In: Tagungsband der GI-Fachtagung ‚Datenbanken in Büro, Technik und Wissenschaft‘ (BTW‘95), nformatik aktuell, Dresden, März 1995, S. 403-422.

Design and Implementation of Advanced Knowledge Processing in the KBMS KRISYS

S. Deßloch[1], N. Mattos[1], B. Mitschang, J. Thomas
Department of Computer Science
University of Kaiserslautern
67653 Kaiserslautern, Germany
e-mail: {thomas | mitsch}@informatik.uni-kl.de

Abstract

Advanced data models and knowledge models together with their powerful query and manipulation languages have already proven to be essential for systems that support non-standard applications such as engineering and knowledge-based application systems. In order to raise their usability and acceptability, it is overly important to provide adequate implementation techniques that guarantee extensible and efficient processing for this advanced DBMS scenario. In this paper we present design alternatives and implementation techniques for such kinds of advanced DBMS, strongly focussing on query and knowledge processing in client/server architectures. To discuss our considerations and implementation technologies, we refer to the knowledge-processing framework of the KBMS KRISYS, although our ideas are generally applicable to (advanced) DBMS.

Keywords: Implementation Issues, Knowledge Processing, Query Processing, Constraint Enforcement, Client/Server Architectures, DBMS, KBMS

1. Introduction

In the last years, the modeling and querying facilities required by advanced applications have consolidated, and standardizations like SQL3 [ISO94] and ODMG [ODMG93] have emerged. Consequently, current research must not focus on data models and their languages only, but should pay increasing attention to improving system performance through adequate processing models and implementation technologies. Prominent examples adhering to that direction come from the area of object-oriented databases, as mentioned in, e.g., [OHMS92], knowledge base management systems (KBMS), e.g. [In84, Ma91, KL89], or other post-relational DBMS, as for example mentioned in [HS93, Gr94, LLPS91, LVZZ94, CR94].

KRISYS (Knowledge Representation and Inference System), a KBMS developed at the University of Kaiserslautern, features an object-oriented knowledge model and a set-oriented, declarative query language as user interface. At the last BTW conference, we reported experiences with the first implementation of KRISYS, which were based on a number of applications modeled with this KBMS [DLMT93]. While the object model turned out to be sufficiently expressive, the lack of an adequate concept for modeling semantic integrity constraints became apparent. Regarding the processing model and implementation of KRISYS, we learned that, for performance reasons, we needed a better adaptation of processing to the workstation/server architecture KRISYS has been conceived for. Application-oriented processing should be performed at the workstation, relying on a buffer for exploiting locality of reference.

As a result we started a major redesign of KRISYS We put special emphasis on knowledge-processing techniques, especially on query processing and constraint management. In this paper we report on the design decisions and implementation techniques that guided the development of advanced knowledge processing in the new version of KRISYS.

1. IBM Database Technology Institute, Santa Teresa Laboratory, 555 Bailey Ave., San Jose, CA, 95161 USA, e-mail: dessloch@almaden.ibm.com, mattos@stlvm14.vnet.ibm.com.

From a knowledge-modeling point of view, the representational framework for semantic integrity constraints resembles the major improvement in the new KRISYS version. Concerning knowledge-processing techniques, the new KRISYS is conceived for client/server environments with most application-oriented processing being done in main-memory at the client side. Consequently, a client infrastructure for efficient and effective knowledge processing close to the application is indispensable. KRISYS supports main-memory query-processing which asks for run-time optimization to dynamically exploit the client buffer contents at run time to achieve efficient overall query processing. In addition, this framework for advanced knowledge-processing supports extensibility at different levels of query processing to cope with later extensions either of the query interface (shifting more application-oriented semantics into the scope of query processing) or of evaluation methods (such as improved join algorithms) [TD93]. Moreover, a new, flexible mechanism for supporting integrity constraints is added to the system features and realized within the same framework.

Another major design decision has been made to improve the interaction between client and server components. The object-server approach, which turned out to be a performance bottleneck in the first KRISYS implementation, is replaced by a query-server architecture that supports the delegation of subqueries. In contrast to traditional query servers, this approach allows to exploit existing buffer contents at the client and improves the overall balance of processing across the client-server architecture. Moreover, it supports set-oriented retrieval of objects from the server, yet avoiding some of the drawbacks encountered with page-server approaches, such as [LLOW91], whose effectiveness strongly relies on appropriate object-clustering mechanisms.

All these design and implementation decisions are thoroughly motivated by the lessons learned from the first implementation of KRISYS [DLMT93] and therefore do not need to be restated in this paper, which is organized as follows. Sect. 2 provides a brief overview of the KRISYS object model and its query language, giving the conceptual starting point for our re-implementation of KRISYS as being described in Sect. 3. A basic understanding of the tasks of each component relevant to knowledge processing is provided and the interaction of these components is demonstrated using a small example. The main motivation for this presentation is to establish a framework for the detailed discussion of architectural components and the subsequent steps of query optimization and processing in Sect. 4. Finally, Sect. 5 sums up the major results of the paper, discusses related work, and gives a brief outlook to future work.

2. The Object Model and Query Language of KRISYS - A Brief Review

In this chapter, we summarize the main features offered by the object model of KRISYS, as adopted from the first version of KRISYS. The model extensions related to integrity constraints will be discussed in Sect. 4.4.

The KOBRA (KRISYS Object Representation) object model supported by KRISYS is comparable to object-oriented data models [Ca91]. 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 for modeling 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. KRISYS supports the abstraction concepts of classification, generalization, association, and aggregation [Ma91] whose semantics (e.g., inheritance along the classification and generalization relationships) is automatically enforced by the system. Objects are typically organized in hierarchies or lattices defined via those abstraction concepts. For the generalization and classification relationships, this means that both multiple inheritance and multiple instantiation (i.e., an object is a direct instance of more than one class) are supported. In addition, the object model of KRISYS provides various other features, as, e.g., integrity constraints and rules, not usually found in object-oriented models [Ma91, De93].

KOALA (KRISYS Object Abstraction Language) [Ma91], a descriptive, set-oriented language, constitutes 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. 1 selects all furnishings costing more than US$1,000, which are suitable for rooms located at the south side of the house. Please note that, using the MESSAGE predicate, a method 'is-suitable-for' is invoked for determining suitability. For reference purposes, we numbered the lines of the query. We assume a KB containing generalization hierarchies for *rooms* and *furnishings*. Symbols with a leading question mark are query variables, similar to tuple variables in SQL. They may appear in the qualification clause and the projection clause. In our example, the projection clause states that the complete objects retrieved constitute the result of that query. The query refers to the abstraction concept of classification and reads as follows: Firstly, *instances of rooms* (direct as well as indirect ones, indicated by the asterisk behind the class' name) are retrieved and bound to query variable ?X (line (1.1)). The resulting set of objects is further restricted by the condition that attribute *orientation* contain value 'South' (line (1.2)). In addition to instances of *rooms*, the query also refers to instances of *furnishings* which must have a price higher than US$1000, represented by the value of attribute 'price' (lines (1.3), (1.4)). Finally, a method is called to check which furnishings are suitable for which rooms (line (1.5)).

```
(1.0)(ASK  ((?X)(?Y))                                      } projection clause
(1.1)    (AND   (IS-INSTANCE ?X rooms *)                   \
(1.2)           (EQUAL South (SLOTVALUE orientation ?X))    |
(1.3)           (IS-INSTANCE ?Y furnishings *)              > qualification clause
(1.4)           (> (SLOTVALUE price ?Y) 1000)               |
(1.5)           (MESSAGE is-suitable-for ?Y ?X)))          /
```

Fig. 1: Sample ASK statement.

3. The New KRISYS Architecture and Processing Model

In the following, the new architecture, as shown in Fig. 2, will be presented to give a basic understanding of the tasks of each system component and to demonstrate the interaction of the components using a small example. The main motivation for this presentation is to establish a framework for the detailed discussion of architectural components in subsequent sections, thus providing the background for understanding the role of each component in a global context.

3.1 Overview of the Architecture

The server part, resembled by the *PRIMA kernel* [HMMS87], concentrates on an efficient and reliable KB management. At its interface, it features a composite-object data model, the MAD (Molecule-Atom Data) model, and its query and manipulation language MQL, for application-independent data management. The workstation side of KRISYS is partitioned into several components organized on three hierarchical layers. The Working Memory is seen as a passive application buffer controlled by the Context Manager, which is keeping a declarative description of the Working-Memory contents and is responsible for loading and unloading sets of objects into or from the Working Memory. To transfer objects between server and workstation, the Context Manager interacts with the Mapping System, which transforms objects from MAD to KOBRA structures and vice versa. This component is also responsible for generating appropriate mapping schemes for the processing phases of an application. At the next layer, the Constraint Manager appears as an additional component besides the KOBRA component. It performs all activities related to checking or processing the constraints of the KB. The topmost layer, the KOALA Processing System, provides the user (and application system) interface. Its task is to prepare and control the processing of KOALA queries.

In the following, we will give an overview of the tasks of each component located at the workstation side. Moreover, we will sketch the overall processing model of the new architecture by showing the interaction of the different parts of the system during query processing.

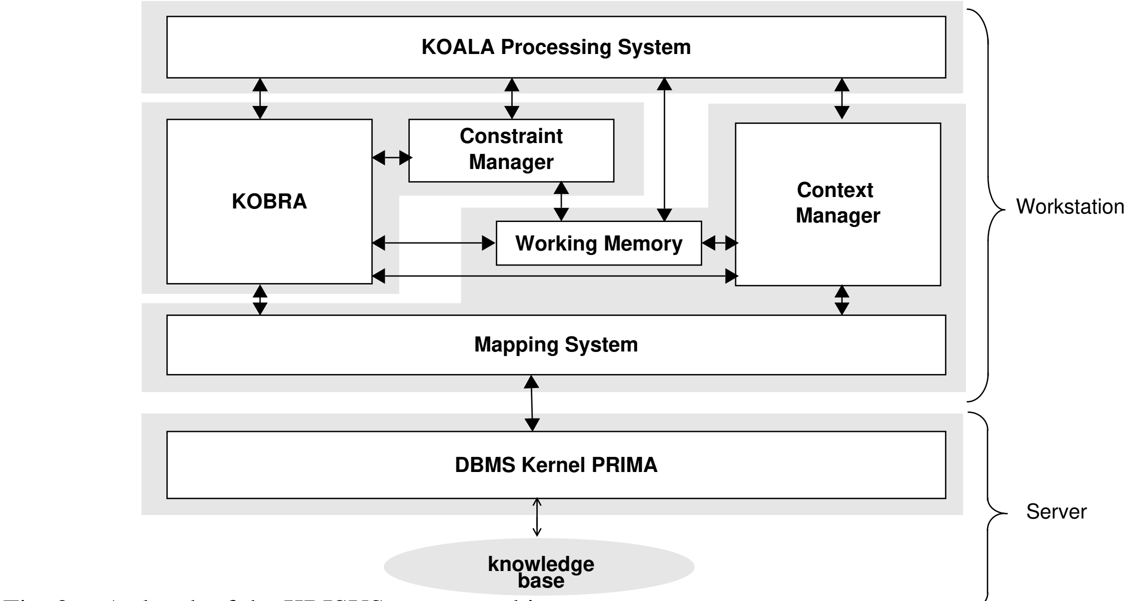


Fig. 2: A sketch of the KRISYS system architecture.

Mapping System

The *Mapping System* provides KOBRA objects as uniform knowledge-representation format for the workstation-based components of KRISYS. Thus it isolates workstation-based knowledge processing from representational aspects of the current server DBMS. Moreover, the Mapping System allows the generation of optimized, application-dependent mapping schemes and their utilization during application processing. Such a mapping allows us, for example, to combine several interrelated classes in a single (PRIMA) table or to split one class across several tables in order to improve the performance of critical DML operations. This task can be divided into the following independent subtasks [Su91]:

- generation of an optimized mapping for a specific application,
- transformation of delegated KOALA subqueries into queries of the PRIMA kernel, and
- adaptation of the mapping in case of changes in the KB structure.

These tasks are accomplished by the following internal components of the Mapping System.

- The *MAD-Schema Generator* establishes an efficient mapping tailored to the needs of the applications [Su91]. It is activated after the design of the KB has been completed, and produces a mapping scheme based on the KB structure as well as processing characteristics.
- According to the mapping information produced by the Schema Generator, the *Mapping Component* handles the delegation of KOALA (sub-)queries [Sch91]. It produces appropriate MQL queries, which are evaluated by the PRIMA kernel, and transforms the corresponding results into the data structures of the Working Memory.
- Operations like the definition or deletion of classes, attributes, etc., which are usually regarded as schema-evolution operations, may induce changes on the mapping produced by the Schema Generator. These changes, as well as transformations of data into the new representation, are accomplished by the *Transformation Component* [Kr93].

Working Memory

The general task of the Working Memory is to support the concept of *near-by-the-application locality of processing* when KOBRA objects are referenced during query and constraint processing. In order to accomplish this task, the Working Memory

- provides data structures and operations for representing and effectively manipulating objects in a format directly reflecting the semantics of the knowledge model,

- allows efficient set-oriented processing of objects by the KOALA Processing System through so-called *Access Structures* (AS), combining functions similar to DB scan-operations with main-memory index facilities, and
- supports pointer-like navigational access or traversal of objects in abstraction hierarchies to optimize the processing of model-inherent constraints.

Transformations of the object format take place whenever objects are transferred from the server and stored in the Working Memory. Such transformations include swizzling pointers representing inter-object and abstraction relationships, construction of appropriate Access Structures, etc. [La91].

While the Working Memory provides basic functions for modifying its contents, buffer management is performed by the Context Manager introduced later in this paper.

KOALA Processing System

The *KOALA Processing System* accepts a KOALA statement, transforms it into an algebra graph, performs rewrite optimizations, and generates a plan-operator graph (i.e. execution plan), which is then compiled and executed [TMMD93].

Obviously, the evaluation of a query should exploit the Working-Memory contents as far as possible. To reach this goal, the KOALA Processing System closely interacts with the Context Manager as well as with the Constraint Manager to identify which parts of the query should be performed at the workstation side and which parts are to be delegated to the server. This decision is reflected by different types of plan operators in the execution plan (e.g. 'Buffer-SELECT' and 'DBMS-SELECT').

Context Manager

During the generation of an execution plan, the KOALA Processing System has to find out which parts of the query may directly be executed on the buffer contents, because the required objects are already present in the Working Memory. A buffer description based on object identifiers is not sufficient for accomplishing this task [De93, DLMT93]. Instead, a declarative description is required.

It is the major task of the *Context Manager* to maintain such a declarative buffer description. It can be incrementally constructed from the subqueries that are delegated to the server since the selection conditions of these queries perfectly describe the results (i.e. the contexts) that are loaded into the Working Memory. To provide the required information for the KOALA Processing System, the Context Manager performs special context-inferencing operations comparing parts of a query to the contexts of the buffer and producing a declarative description of those object sets that must be fetched from the server.

KOBRA

The KOBRA component provides the other components, mainly the KOALA Processing System, with a basic set of functions for modifying and retrieving information in the Working Memory on a 'per object' basis. This functionality, which includes reading/changing attribute values, object creation/deletion, connection/disconnection of abstraction relationships, method execution, etc., incorporates the semantics of the KOBRA knowledge model.

Constraint Manager

The task of maintaining KB consistency according to the given constraints is fulfilled by an additional component, the *Constraint Manager* [De93]. Based on events reported by the KOALA Processing System or the KOBRA component (e.g., atomic write/read operations, begin/end of composed activities, etc.), the Constraint Manager initiates actions to ensure consistency, or stores the events for later, deferred activation. Moreover, the creation, deletion, or modification of constraints is reported to this component.

Additionally, the Constraint Manager provides information about certain types of constraints to the KOALA Processing System necessary for rewriting purposes during query optimization.

3.2 Interaction of System Components During Query Processing

To illustrate the interactions and dependencies between the different system components of the new KRISYS architecture, we sketch the evaluation of a simple example query. We refer to the query already presented in Fig. 1.

After having been submitted to the KOALA Processing System (Fig. 3 ①), the statement is transformed into an algebra graph, on which algebraic optimizations are performed (Fig. 3 ②). These involve query rewrites commonly applied in relational DBMS, such as subquery to join transformation, selection-push-down, etc.

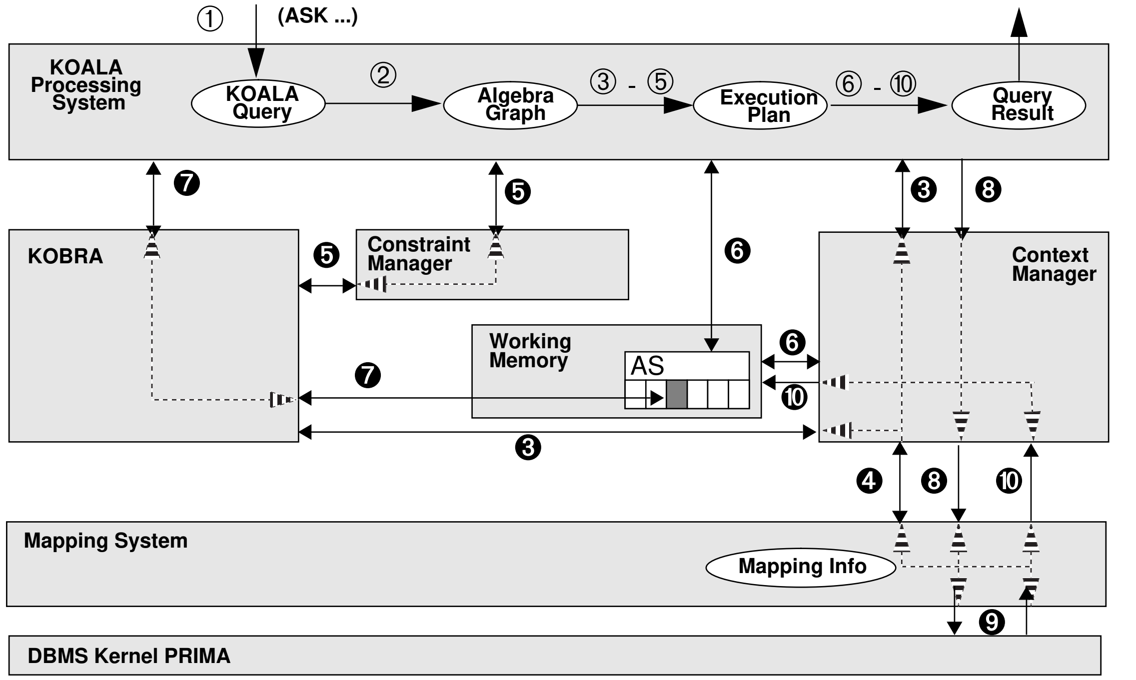


Fig. 3: Interaction of components during the evaluation of a KOALA query.

In the next step, an appropriate execution plan will be generated. At this stage of processing, the KOALA Processing System will interact with the Context Manager (Fig. 3 ❸) to determine which parts of the query can be executed on the Working-Memory contents and which have to be delegated. Not all parts of the query are considered for delegation. For example, all operations involving method calls, like the join operation of rooms and furnishings resulting from the activation of method 'is-suitable-for', can only be performed at the workstation side [De91]. To provide the required objects, the Context Manager analyzes the descriptions of the contexts already installed in the Working Memory. There may, for example, be no context containing rooms, so that an appropriate answer is given to the KOALA Processing System, which will then consider the delegation of the corresponding subquery. It will, however, not always be that easy. In many cases there will be contexts that somehow overlap with the set of objects requested by the subquery. For this purpose, the Context Manager supports specialized inference capabilities that allow to determine a declarative description of those objects that are still missing and consequently have to be fetched from the server.

In our example, the Context Manager would return the answer that the selection subquery involving furnishings can completely be supported by an existing context. Using the information provided by the Context Manager, the KOALA Processing System produces an appropriate execution plan. For this task, the KOALA Processing System additionally needs an estimation of execution costs. Here, the mapping scheme chosen for the current application plays a very important role. To provide the required cost estimations, the Context Manager therefore enriches its description of execution alternatives with cost information provided by the Mapping System, before passing it to the KOALA Processing System (Fig. 3 ❹).

Depending on cost information, the KOALA Processing System may even choose not to exploit some of the inferences drawn by the Context Manager.

Additionally, the KOALA Processing System must interact with the Constraint Manager (Fig. 3 ❺)[2]. This is necessary because the evaluation of predicates in the selections to be delegated might involve the activation of constraints, which cannot be performed by the server [De91]. For example, the attribute 'price' of *furnishings* may be involved in a constraint relating it to the additional features of the furnishing. Depending on how the KB designer has chosen to represent this constraint (e.g., defining the price of *furnishing* as a virtual attribute, whose value is computed on demand), additional rewrite operations may be necessary.

Let us assume that the subqueries chosen for delegation do not require the activation of constraints, so that the execution plan generated by the KOALA Processing System is confirmed and can be compiled and executed (steps ⑥ - ⑩). The execution of Working-Memory plan operations is based on Access Structures containing sets of object tuples (Fig. 3 ❻). Working-Memory operators appearing as leaves of the plan-operator graph rely on contexts residing in the Working Memory. To this end, the Context Manager provides initial access to the associated contexts organized in particular Access Structures managed by the Context Manager. In our example, an Access Structure containing the furnishings is provided. Each operator can be understood as producing a temporal Access Structure to be consumed by its successor. The functionality required to implement the operations performed on each element of the Access Structure during the execution of a plan operator (e.g., accessing the attribute 'price' of the instances of *furnishings*) is provided by the KOBRA component (Fig. 3 ❼).

The execution of DBMS plan-operators is performed in several steps. First, the Mapping System is consulted to produce an equivalent server DML operation based on the actual mapping scheme (Fig. 3 ❽). This DML operation is sent to the server and executed (Fig. 3 ❾). The result of the query is then returned to the Mapping System, which transforms it into the Working-Memory representation (i.e., KOBRA objects). Finally, the resulting objects are inserted into the Working Memory and collected in a new Access Structure (Fig. 3 ❿). This last step is performed by the Context Manager, which registers the result of the delegated subquery as a new context and provides it as an Access Structure to subsequent plan operators.

Plan execution is continued in the above described manner and completed by returning the result of the query to the user or application.

4. Knowledge Processing in KRISYS

4.1 Working Memory

As described above, processing of the Working-Memory contents is not performed ‘directly’ by the application program, but is carried out through the KOALA Processing System, the Constraint Manager, and KOBRA. The Working Memory therefore must provide functionality for efficiently supporting the processing requirements of these components [La91].

Efficient access of information on a ‘per-object’ basis

An important processing requirement is the fast localization of objects based on their identifiers. This is achieved through an object hash-table. Moreover, efficient access to information about the objects (i.e., attribute information, aspect information, etc.) is also supported. Such access usually occurs repeatedly to different attributes of the same object, or to different aspects of the previously accessed attributes, and can therefore be seen as a kind of ‘navigation within the object’. For example, an update operation involves the localization of the object, the write

2. Please note that points ③ - ⑤ are not necessarily executed in the sequential order chosen above for illustrating the interactions.

access on an attribute within the object, and additional accesses to aspect information associated with the attribute in order to record events and notify the constraint affected by the update.

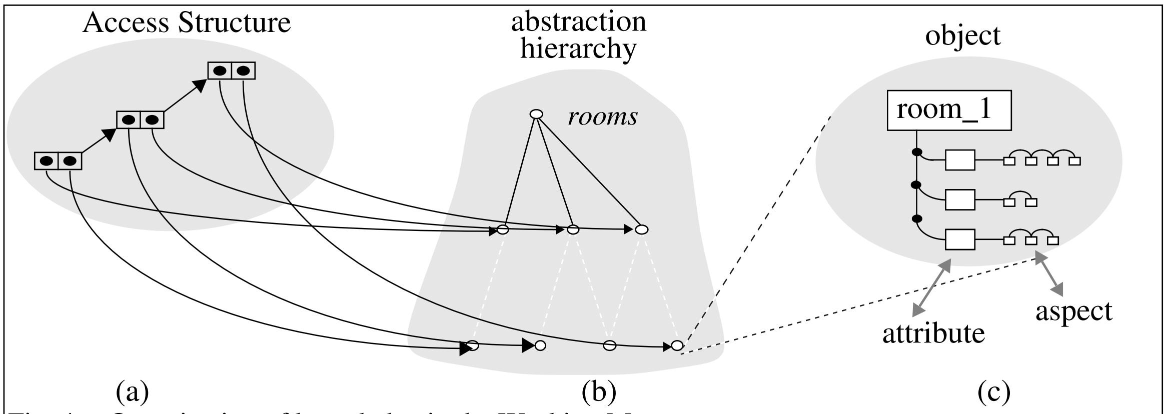


Fig. 4: Organization of knowledge in the Working Memory.

The different internal representational levels (object, attribute, and aspect) are directly reflected in the Working-Memory representation (Fig. 4 c), and are linked via main-memory pointers. The pointers allow an efficient retrieval of attribute and aspect information based on the data structure of the object. In Fig. 4 c, we have sketched this representation for the object *room_1*.

The chosen representation has the additional advantage to support the structural heterogeneity of objects in a single, uniform data structure for accessing object information. For example, even objects belonging to the same class may have varying structures because they belong to structurally different subclasses. Moreover, some objects might be instances of multiple classes, or some attributes may have been defined only for individual objects. Due to the above representation scheme, information about objects can be retrieved and modified on a uniform basis without having to access additional meta information (e.g. class descriptions) in order to interpret the data structures.

Moreover, the Working Memory offers functions for creating and deleting data structures for objects, attributes, and aspects, as well as for read/write access.

Fast navigation across abstraction hierarchies

It is important to speed up the retrieval of objects via abstraction relationships (e.g., all 'transitive' instances of a class) and provide means for efficiently guaranteeing model-inherent integrity constraints. For example, the creation of a new attribute in a class requires the traversal of the class hierarchy to perform inheritance. Consequently, the abstraction relationships between objects are materialized as main-memory pointers. This materialization has been depicted in Fig. 4 b for the generalization/classification hierarchy of *rooms*.

Besides operations for establishing/deleting abstraction links among objects, the Working Memory offers additional functionality to traverse abstraction hierarchies and perform operations on the traversed objects. These functions, which are mainly used for maintaining model-inherent integrity, can be supplied with parameters that determine the relationships to be followed, specify a search strategy for the traversal (e.g., breadth-first), or denote operations to be performed at each node during the traversal. For example, attribute inheritance was easily implemented as a breadth-first traversal following the *subclass-of* and *instance-of* relationships, performing the creation of a data structure for an attribute every time a node is reached, and testing for possible inheritance conflicts which will determine the next step in the traversal.

Direct support of set-oriented processing of Working-Memory objects

For set-oriented processing of objects, the Working Memory allows the KOALA Processing System to create, maintain, and exploit collections of objects organized as Access Structures (cf. Fig. 4 a). In order to be suitable for the purposes of the KOALA Processing System, an Access Structure must contain items that match the internal format used during query processing, the so-called KOALA tuple-format, which will be described in detail in section 4.2.2 and basically consist of a single object or several associated objects.

As shown in Fig. 4 a, these tuples do not contain copies of Working-Memory objects, but are associated with the objects via main-memory pointers. In this example, the Access Structure contains pairs of objects resulting from a join. This ensures that during query processing no redundancies are introduced by the KOALA Processing System. Intermediate results are produced by employing a sophisticated concept for sharing object information even at the attribute and aspect level, using multiple pointers to the same information.

The Working Memory provides the following functionality for exploiting Access Structures.

- Creation and deletion of Access Structures.
- Opening and closing cursors for Access Structures, which allow to scan Access Structures in forward or backward direction. Multiple cursors can be defined for the same Access Structure, so that an intermediate query result can be exploited by several 'threads' of the query execution simultaneously.
- Functions for reading, inserting, removing, and replacing tuples of Access Structures relative to the cursor position.

In its basic form, an Access Structure is organized as a list of KOALA tuples. In addition, Access Structures can also be organized as trees or hash tables, thereby supporting the maintenance of main-memory indices. In such a case, additional information must be provided at the creation of Access Structures, describing the key attributes of objects to be indexed, and the associative access to the contents of Access Structures is supported through additional functionality. With these facilities, the KOALA Processing System may fully exploit the contents of the Working Memory during query processing. Further optimizations are provided through the usage of Access Structures as main-memory indices, which may be introduced dynamically or temporarily (i.e., in the scope of a single query) during query processing.

In summary, the Working Memory directly and effectively supports the requirements of the other system components concerning the processing of object information, thereby providing a suitable basis for knowledge processing in the workstation component of KRISYS.

4.2 Query Processing

Query Processing is performed by the KOALA Processing System. To guarantee a semantically clear and streamlined system design, we partitioned its overall tasks into a processing framework and a part responsible for knowledge-model semantics. While the processing framework is based on an algebraic model that allows conventional (relational) algebraic optimizations to be used to a large extent, knowledge-model semantics is founded on the functionality provided by KOBRA. In the following, we will discuss both issues in more detail.

4.2.1 Knowledge-Model Semantics

Except for the notion of object structures, the processing framework of the KOALA Processing System is completely independent of knowledge-model semantics which is introduced via *base predicates*. Base predicates represent an intermediate level between KOALA and KOBRA. While KOALA expressions are declarative, state-oriented, and set-oriented, base predicates operate object-wise, however still being declarative and state-oriented. Since base predicates resemble assertions on single objects, they can be easily mapped to the procedural level of KOBRA. Fig. 5 depicts the different representational levels and their processing characteristics.

To the right side of Fig. 5, we sketched how an example KOALA statement is translated to base predicates and the KOBRA level. We use a TELL asserting that all corridors (being instances of that class, denoted by query variable ?C) are adjacent to any room ?R lying in the same private area ?A. Let us have a look at the way the assertion is translated to base predicates. The assertion to be met is that a qualifying corridor ?C is a value of attribute 'neighboring-rooms' of any adjacent room of that private area. It is translated into a piece of code at the KOBRA level that reads the actual value of attribute ‘neighboring-rooms’ and adds the current value of ?C to the attribute values if it is not yet included.

level	processing	example
KOALA	declarative ***set-oriented*** state-oriented	(TELL (IS-IN ?C (SLOTVALUES neighboring-rooms ?R)) WHERE (EXIST ?A (IS-INSTANCE ?A private-areas *) (IS-INSTANCE ?C corridors *) (IS-INSTANCE ?R rooms *) (IS-AGGREGATION ?A has-rooms ?C) (IS-AGGREGATION ?A has-rooms ?R)))
base predicates	declarative ***object-wise*** state-oriented	**Conditions:** is-inst(?A, private-areas), is-inst(?C, corridors), is-inst(?R, rooms) is-aggr(?A, has-rooms, ?C), is-aggr(?A, has-rooms, ?R) **Assertions:** has-attval-member(?R, neighboring-rooms, ?C)
KOBRA	procedural object-wise ***state-dependent***	..... actval:= read-attr(?R, neigboring-rooms) (if not(member (?C, actval)) add-attr-value(?R, neighboring-rooms, ?C) ......

Fig. 5: Mapping of KOALA to KOBRA.

4.2.2 Processing Framework

The overall steps of query processing proceed in a similar fashion as those in relational DBMS [HFLP89]: first, an algebra graph is generated and subsequently optimized, i.e., rewritten; thereafter, a plan-operator graph is constructed; finally, executable code is assembled, and the query is actually evaluated. Fig. 6 gives an overview of the steps and representational levels of query processing. We will discuss them more concisely in the following.

Algebra Level

Algebra operators work on data streams consisting of sets of n-tuples which they accept as input and also produce as output. A data stream can be seen as a table made up of n columns bound to query variables. A table is represented as an Access Structure in Working Memory, and each n-tuple (table entry) represents an Access-Structure entry and comprises n elements, each featuring object level, attribute level, and aspect level. The elements of a column may be unnested on the attribute level and/or the aspect level, depending on the operations to be performed on that column. Fig. 7 depicts an example table consisting of 2-tuples and demonstrates the effects of unnesting/nesting the first column on the attribute level[3].

KOALA algebra consists of three kinds of operators. The first kind comprises operators that are responsible for handling columns or object structures (e.g., COL-COPY, COL-PROJECT, COL-UNION, NEST, UNNEST). The operators of the second kind provide functionality comparable to conventional relational algebras, e.g., EXIST, FORALL, JOIN, PRODUCT, or SELECT. The third kind is responsible for modifications of the KB. As described above, KOALA algebra employs state-oriented base predicates for realizing knowledge-model semantics. Consequently, the algebra level need not consider the actual state of the KB, and needs only a single operator, ASSERT, to carry out modifications. In relational algebras, however, where state-orientation is not known, several operators are required to carry out changes in the database (e.g., UPDATE, INSERT, DELETE).

To illustrate how a query is translated into an algebraic representation, we refer to the sample TELL statement and the way it is decomposed into base predicates shown in Fig. 5. First, an algebra graph is constructed (cf. step ❶ from Fig. 6). It is shown in Fig. 8 (a). Firstly, instances

3. Nesting and unnesting of attributes is used, for example, during projections.

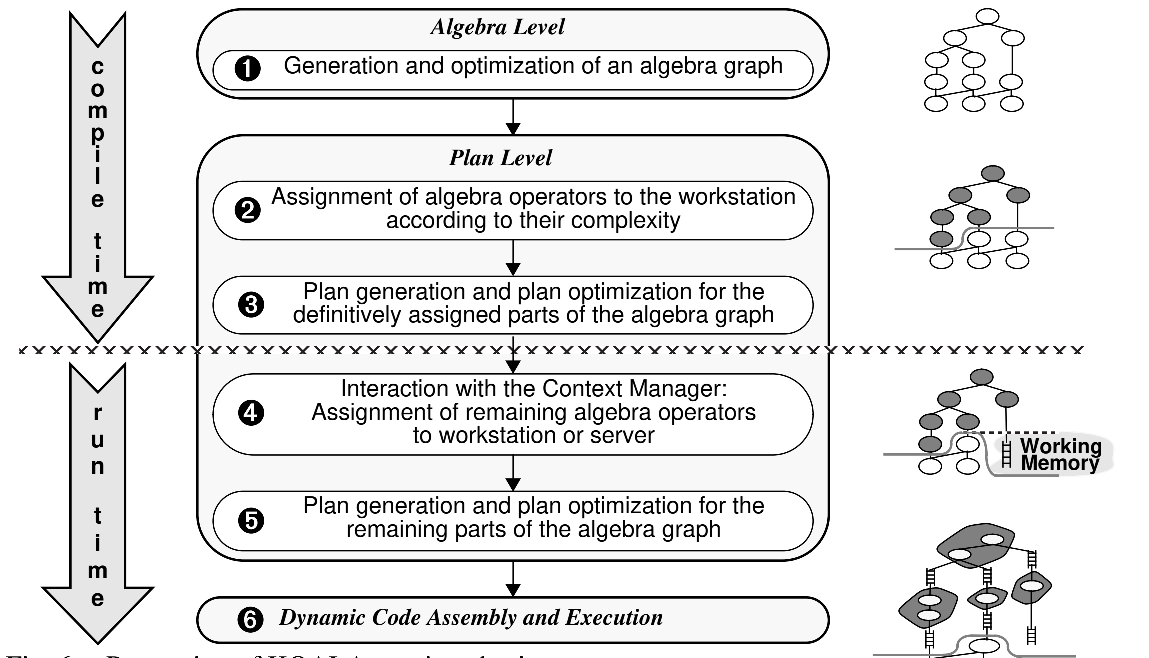


Fig. 6: Processing of KOALA queries - basic steps.

of *area*s are selected. Since relevant rooms and corridors must be components of some private area, the corresponding object identifiers can be retrieved from attribute *has-rooms* of each selected area. To access this attribute, each area object must be unnested on the attribute level. Thereafter, all objects referenced by attribute *has-rooms* of a given area can be retrieved. Since such an evaluation of object references is quite a frequent operation, a special operator FOLLOW-UP has been added to the KOALA algebra. After the FOLLOW-UP, rooms and corridors are selected separately. Those belonging to the same private area are joined and provided as input to the assertion part of the TELL statement.

4.2.3 Plan Level

Our plan-operator approach involves the concepts shown in Fig. 9. We briefly recapitulate the salient features of the plan level; for a detailed description we refer to [TD93, TGHM94]. Plan-operator templates realize a *simple processing paradigm for plan operators*, as well as *extensibility at the plan-operator level*. Knowledge-model semantics is introduced into plan-operator processing via base predicates supplied as parameters to the plan operators. This guarantees *extensibility of the query language* without affecting existing plan operators. Subgraphs of a plan-operator graph are combined to units of execution, called *blocks*. Blocks are constructed such that intra-block processing works in a pipelining mode, i.e. tuple-wise, without the need for intermediate result materialization. The concept of *LAS* (logical Access Structures) provides an adequate data structure for this kind of internal data flow. Data streams between blocks are

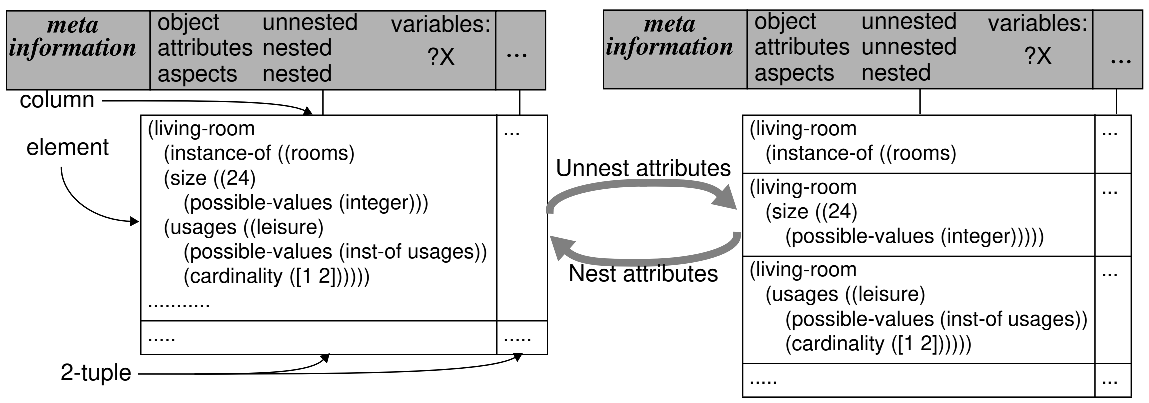
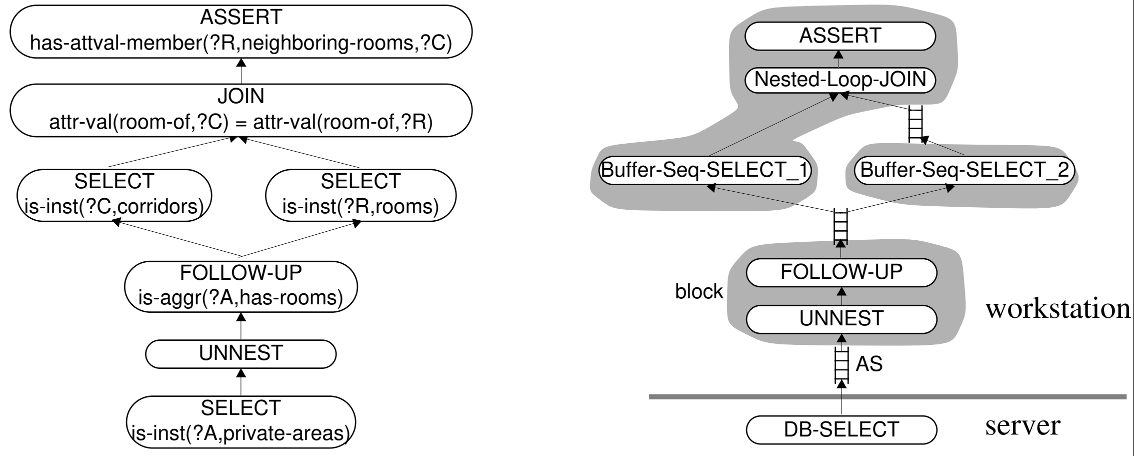


Fig. 7: Unnesting/nesting of columns.



(a) algebraic representation (b) plan-level representation

Fig. 8: Example TELL statement.

materialized in the Working Memory and mapped to Access Structures, thus ensuring *efficient data flow between blocks*. Moreover, the way in which all these concepts are combined warrants *efficient dynamic query optimization* and the construction of *flexible units of execution* even at run time. These characteristics were achieved by a modular design and realization of the plan level.

Due to the workstation/server environment in which query processing is performed, determining the evaluation site of each algebra operator is a crucial issue (step ❷ in Fig. 6). By delegating operations to the server, the amount of data to be transferred into Working Memory can be reduced. This also results in less objects to be installed in Working Memory allowing a better exploitation of its storage capacities. Deciding on the evaluation site of each operator is based upon two criteria. Firstly, those algebra operators must be assigned to the workstation that are either too complex to be evaluated by the server DBS or that cannot be transformed into queries to the server due to the current mapping to the server DBMS.[4] Secondly, for performance reasons, the KOALA Processing System must exploit the contents of the Working Memory (including indices, sort orders, etc.).

The first criterion can be tested at compile time so that a preliminary borderline between workstation-based and server-based operations can be drawn (cf. Fig. 6, right side). Depending on the contents of the Working Memory at run time, the operators below the borderline may be assigned to workstation or server. Hence, plan-level manipulations can be definitively completed only at run time, yet preliminary plan optimizations may be performed for those operators definitively assigned to the workstation to save run-time effort (step ❸ in Fig. 6).

At run time, the KOALA Processing System interacts with the Context Manager to compare the actual contents of the Working Memory to the information referred to by the query at hand. If the input to an operator already resides in Working Memory (as an Access Structure), the producing subgraph[5] is pruned and replaced by a pointer to the appropriate Access Structure.

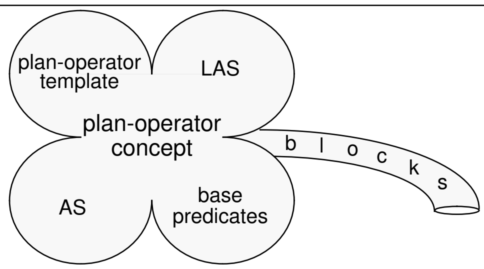


Fig. 9: Constituents of our plan-operator concept.

4. For simplicity reasons, we shall not consider this aspect in this paper.
5. Consisting of one or more plan operators.

While this applies to operators above as well as below the preliminary borderline, for the latter it also implies that these operators are assigned to the workstation, i.e., the borderline is moved downward, and less operators must be delegated to the server (sketched in Fig. 6, right side). For those subgraphs not yet assigned to workstation or server, two further situations may arise. If the Working Memory does not contain any required input for a subgraph, the whole subgraph must be evaluated at the server DBMS[6], and the border between workstation and server remains where it has been put at compile time. The second situation occurs if only part of the required input is residing in Working Memory, and the rest must be fetched from the server. In this case, basically two processing strategies are possible: to completely delegate the query to the server, requiring to previously write back to the database the potentially updated portion of knowledge installed in Working Memory, or to only complement the Working-Memory contents such that the query can be performed in Working Memory. To solve this optimization problem, the KOALA Processing System interacts with the Context Manager (step ❹ in Fig. 6). This interaction and its outcome for our example query will be described in Sect. 4.3. Fig. 8 (b) shows the resulting plan-operator graph[7] assuming that no instance of *area* is residing in Working Memory. Hence, the corresponding selection must be executed in the server and is transformed into a server plan-operator (DB-SELECT). Since the subsequent UNNEST operator refers to object structures of the knowledge model, it must be carried out at the workstation side. Consequently, all its successors must be executed there as well, although evaluating the FOLLOW-UP operator may result in additional queries to be sent to the server. Since making assertions over the KB may involve the full functionality of the knowledge model, algebra operator ASSERT is always transformed into a workstation-based plan-operator (of the same name).

The resulting plan-operator graph can be further optimized (step ❺ in Fig. 6).

4.2.4 Dynamic Code Assembly and Execution

These tasks complete overall query processing (step ❻ in Fig. 6). The plan-level representation of a query is transformed into a graph made up of *blocks* (sketched in Fig. 6, right side) which are the units of execution in our query-processing approach. Blocks rely on the plan-operator level, both conceptually and concerning their implementation [TGHM94]. Just like plan operators, blocks accept one or more input streams and produce a single output stream. Blocks are constructed based on the processing characteristics of plan operators to minimize the amount of materialized intermediate results during query processing. Fig. 8 (b) shows the blocks constructed from the plan-operator graph at hand. The UNNEST and FOLLOW-UP operators can work in a pipelining fashion, and are therefore combined into a single block. The same holds for the Buffer-Seq-SELECT_1 (alternatively Buffer-Seq-SELECT_2), NESTED-LOOP-JOIN and ASSERT operators.[8]

Evaluating a query means executing the corresponding blocks. The most straightforward way is to perform blocks in a sequential order defined by the inter-relationships of the block-structured graph. Additionally, our query-processing approach also permits parallel execution of blocks [TMMD93].

Opposed to conventional query-processing systems requiring *strict compilation*, we assemble executable code by putting together precompiled functions, yet we may still choose to compile a query, e.g., for complex queries or large amounts of data to be processed. We call this approach *dynamic code assembly* [TGHM94], allowing to assemble executable code using fully compiled functions by data structures containing function pointers.

6. Note that it has already been checked at compile time that all operators below the borderline can be evaluated at the server.
7. For simplicity, we did not repeat the base predicates for the plan operators.
8. Note that, for this block being able to operate as a pipeline, the complete results of Buffer-Seq-SELECT_2 must be computed previously. Only in this case, the NESTED-LOOP-JOIN can directly process any new result being piped from Buffer-Seq-SELECT_1.

4.3 Context Management

It is the task of the Context Manager to provide the KOALA Processing System with information about the Working-Memory contents during plan generation. Due to the declarative query interface to the server component, the Context Manager should maintain its description of the Working-Memory contents in a declarative form as well. The Context Manager perceives the Working Memory as a collection of *contexts*. A context represents a set of Working-Memory objects being the complete extension of a logical condition, the *context description*.

Contexts directly correspond to the results of (sub-)queries that have been delegated to the server and whose results have been brought into Working Memory. For each set of query results received from the server, the Context Manager keeps the query condition as a context description and maintains an Access Structure that contains the set of result objects.[9] The language for context descriptions is therefore equivalent to the subset of KOALA that can appear as a condition of a DB-SELECT plan operator. In the following, we will illustrate the main activities performed by the Context Manager in coordination with the KOALA Processing System using the example query already introduced above.

Context Description

Let us assume, that a previous query retrieved from the server all instances of *areas* having more than three rooms. The Context Manager has therefore registered the following context description.

```
((?X)                                   ← projection
     (IS-INSTANCE ?X areas *)           ← variable definition
     (> (SLOTVALUE no-of-rooms ?X) 3))  ← selection
```

The description is divided into three parts. The *variable definition part* (V) characterizes the domain of the context's objects in terms of predicates referring to the abstraction concepts, while the *selection part* (S) states further selection conditions applying to the context. The *projection part* (P) completes the description, listing those attributes that have been brought into the Working Memory.

Context Comparison

When consulting the Context Manager, the KOALA Processing System submits a description of a 'wanted' context, resembling the subquery currently under consideration. The Context Manager compares the wanted context W with a 'given' context G, i.e., with a context available in Working Memory. To this end, we developed an algorithm that basically compares the different parts of W with the corresponding parts of G. For our example query, the KOALA Processing System will ask the Context Manager about contexts available for supporting the selection on *private-areas*. The result of the involved context comparison is depicted in Fig. 10.

First of all, the projection parts of the contexts are compared. Since both projection parts preserve the complete object structure, they turn out to be equivalent. Next, the variable definitions are compared. To determine the result, the Context Manager will at this point have to inspect the abstraction relationships defined in the KB. Since *areas* is known to be a superclass of *private-areas*, the result of the comparison is the set inclusion $V(G) \supseteq V(W)$. Finally, the selection parts are compared. Since no additional selection is defined for W (i.e., all *private-areas* are contained in the context), the comparison results in the set containment $S(G) \subseteq S(W)$.[10] The comparison of predicates in both the selection and the variable-definition parts relies on the interpretation of set relationships as logical relationships, where set containment is equivalent to logical implication.

9. The Access Structure can later be handed to the KOALA Processing System for accessing the context.
10. If several predicates are involved in a selection (or a variable-definition) part, each predicate of G must be compared with each one of W. For complex selection conditions (involving disjunctions, etc.), a disjunctive variant of the algorithm is supplied in addition to the above described (conjunctive) version.

G(iven)	Rel.	W(anted)
P: (?X)	≡	P: (?X)
V: (IS-INSTANCE ?X areas *)	⊇	V: (IS-INSTANCE ?X private-areas *)
S: (> (SLOTVALUE no-of-rooms ?X) 3)	⊆	S: 'true'
context G	'O'	context W

$G \cap W$: SELECT
(IS-INSTANCE ?X private-areas *)
FROM G

$W \setminus G$: LOAD
(?X)
(AND (IS-INSTANCE ?X private-areas *)
(NOT (> (SLOTVALUE no-of-rooms ?X) 3))

Fig. 10: Context comparison (example).

To obtain the overall relationship between G and W, the individual comparison results for P, V, and S must be combined. In our example, the relationship 'G overlaps with W', (denoted by 'O') is achieved, because we have obtained two 'inverse' set inclusions in V and S. The 'overlap' result means that we can exploit the context existing in the Working Memory for answering the query. However, we still need to query the server for those objects not covered by the context. Therefore, the Context Manager additionally produces descriptions how to filter the existing context for the required result set (i.e., how to obtain $G \cap W$ from G), and how to retrieve the remaining objects from the server (i.e., how to retrieve $W \setminus G$). These results are passed to the KOALA Processing System for modifying the query plan accordingly. Moreover, a pointer to the AS containing G is passed on to make it accessible for the KOALA Processing System.

In our example, the KOALA Processing System chooses not to consider the private areas already in the Working Memory but to fetch all instances of *private-area* from the server. Before executing the query, however, the Context Manager must write back to server all private areas residing in the workstation buffer. Analogously, the Context Manager is asked about contexts available for rooms and corridors, the other classes involved in the example query. For reasons of simplicity we assume that these subqueries can be fully supported by contexts at the workstation component. The resulting query execution plan is depicted in Fig. 8 (b).

The above algorithm, which is outlined in detail in [De93], exhibits polynomial time complexity w.r.t. the number of predicates involved in the comparison. It is important to note that, although we retrieve only objects in $W \setminus G$ from the server, we might well retrieve objects that are already in the Working Memory. For example, other contexts might be present there that overlap with W, but are not exploited for the query because the 'amount of overlap' is not promising enough. The Working Memory is capable of handling this situation simply by ignoring already installed objects (i.e., no additional copies are introduced into the Working Memory).

Additional Tasks of the Context Manager

Although maintaining and comparing context descriptions can be seen as the central task of the Context Manager, additional activities are performed by this component to realize consistent buffer management based on the notion of contexts.

For instance, the Context Manager is involved in the process of *update propagation* to the server component. Before a query is delegated to the server, updates that have occurred on Working-Memory objects must be propagated to the server. Otherwise, inconsistencies between server DB and Working Memory may result in wrong query results. Using the Context Manager, we can realize a partial update delegation approach, i.e., not all updates, but only those updates (or a relatively small superset) that are needed to guarantee a correct query result are propagated.

Additional activities are required by the Context Manager to keep context extensions 'up-to-date' after updates, and for discarding contexts from the Working Memory. A more detailed discussion of these tasks can be found in [De93].

4.4 Constraint Management

In this paper, we can only briefly outline the modeling and processing concepts involved with integrity constraints. An elaborate description can be found in [De93].

Constraint Modeling

KRISYS supports a layered approach for representing constraint characteristics at different abstraction levels. The central part of a constraint is its condition, i.e., a logical condition that has to be valid in a consistent KB state. At the operational level, the constraint is described in terms of adjustments telling the system how to correct inconsistencies. For a single constraint, multiple, alternative adjustments can be specified, which are selected and executed according to various criteria. At the realization level, the ‘implementation’ of a constraint is described in terms of event patterns, whose occurrence will lead to the execution of particular checking and adjustment operations. Among other things, the constraint designer may choose either a data-driven realization (i.e., where violating actions trigger corrections), or a demand-driven semantics (i.e., ‘dependent’ information is recomputed each time it is needed).

An example illustrating the layered approach is depicted in Fig. 11 showing a constraint stating that for all rooms ?X that have a corridor ?Y as a neighboring room, ?Y must have ?X as one of its accessible rooms (i.e., the corridor allows access to room ?X). One of the adjustments defined for this constraint states that for pairs of rooms and corridors violating the constraint, the room should be added as an attribute value of ‘accessible-rooms’ of the corridor. Essentially, KOALA is used (in a slightly modified form) for specifying constraint conditions and adjustments. The user may define constraints by specifying logical formulas with both existential and universal quantification. Moreover, user-defined methods can be evaluated in the query and constraint language environment, and can therefore be used for checking consistency and for carrying out reactive operations. At the realization level, a data-driven implementation was chosen, meaning that the adjustment will be performed on occurrence of an update event on the ‘neighboring-rooms’ attribute of a room. Alternatively, a demand-driven implementation would have been possible, which is also depicted in Fig. 11.

Along the specification of the constraint, the system shields the user as far as possible from (event-oriented) realization details by automatically determining constraint characteristics at the realization level. For our example constraint, only the condition and the adjustment (in the syntax presented in Fig. 11) were actually specified by the user. This support allows a high-level, implementation-independent constraint specification. Determining event-patterns describing in which situations an adjustment should or should not be applied is performed by the system based on the assumption that adjustments should always be conflict-avoiding. This means that an adjustment should, if possible, not contradict (or undo) user operations that caused the inconsistency. If this assumption does not hold, the person having defined the constraint may replace the event-patterns produced by the system, thereby specifying his own ‘implementation’.

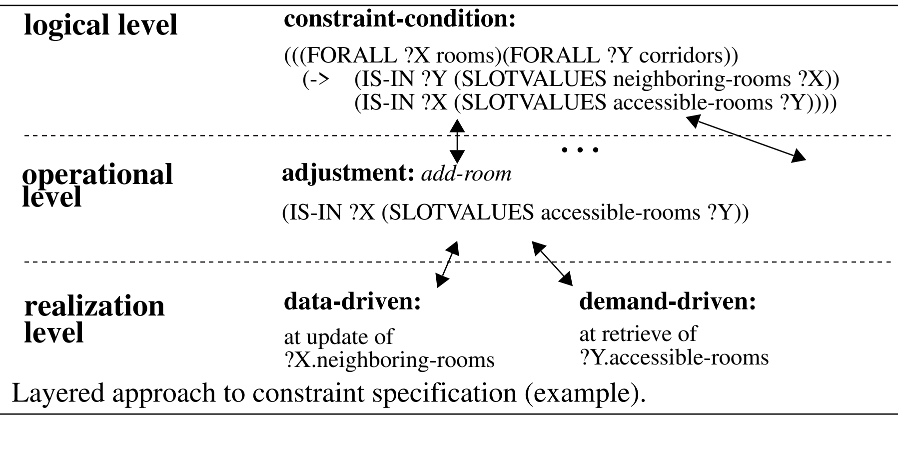


Fig. 11: Layered approach to constraint specification (example).

Additional aspects of the constraint mechanism can only be briefly listed here. We refer to [De93] for details.

- Methods in KRISYS are executed in a nested-transaction scheme. Both methods and integrity constraints can be associated with integrity levels describing certain degrees of partial consistency to be guaranteed by methods executing on that level.
- Constraint violations can be tolerated and defined as exceptions. This is especially important for long-running activities, such as design applications, where a rollback of work is not desirable.
- Constraints can be specified not only for classes, but also for individual object instances. This allows, for example, to represent design goals specific to a product under development as constraints.
- Method calls can be used in both conditions and adjustments, allowing to employ procedures for testing consistency and implementing corrections.
- Constraints are represented as objects in the KRISYS knowledge representation framework, allowing them to be organized using abstraction concepts and queried using KOALA.
- Constraint templates allow the definition of parameterized constraints, thereby permitting constraints to be tailored to application-specific needs.

Constraint Monitoring

Constraint monitoring, being essential for achieving effective and efficient integrity control for the applications KRISYS is intended for, is performed at the workstation side. Consequently, constraint monitoring activities can be realized in the knowledge representation framework of KRISYS, i.e., they are implemented in a natural way as the 'behavior' of the objects (such as constraint, adjustment, and integrity-level objects) used for constraint modeling. Moreover, the processing concepts, such as the processing of KOALA and the functionality of the Working Memory can be directly employed.

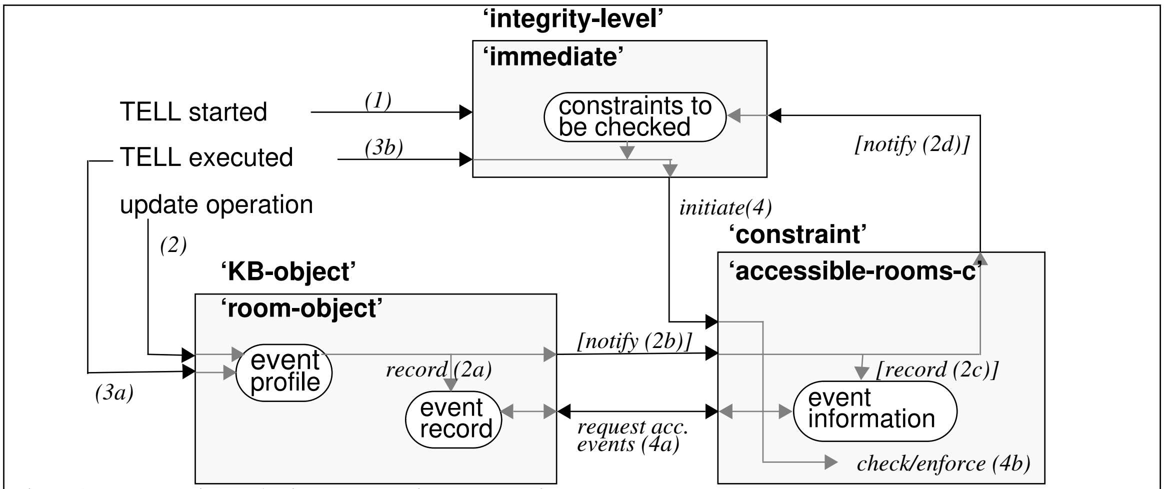


Fig. 12: Interactions during constraint processing.

As shown in Fig. 12, constraint-monitoring activities are distributed across three major components. *Event management* is integrated into the regular KB objects by

- keeping an event profile for each object that contains information about the events to be recorded and about the constraints that are affected, and
- storing an event record with the objects, thereby supporting local event accumulation.

These activities are initiated by base predicates responsible for updating an object in the Working Memory (see Sect. 4.2.1). After the initiation of a TELL is reported to the appropriate integrity-level object (Fig. 12, step 1), the execution of the base predicate (Fig. 12, step 2) modifying attribute values will cause the appropriate event description to be stored with each object added to class *rooms*. Moreover, the constraint object representing the constraint defined in Fig. 11 will be notified.

Constraint scheduling is performed by those integrity-level objects responsible for initiating the processing of individual constraints in an appropriate order. For example, the object representing the 'immediate' integrity-level will be directly activated after the TELL statement is executed (Fig. 12, step 3), but before the transaction associated with the statement is committed. It will then schedule constraint monitoring according to a priority scheme, if multiple constraints need to be activated, and in turn activate the constraint objects (Fig. 12, step 4).

Constraint enforcement is realized by the individual constraint objects. They have the task to check the constraint condition according to the event information supplied by the event management and take the appropriate actions to handle the occurring violations. In our example, the constraint object will try to use the single adjustment defined for it to correct the violations, and will be able to apply the adjustment because it does not contradict the operation performed by the user (i.e., it will not undo the effects of the TELL statement). The adjustment is again realized as a TELL statement that exploits the event information passed to the constraint object for adjusting only inconsistent objects. This TELL statement has been generated by the system by specializing the adjustment supplied by the user to specific event patterns.

5. Conclusions

In this paper we described the design and implementation of advanced knowledge processing in the KBMS KRISYS. The most important issues to be addressed by this framework are the workstation/server environment as well as its impact on overall knowledge processing resulting in main-memory-based query processing. The processing framework of KRISYS founds on the KOBRA knowledge model and benefits from well-known query-processing techniques, especially from the areas of relational, main-memory, object-oriented, and parallel database systems [HFLP89, IEEE92, Ca91, MPTW94]. The applicability of our approach as well as of the mechanisms necessary for implementing it are not restricted to KRISYS but are generally valid for (advanced) DBMS requiring client-based query processing. Therefore, we see our knowledge-processing framework and its implementation, i.e. KRISYS, as a valuable contribution to current research in advanced DBMS.

Our approach can be best characterized by its major components

- Working Memory
  Its task is to support the concept of near-by-the-application locality of processing when KOBRA objects are referenced during query and constraint processing. To this end, the Context Manager guarantees that the Working-Memory contents is exploited for query processing, thus reducing data transfer between workstation and server to a minimum.
- KOALA Processing System
  The query language KOALA is processed following an algebraic approach that is sufficiently flexible to adapt to language extensions. Along the same lines, the plan-operator concept for client-based query processing has been designed to be extensible and to allow run-time optimizations.
- Constraint Manager
  A new mechanism for integrity management has been outlined and integrated into the knowledge-processing framework of KRISYS.

Although object-oriented DBMS aim at the same application domains as KRISYS, to the best of our knowledge, there are no such systems that offer comparable concepts for optimizing and processing arbitrary queries on the workstation's buffer. ObjectStore [OHMS92], for example, provides simple search arguments (path expressions) for navigating the buffer. Selecting appropriate indices handling simple search arguments and execution are interleaved. This optimization measure differs from our approach of employing run-time optimization before execution.

Run-time optimizations may be motivated either by the desire to flexibly adjust execution strategies, as pursued by Volcano [Gr94], or by the desire to dynamically exploit buffer contents, as proposed for ADMS [CR94]. ADMS integrates matching and query optimization. The query graph is reduced by those parts that match cached query results stored in a cache space and organized by a specific data structure called *logical access-path schema*. In our case, representations of buffered and intermediate (cached) data coincide. Hence, matching and optimization operate on a single representation, thus simplifying query-processing implementation.

Re-implementation of KRISYS started with the Working-Memory representation, Access Structures, and their functionality. This provided the basis for the KOBRA model and its internal interface which have been fully operational since 1993 as well. At the same time, the Mapping System had been completed so that we could start implementing the KOALA Processing System. The transformation of a KOALA query into an algebraic representation and the subsequent rewrite are already realized, as well as all constituents of the plan-operator level, including blocks and the corresponding functionality. Currently, the KOALA Processing System allows to sequentially execute block-structured queries.

The availability of query-processing facilities opens up a range of research activities we are currently working on or which will be part of our future work:

- implementing the Context Manager to practically investigate the interplay between the knowledge referenced by queries and the costs and benefits of context maintenance,
- realizing the Constraint Manager starting from the basic functionality linking query processing and constraint management, i.e., event management, constraint scheduling, and constraint enforcement,
- considering non-algebraic optimization, i.e., establishing a cost model for query processing taking into account features of the knowledge model (method calls, transitive closure operations like inheritance, etc.), context management and mapping information.

References

Ca91 Cattell, R. (ed.): Next Generation Database Systems, in: Special issue of Communications of the ACM, Vol. 34, No.10, 1991.

CR94 Chen, C.M., Roussopoulos, N.: The Implementation and Performance Evaluation of the ADMS Query Optimizer: Integrating Query Result Caching and Matching, in: Advances in Database Technology - EDBT ‘94, Jarke, M., Bubenko, J. (eds.), Lecture Notes in Computer Science 779, Springer-Verlag, 1994, 323-336.

De91 Deßloch, S.: Handling Integrity in a KBMS Architecture for Workstation/Server Environments, in: Proc. of the GI-Fachtagung "Datenbanksysteme in Büro, Technik und Wissenschaft", Kaiserslautern, März 1991, Hrsg. H.-J. Appelrath, Informatik-Fachberichte 270, Springer-Verlag, S.89-108.

De93 Deßloch, S.: Semantic Integrity in Advanced Database Management Systems, Doctoral Thesis, Dept. of Computer Science, University of Kaiserslautern, Sept. 1993.

DLMT93 Deßloch, S., Leick, F.J., Mattos, N., Thomas, J.: The KRISYS Project - A Summary of What We have Learned so far, in: Stucky, W., Oberweis, A. (eds.): Datenbanksysteme in Büro, Technik und Wissenschaft,, Springer (Informatik Aktuell), 1993, 124-143.

Gr94 Graefe, G.: Volcano, an Extensible and Parallel Query Evaluation System, in: IEEE Transactions on Knowledge and Data Engineering, Vol.6, No.1, 1994, pp. 120-135.

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

HMMS87 Härder, T., Meyer-Wegener, K., Mitschang, B., Sikeler, A.: PRIMA - A DBMS Prototype Supporting Engineering Applications, in: Proc. of the 13th Int. VLDB Conf., Brighton, UK, 1987, pp. 433-442.

HS93 Hong, W., Stonebraker, M.: Optimization of Parallel Query Execution Plans in XPRS, Distributed and Parallel Databases, Vol. 1, 1993, 9-32.

IEEE92 Eich, M. (ed.): IEEE Transactions on Knowledge and Data Engineering, Special Issue on Main-Memory Databases, Vol. 4, No. 6, 1992.

In84 IntelliCorp Inc.: The Knowledge Engineering Environment, IntelliCorp, Menlo Park, CA, 1984.

ISO94 ISO/IEC JTC1/SC21/WG3: ISO/ANSI working draft Database Languages - SQL3, American National Standards Institute, 1430 Broadway, New York, NY 10018, Sept. 1994.

KL89 Kifer, M., Lausen, G.: F-Logic, a Higher-Order Language for Reasoning about Objects, Inheritance and Schema, Proc. of the ACM SIGMOD Int. Conf. on Management of Data, 1989, pp. 134-146.

Kr93 Krivokapic, N.: Schema Evolution in KRISYS (in German), Diploma Thesis, University Kaiserslautern, 1993.

La91 Langkafel, D.:A Component for Graph-oriented Management of Knowledge-base Excerpts (in German), Undergraduation Final Work, Dept. of Computer Science, University Kaiserslautern, 1991.

LLOW91 Lamb, C., Landis, G., Orenstein, J., Weinreb, D.: The ObjectStore Database System, in: Communications of the ACM, special issue on next-generation database systems, vol. 34, no. 10, 1991, pp. 50-63.

LLPS91 Lohman, G. Lindsay, B., Pirahesh, H., Schiefer, B.: Extensions to Starburst: Objects, Types, Functions, and Rules, in: CACM, Vol. 34, No. 10, 1991, pp. 94-109.

LVZZ94 Lanzelotte, R. Valduriez, P., Zait, M., Ziane, M.: Industrial-Strength Parallel Query Optimization: Issues and Lessons, in: Information Systems, Vol. 19, No. 4, 1994, pp.311-330.

Ma91 Mattos, N.: An Approach to Knowledge Base Management, in: LNCS 513, Springer-Verlag, 1991.

MPTW94 Mohan, C., Pirahesh, H., Tang, W., Wang, Y.: Parallelism in relational database management systems, in: IBM System Journal, Vol.33, No. 2, 1994, pp. 349-371.

ODMG93 Cattell, R. (ed.): The Object Database Standard: ODMG-93, Morgan Kaufmann, CA, 1993.

OHMS92 Orenstein, J., Haradhvala, S., Margulies, B., Sakahara, D.: Query Processing in the ObjectStore Database System, Proc. of the 1992 ACM SIGMOD Conference, 403-412.

Sch91 Schulte, D.: Flexible mapping of Knowledge Models to Data Models exemplified using the knowledge Model KOBRA and the Relational Model (in German), Diploma Thesis, University Kaiserslautern, 1991.

Su91 Surjanto, B.: Design and Implementation of a Knowledge-based System Generating Application-specific KB schemata for KRISYS (in German), Diploma Thesis, University Kaiserslautern, 1991.

TD93 Thomas, J., Deßloch, S.: A Plan-Operator Concept for Client-Based Knowledge Processing, Proc. 19th VLDB Conference, Dublin, Ireland, August 1993.

TGHM94 Thomas,J., Gerbes, T., Härder, T., Mitschang, B.: Implementing Dynamic Code Assembly for Client-Based Query Processing, submitted for publication.

TMMD93 Thomas, J., Mitschang, B., Mattos, N., Deßloch, S.: Enhancing Knowledge Processing in Client/Server Environments, Proc. 2nd Int. Conf. on Information and Knowledge Management, Washington, D.C., 1993, 324-334.