systems, Springer-Verlag, New York, 1986.

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

[5] Cluet, S., Delobel, C., Lecluse, C., Richard, P.: Reloop, an Algebra Based Query Language for an Object-Oriented Database System, Proc. 1st Int. Conf. on Deductive and Object-Oriented Databases, Kyoto, Japan, 1989, 294-313.

[6] Deßloch, S.: Handling Integrity in a KBMS Architecture for Workstation/Server Environments, Proc. GI-Fachtagung "Datenbanksysteme in Büro, Technik und Wissenschaft", Kaiserslautern, March 1991, ed. H.-J. Appelrath, Informatik-Fachberichte 270, Springer-Verlag, 89-108.

[7] DeWitt, D.J., 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.

[8] 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.C. (eds.): Objektbanken für Experten, Springer (Informatik Aktuell) 1992.

[9] 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.

[10] Deux,O. et al: The Story of O2: IEEE Transactions on Knowledge and Data Engineering, Vol. 2, No. 1, March 1990, 91-108.

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

[12] Graefe, G.: Volcano, an Extensible and Parallel Query Evaluation System, University of Colorado at Boulder, Technical Report No. 481, 1990.

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

[14] Härder, T., Hübel, C., Meyer-Wegener, K., Mitschang, B.: Processing and Transaction Concepts for Cooperation of Engineering Workstations and a Database Server, Data and Knowledge Engineering, Vol. 3, 1988, 87-107.

[15] Härder, T., Meyer-Wegener, K., Mitschang, B., Sikeler, A.: PRIMA - A DBMS Prototype Supporting Engineering Applications, Proc. 13th VLDB Conf., Brighton, UK, 1987, 433-442.

[16] 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.

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

[18] Küspert, K., Dadam, P., Günauer, J.: Cooperative Object Buffer Management in the Advanced Information Management Prototype, Proc. 13th VLDB Conf., Brighton, England, Sept. 1987, 483-492.

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

[20] Kim, W., Lochovsky, F.H. (eds.): Object-Oriented Concepts, Databases, and Applications, ACM Press, New York, 1989.

[21] Lamb, C., Landis, G., Orenstein, J., Weinreb, D.: The ObjectStore Database Management System, Special issue of Communications of the ACM, Vol. 34, No. 10, 1991, 50-63.

[22] 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.

[23] Mattos, N.M.: An Approach to DBS-based Knowledge Management, 1st Workshop on Information Systems and Artificial Intelligence, D. Karagiannis (ed.), Lecture Notes in Computer Science 474, Springer-Verlag,127-152.

[24] Mattos, N.M.: KRISYS - a KBMS Supporting Development and Processing of Knowledge-based Applications in Workstation/Server Environments, ZRI-Report 5/91, Universität Kaiserslautern, submitted for publication.

[25] Mattos, N.M., Meyer-Wegener, K., Mitschang, B.: Grand Tour of Concepts for Object-Orientation from a Database Point of View, to appear in: Data and Knowledge Engineering.

[26] Mattos, N.M., Michels, M.: Modeling with KRISYS: the Design Process of DB Applications Reviewed, Proc. the 8th Int. Conf. on Entity-Relationship Approach, Toronto - Canada, Oct. 1989, 159-173.

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

[28] Schek, H.-J., Scholl, M.H.: Evolution of Data Models, in Database Systems of the 90's, Lecture Notes in Computer Science 466, (Ed. A. Blaser), Springer Verlag, 1990, 135-153.

[29] Strobel, M.: Conception of a Component for Declarative Buffer Management in the KBMS KRISYS (in German), Undergraduation Final Work, University of Kaiserslautern, Kaiserslautern, June 1992.

[30] Schmidt, J.W., Thanos, C. (ed.): Foundations of Knowledge Base Management, Topics in Information Systems, Springer-Verlag, 1989.

[31] Shaw, G.M., Zdonik, S.B.: A Query Algebra for Object-Oriented Databases, Proc. Int. Conf. on Data Engineering, Los Angeles, 1990, 154-162.
[32] Thomas, J., Mitschang, B., Mattos, N., Deßloch, S.: Knowledge Processing in Workstation/Server Environments - The KRISYS Approach, ZRI Report 7/92, University of Kaiserslautern, 1992.
[33] Vandenberg, S.L., DeWitt, D.J.: Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance, Proc. ACM SIGMOD Int. Conf. on Management of Data, Denver, Colorado, 1991, 158-167.