# *Enhancing Knowledge Processing in Client/Server Environments*

J. Thomas, B. Mitschang, N. Mattos‡, S. Deßloch
Department of Computer Science, University of Kaiserslautern
P.O.Box 3049, 6750 Kaiserslautern, Germany
e-mail: {thomas | mitsch | dessloch}@informatik.uni-kl.de

## Abstract

A great variety of techniques has been developed to optimize and enhance query processing for relational, client/server, distributed, parallel, and heterogeneous database systems (DBS). Based on that work and experience, we investigate how far those techniques are applicable to query processing in Knowledge Base Management Systems (KBMS). Our reference system is the KRISYS KBMS that consists of a knowledge-processing system at the client (client-based processing) and a data-processing system at the server (database backend). We describe a unifying framework for query processing incorporating both processing systems (as realized in KRISYS). This allows to distribute and balance the amount of work done in the client and in the server. Based on an evaluation of that framework, several approaches to further enhance knowledge processing are reported.

## 1. Introduction

During the last several years, *Knowledge Base Management Systems* (KBMS) have emerged as an important research area in the field of databases. They do not only support a reliable and efficient management of large amounts of knowledge, but also offer modeling constructs to build such knowledge bases. Thus, KBMS naturally integrate aspects of knowledge representation and database technology.

KBMS are intended to support non-standard applications such as, e.g., intelligent CAD [MDL91]. These applications generally run on powerful clients equipped with sufficient processing capability, main memory, and private disk space, which are typically dedicated to single users or knowledge engineers and may provide special functions (e.g., suitable graphic interfaces) for them. The clients access a central server component, whose task is to maintain centralized information (i.e., the knowledge base, for short KB) and to control its shared use. Therefore, and also from a hardware point of view, KBMS architectures fit into a client/server environment with decentralized and autonomous processing sites. Among others, failure isolation, extensibility, and scalability of the entire system can be considered as key advantages of such architectures.

Client and server must be linked via an appropriate interface that minimizes communication traffic and KB accesses. Such an interface must take into account the functionality provided at the server site and at the client site in order to determine the amount of processing to be performed by either component. This gives rise to the question of where to place the 'borderline' between the components when considering the required processing capabilities. Introducing more semantics in the server by enhancing its functionality allows shifting a lot of processing from the client to the server (because the operations performed by the server become more powerful), however hinders the exploitation of available computing resources and processor 'power' in the client, increases communication overhead, and causes some kind of dependency of the client on the server. Less semantics on the server side means less functionality, tending to leave more processing at the client, thereby not fully exploiting the server's processing capabilities. Hence, it is necessary to balance the amount of processing done in the client and in the server. In general, such a balance is achieved by placing most of the semantics provided by the KBMS on the client side (close to the user/application, where it is needed) and most of the data management tasks on the central server. Consequently, in the client, a knowledge model and a query language comprising operations for defining and manipulating knowledge constitute the interface to users and applications. In the server, a DBS manages the KB. Hence, from a software point of view, KBMSs clearly separate between what we call *knowledge processing* in the client, and *data processing* in the server pre-processing portions of the KB to be loaded into the client.

In this architectural scenario, whenever queries are being processed, a precise coordination between client and server must take place to achieve the desired performance. In contrast to conventional query processing, where all accesses are performed under the absolute control of one query-processing system, we need here coordination of two autonomous systems taking part in the overall evaluation of queries. Further, both query-processing systems (i.e., the knowledge-processing system at the client and the data-processing system at the server) are faced with problems of efficiency and, for this reason, must apply optimization techniques to improve their performance. Hence, it is worthwhile to investigate the similarities as well as the dissimilarities of the two query-processing approaches in order to achieve a fruitful coordination. It is also important to analyze whether the well-founded (especially relational) technology for query optimization and query evaluation is applicable in this new architectural scenario.

At a general level this "divided" processing approach, as adopted by the KRISYS KBMS, is very similar to those used in current pure Object-Oriented Data Base Systems (OODBS), such as, e.g., ObjectStore [OHMS92]. However, a more detailed view reveals important differences: In many OODBSs much of the query and DBS processing is done on the client side, whereas the server mostly stores and retrieves pages of data in response to requests from the clients. In contrast to this, KRISYS asks for arbitrary sets of objects to be loaded into the client issuing queries to the server DBS. However, when comparing the work to be done in the client, similarities show up. Both, KBMS and OODBS manage an object cache and process queries over the cache contents. Due to these similarities, we are quite optimistic that the concepts and approaches discussed in this paper will be useful for OODBS as well. In addition to that, we assume that our work somehow inversely affects the areas from where we have taken our motivation, the processing concepts, as well as the optimization issues. Those areas are located in the realm of database query processing and were already mentioned above.

‡) IBM Database Technology Institute, Santa Teresa Laboratory, 555 Bailey Ave., San Jose, CA, 95150 USA, e-mail: mattos@stlvm14.vnet.ibm.com



CIKM '93 - 11/93/D.C., USA

© 1993 ACM 0-89791-626-3/93/0011 ....$1.50

Summing up, the observations lead to a set of requirements that must be met in order to guarantee effective and efficient overall query processing in KBMS. The evaluation of these requirements as well as the investigation of several solutions and their implementation in the KRISYS KBMS determine the goal and purpose of this paper. More specifically, Sect. 2 gives a concise overview of knowledge processing in the client/server environment set by the KRISYS KBMS. Sect. 3 outlines measures to considerably enhance knowledge processing, mostly applying well-known techniques for query processing in DBS. General as well as specific enhancements to knowledge processing are investigated as well as qualitatively evaluated. Moreover, we discuss how these enhancements can be realized in the framework of KRISYS. The final section summarizes the results, reports on the current state of system implementation, and concludes with an outlook to future work.

## 2. Knowledge Processing in the KRISYS KBMS

### 2.1 Knowledge Model and Query Language

The knowledge model of KRISYS is comparable to object-oriented data models [CACM91] [Ki91]. 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 [Ma88] [MM89]. The special semantics of these relationships is guaranteed by the system (e.g., inheritance along the classification and generalization relationships). In contrast to DBS, KRISYS does not distinguish between schema-information and instance-information - both are represented using the concept of objects (a similar approach is taken in [KL89]). 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, such as, e.g., integrity constraints and rules, not usually found in object-oriented models. In the scope of this paper, an in-depth discussion of these concepts is not necessary (see [Ma91] for details on rules and [De91] for details on integrity constraints).

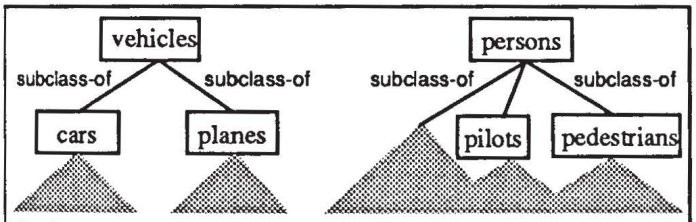


Fig. 2.1: The example KB showing class hierarchies and instances

An abstract view of our sample KB is given in Fig. 2.1. There, we assume a KB containing generalization hierarchies for persons and vehicles. Classes are drawn as rectangles, subclass/superclass relationships are the edges between superclass and subclass, and the shaded areas visualize the remaining parts of the hierarchies including all instances.

Retrieval and modification of a KB is supported by KOALA [DLM90] [Ma91], 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. For example, the ASK statement given in Fig. 2.2 and applied to the KB shown in Fig. 2.1 retrieves the names of all persons being either pedestrians, drivers of a car (referenced via slot 'driver'), or pilots of a plane.

```
(ask ((?y))
  (and (is-instance ?y persons)                                  ①
    (or (is-instance ?y pedestrians)                             ②
        (exist ?x (is-instance ?x cars)                          ③
              (is-in ?y (slotvalues driver ?x)))
        (exist ?z (is-instance ?z planes)                        ④
              (is-instance ?y pilots)
              (is-in ?z (slotvalues flies ?y)))))
```
Fig. 2.2: Sample ASK statement

Symbols with a leading question mark are query variables, similar to tuple variables in SQL. These variables may also appear in the projection clause (the first clause in the ASK statement). In our example, the projection clause states that just the object names are to be included in the result of the query. The query refers to the abstraction concepts of classification and instantiation and to references between the *vehicles* hierarchy and the *persons* hierarchy, which are due to the fact that a car has a driver (being an instance within the *persons hierarchy*) and a plane is flown by a pilot (being an instance of the pilot subclass in the persons hierarchy). With a (relational) evaluation concept in mind, the query reads as follows: Firstly, the *instances of persons* are retrieved and bound to the query variable ?y (operation 1), i.e., not only the direct instances but also those belonging to the whole class hierarchy beneath *persons*. Some of these instances may also be *instances of pedestrians* (operation 2). If this condition does not hold, the person may be among the drivers of an *instance of cars* (operation sequence 3) indicated by variable ?x, or a person qualifies, if he/she is a pilot and flies an *instance of planes* (operation sequence 4) referred to by query variable ?z. We will use this query example throughout the paper.

### 2.2 Knowledge Processing in KRISYS - the Main Ideas

KRISYS is a prototypical implementation of a KBMS developed at the University of Kaiserslautern [Ma91]. 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 shared information. For the purpose of this paper, it is sufficient to take a general view of the KRISYS architecture as shown in Fig. 2.3. (see [DHLM92] for more details).

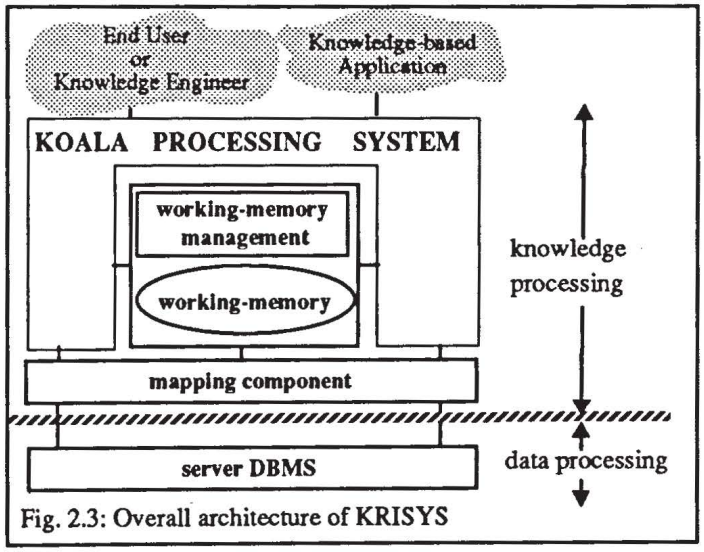


Fig. 2.3: Overall architecture of KRISYS

The overall architecture is motivated by two primary goals necessary to achieve efficient knowledge processing in client/server environments. Firstly, *application-oriented processing at the client site* must be supported to exploit the processing capabilities of the client components and to keep the server component from being overloaded. The server component may concentrate on the effective and efficient management of data [HR83]. Secondly, for efficiency and reliability, a *loose coupling* of client and server components that reduces communication efforts and dependencies between both sides must be achieved [HHMM88].

The first measure taken to fulfill these requirements was the introduction of a system-controlled application buffer (called *working-memory*, WM) at the client site, handling the applications' locality of reference. The WM is used by the KOALA Processing System (KPS) as resource to obtain its input data and as storage medium to which to write its query results; in other words, as a medium to maintain intermediate results. Hence, currently needed parts of the KB have to be transported into the WM only once, thereby minimizing the communication between client and server. Thus, if the application only refers to information already residing in WM, no calls to the DBS 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-specific processing tasks using e.g., methods or rules. Since these kinds of operations work on the knowledge model, it is clearly advantageous to perform them at the client site. Otherwise, the client would be idle, whilst the server might be overloaded. This holds also for the processing of KOALA (ASK or TELL) statements and for significant parts of the knowledge model (the semantics of the abstraction concepts, such as, e.g., inheritance) as well. Thus, only a portion of the processing related to the evaluation of a query, is performed by the server DBS[1], whilst the remaining, more complex tasks are carried out on the client side.

Knowledge processing in KRISYS, thus, involves both the server DBS and the system components of the client. If information referenced by a query must be fetched from the server, the *mapping component* (MC) is invoked. Its main purpose is to conceal details of how knowledge is actually mapped to the primitives of the underlying DBS[2], thus making the upper system components independent from the actual mapping. Queries posed to the MC are expressed in a functional subset of the KOALA language. This subset corresponds to the kind of simple queries that can be evaluated by the server component. Since the knowledge model of KRISYS cannot be mapped directly to the data model of the server component, the MC usually generates (a set of) queries formulated in the query language of the server DBS. Results coming in from the server consequently have to be transformed into a main-memory representation of the knowledge model. Hence, knowledge processing in the client can be realized based on this data structure. Thus, only the MC refers directly to the actual representation in the DBS.

## 2.3 Working-Memory Management and Representation

The above described architectural decisions indicate that the processing of associative queries (inherent in KOALA) indispensably requires the exploitation of the buffer contents. For this reason, KRISYS must be able to relate the knowledge requested 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 the specification of collections of objects (e.g., the WM contains 'all planes'). Thus, subsumption tests are performed by comparing the predicates of the query with the ones describing the buffer. These tests must be performed at run-time, and their results are then used for deciding which parts of a query are carried out in WM and which ones have to be delegated to the server. The component of KRISYS responsible for maintaining the buffer description and performing the tests is called *working-memory manager*. A discussion of 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 (cf. Fig. 2.4). Firstly, there are the relationships within an object, consisting of links from the object name to its attributes and their descriptions. Secondly, there are the relationships among objects. The most important of these relationships are the abstraction concepts, forming abstraction hierarchies frequently traversed during knowledge processing. Both types of relationships are materialized in WM using main-memory pointers[3], 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 so-called *access structures* (AS), taking the role of main-memory indices. 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 if advantageous.

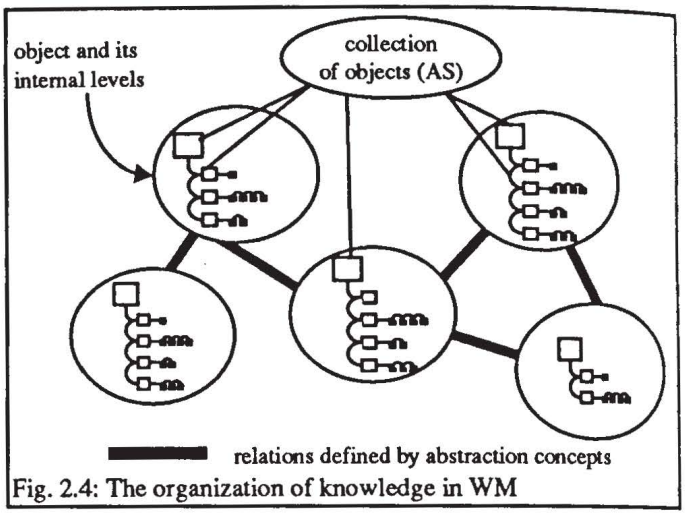


Fig. 2.4: The organization of knowledge in WM

## 2.4 Steps in Knowledge Processing

KPS realizes an algebraic processing model allowing conventional algebraic optimizations to be used to a large extent [JK84]. Thus, the overall steps of knowledge processing proceed in a similar fashion as the well-known steps of data processing in relational DBSs [HFLP89]: first, an algebra graph that represents a flexible internal representation of a query is generated and subsequently optimized; then, a plan operator graph is constructed; finally, executable code is generated, and the query is actually evaluated (Fig. 2.5). However, when analyzed in detail, several differences arise. They are due to the specific hardware environment and the different semantic levels of the knowledge model in the client and the data model at the server. In the following, we will discuss the steps of knowledge processing in detail. The starting point is the comprehensive sample query given in Fig. 2.2 and applied to the KB shown in Fig. 2.1 (cf. Sect. 2.1).

1. We exploit a relational or extended relational model at the server DBS [Mi88].
2. E.g., which relations are actually employed to represent an object class, which indices are defined over those relations, etc.
3. This is comparable to the 'pointer-swizzling' concepts applied in OODBS [Mo92].

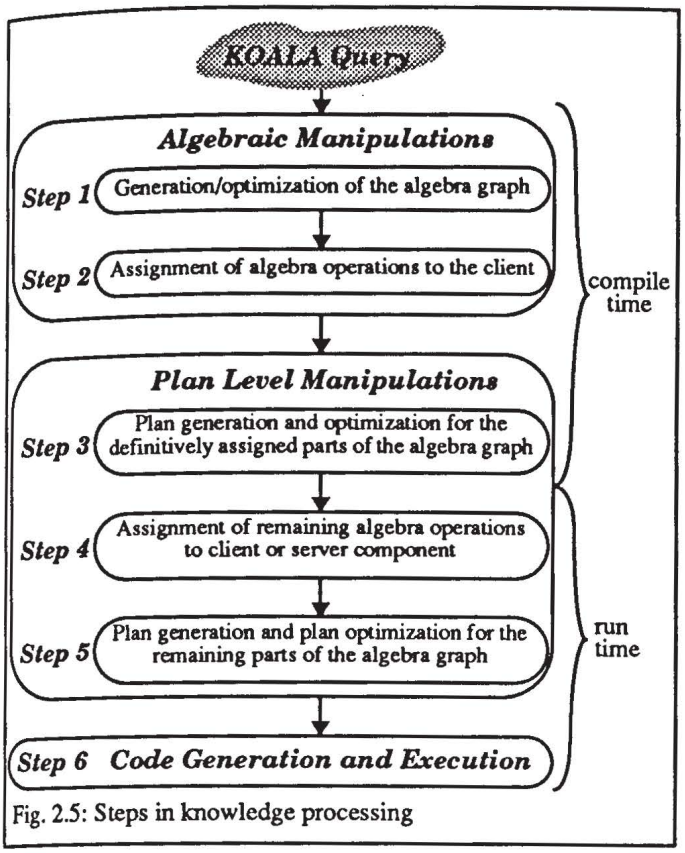



Fig. 2.5: Steps in knowledge processing

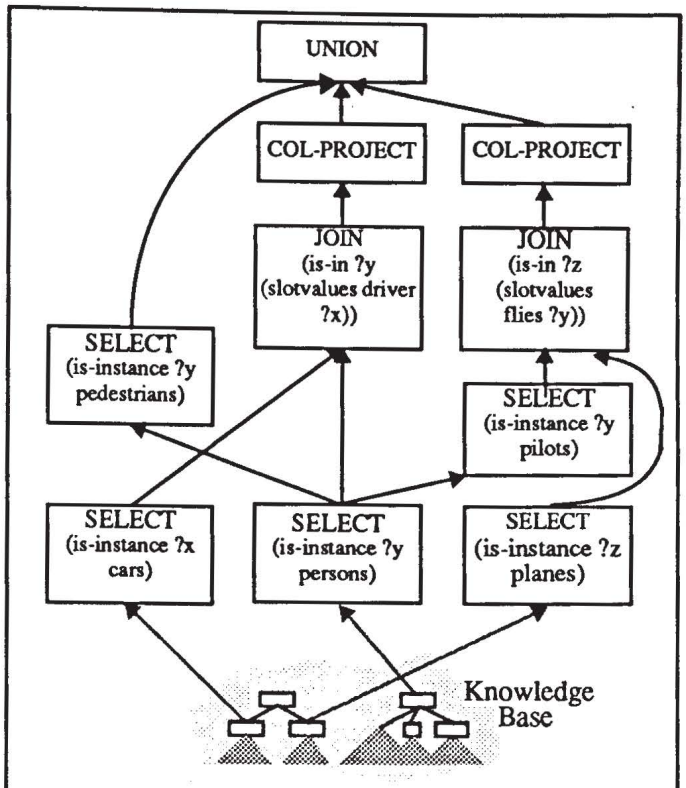



Fig. 2.6: Optimized algebra graph

### 2.4.1 Compile-Time Activities

**Generation of Algebra Graph and Algebraic Optimization (Step 1)**

In the first step of knowledge processing, an initial algebra graph is constructed from the incoming query. This algebra graph is subsequently rewritten by means of graph transformations performing algebraic optimization measures. As the result of the first step, an optimized algebra graph is generated as shown in Fig. 2.6 referring to our sample query mentioned before. When looking at this graph, we can easily recognize well-known algebra operators like, e.g., selection, projection, join, or union. Further, the set of legal algebra operators comprises those known from relational algebra (e.g., push-down of selections and projections, combination of sequences of unary operations, treatment of common subexpressions, etc.). In addition, there are some specific operators related to the special semantics and characteristics of the knowledge model [Ro92], e.g., an operator to follow object references and materialize the corresponding objects.

**Assignment of Algebra Operators to the Client Component (Step 2)**

Due to the client/server environment in which knowledge processing is performed, a crucial issue is to determine the evaluation site of each algebra operator. The more operations of an algebra graph that can be performed in the server component, the more reduced is the amount of data to be transferred into WM. This reduction of data volume also results in less objects to be installed in WM allowing a better exploitation of its storage capacity. Deciding on the evaluation site of each operator is based upon two criteria, namely,

- the complexity of the operations[4] compared to the query capabilities provided by the server DBS, and
- the current contents of the WM.

The first criterion can be evaluated at compile time, and we will discuss the underlying considerations in the following. The contents of the WM, however, is known only at run time and, consequently, must be dealt with in a later step of our processing framework (step 4).

With respect to the first criterion, those algebra operations have to be assigned to the client (and are indicated by a 'C' symbol) that are either too complex to be evaluated by the server DBS or that cannot be transformed into queries to the server due to insufficient expressive power of the DBS's query facilities. All other operators are preliminarily assigned to the server and marked by a '?' symbol. Let us assume that this procedure results in an operator assignment as shown in Fig. 2.7, left side (for reasons of simplicity, the operations are numbered, but directly correspond to their counterparts in Fig. 2.6).

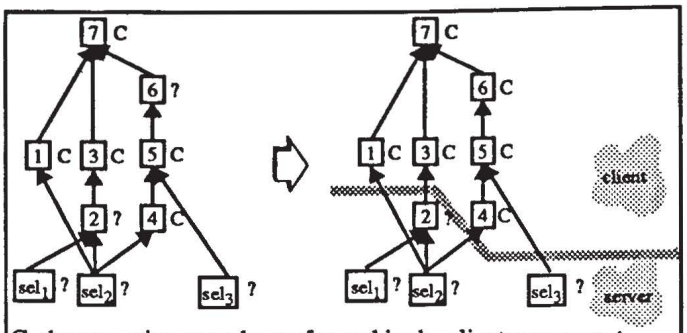
Now, a borderline between client and server can be drawn (Fig. 2.7, right side). Those operations whose evaluation site could not yet be decided (because they are potentially evaluable at the server), but which are preceded and followed by operations to be performed in the client, are reasonably re-assigned to this component. This applies, e.g., for operation 6. The operations still marked by a '?' can be assigned to client or server only at run time.

**Plan Generation and Optimization (Step 3)**

All the operations of the operator graph above the borderline expect their input to be in WM. Therefore, plan operators can be chosen for those operations, and the resulting (partial) plan operator graph can be optimized. Here again, the approaches to conventional plan optimization [Lo88] are applicable. However, the cost models used in plan optimization have to be adjusted to the main-memory query-processing environment[5]. The optimization measures remain valid

4. Note, this complexity is also based upon the way the KB is currently mapped (by the MC) to the DBS in the server. For simplicity reasons, we shall not consider this as a separate aspect.

5. This means that the concepts known from main-memory DBS are becoming applicable [De84]. These concepts are also being applied for query processing in OODBS [OHMS92].




C: the operation must be performed in the client component
?: the evaluation site may be either client or server

Fig. 2.7: Assignment to client or server

at run time despite the borderline possibly being moved "downward"; i.e., more operations being performed in the client. This observation will be justified subsequently. However, we can already note that the borderline will never be moved "upwards", because the assignments of operations to the client site is static.

### 2.4.2 Run-Time Activities

**Assignment of Remaining Algebra Operators (Step 4)**

At run time, the remaining algebra operators must be assigned to an evaluation site. Due to their reduced complexity, they may be performed in the client or in the server. The decision is solely based upon the current contents of the WM; i.e., the contents' descriptions maintained by the working-memory manager. When trying to decide on the evaluation site of a certain operator, one has to take into account the whole subgraph of the algebra graph rooted at that operator. Three situations may occur depending on the amount of requested knowledge already residing in WM.

If the WM does not contain any of the required input for a subgraph, the whole subgraph has to be delegated to the server DBS. Consequently, the border between client and server remains where it has been put at compile time. The plan operators above the borderline, which receive their input from such subgraphs, are not affected, because the results of the delegated operations are to be made available for further processing in WM by the MC. Additionally, the corresponding optimization measures already done at compile time are valid further on.

If the complete input for a subgraph already resides in WM due to previous queries evaluated in the client, the borderline is moved downwards because now this subgraph can be evaluated in the client. However, the optimization decisions for all direct successors, already set at compile time, are still valid, since they assumed their input to be in WM, and this will not change by moving down the client/server borderline. The only reason why the resulting graph may not look optimal is because it may contain sequences of plan operators that could be combined into a single operator to prevent intermediate results (e.g., sequences of selects). This can actually be handled subsequently or an appropriate execution control is employed at run time that avoids intermediate results for those operator sequences (cf. Sect. 3).

The third possibility arises if only part of the required input is residing in WM, and the rest is still residing in the server. In this case, basically two processing strategies are possible. One alternative is to completely delegate the query to the server, requiring to previously write back to the database the potentially updated portion of knowledge installed in WM. The second viable solution is to only complement the WM contents such that the query can be performed in WM. Using a cost model will help in deciding between the two alternatives. This discussion is beyond the scope of our paper. However, it is important to note that in either case, the implications for the already optimized plan operator graph above the borderline are a combination of those from the previous two cases. Thus, the optimality of this graph is guaranteed here as well.

**Plan Generation and Optimization (Step 5)**

Choosing an optimal plan for an algebra operator depends on its execution site. The plan operators to be evaluated at the server site are mapped to server queries (or query) by the MC and succinctly optimized by the DBS query optimization as normal database queries. (This aspect will be brought up later in Sect. 3.5). In case the information to be processed is already completely installed in WM, choosing a good plan operator mainly depends on the organization and size of the AS in which the knowledge is stored: whether the information is available in some sort order that might be exploited, or whether it might be worthwhile reorganizing the knowledge first, before actually processing it.

**Code Generation and Execution (Step 6)**

This step is in analogy to conventional data processing. If a compilation approach is taken, executable code has to be generated before the query can actually be evaluated, whereas otherwise, the query is executed by interpreting the plan operator graph.

During execution, the tasks of each plan operator are carried out by either sending queries to the server DBS or by accessing the knowledge residing in WM. To this end, each plan operator exploits the WM data structures for efficient access to its input data and organizes its computed results in terms of the same representational framework. Again, the concepts and techniques applied here can also be found in similar forms in OODBMS [CACM91] [OHMS92], as well as in main-memory DBMS [De84].

### 2.5 Summary

In this section we have presented the architecture of KRISYS and an overview of the knowledge-processing techniques applied. The basic concepts of this approach are supported in the prototypical implementation of the KPS in the KRISYS KBMS. In that system prototype we have concentrated on main-memory query processing in WM pre-loaded with a portion of the KB. Queries are transformed into an optimized algebraic representation, which is then used to construct a straight-forward plan operator graph with all the plan operators now being evaluated on the client side. Consequently, the next implementation step shall take server processing into account, as well. Conceptual and implementational issues to reach that goal are discussed in the next section.

## 3. Enhancements to Knowledge Processing

From the previous section we can conclude that knowledge processing in KRISYS proceeds along the same steps as query processing in conventional DBS. Due to the given architectural environment, a central issue for efficient knowledge processing is to decide on the execution site of each operator. In addition, other measures for improving performance can be taken. They are best classified into one of the following areas:

- approaches to enhance client/server communication,
- approaches to enhance client processing, and
- approaches to enhance server processing.

To discuss the optimization potential offered by these approaches, we apply a simple, yet sufficiently expressive, analytical model. We assume that all operations in the client component require the same amount of processing time, and that selections on the server take three times longer than operations to be performed in the client. In reality, this ratio is even higher, because we compare main-memory operations to database operations that might run into disk I/O, and because the communication overhead between client and server is also not taken into account. Moreover, we consider CPU costs as the only cost parameter for client processing, since it is carried out in WM only. This assumption even holds in the case

of a shared-memory multi-processor client, in which knowledge processing is distributed among the processors and all client processing takes place in the WM residing in shared-memory.

Before actually looking at the optimization potential offered by these approaches and their reflection in KRISYS, we will introduce the plan operator concept underlying the processing of KOALA queries. This detailed view to query processing is necessary in order to understand the subsequent discussions.

### 3.1 Generic Plan Operator Concept and Flexible Processing Model

Due to the client/server environment of KRISYS, its plan operators must take into account client processing, server processing, as well as interactions between these two processing systems. To accomplish that, the plan operators provide for a high level abstraction that allows the unification of several processing issues applicable in that heterogeneous processing environment.

At a logical abstraction level the plan operators are seen as producers and/or consumers of *tuple streams*[6] that define a high level connection of producers and consumers abstracting from both the structure of the tuples and from the way how the tuples get from the producer to their consumers. (Later in the discussion we shall elaborate on the abstraction and flexibility achieved by this). Each plan operator has at least one input stream and exactly one output stream that might be fed into several subsequent operators. All plan operators are realized as iterators; i.e., they are controlled by an *open-next-close* interface. The *open* function initializes a plan operator's processing, the *next* function asks for the next result tuple to be produced by the plan operator, and the *close* function terminates plan-operator processing. This open-next-close protocol is applicable to all plan operators meaning that an operator has firstly to *open* its directly subordinate operators before being able to ask them for the production of input tuples (*next* calls) that are needed for the operator's own processing and for the production of its output tuples. When processing of an operator is finished, its subordinate operators will be terminated (*close* call), too. Hence, the generic processing model for an entire plan operator graph consists of three phases. During initialization, an *open* call is sent to the root plan operator initiating it and causing *open* calls to all its input streams down the plan operator hierarchy. In the second phase, a *next* call is repeatedly issued to the topmost plan operator, which passes on the *next* call to its subordinate operators until the end of input is reached. As soon as this situation occurs, the root plan operator closes its input streams propagating the *close* signal to its predecessors and then terminates itself. Thus, all iterators in a plan operator graph are recursively shut down.

This generic interface defines an evaluation model that is purely demand-driven and that allows for different realizations, thus adapting evaluation to the current environment, i.e., to client or server processing. Further, it separates evaluation control from the specific tasks of a plan operator (e.g., selection, projection, join, union, etc.). This offers easy extensibility to new plan operators because a new plan operator only has to obey the open-next-close protocol to be utilizable in that framework [TD93]. This simple but flexible plan operator concept has also been adopted by other query-processing systems, e.g., Starburst [HFLP89] and Volcano [Gr90b], and has proven its applicability.

### 3.2 Enhancing the Communication between Client and Server Processing Systems

For the discussion of these and the following enhancements we assume that an optimized plan operator graph has already been constructed from the algebra graph given in Fig. 2.6. All its operators have been assigned to either client or server, and the selections are the only operations to be delegated to the server DBS (cf. Fig. 3.1).

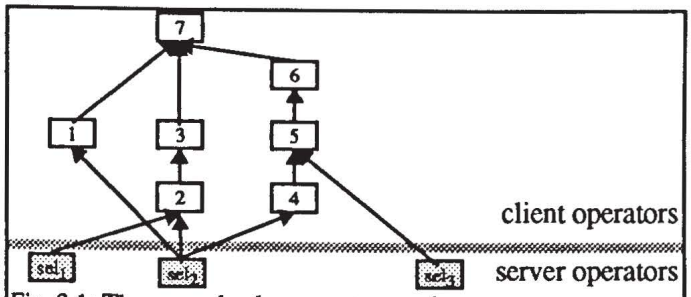
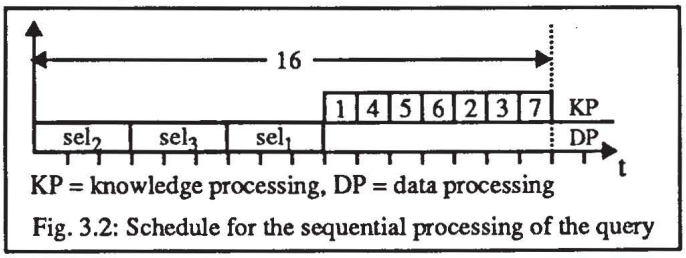


Fig. 3.1: The example plan operator graph.

#### 3.2.1 Asynchronous Interface to the Server DBS

The simplest strategy for processing the plan operator graph of Fig. 3.1 is to execute all operators in a sequential order, e.g., by evaluating selection $sel_2$ first, then performing selection $sel_3$, and finally carrying out selection $sel_1$. Thereafter, operations 1 to 7 can be executed. Other sequential strategies are only permutations of this operator sequence and thus can be treated analogously.

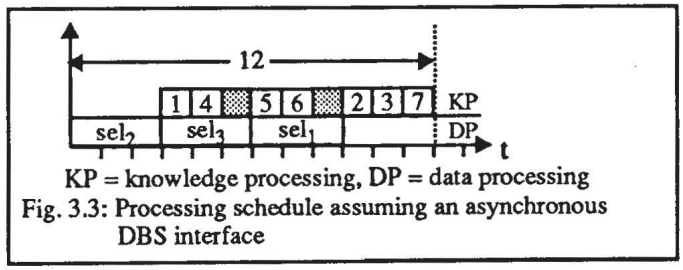
A schedule for this sequential evaluation is outlined in Fig. 3.2. This diagram does not allow any quantitative judgments on the duration of the evaluation process, yet qualitative conclusions may be drawn. Although being logically correct, this evaluation order sequentializes even those operations that are independent from one another, e.g., selection $sel_1$ is independent from any schedule for the operator sequence (4,5,6). In this case we can't do better, because it is assumed that the interface to the server DBS is synchronous, i.e., blocking. Sending a query to the server **blocks** the callee until the **complete** result is available. However, if the interface to the server DBS is asynchronous, a better processing scheme may be achieved.



KP = knowledge processing, DP = data processing

Fig. 3.2: Schedule for the sequential processing of the query

While processing a subtree (whose input is already residing in WM), the evaluation of another subtree can already be requested from the server. It does not suffice just to send a query to the server DBS, it must also be guaranteed that all results, arriving asynchronously in the client, are stored in WM for later processing. Under these circumstances and referring to the processing sequence mentioned above, the schedule given in Fig. 3.3 becomes possible.



KP = knowledge processing, DP = data processing

Fig. 3.3: Processing schedule assuming an asynchronous DBS interface

Although diagrams 3.2 and 3.3 do not allow any quantitative conclusions, it becomes clear that by using an asynchronous interface for the server DBS considerable improvements show up,

6. Here we use 'tuple' as a generic term. It refers, in general, to a processing element, which in our case can be a complete (knowledge) object or even specific parts thereof.

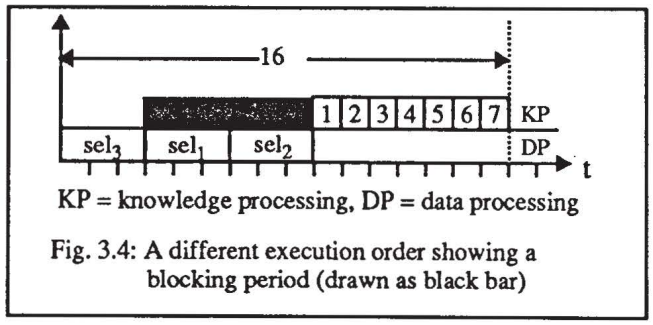
since now client knowledge-processing and server query-processing do work concurrently.

### 3.2.2 Multi-Query Interface to the Server DBS

From the client's point of view it does not matter whether the queries are sent to the server one after the other or whether they are initiated together. This may, however, be decisive for query processing in the server DBS if it is able to optimize such *multi-queries*. In this case, the evaluation of the single queries can be combined resulting in a reduction of execution overhead. If, e.g., the server DBS is supplied with several read-only queries, they need not be executed each within a separate transaction. It is sufficient to carry them out as independent queries within a single transaction, thus saving processing overhead, e.g., for locking. A further major gain of performance may be achieved by avoiding redundant accesses and operations on the same data by identifying common subexpressions referenced by several (sub)queries [Se88].

If the server DBS cannot process the set of queries or parts thereof in parallel, it has to evaluate them sequentially, thus also defining the order in which the answers to the queries are given back to the client. A particular order set by the server DBS might influence knowledge processing in the client such that an optimal order cannot be achieved any more. If, e.g., in our sample scenario the portions delegated to the server are answered and returned in the order $sel_3$, $sel_1$, $sel_2$, total execution takes longer, i.e., more time is needed for processing the entire query. Execution time even corresponds to sequential processing (cf. Fig. 3.4). Although we assume an asynchronous interface between client and server, client processing is blocked until (sub)query $sel_2$ has been processed completely. This blocking period is drawn as a black bar in Fig. 3.4.

For the client it is therefore desirable that its optimal execution order is respected by DBS processing. To avoid that the optimization efforts in client and server contradict each other concerning execution sequences, the DBS should be able to adapt its optimal evaluation to the needs of the client. To do that, it not only needs a set of queries subject to multi-query optimization, but also the optimal evaluation sequence is required. However, these considerations may not be valid any longer if the server DBS is a parallel database system (cf. Sect. 3.5).

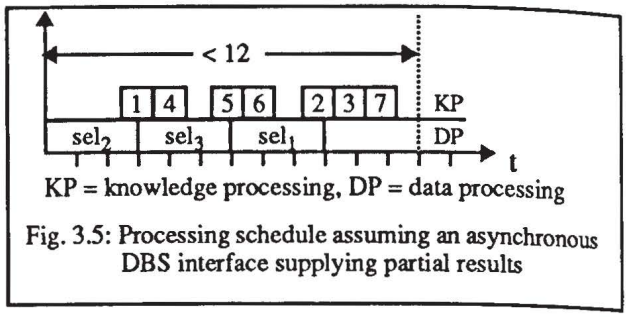



KP = knowledge processing, DP = data processing

Fig. 3.4: A different execution order showing a blocking period (drawn as black bar)

### 3.2.3 Supply of Partial Results of a Query by the Server DBS

In the evaluation scenario given by Fig. 3.3, the client has to wait at two points (shaded areas) for the results of a database query, although the interface to the server is non-blocking. After operation 4 has been completed, the client cannot continue until the results of $sel_2$ are completely available. The same applies for selection $sel_1$, the missing input for operator 2. This effect is due to the server DBS returning only complete answers to the client. However, if the server DBS is able to provide partial results, the client can already start to process them without having to wait for the server to finish. Thus, the duration of overall processing can be reduced considerably as depicted in Fig. 3.5. Note, operation 5 can start with the first results from $sel_3$ right after operation 4 is completed. Analogously, operation 2 can start when the first results generated from $sel_1$ show up supposing that $sel_2$ is already done.




KP = knowledge processing, DP = data processing

Fig. 3.5: Processing schedule assuming an asynchronous DBS interface supplying partial results

### 3.2.4 Combining the Concepts

In the previous sections we identified and evaluated three important measures enhancing the communication between client and server processing systems. We introduced them as separate concepts, being independent from each other. However, their combined usage is advisable to achieve best processing performance. Thus, an asynchronous interface between client and server processing systems that supports multi-queries and supplies partial results will yield the best results.

For query processing in KRISYS we employ exactly this combined approach (cf. Sect. 3.5). Its realization is based on a new plan operator (to be described in detail in Sect. 3.4) and thus integrated into the overall processing concept of the *open-next-close* protocol described in Sect. 3.1.

## 3.3 Enhancing Client Processing

With the appearance of multi-processor clients [Co89] [Se90], query processing is entering a new dimension. Parallelism in connection with asynchronicity is becoming effective. The application of these techniques to client query processing systems is discussed in this section.

Since up to now the client is seen as a single-processor system, query processing in the client has to be performed sequentially. For effectively exploiting parallelism in the client, a shared-memory multiprocessor architecture must be available. The degree of parallelism may vary and, with this, also the respective benefits, both being determined by the dependencies given in a plan operator graph. Thus, the central issue is to detect reasonable 'units of parallelism' within such a graph. Referring to the plan operator graph of Fig. 3.1, there are subgraphs that are logically independent from one another (due to their disjointness) and thus can be processed in parallel, such as, e.g., those defined by the operators (2, 3) and (4, 5, 6). Consequently, the first criterion applied for the construction of units of parallelism is the logical independence of subgraphs in the plan operator graph.

The second important criterion for deciding on units of parallelism takes into account the processing characteristics of the plan operators. There are two types of operators to distinguish. The first class is called *tuple-oriented*, since the operator processes (the input tuples) and decides (about an output tuple) on a per tuple bases. Obviously, the selection and projection operators fall into that category. Contrary to that are the so-called *set-oriented* operators. They process (the input tuples) and decide (about an output tuple) on a set-of-tuple basis. Sometimes the whole input has to be seen, before an output tuple can be generated, as is the case, for example, with the sort or duplicate-removal operators. This is, in general, true for all so-called non-monotonic operators. A similar situation holds for the join operation. Depending on the join strategy to be applied and on the type of join (e.g. 1:1, 1:n, or n:m), it can be decided based on the tuple or set-orientedness of the operator at hand. For example, a 1:1 join employing a sort-merge

strategy can be processed in a tuple-oriented manner, whereas a nested-loop join always votes for a set-oriented processing (w.r.t. the inner loop[7]). Sequences of tuple-oriented operators which are in a producer-consumer relationship, therefore being dependent on each other (e.g., operators 2 and 3), can process the tuples without the need to store or materialize intermediate results. However, this pipelining mode is hindered by set-oriented operators (as may be the case, e.g., for operator 5) that have to wait for tuples to be available at their input streams before being able to start or continue processing. Therefore, it is necessary to buffer the tuples of the producer(s) to make both consumers and producers independent from one another's processing speed, in a way that the producers can keep on generating results even if their successors cannot immediately consume them. We call this intermediate 'storage' *input/output queue*. This is our second criterion for finding groups of operators. Operators within such a group do not need input/output queues and, for that reason, apply the pipelining processing mode. Consequently, operators with input/output queues in between are assigned to different operator groups that can be processed independently, thus being subject to parallel execution at the group level.

Applying both criteria introduced to the plan operator graph of Fig. 3.1 results in the following operator groups (units of parallelism) and their associated input/output queues all shown in Fig. 3.6.

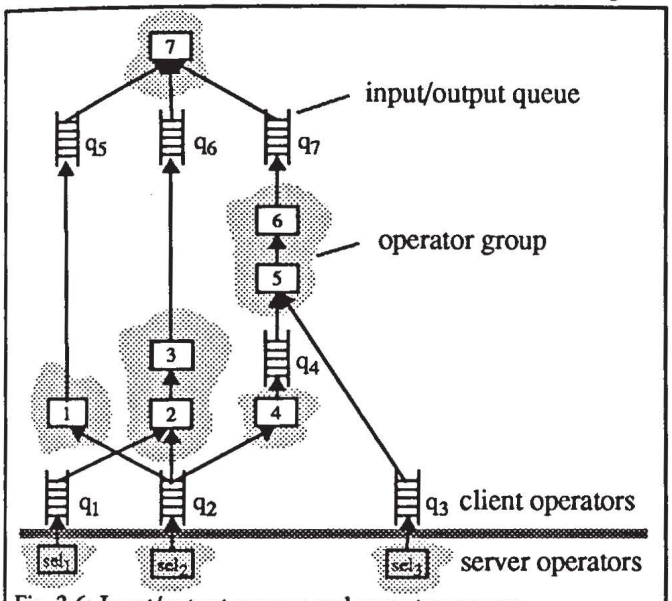


Fig. 3.6: Input/output queues and operator groups

Operation 4, e.g., has become a unit of parallelism because its successor - operation 5 - has to wait for yet another input stream. Due to the asynchronicity between client and server, the results arriving from the server always have to be made available via input/output queues. On the other side, plan operators 5 and 6 as well as the operators 2 and 3 form operator groups that can apply a pipelining mode internally.

In the query-processing model for a multi-processor client, these groups are assigned to different processes that can be distributed among the available processors. Thus, processing of our example graph results in the schedule depicted in Fig. 3.7. There, we assume that data from the server is already available in the client. Operator groups (1), (2,3), and (4) can be started and processed in parallel, whereas operator group (5,6) starts later with the first result of operator group (4) being available at its input stream (i.e., in input queue $q_4$). The same happens to operator group (7), which starts with the first results of groups (5,6), (1), and (2,3) available at the respective input queues. As one can clearly see, total execution time for client processing is reduced considerably.

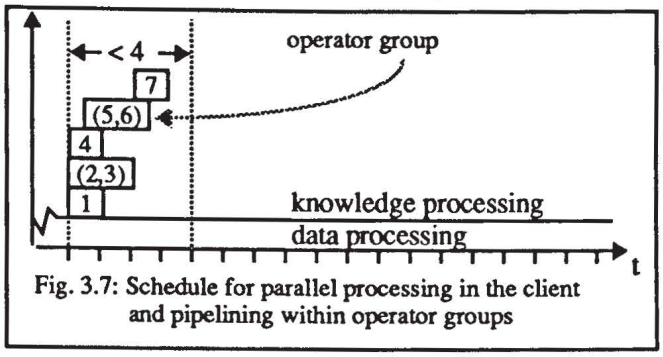


Fig. 3.7: Schedule for parallel processing in the client and pipelining within operator groups

### 3.4 The Plan Operator Transmit

From the previous observations we know that there are basically two different 'coupling modi' between the plan operators that have to be supported:

- The *pipelining mode* comes along without storing the intermediate results from producer operators.
- In the complementary *'buffering' mode*, there is a need for storing (intermediate) results in queues.

To support the abstraction of a tuple stream being the input as well as output medium for the plan operators (as introduced in Sect. 3.1), a general concept is needed that on one hand links producers and consumers supporting the two coupling modi, and that on the other hand is compatible with the iterator paradigm of the plan operators.

These requirements can be met most elegantly by introducing a new plan operator that is responsible for the proper connection of plan operators (a similar operator has been proposed in [Gr90a]). This plan operator, we call it *transmit*, firstly conceals the actual coupling modus and secondly adheres to the generic open-next-close protocol. With this, the existence of queues in the plan operator graph can be made transparent. To that end, the *transmit* operator simply manages the queue itself, by realizing the *next* functions issued by its consumer operator as a read operation to the internal queue and by having the queue filled by issuing *next* operations to its producer operator.

*Transmit* is also capable of masking process and even processor boundaries. Therefore, we can simply replace all input/output queues of an operator graph by instantiations of the *transmit* operator. Moreover, *transmit* provides control structures for the operators of operator groups that are to be executed in pipelining mode. This implementational aspect, although being important for performance, is beyond the scope of this paper[8]. The operator graph that results from introducing *transmit* in the graph of Fig. 3.6 is shown in Fig. 3.8. It now reflects the abstraction level we wanted to achieve: the whole operator graph consists only of operators obeying the open-next-close protocol, thereby abstracting from the actual processing environment like shared or 'distributed' memory, data-driven or control-driven evaluation, etc.

Fig. 3.8 also zooms into a *transmit* operator that decouples two operator groups (operator group (5,6) and (7)) assigned to separate processes. *Transmit* offers synchronous read and write operations at the consumer and producer side, respectively, and does inter-process communication as well as the efficient management of the queue holding the tuples to be passed on. As a result, both producers and consumers are freed from those issues and employ

7. The inner tuple stream must be completely available to process a tuple of the outer input stream.

8. For a detailed discussion of the realization of plan operators and their communication, see [TD93].

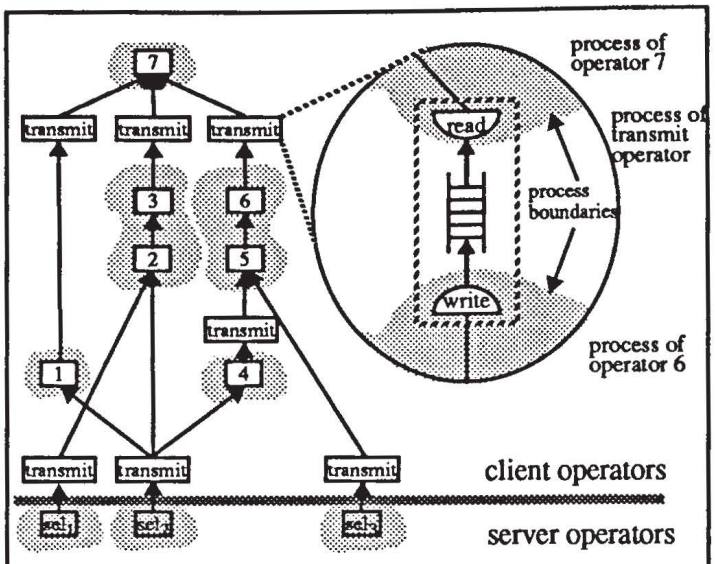


Fig. 3.8:The use of *transmit* in a plan operator graph

the same simple interface (*open-next-close* protocol) at their input and output tuple streams.

If producer and consumer have access to shared memory, then *transmit* employs efficient communication primitives (e.g. semaphores) available for this environment. For example, this might be the case for the *transmit* between operator groups (5, 6) and (7) in Fig. 3.7. However, if consumer and producer work on separate machines (as, e.g., the three *transmit* operators at the client/server boundary in Fig. 3.7), *transmit* has to implement the input/output queue differently (e.g., via TCP/IP datagram services). In any case, *transmit* will choose the best realization strategy depending on its execution environment.

The *transmit* operator directly following $sel_2$ accepts information arriving asynchronously from the server DBS and has to pass it on to the operations 1, 2, and 4. For this purpose it does not replicate the information but keeps a single data structure with multiple pointers, one for each consumer. It is important to mention that in the case of $sel_2$, as well as in the case of $sel_1$ and $sel_3$, the input for *transmit* - although originating in the database - comes in a form corresponding to the knowledge model of KRISYS, since the functionality of the MC is exerted by the plan operators that issue the queries to the server DBS.

In that general model, the *transmit* operator is free to 'drive' its input operators, thus being able to move smoothly from strict demand-driven control to pure data-driven control. For example, in the scenario from before, it can guarantee that its internal queue is always filled to a certain limit by simply calling *next* repeatedly on the input operator. Or, if the *transmit* operator perpetually calls *next* on the input operator, then it pursues a data-driven approach. In a data-driven scenario each single operator or operator group is activated according to the availability of input data in its input streams. Thus, execution control is determined by data flow, and there is no need any more for explicit and maybe centralized execution control. This, of course, considerably simplifies the overall processing and execution model.

Summing up, *transmit* encapsulates parallelism, process and processor boundaries, thus defining the basis for flexible query processing. Data parallelism, i.e., splitting of queues and multiple instantiation of the operator (to work on the split data parts) is possible but not yet considered. Currently, a first version of *transmit* is being implemented in the KRISYS KBMS.

### 3.5 Enhancing DBS Server Processing

The effectiveness of server processing is determined on one side by its interface to the client processing system and on the other side by server internal measures to query processing. In Sect. 3.2 we have already discussed the salient characteristics of an effective interface between client and server processing system, e.g., asynchronicity, supply of partial results or the ability to do multi-query optimization. In this section we concentrate on enhancements to the server processing system. Since this is a DBS, all known measures to enhance DB query processing apply here as well. Therefore, the measures for parallelism combined with a dataflow approach [PMCLS90] [Gr90b] [DG90] that have been successfully applied to client processing (as documented in the previous section) do also work in the context of server processing and will not be repeated here. Instead, we will show in this section how server processing **complements** the concepts given at its client interface.

From the server processing system's point of view it is important to observe the following characteristics and requirements determined by the client/server interface mentioned in Sect. 3.2:

(1) The interface should be non-blocking, i.e., asynchronous (cf. Sect. 3.2.1 and Sect. 3.4).

(2) A single request to the server DBS might consist of a set of queries together with some priority information (cf. Sect. 3.2.2).

(3) Partial results of a query should be delivered to the client as they are derived (cf. Sect. 3.2.3).

Requirements (1) and (3) guarantee that any (partial) results being derived by the server processing system can be made available instantaneously for further processing in the client without blocking both client and server processing. In Sect. 3.4 we have shown that there are conceivable realizations of the *transmit* operator that provide this kind of interface. Requirement (2) basically asks for multi-query processing in the server DBS. In the following, we will elaborate on that in more detail.

In [Se88] two types of algorithms are considered for realizing multi-query optimization (MQO). The algorithms of the first type consider exactly one (locally optimal) access plan per query. An algorithm of the second type builds a global access plan by choosing among local (not necessarily optimal) access plans for each query. Since merging locally optimal plans does not generally result in an optimal global access plan, the second type of algorithm is more desirable. The related optimization problem can be modeled by an A* algorithm, i.e., as a search space problem. Finding an optimal global access plan therefore depends on a good search function, i.e., a fast convergence of the algorithm.

Our multi-query processing framework looks a little bit different to the conventional one sketched in [Se88]. For example, assume the scenario given on the right side of Fig. 2.5 with all the operators that are under the client/server borderline (marked by '?') being delegated to the server DBS in one single request. In that scenario, the following knowledge queries have to be mapped to database queries and issued to the server processing system for evaluation:

Query Q1: to retrieve the knowledge objects specified through $sel_1$,

Query Q2: to retrieve the knowledge objects specified through $sel_2$,

Query Q3: to retrieve the knowledge objects specified through $sel_3$,

Query Q4: to further restrict the result of query Q2 joined with query Q3 to get the result for operator 2.

In addition to this set of separate queries a priority list is specified telling that $sel_1$ and $sel_2$ be best evaluated before operation 2 is done and $sel_3$ is independent from all others.

There are two things worthwhile mentioning that are in contrast to conventional MQO. Firstly, the queries in a set need not be independent from each other, i.e., one query might be defined on the result of another query in the same set as we can see from the example list given above: query Q4 is defined based on query Q2 and Q3. Secondly, precedence information is associated with the query set.

Conventional MQO does not support dependent queries. One possible solution is to duplicate the dependent parts of the query and to replicate their results at the client interface. This means for our example from above that query Q2 (i.e., operation $sel_2$) has to be duplicated twice, once for operation 1, a second time for operation 4, and finally as a part of query Q4 (i.e., operation 2). This is obviously not the best solution since it provokes redundant query processing as well as increased communication between server and client. An approach to evaluate (and optimize) a set of dependent queries and to return this multi-query in addition to eventually needed intermediate results has recently been proposed in the framework of a composite object extension to the Starburst extensible database system [MP91] [MPPLS93]. The application of these concepts to the query-processing framework adopted by KRISYS are currently under investigation. To the best of their knowledge, the authors are not aware of any other approach that is suitable for the KRISYS query-processing framework. Due to space restrictions, a more detailed description and discussion has to be postponed to another publication.

Further, if the client supplies an execution order together with the set of queries, this information can be used by the A* algorithm to reduce the search space and thus to speed up optimization. However, strictly respecting the execution order optimal for the client may prevent finding an optimal global access plan. The priority that is given to the control information supplied by the client therefore remains to be investigated. If parallel query evaluation is present, then the importance of this control information is less since the multiple requests might be dealt with concurrently.

## 4. Summary, Related Work, and Outlook

The scope of this paper is query processing for KBMS. Such systems are designed for client/server environments and involve a DBS on a central server and system components responsible for providing a knowledge model and its operations on a client. This architectural scenario causes the overall query processing to be split into data processing in the server and knowledge processing in the client. Therefore, an important step in knowledge processing for KBMS is to draw the borderline between the operations to be performed in the client and those to be delegated to the server. Starting from this decision, we showed how knowledge processing, data processing as well as their interaction can be enhanced by using techniques known from the field of DBS, as, e.g., operator parallelism, asynchronicity at the client/server interface or multi-query optimization in the server. Most of the techniques could be directly applied to query processing for KBMS, and the issues related to parallelism and processing in a heterogeneous hardware environment could be managed by a single plan operator (*transmit*). This newly defined operator adheres to the open-next-close protocol and, thus, perfectly fits into the knowledge-processing framework set by the existing operators of the KPS. *Transmit* defines an evaluation model that allows for different realizations, adapting evaluation to the current environment, i.e., to client or server processing. Further, it decouples all other plan operators from the (hardware) characteristics of the execution environment at hand.

The current state of implementation allows a sequential evaluation of KOALA queries in the client on a pre-loaded KB portion. Our experiences so far are restricted to the descriptive portion of the knowledge model/language. Since, in our case, rules are formulated via KOALA, we perceive the same concepts to be applicable. However, handling methods the right way is different. Still, method optimization is one of the hard problems (also in OODBSs). To that end, we developed a method description that declaratively comments the behavior of the method w.r.t. querying/processing issues. Based on that information, we want to investigate whether the given concepts apply, or have to be adapted. It is also important to get the full spectrum of knowledge processing tasks at work (i.e., the six steps given in Fig. 2.5). We primarily concentrate on the concepts that enhance client processing, as detailed in [TD93]. Especially, we have developed algorithms that perform automatic grouping of plan operators and algorithms that provide for optimal scheduling in a multi-processor client. At the moment we have done only preliminary work in the area of transaction support for our processing environment, which is distributed among client and server.

**References**


CACM91 Cattell, R. (ed.): Next Generation Database Systems, in: Special issue of Communications of the ACM, Vol. 34, No.10, 1991.

Co89 Concurrent Computer Corporation, Quick Reference Guide, 4/89.

Da90 Date, C.J.: An Introduction To Database Systems, Vol. I, Addison-Wesley, 1990.

De84 DeWitt, D., et al.: Implementation Techniques for Main Memory Database Systems, in: Proc. ACM SIGMOD Int. Conf. on Management of Data, Boston, June 1984, 1 - 8.

De91 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.

DG90 DeWitt, D.J., Gray, J.: Parallel Database Systems: The Future of Database Processing or a Passing Fad?, SIGMOD RECORD, Vol.19, No. 4, December 1990, 104-112.

DKS92 Du, W., Krishamurthy, A., Shan, M.-C.: Query Optimization in Heterogeneous DBMS, Proc. 18th Int. Conf. on Very Large Databases, Vancouver, 1992, 277-291.

DHLM92 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.

DLM90 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.

Gr90a Graefe, G.: Encapsulation of Parallelism in the Volcano Query Processing System, in: Proc. ACM SIGMOD Int. Conf. on Management of Data, Atlantic City, May 1990, 102-111.

Gr90b Graefe, G.: Volcano, an Extensible and Parallel Query Evaluation System, University of Colorado at Boulder, Technical Report No. 481, 1990.

HFLP89 Haas, L., Freytag, J.C., Lohman, G., Pirahesh. H.: Extensible Query Processing in Starburst, in: Proc. of the ACM SIGMOD Conf., Portland, 1989, 377 - 388.

HHMM88 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.

HR83 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.

JK84 Jarke, M., Koch, J.: Query Optimization in Database Systems, Computing Surveys, Vol. 16, No. 1, June 1984, 111-152.

Ki91 Kim, W.: Introduction to Object-Oriented Databases, Computer System Series, MIT Press, 1991.

KL89 Kim, W., Lochovsky, F.H. (eds.): Object-Oriented Concepts, Databases, and Applications, ACM Press, New York, 1989.

Kr89 The KBMS Prototype KRISYS - User Manual, Version 2.3, University of Kaiserslautern, 1989.

Lo88 Lohman, G. Grammar-like Functional Rules for Representing Query Optimization Alternatives, in: Proc. ACM SIGMOD Int. Conf. on Management of Data, Chicago, June 1988, 18 - 27.

Ma88 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.

Ma91 Mattos, N.: KRISYS - a KBMS Supporting Development and Processing of Knowledge-Based Applications in Workstation/Server-Environments, Internal Report, Dept. of Computer Science, University of Kaiserslautern, 1991.

MDL91 Mattos, N.M., Deßloch, S., Leick, F.-J.: A Knowledge-Based Approach to Intelligent CAD for Architectural Design, ZRI Report 4/91, Dept. of Computer Science, Univ. of Kaiserslautern, 1991.

Mi88 Mitschang, B.: Towards a Unified View of Design Data and Knowledge Representation, in: Proc. of Second Int. Conf. on Expert Database Systems, Tysons Corner, Virginia, pp. 33-49, April 25-27, 1988. Further publication by the Benjamin/Cummings Publishing Co.

MM89 Mattos, N.M., Michels, M.: Modeling with KRISYS: the Design Process of DB Applications Reviewed, in: Proc. the 8th Int. Conf. on Entity-Relationship Approach, Toronto - Canada, Oct. 1989, 159-173.

Mo92 Moss, E.: Working with Persistent Objects: To Swizzle or Not to Swizzle, in: IEEE, TOSE, Vol.18, No.8, August 1992, pp. 657-673.

MP91 Mitschang, B., Pirahesh, H.: Integration of Composite Objects Into Relational Query Processing: the SQL/XNF Approach, in: Proc. of Int. Workshop on Query Processing in Databases with Object-Orientation, Complex Objects, and Nested Relations (to be published by Morgan Kaufman), FRG, 1991.

MPPLS93 Mitschang, B., Pirahesh, H., Pistor, P., Lindsay, B., Südkamp, N.: SQL/XNF - Processing Composite Objects as Abstractions over Relational Data, in: Proc. of Ninth Int. Conf. on Data Engineering, April 1993, Wien, pp. 272-282.

OHMS92 Orenstein, J., Haradhvala, S., Margulies, B., Sakahara, D.: Query Processing in the Objectstore Database System, in: Proc. ACM SIGMOD Int. Conf. on Management of Data, San Diego, June 1992, 403 - 412.

PMCLS90 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, September 1990.

Ro92 da Rocha, R.: Transformation and Rewrite in the Query-Processing System of KRISYS, Master Thesis (in portuguese), Universidade Federal do Rio Grande do Sul, Porto Alegre, May 1992.

Se88 Sellis, T.K.: Multiple Query Processing. ACM Transactions on Database Systems, Vol. 13, No. 1, March 1988, 23-52.

Se90 Sequent Computer Systems: System Summary, 1990.

TD93 Thomas, J., Deßloch, S.: A Plan-Operator Concept for Client-Based Knowledge Processing, Proc. 19th Int. Conf on Very Large Databases, August 1993, Dublin, Ireland.