View-Concepts: Knowledge-Based Access to Databases[*]

Jon A. Pastor
(215) 648-2769
pastor@vfl.paramax.com

Donald P. McKay
(215) 648-2256
mckay@vfl.paramax.com

Unisys Government Systems Group
Valley Forge Engineering Center
Research and Development
PO Box 517
Paoli, PA 19301

Abstract

Semantic data models for database systems provide powerful tools to assist database administrators in designing and maintaining schemas, but provide little or no direct support for users of the database. Some research has been done on mapping user models of a domain to the underlying database using semantic schemas. Little has been done, however, on mapping conceptually meaningful data structures to a database lacking a semantic schema, or to a multi-database system that lacks a consistent semantic schema. We argue for the appropriateness of a knowledge representation language for describing the database schema, user data structures, and the mapping between them; present a problem domain in which an existing relational database without a semantic schema must be accessed by a knowledge-based application; and describe our implementation of a system that provides access to a relational database from a KL/ONE-style knowledge representation language. With this background, we highlight recently-added capabilities of the implementation, and provide detailed examples.

1. Introduction

The integration of Artificial Intelligence (*AI*) and database management system (*DBMS*) technologies promises to play a significant role in shaping the future of computing. As noted in [[7]], AI/DB integration is crucial not only for next-generation computing, but also for the continued development of DBMS technology and, in many cases, for the effective application of AI technology. The motivations driving the integration of these two technologies include the need for

Ÿ access to large amounts of shared data for knowledge processing,
Ÿ efficient management of knowledge as well as data, and
Ÿ intelligent processing of data.

In addition, AI/DB integration at Unisys was motivated by the desire to preserve the substantial investment in most existing, or *legacy*, databases. To that end, a key design criterion was that our integration technology support the use of existing DBMSs as independent system components. Finally, the development of large-scale, heterogeneous, distributed AI systems involving a number of discrete cooperating agents, as well as a general need to retain results of AI processes for the long-term, have led to a need for *persistent storage* of knowledge to permit *knowledge sharing* among those agents.

1.1 Technical Approach

We distinguish four approaches to the integration of AI and DBMS technologies:

Ÿ extended AI system,
Ÿ extended DBMS system,
Ÿ loosely coupled AI/DB interface, and
Ÿ enhanced AI/DB interface.

*This work is supported by Rome Laboratory and the Advanced Research Projects Agency under USAF contract F30602-91-C-0040. The views and conclusions are the authors' and should not be interpreted as the official opinion or conclusions of the US Government, the USAF, Rome Laboratory, or ARPA.

Our previous work on the Cache-based Intelligent Database Interface (*CIDI*) [[11]] focused on an enhanced AI/DB interface for logic-based systems; the work described in this paper builds on the CIDI by defining a *View-Concept* model demonstrated using the Loom Interface Module (*LIM*).

Some work has been done toward implementing a *View-Object* model [[15]] on relational DBs for which a semantic schema exists. In particular, Barsalou and Wiederhold [[3],[4]] describe a system based on the *Structural Model* [[14]], an extended entity-relationship model. In this system, the user selects a *pivot* relation that includes the intended key(s) for the object being defined. The various link-types in a *structural schema* are then traversed, and, using a relevancy metric, a tree of candidate relations—rooted at the pivot—is generated. The user then prunes this tree, leaving only the relations and attributes that s/he wishes to have in the defined object. Once the object is defined, it is linked into an object hierarchy. Barsalou and Wiederhold provide algorithms that assure that objects are retrievable, and, if desired, updatable. The *View-Concept* model described here is intended as a knowledge-based extension of the View-Object model; it also draws upon work done at Unisys on an Ingres DB interface for the CYC KRS [[12]].

2. Architecture

LIM acts as an intermediary between a Loom application and one or more DBs, using the services of the CIDI to access the DBs. The inter-relationships among the various components of the overall system are illustrated in Figure 1. LIM uses the CIDI to read the DB schema, building a Loom representation of the schema based on this information. Subsequently, in response to a query or update request from a Loom application that requires access to the DB, LIM parses the request and uses the CIDI to generate the appropriate data manipulation language (*DML*) statements for the DBMS; in the case of a query, it then processes the tuples returned to it by the CIDI into the form requested by the application.

Processing within LIM is directed by a multi-layer KB architecture that is built in a mixed-initiative process. Figure 2 depicts the layers in this architecture.

2.1 Semantic Mapping KB (SMKB)

The Semantic Mapping KB (*SMKB*) is an isomorphic representation of the DB schema. It is constructed by a Knowledge Base Administrator (*KBA*)—analogous to the database administrator for a DB—from an automatically-generated schema model. LIM creates this schema model by using the CIDI to read the DB schema; based on this information, it defines one Loom concept for each table and one Loom relation for each column. The relation representing each column is defined as a role on the concept representing the appropriate table, value-restricted to a type that represents the DB type (domain) of the column; these DB types are typically simple types like `integer` and `string`. The SMKB is created primarily by defining *semantic* types, and then substituting these for the simple DB types that appear in the schema model. LIM stores information identifying the DB, table, and column with the concepts and relations in the SMKB.

2.2 Application KB (AKB)

Application KBs (*AKB*s) refer to concepts and relations in the SMKB.

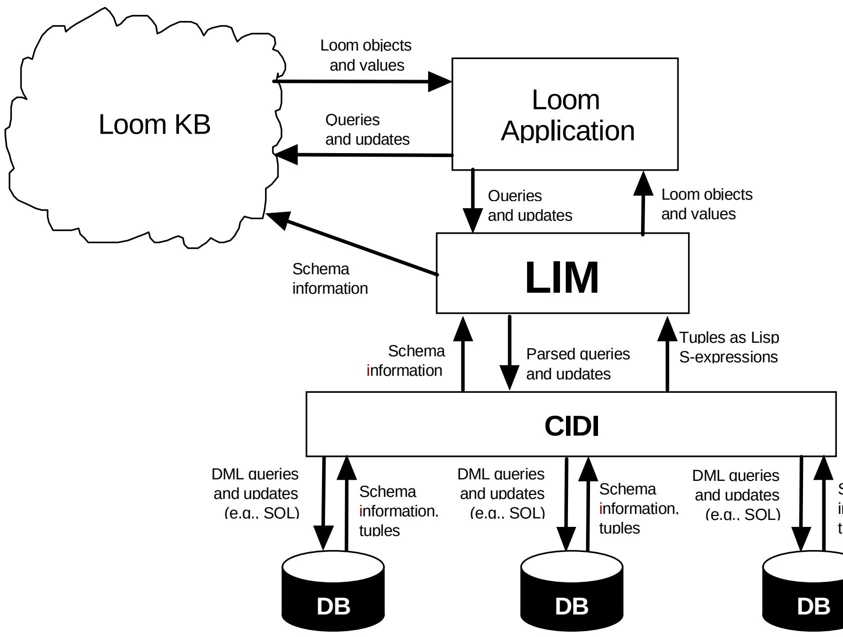
Loom KB
Loom objects and values
Queries and updates
Loom Application
Queries and updates
Loom objects and values
Schema information
LIM
Schema information
Parsed queries and updates
Tuples as Lisp S-expressions
CIDI
DML queries and updates (e.g., SQL)
Schema information, tuples
DML queries and updates (e.g., SQL)
Schema information, tuples
DML queries and updates (e.g., SQL)
Schema information, tuples
DB
DB
DB

Figure 1: LIM Overview

Unlike concepts in the SMKB, which are isomorphic to tables in the DB, and which presently have no hierarchical structure, concepts in the AKB do not necessarily map in any simple way to the tables in the DB, and can have arbitrary hierarchical structure. Connections to the DB are implemented via *DB-mapping* declarations, in which a concept-role pair in the AKB is mapped to a SMKB role.

3. Operation

LIM, given a query or update request involving a concept in the SMKB or AKB,

- obtains schema mapping information from the SMKB;
- translates the request into an equivalent DML statement with the aid of the CIDI, which submits the statement to the DBMS and assembles the result; and
- Ÿ for a query, restructures the returned tuples as necessary, generating any KB structures required to satisfy the query.

With regard to the last point, a fundamental principle of LIM is that KB structures are created only on demand: queries are satisfied without creation of KB objects whenever possible, to minimize overhead and bookkeeping. Control over object creation is entirely at the discretion of the application.

The processing performed by LIM is illustrated in Figure 3.

3.1 Schema Generation

As described above, when a DB is opened via the CIDI, schema information is cached by the CIDI in local data structures. The schema generation module reads this information, and generates one Loom concept per table and one Loom relation per column. The relations corresponding to the columns of a table are then added as roles of the corresponding concept via value-restrictions.

3.2 Schema Augmentation

The automatically-generated schema model is a literal representation of the DB schema. One implication of this is that the semantics of relationships among tables are not explicit, since the schema does not identify the columns in the DB over which joins are semantically reasonable. In creating the SMKB, the KBA augments this literal schema representation by defining semantic types to explicate the semantics of joins. In particular, where two relations represent columns over which a join is semantically reasonable, the value restrictions on these relations in their respective concepts are changed to the same KB type. For example, two DB columns whose DB type is `integer`, but which both represent a particular kind of identification number, would have their value restrictions specialized to a concept representing that kind of identification number.

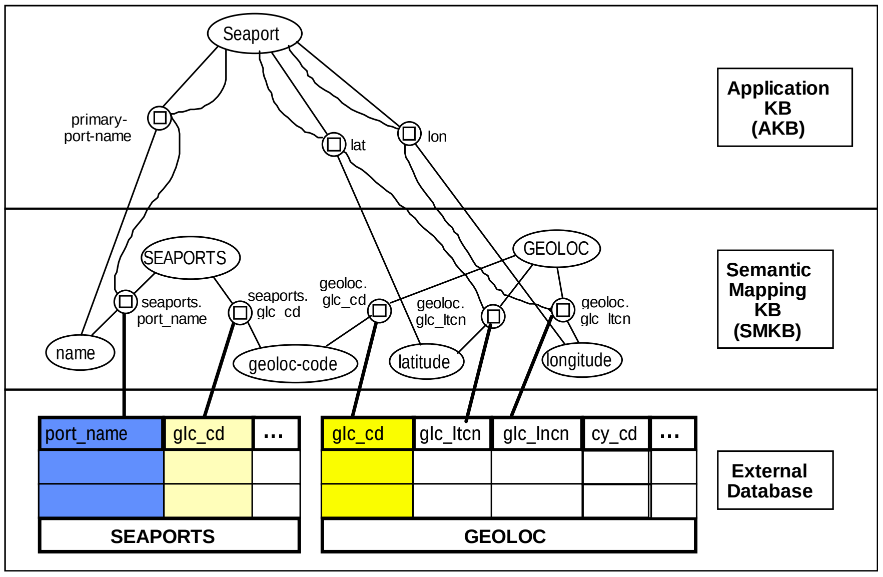

Figure 2: LIM Knowledge Base Architecture

In addition to modification of role value restrictions, it is usually desirable to represent the structural semantics of the domain more closely than is possible in the relational model. Such restructuring may be specified in the AKB by defining View-Concepts and mapping their roles to those of SMKB concepts. When such semantic restructuring takes place, it is necessary to ensure that the proposed structures are retrievable, and—if desired—updatable. *Retrievability* requires that all participating tables can be joined, and that sufficient information (e.g., keys) is preserved in the semantic representation to permit unambiguous access of all necessary tuples. *Updatability* requires that all key, index, and non-null columns in the DB tables underlying a View-Concept in the AKB are included in the View-Concept.

Retrievability of a View-Concept is assured via a mixed-initiative dialog, in which the system computes all semantically meaningful ways of joining all of the DB tables required for the construction of a View-Concept; if there is more than one such alternative, LIM presents them for selection by the user. It is not possible for the system to compute join paths without user intervention, since any given pair of tables might be joinable in several ways, not all of which are semantically equivalent.

Updatability of View-Concepts, when required, is checked automatically by the system. If any necessary information is unavailable in the view, the system identifies the missing information, and suggests that it be added to the model.

3.3 Query Translation

Given a LIM query, the query translation module:

Ÿ identifies variables in the query corresponding to Loom relations that are derived from the DB,
Ÿ identifies variables in the query corresponding to Loom concepts having roles derived from the DB, and
Ÿ constructs a DML query and submits it to the CIDI for processing against the DB.

If the query requests the return of Loom objects, rather than just values from the DB, the DML query will select and return values in each tuple to permit generation of the appropriate Loom objects.

3.4 Update

The design of a DB update facility for Loom is not as straightforward as might appear at first. In this section, we will address the semantic issues involved in update; discussion of additional technical issues will be deferred to Section 4.3.

Loom instances can be constructed incrementally, by asserting the existence of the instance, and then subsequently asserting facts about it. Furthermore, classification of an instance does not take place as a result of asserting its existence, or asserting facts about it, but must be explicitly requested. In addition to the incremental nature of instance creation, certain semantic requirements must be taken into consideration: as noted in Section 3.2, if an instance is to be stored into a DB, an update cannot be performed unless sufficient information (e.g., values for keys and indices) is available.

While it would be possible to determine when a user considered a given instance's definition complete (i.e., when classification is requested), or to determine the point at which sufficient information has been asserted about an instance to permit storage in the DB, we saw no justification for presuming that either of these conditions should necessarily imply that the user intended to make the instance in question persistent. We therefore chose to separate instance creation from a request for storage in the DB.

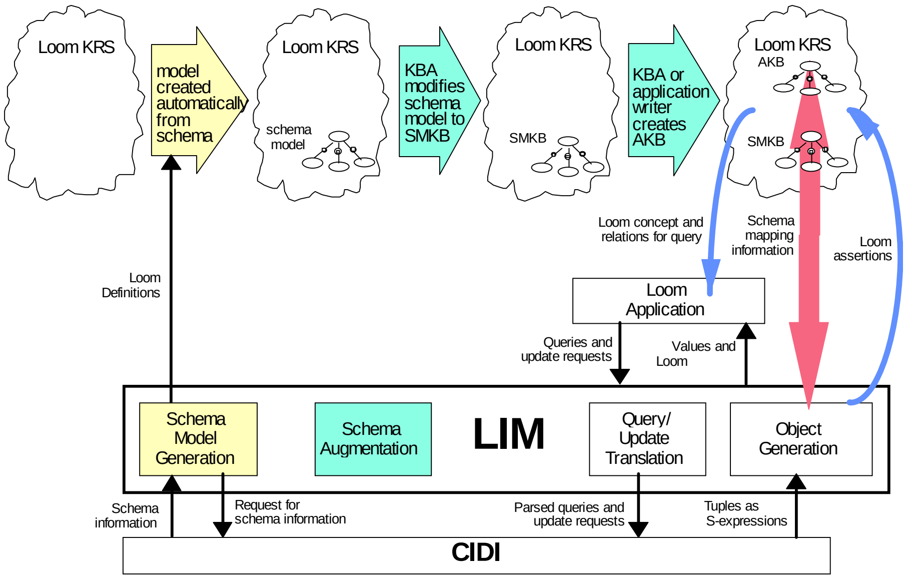
Loom KRS
model created automatically from schema
Loom KRS
schema model
KBA modifies schema model to SMKB
Loom KRS
SMKB
KBA or application writer creates AKB
Loom KRS
AKB
SMKB
Loom concept and relations for query
Schema mapping information
Loom assertions
Loom Definitions
Loom Application
Queries and update requests
Values and Loom
Schema Model Generation
Schema Augmentation
LIM
Query/ Update Translation
Object Generation
Schema information
Request for schema information
Parsed queries and update requests
Tuples as S-expressions
CIDI

Figure 3: LIM Internal Architecture and Processing

When a user issues a request to store a DB-derived Loom instance, LIM verifies that it has values for all roles mapped to key- and index-columns for underlying tables, as well as for all columns for which NULL values are not permitted. If the instance contains all requisite information, the DB is modified via the CIDI. Note that a request to store an instance of a View-Concept that draws information from more than one table may result in DB operations against all such tables. Note also that the it must be decided, for each such table, whether to modify (*update*) an existing tuple, or to create (*insert*) a new tuple. Both of these issues will be discussed in Section 4.3.

3.5 Object Generation

A LIM query consists of a list of output variables to be bound, and one or more statements (in a syntax similar to that of the Loom assertional language) that produce sets of bindings for these variables. It is easily determined from the positions of variables in the output list and the query expressions whether a particular output variable corresponds to a role value or a concept. For a variable corresponding to a role value, the value retrieved from the DB can be returned to the application, possibly with some conversion due to the differences between semantic types used in the KB and simple DB types (cf. Section 4.2). For a variable corresponding to a concept, however, the application will expect to have returned to it an instance of that concept; this requires that LIM be capable of creating Loom instances using values retrieved from the DB. LIM's object generation module extracts from the returned tuples all values requested specifically for the purpose of building Loom objects, creates the objects, and returns them to the application.

4. Technical Details

The descriptions in Sections 2 and 3 glossed over several critical technical issues in the design, implementation, and operation of LIM. Among these are caching, data inference, complexities in update, and Meta-View-Concepts.

4.1 Caching

LIM uses three different caching schemes, for two purposes. The first purpose is the conventional one of improving performance; the second is related to preserving referential integrity in an Object-Oriented system. For improved performance, LIM can make use of the CIDI's *results cache*: the CIDI stores the result of every query it processes, indexed under a canonicalized version of the CIDI's internal representation of the query; if a subsequent query is translated to the same canonical form, *modulo* variable names, the result from the earlier query is returned. In addition, LIM employs a similar cache based on a canonicalization of the original user query: the results of each LIM query are stored, indexed under this canonical form, and a subsequent query that is identical, *modulo* variable names, will return the cached result.

Both of these can result in dramatic performance increases, because DB execution time dominates total LIM processing time for all but the largest of queries (as detailed in Section 5). A "hit" on the LIM results cache returns results almost intantaneously, since the query need not even be translated and submitted to the CIDI, and a hit on the CIDI results cache incurs only LIM translation time; in neither case will the DBMS be involved.

A query will hit the LIM cache only if it is virtually identical to a previous query, *modulo* variable names; on the other hand, a query might hit the CIDI cache even if it is superficially quite dissimilar to an earlier query, provided that both translate to an identical CIDI query. The extreme case of this is a query for an instance following a query for all of the instance's values: both will translate to the same CIDI query, even though they look quite dissimilar at the surface syntactic level.

Use of both LIM and CIDI caching is controllable both globally and at the individual query level. This is important in applications where the contents of the DB are known to be dynamic, and it cannot necessarily be presumed that a subsequent query *should* return the same results as an earlier identical query. We intend to augment the current caching schemes with "smart" caching techniquies, such as subsumption-based caching, in future versions of LIM.

The caching used to assure referential integrity uses a similar mechanism, but for a completely different purpose. When a user queries LIM for an instance of a View-Concept, and then subsequently queries for an instance with the same key values, it is usually the case that s/he expects the *same* KB object to be returned in both cases. Since both results caches presently used by LIM are based purely on syntax, they are incapable of recognizing syntactically-distinct queries that should (semantically) return the same object.

A "smart" cache will alleviate this problem to a large degree, but it is still necessary to assure that syntactically-distinct queries referencing the same instance return the same object. For this reason, LIM checks the Loom instance database (ABox) prior to creating instances. Given an object query, after submitting a query to the CIDI and receiving return values, LIM queries the Loom ABox before creating a new instance. If the View-Concept that is to be the type for the instance has keys defined, LIM uses these (in conjunction with Loom's indexing capabilities) to speed the seach; otherwise, all values are used.

Note that "ABox cache" checking is ***not*** an efficiency measure: on the contrary, it carries a performance penalty that can become significant on extrememly large queries (many hundred to several thousand objects). For this reason, and because of situations (like that of dynamic DB contents) where ABox cache checking is undesirable, it is controllable both globally and at the individual query level.

4.2 Data Inference

Data values in a DB not are necessarily in the form required or desired for the KB. LIM presently supports *data value rendering*, which maps from simple data values in the DB to scalar KB types; e.g.,

```
"M" ⇒ |C|Military
```

Future releases of LIM will support *data object rendering*, which maps from simple data values in the DB to instances of concepts; e.g.,

```
"BSRL" ⇒ |i|Bizerte(Geoloc)
```

where `"BSRL"` is the `GEOLOC_CODE` for the `GEOLOC` Bizerte.

Finally, LIM will support *data classification*, which results in classification of instances in the KB after creation; e.g.,

```
> (db-retrieve ?ship
     (:and (Ship ?ship)
           (name ?s "Constitution")))

|i|Constitution(Frigate)
```

Here, it has been determined that the instance named `"Constitution"` is an instance of the concept `Frigate`, even though the query specified the more general concept `Ship`.

4.3 Update Issues

Update of simple View-Concepts that are mapped to single DB tables is relatively straightforward and unambiguous: if the View-Concept has roles mapped to all requisite columns of the underlying table (i.e., key-, index-, and no-nulls-columns), and has values for all roles so mapped, tuples underlying it can be updated or inserted. The principal subtlety in this case is determining whether an insert or update is appropriate. Our approach to making this determination is to keep track, for each instance retrieved from the DB, of the values actually retrieved. When an object is stored, LIM checks to see whether it was originally retrieved from the DB, and, if so, checks the current values against those recorded originally. If the object was not retrieved from the DB, the request to store it will be treated as an insert, and will succeed if there is no matching record in the DB. If the object was retrieved, and its key values have changed, the store request will be treated as an insert; otherwiseit will be treated as an update.

When a View-Concept involves one or more joins, or has sub-View-Concepts (i.e., roles that are value-restricted to a View-Concept), the situation becomes more complex. In either case, the View-Concept is only considered updatable if all information required for the joins is stored in the View-Concept(s) involved; the complexity lies in the fact that LIM has no control over which of the two (or more) columns involved in the join(s) are mapped to the View-Concept, and it is typically the case that the values must be propagated to all of the relevant tables, even to columns that are not DB-mapped by the View-Concept. In such cases, LIM must use information about joins and subview-joins to propagate values from DB-mapped roles to both the SMKB roles to which they are mapped *and* those representing the columns across which joins are required.

The other subtlety introduced with joins is the fact that not all component tables will necessarily require either update or insert operations for all store requests. For example, short of the creation of a new seaport or airport, it is extremely unlikely that any new `GEOLOC`s (the DB table used to represent geographic locations of all kinds in the DB) would be created by any store request, and yet many application concepts in our domain contain information drawn from (i.e., have roles DB-mapped to) the `GEOLOC` table; it will seldom, if ever, be the case that a request to store an instance of one of these concepts should result in an update or insert to the `GEOLOC` table. Other, even more extreme, examples of this include tables like the one for `COUNTRY-CODES`, which maps arbitrary country codes to country names: it will seldom be the case that a routine update will want to add to, or modify, such tables.

4.4 Meta-View-Concepts

In addition to data about individual real-world objects like seaports and ships, some DBs (including those used by the ARPI) contain data about *classes* of real-world objects. Examples of this include data about the dozens of types of ships that are recognized. The need to represent this information in a domain model should be obvious; it should also be obvious that it is inappropriate, and potentially dangerous from the standpoint of data integrity, to permit this data to be manually encoded in the domain model. Furthermore, it is highly desirable, if not absolutely critical, that information about classes of objects be accessible from the context of the objects themselves.

For this purpose, LIM supports a Meta-View-Concept facility. In Object-Oriented systems, the term *metaclass* is typically used to describe classes whose instances are all classes: just as a class describes the structure (i.e., slots or roles) of objects of that class, so a metaclass describes the structure of a class. In the case of ships, for example, a ship meta-class would describe the structure of something that represents a collection of similar ships—i.e., a class of ships—while a ship class would describe the attributes of something that represented an individual ship. While it is possible, and in many cases reasonable, for a metaclass and its classes to share attributes, it is typical for each to have attributes that are not meaningful for the other. For example, while it is reasonable that both ship classes and ships share some common attributes such as length and draft, it is meaningless for ship classes—i.e., instances of the ship metaclass—to have a hull-number attribute; and, while it is possible to represent the number of ships in its class in the object representing each individual ship—i.e., as an attribute of individual ships, defined in ship classes—it is clearly more appropriate to model this as an attribute of ship classes, defined at the ship metaclass.

LIM's Meta-View-Concept facility provides the ability to define meta-level View-Concepts, and to retrieve data for instances of these—i.e., View-Concepts—from the database. In addition to supporting queries about class-level concepts— say, for the number of ships in a particular class—meta-View-Concepts provide the basis for a mechanism implementing any of several varieties of defaults. For example, if an instance is missing a value for a particular attribute, it may be quite reasonable to obtain a proxy value by default from the instance's class; in the case of a particular type of ship, if the `length` is missing from one instance, for example, it may be reasonable to use the `length` value from the ship's class in its place. It may also be reasonable to "inherit" class-level values, such as the number of ships in a class, at the instances.

5. Performance

Early versions of LIM performed quite reasonably with value queries and simple object queries; LIM overhead — execution time attributable to LIM—was less than 20 percent of total execution time. However, performance degraded significantly as the number of objects returned increased. For example, retrieving one object with approximately 115 sub-objects (i.e., role fillers whose type is a View-Concept) took over one minute (see Figure 5); furthermore, the ABox cache check (cf. Section 4.1) took over half of the total time for the query. Analysis of these results led us to make radical changes in the ABox cache checking algorithms, as well as the LIM translation logic. As a result, worst-case performance was improved by more than an order of magnitude, while variation in execution times was reduced similarly.

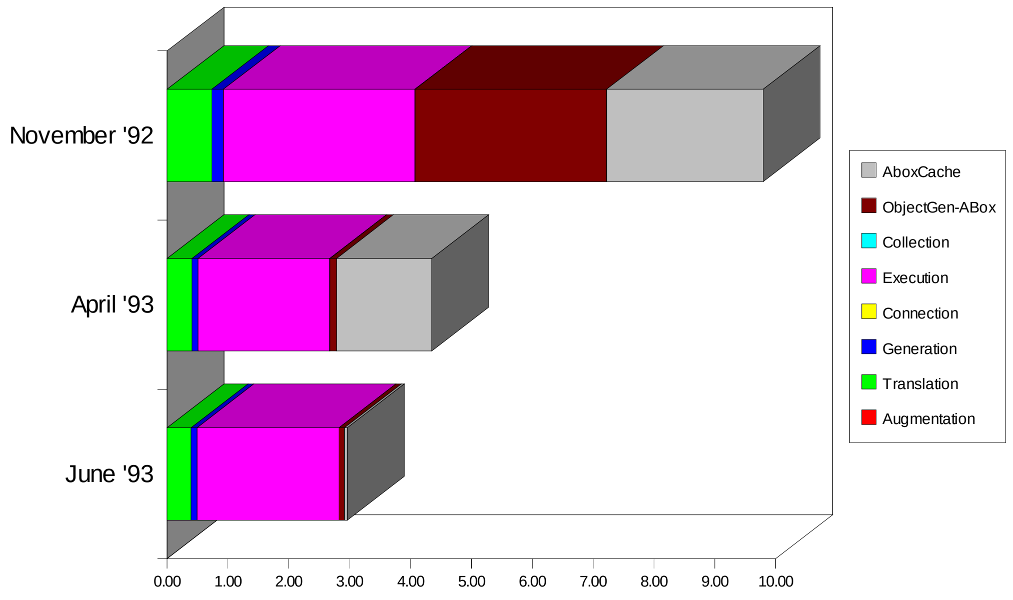

Figure 4:Distribution of Execution Times

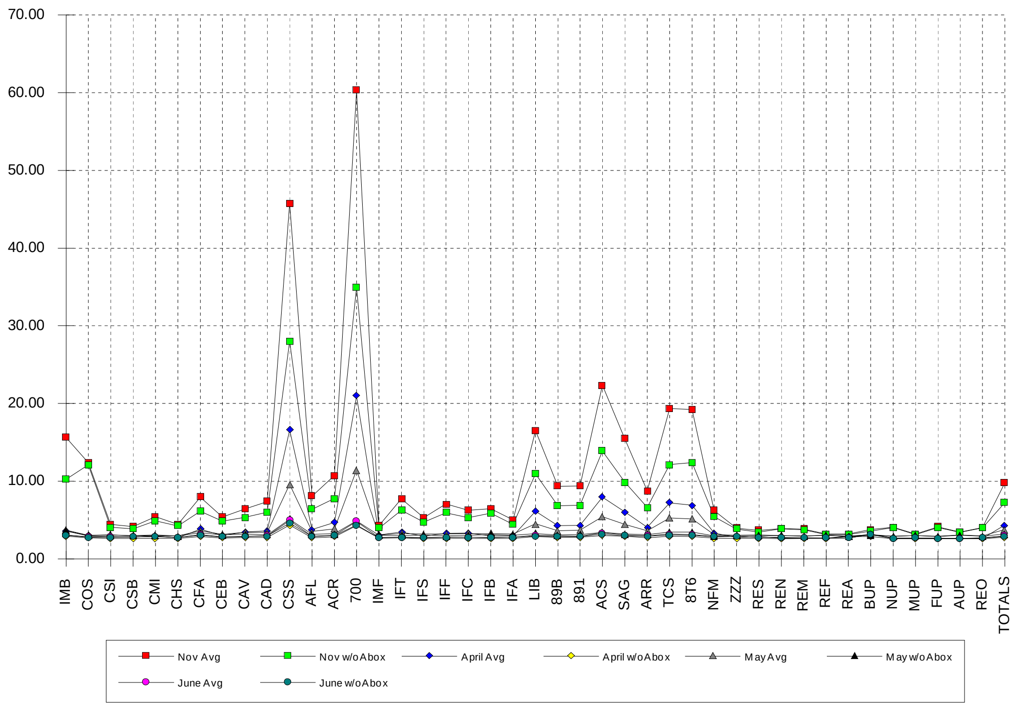

Figure 5: Performance Improvements

The reductions in the various components of total query execution time can be seen in Figure 4. On the average of all queries in our test suite, the time taken to query for and create Loom instances (“ObjectGen-Abox” and “AboxCache”, respectively) was dramatically reduced while times for translation of the source query (“Translation”) and generation of DML ("Generation") were reduced significantly. We do not regard the reduction in database execution time ("Execution") as significant, since this is highly variable and dependent almost entirely on factors beyond LIM's control.

6. Example

To illustrate the processes described in Section 3, we present a small, fairly simple example. Let us presume that an application requires information about the location of various seaports. In the USTRANSCOM databases with which we are working, information about seaports is stored in a table called `SEAPORTS`, and information about geographic locations in a table called `GEOLOC`. The various KB layers representing the mapping from application to DB are shown in Figure 2. Note that from a user's perspective, the SMKB and DB pre-exist, and definition of application concepts thus appears to be a top-down process; however, in order to illustrate the process of defining the mappings, we will proceed bottom-up.

The bottom panel shows a simplified tabular representation of the schema definitions for the two tables, `SEAPORTS` and `GEOLOC`. The middle panel shows the SMKB concepts representing the two tables. These were created by modifying the value-restrictions in the Loom definitions automatically generated by the schema generation module. For example, the initial Loom concept definition for the portion of the `GEOLOC` table shown is:

```
(defconcept Geoloc
  :is-primitive
     (:and db-concept
        (:the Geoloc.glc_cd String)
        (:the Geoloc.glc_lncn Number)
        (:the Geoloc.glc_ltcn Number)))
```

Note that role value restrictions correspond to the simple data types (e.g., string, number) that appear in relational databases. This definition is modified by the KBA to produce the SMKB definition:

```
(defconcept Geoloc
  :is-primitive
     (:and db-concept
        (:the Geoloc.glc_cd Geoloc_Code)
        (:the Geoloc.glc_lncn Longitude)
        (:the Geoloc.glc_ltcn Latitude)))
```

For example, the role of `Geoloc` that corresponds to the column `glc_cd` has type `String`; this has been modified in the SMKB to `Geoloc_Code`. This permits LIM to infer that `Seaports` and `Geoloc` can be joined over their `glc_cd` roles.

Loom definitions for the semantic type hierarchies above the types used in `Geoloc` are:

```
(Defconcept Identifier :Is-Primitive Thing)
(Defconcept Code :Is-Primitive (:and String Identifier))
(Defconcept Location :Is-Primitive Code)
(Defconcept Geoloc_Code :Is-Primitive Location)
(Defconcept Measured_Qty :Is-Primitive Number)
(Defconcept Degrees :Is-Primitive Measured_Qty)
(Defconcept Latitude :Is-Primitive Degrees)
(Defconcept Longitude :Is-Primitive Degrees)
```

Finally, the top panel shows a simple application-level concept derived from information in both DB tables. The following is the Loom concept definition for the AKB concept `seaport`, which was created manually:

```
(defconcept Seaport
  :is-primitive
     (:and View-Concept
        (:the primary-port-name String)
        (:the lat Latitude)
        (:the lon Longitude)))
```

This is mapped to the DB by making the following declarations, which are stored as assertions in the Loom KB:

```
(def-db-mapping primary-port-name seaport seaports.port_name)
(def-db-mapping lat Seaport geoloc.glc_ltcn)
(def-db-mapping lon Seaport geoloc.glc_lncn)
```

Queries can be posed against either the SMKB or the AKB. (Note: the names used in the following examples have been changed; we have not yet obtained permission to publish the data in our test database.) For example, the query:

```
(db-retrieve (?name)
  (:and
     (Seaports ?port)
     (Geoloc ?geoloc)
     (Seaports.Glc_cd ?port ?geocode)
     (Geoloc.Port_Code ?geoloc ?geocode)
     (Seaports.port_name ?port ?name)
     (Geoloc.Country_State_Code ?geoloc "DP")
     (Seaports.Clearance_Rail_Flag ?port "Y")))
```

("What are the names of seaports in Dogpatch that have railroad capabilities at the port?") can be posed against the SMKB. The SQL generated by LIM and the CIDI for this query is:

```
SELECT DISTINCT RV1.name
FROM SEAPORTS RV1, GEOLOC RV2
WHERE RV2.glc_cd = RV1.glc_cd
  AND RV2.country_state_code = 'DP'
  AND RV1.clearance_rail_flag = 'Y'
```

The values returned are:

```
("Cair Paravel" "Minas Tirith" "Coheeries Town"
                "Lake Woebegon" "Oz")
```

The query:

```
(db-retrieve ?port
  (:and
     (seaport ?port)
     (primary-port-name ?port "Oz")))
```

("Return a `seaport` object for the port whose name is 'Oz'") can be posed against the AKB. The SQL generated by LIM and the CIDI for this query is:

```
SELECT DISTINCT RV1.name,
          RV2.latitude,
          RV2.longitude
FROM SEAPORTS RV1, GEOLOC RV2
WHERE RV2.glc_cd = RV1.glc_cd
  AND RV1.name = 'Oz'
```

The value returned by this query is an object whose Loom definition is:

```
(TELL
  (:ABOUT SEAPORT59253
     SEAPORT
     (LON 98.6)
     (LAT 3.14159)
     (PRIMARY-PORT-NAME "Oz")))
```

7. Status

Our current system is implemented in Lucid Common Lisp and runs on a SUN SPARCstation 2. LIM uses both the CIDI [[11]] and the Loom knowledge representation language [[10]]. The CIDI uses one of several protocols to access Oracle databases on a remote server. LIM can be used either in a stand-alone Lisp image, or via a client-server architecture.

The present released version of LIM supports all definition and retrieval operations, as well as update of SMKB concepts and simple View-Concepts, and basic Meta-View-Concept functionality. The present development version of LIM includes the basic functionality of update of complex View-Concepts (i.e., those requiring joins); this version should be released by the time of the publication of this paper.

Several other participants in the ARPI—including ISI, ISX, UCLA, MITRE, and BBN—are presently using LIM. All of these are using LIM internally in a stand-alone Lisp image, but most are also using LIM via a client-server architecture from a remote site, typically accessing one or more servers at Unisys via KQML.

In addition, groups at ISI and UCLA have integrated LIM with SIMS [[1],[2]] and CoBASE [[8]]; the combined systems provide integration of access to multiple sources, approximate queries and answers, and fault tolerant knowledge base access to databases. The LIM/CoBASE/SIMS system has been used in both local-Lisp-image and client/server realizations; the ultimate objective of the joint project is to have a fully-distributed, multi-database system in operation.

8. Future Work

Our plans for extensions to LIM include support for:

- Data Inference
  - Data *object* rendering
  - Data classification
- Multi-DB/Cooperative Query Processing
  - Multi-DB access provided by ISI SIMS
  - Cooperative query processing provided by UCLA CoBASE
- Object-Oriented DB Storage
- “Smart” caching

9. Conclusion

We have described a View-Concept model which uses a knowledge representation language, Loom, to define the semantic schema of a database. This definition has two levels, each of which is of utility to a knowledge-based application. Both are based on a verbatim model of the database; for legacy databases, this can be generated automatically from the database schema, and can be used by any knowledge-based application which would assist a knowledge base administrator in the development of the semantic mapping layer (i.e., a knowledge-based semantic schema).

The semantic mapping layer defines the relevant concepts supported by the database domain; in our current knowledge bases, the semantic mapping layer adds semantic types to the automatically-generated schema model. We envision additional information in the semantic mapping layer, including composites of database objects which form larger conceptual structures.

Finally, the View-Concept model includes an application-specific layer that defines the mapping between an application domain’s conceptual structures and the semantic definition of database concepts. We believe that the structured approach embodied in the View-Concept model significantly elucidates the knowledge-base-to-database interface problem. Further, we expect that grounding the implementation in the CIDI will support reasonable performance.

Our preliminary implementation includes algorithms for properly defining objects to determine retrievability and updatability, as well as retrievals against a database. In the coming year, we will be developing a more sophisticated application for the military transportation logistics domain using LIM. We expect feedback from this experience primarily to concern the completeness of the application knowledge base, and to give us valuable performance data.

10. Acknowledgments

The authors wish to acknowledge the contributions of current and former members of the Unisys Knowledge-Based Planning group, specifically Rebecca Davis, Robin McEntire, Rich Fritzson, and Barry Silk (US Government). In addition, the first version of the Intelligent Database Interface was developed by Tony O’Hare, currently at IBM, Research Triangle, and Larry Travis of the University of Wisconsin.

11. References

[1] Yigal Arens, “Services and Information Management for Decision Support,” *AISIG-90: Proceedings of the Annual AI Systems in Government Conference*, George Washington University, Washington, DC, May, 1990.

[2] Yigal Arens, “Planning and Reformulating Queries for Semantically-Modeled Multidatabase Systems,” *Proceedings of the First International Conference on Information and Knowledge Management*, Baltimore, MD, November, 1992.

[3] Thierry Barsalou and Gio Wiederhold, “Applying a Semantic Model to an Immunology Database,” 1987.

[4] Thierry Barsalou, “An Object-Based Architecture for Biomedical Expert Database Systems,” 1988.

[5] Thierry Barsalou, Arthur M, Keller, Niki Siambela, and Gio Wiederhold, “Updating Relational Databases through Object-Based Views,” ACM, 1991.

[6] Ronald Brachman and James Schmolze, “An Overview of the KL/ONE Knowledge Representation System,” *Cognitive Science* 9, 1985, pages 171–216.

[7] M. Brodie, J. Mylopoulos, and J. W. Schmidt, editors, *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*, Springer-Verlag, 1984.

[8] Wesley W. Chu, Andy Y. Hwang, Rei-Chi Lee, Qiming Chen, Matthew Merzbacher, and Herbert Hecht, “Fault Tolerant Distributed Database System via Data Inference,” *Proceedings of the Ninth Symposium on Reliable Distributed Systems*, Huntsville, Alabama, October 9–11, 1990.

[9] R. Kowalski, *Logic for Problem Solving*, Elsevier, 1979.

[10] Robert MacGregor and Robert Bates, “The Loom Knowledge Representation Language,” *Proceedings of the Knowledge-Based Systems Workshop*, April 1987.

[11] Don McKay, Tim Finin, and Anthony O'Hare, “The Intelligent Database Interface,” *Proceedings of the 7th National Conference on Artificial Intelligence*, 1990.

[12] G. Christian Overton, Kimberle Koile, and Jon A. Pastor, “GeneSys: A Knowledge Management System for Molecular Biology,” *Computers and DNA*, Santa Fe Institute, G. Bell and T. Marr, editors, Addison-Wesley, Reading, MA, 1990.

[13] Michael Stonebraker and Larry Rowe, *The Postgres Papers*, University of California, Berkeley, 1987.

[14] Gio Wiederhold and R. ElMasri, “The Structural Model for Database Design,” in *Entity-Relationship Approach to System Analysis and Design*, pages 237–257, North Holland, 1980.

[15] Gio Wiederhold, “Views, Objects, and Databases,” *IEEE Computer*, Vol. 19, no. 12, December 1986, pages 37–44.