CoBase: A Cooperative Database System *

Wesley W. Chu, Qiming Chen and Matthew Merzbacher
Computer Science Department
University of California, Los Angeles

Abstract

This chapter proposes the use of the *type abstraction hierarchy (TAH)* as a framework for deriving *cooperative query answers* (CQA). The TAH integrates abstraction with the subsumption (is-a) and composition (part-of) semantics found in the type hierarchy. This framework provides a multi-level object representation that is an important aspect of cooperative query answering. *Patterns*, which specify one or more conditions on an object, are introduced as a small granularity structure with specific semantic information. Cooperative query answering uses the TAH and patterns to provide query relaxation, generalization and specialization. Relaxation can be explicitly controlled by the user, implicitly performed by the system, or both. An explanation system is also included to present the relaxation path as well as a nearness measure between the approximate answer and the exact answer.

The operations required for CQA may be added to any conventional query language. As an example, we present *CSQL*, a cooperative extension of SQL used in the CoBase project at UCLA. Experimental results reveal that the proposed type abstraction hierarchy and relaxation control provide an organized structure for representing concepts at different knowledge levels in various domains, and offers a systematic and efficient method for cooperative query answering.

Index Terms: Knowledge Representation, Type Hierarchy, Type Abstraction Hierarchy, Query Language, Query Rewrite, Cooperative Query Answering, Semantic Distance, Relaxation Control

1 Introduction

When making daily decisions, humans seldom have complete or exact information. Yet, traditional query processing systems accept only precisely specified queries, requiring users to fully understand the problem domain and the database structure and content. Further, traditional systems offer only exact answers, returning limited null information if the precise answer is not available.

*This work supported by DARPA contract N00174-91-C-0107

Cooperative query answering (**CQA**) [4, 5, 6, 9, 11, 13, 20] is an extension to the classical database notation that remedies these shortcomings.

Cooperative query answering provides neighborhood or generalized information relevant to the original query and within a certain semantic distance to the exact answer. To do this, the CQA system enlarges the scope of the query by broadening the search range to areas near to the original query. Queries with no exact answer can be broadened until an approximate answer is found; even queries with limited answers can be extended to find additional approximate answers. For example, in response to a query about a specific flight departing at 9am from LAX airport in Los Angeles to Dulles airport near Washington, the system may return all the *morning* flights from airports within the Los Angeles *area* to airports within the Washington *area*. Using the same techniques, cooperative query answering is also able to answer imprecisely specified queries.

A cooperative query answering system requires the support of different knowledge representations at different abstraction levels and query transformations between these levels. Enlarging and shrinking a query scope is accomplished by shifting the queried objects between different levels of abstraction. Thus, our solution to cooperative query answering uses a multi-level knowledge representation.

Although the link between object representations at different knowledge levels can be made using explicit rules [9, 21, 14], such rules lack a systematic organization to guide the query transformation process. As a result, it is difficult to scale-up or combine rule based systems. We propose the **type abstraction hierarchy** (TAH) data structure as an efficient organized framework for coupling data and knowledge for cooperative query processing.

In this chapter we introduce the type abstraction hierarchy and demonstrate its use for neighborhood and conceptual query answering. We introduce the cooperative extension to SQL, *CSQL*, and show how it can be used to control relaxation and search in the TAH. Based on query context and the user profile, a relaxation manager trims the TAH to limit the search scope and reduce the number of relaxed query answers. Finally, we present an implementation of CoBase.

2 Cooperative Answering via Abstraction Hierarchies

There are three kinds of objects in a relational database, each of which may be abstracted. Attribute values, tuples, and relations can be abstracted separately or in conjunction.

Attribute value abstraction, or *value abstraction*, is a method for changing a single attribute value into a range of nearby values. A query is relaxed by taking one or more of its conditions and relaxing it to an approximate range.

For example,

$airport$ = "LAX"

can be abstracted to the conceptual attribute

$area$ = "Los Angeles"

and then specialized back to a set of values

$airport$ = "LAX", "Burbank" or "Long Beach"

Multiple levels of abstraction are allowed, so the *area* conceptual attribute may be generalized further to

$region$ = "Southern California"

then specialized back, and so on.

Tuple abstraction organizes the tuples in each relation into abstraction groups. A tuple may belong to multiple abstraction groups. To find the tuples near to a given target tuple, first determine the abstraction groups to which the tuple belongs and then find the other tuples in those abstraction groups. Those tuples are valid approximations of the target.

Tuple abstraction is approximated in CoBase by simultaneous relaxation of the attributes forming the tuple using value abstraction. For example, in an airline database with each airline storing its flights in a separate relation, the American Airlines tuple,

AA_flight(LAX, Dulles, 10am, 6pm, #076)

may be relaxed by respectively relaxing each of the attribute values to:

AA_flight(Los Angeles, Washington, morning, afternoon, any flight)

Since a higher-level and more abstract object representation corresponds to multiple lower-level and more specialized object representations, querying an abstractly represented object is equivalent to querying multiple specialized objects.

Relation abstraction allows the relation itself to be approximated. In the example above, the relation "AA_flight" might be relaxed to the abstraction cross-country flight ("CC_flight") which also represents flights in the relations for other airlines. Further, since CC_flight is a conceptual relation, it only need contain the attributes distinguishing it. The flight number attribute can be dropped, since it may have any value at this level of abstraction.

CC_flight(Los Angeles, Washington, morning, afternoon)

As in the other cases, multiple layers of abstraction are possible. CC_flight may join with cross-country Bus and Train trips to form:

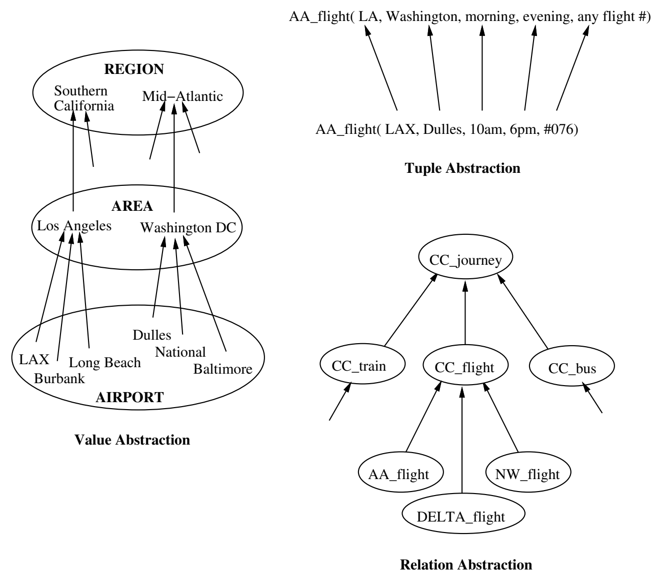

Figure 1: Abstract Representations

CC_journey(Los Angeles, Washington)

In this case the departure and arrival times have been removed from the abstraction. Queries conditions on departure and arrival will be ignored at this level of abstraction, but enforced when the query is specialized back down to the instance level.

Together, the three kinds of abstraction (shown in Figure 1) form the *Type Abstraction Hierarchy*, the principal data structure for cooperative query answering used in CoBase.

The notion of multi-level object representation is not captured by the conventional semantic network and object-oriented database approaches. Grouping objects into a class and grouping several classes into a super-class only provide a common "title" (type) for the involved objects without concern for the object instance values and without introducing abstract object representations. Grouping several objects together and identifying their aggregation as a single (complex) object does not provide abstract instance representations for its component objects. Therefore, an object-oriented database deals with information only at two general layers: the meta-layer and the instance layer. Since forming an object-oriented type hierarchy does not introduce new instance values, it is impossible to introduce an additional instance layer. In the type abstraction hierarchy, instances of a super-type and a sub-type may have different representations and can be viewed at different instance layers. Such multiple layer knowledge representation is essential for cooperative query answering.

The type abstraction hierarchy is instance-representation-based and consists of the following key aspects:

- The value abstraction mechanism for atomic and composed object representations.
- An abstraction relationship between all layers in the hierarchies. In all cases, a super-type object conveys a *more abstract representation* than its sub-type object. For example, attribute type *area* is a super-type of attribute type *airport*, and tuple type *CC_flight* is a super-type of tuple type *AA_flight*.
- A formalism for integrating the abstraction view of type hierarchy with the subsumption (is_a) and composition (part_of) views of the type hierarchy, described in Section 3.1.

As shown above, our notion of abstraction originates from different representations between atomic values at different levels. Context based relationships are stored in the database and can be combined to form an abstraction of complex values. Rewriting mechanisms are provided to transform object representations between different levels.

Translating a query between different abstraction levels is achieved by *query abstraction* and *query refinement*. This consists of converting object types, attribute names, and domain values between different knowledge levels. In general, a *query abstraction process* converts a query Q into a more abstract representation Q' through the following steps:

1. Identify a target to be abstracted. The target will usually be an attribute value, but can be a tuple or relation.
2. Find the appropriate super-type object for the target.
3. Convert any sub-parts of the target to their corresponding types (attributes associated with the same type may be named separately). This may be necessary for complex objects at any level.
4. Transform attributes referred to in the query to that of the super-type object through type generalization rewrite.
5. Transform conditions referred to in the query to those related to the super-type object through both type generalization rewrite and term generalization rewrite.

A *query refinement process* converts a query Q to more specific query representations $Q_1, Q_2, \ldots, Q_n$ according to the type abstraction hierarchy through the following steps:

1. Find the set of sub-type objects through type specialization rewrites.
2. Provide conversions between the attributes and the corresponding types.
3. Based on each sub-type, use type specialization rewrite to transform attributes referred to in the query to those in the sub-type object.
4. Based on each sub-type, use both type specialization rewrite and term specialization rewrite to transform conditions referred to in the query to those related to the sub-type object.

The query modification, either upward or downward, is invoked recursively depending on the requirement and knowledge availability. In the following sections we shall present two typical cooperative query answering mechanisms based on the notion of type abstraction hierarchy.

2.1 Neighborhood Query Answering

We first discuss the use of type abstraction hierarchy for supporting neighborhood query answering. Instead of providing exact answers, neighborhood query answering provides information of a wider range which may be helpful for the user. For example, assuming that a user tries to reserve a flight on DELTA airlines from LAX airport in Los Angeles to National airport in Washington D.C., if the request flight is unavailable, then an alternative but similar flight on DELTA or other airlines may be provided.

A neighborhood query answer is obtained by transforming the given query Q up and down along the type abstraction hierarchy to reach the neighboring objects. For an original query Q, the general processing steps are:

1. Search the exact type of objects required by the query Q. If this fails then go to the next step.
2. Move upward along the type abstraction hierarchy and rewrite the query to a more abstract one through a query abstraction process, i.e., $Q \rightarrow Q'$.
3. Move downward along the hierarchy and rewrite the query to more specific ones through a query refinement process, i.e., $Q' \rightarrow Q''$.

Three kinds of tables are used for neighborhood query processing: the *relation_abstraction hierarchy* which describes the subtyping relationship among relations; the *domain-mapping table* that gives the relationship between attribute names and domain names, allowing attributes from the same domain to be differently named; and the *attribute–value abstraction* which shows the mapping of instances for each pair of super-type and sub-type or the corresponding instance values between a super-type and a range of sub-type (e.g. "morning" corresponds to "7am to 11am"). These tables describe knowledge representations at different knowledge levels which are stored in and managed by the database. Certain additional query language constructs to SQL are introduced, such as **within** for indicating set membership, **between** for indicating a range with an upper and a lower bound, and the relaxation operation '∼' for object types or attributes.

We shall show some examples based on the relation abstraction hierarchy shown in Figure 2, where "DELTA" is the abbreviation for "DELTA Airlines" and "NW" for "Northwest Airlines". The notation '∼' following an object type or an attribute in the original query indicates that it is relaxable. The domain mapping table and the attribute value abstraction mapping tables partially illustrated in Tables 1 and 2 are used in these examples.

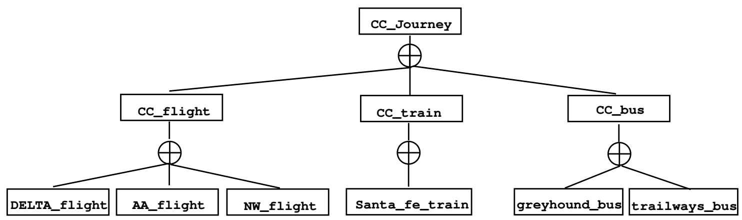

Figure 2: A Relation Abstraction Hierarchy

Attribute Name	**domain**
departure_time, arrival_time	time
departure_period, arrival_period	period
departure_airport, arrival_airport	airport
departure_area, arrival_area	area

Table 1: Domain Mapping Table

In this example, only attribute relaxation occurs. **9am** on attribute *departure_time* is relaxed to **morning** on attribute (type) *departure_period* through a **query abstraction** process, which is further refined to **between <7am, 11am>** through a **query refinement** process. Since the resulting query condition **between <7am, 11am>** has wider coverage than the query condition **9am**, it provides query answers with a wider range.

Original Query

select * from AA_flight
where *departure_time* ∼= 9am
and *departure_airport* = "Dulles"
and *arrival_airport* = "LAX".

time	period
<7am, 11am>	morning
<11am, 1pm>	noontime
<1pm, 7pm>	afternoon
<7pm, 11pm>	evening

airport	area
LAX, Burbank, Long Beach	Los Angeles
National, Dulles, Baltimore	Los Angeles

fare	flight_cost	cost
<301, 400>	flight_cost_low	reasonable
<401, 600>	flight_cost_medium	high
<601, 800>	flight_cost_high	high

fare	train_cost	cost
<251, 300>	train_cost_low	low
<301, 350>	train_cost_medium	reasonable
<351, 400>	train_cost_high	reasonable

fare	bus_cost	cost
<201, 250>	bus_cost_low	low
<251, 300>	bus_cost_medium	low
<301, 350>	bus_cost_high	reasonable

Table 2: Attribute Value Abstraction Tables

Query Abstraction

select * from AA_flight
where *departure_period* = "morning"
  and *departure_airport* = "Dulles"
  and *arrival_airport* = "LAX".

Query Refinement

select * from AA_flight
where *departure_time* **between** < 7am, 11am>
  and *departure_airport* = "Dulles"
  and *arrival_airport* = "LAX".

In the next example, both the attribute values and the relation of queried objects (airlines) are relaxed, and therefore the returned information covers more options on airlines, departure and arrival airports, and fares.

Original Query

select * from **DELTA_flight** ∼
where *departure_airport* ∼= "Long Beach"
and *arrival_airport* ∼= "Dulles"
and *fare* ∼**between** <320, 350>

Query Abstraction

select * from **CC_flight**
where *departure_area* = "Los Angeles"
and *arrival_area* = "Washington"
and *flight_cost* = "flight_cost_low"

Query Refinement

select * from **DELTA_flight**
where *departure_airport* **within** {"LAX", "Burbank", "Long Beach"}
and *arrival_airport* **within** {"National", "Baltimore", "Dulles"}
and *fare* **between** <301, 400>

select * from **AA_flight**
where *departure_airport* **within** {"LAX", "Burbank", "Long_Beach"}
and *arrival_airport* **within** {"National", "Baltimore", "Dulles"}
and *fare* **between** <301, 400>

select * from **NW_flight**
where *departure_airport* **within** {"LAX", "Burbank", "Long_Beach"}
and *arrival_airport* **within** {"National", "Baltimore", "Dulles"}
and *fare* **between** <301, 400>

As shown in Figure 3, the last example can be extended to a two level query abstraction and refinement. First the query on **DELTA_flight** is abstracted to **CC_flight** that also covers airlines of type **AA_flight** and **NW_flight**, then the modified query is further abstracted to cover **CC_journey** that includes not only **CC_flight** but also **CC_train** and **CC_bus**. As a result, even more options may be provided by answering the queries refined from the above abstract query as shown in the following.

Original Query

select * from **DELTA_flight** ∼
where *departure_airport* ∼= "LAX"

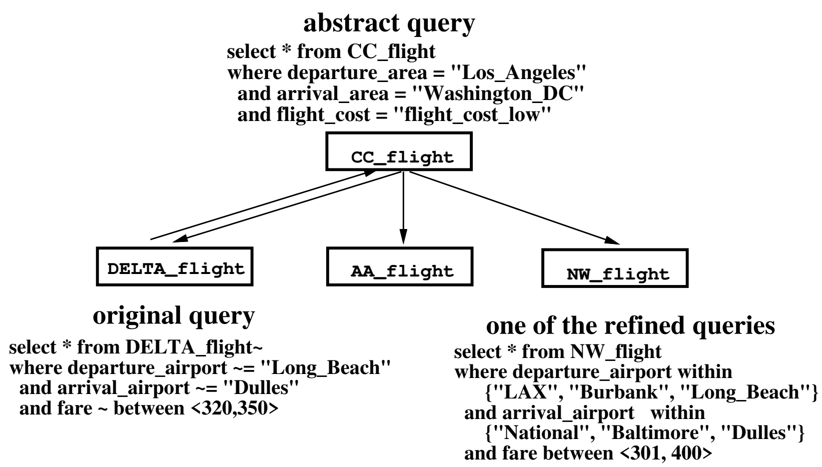

Figure 3: An Example of Neighborhood Query Answering

    and *arrival_airport* $\sim=$ "National"
    and *fare* $\sim$**between** $<$320, 350$>$.

First Step Of Query Abstraction From DELTA_flight

select * from **CC_flight**
where *departure_area* = "Los Angeles"
    and *arrival_area* = "Washington"
    and *flight_cost* = "flight_cost_low".

Further Query Abstraction From CC_flight

select * from **CC_journey**
where *departure_area* = "Los Angeles"
    and *arrival_area* = "Washington DC"
    and *cost* = "reasonable"

First Step Of Query Refinement From CC_journey

select * from **CC_flight**
where *departure_area* = "Los Angeles"
    and *arrival_area* = "Washington"
    and *flight_cost* = "flight_cost_low"

select * from **CC_train**
where *departure_area* = "Los Angeles"
    and *arrival_area* = "Washington"

and *train_cost* **within** {"train_cost_medium", "train_cost_high"}

select * from **CC_bus**
where *departure_area* = "Los Angeles"
and *arrival_area* = "Washington"
and *bus_cost* = "bus_cost_high"

Further Query Refinement From CC_flight

select * from **DELTA_flight**
where *departure_airport* **within** {"LAX", "Burbank", "Long_Beach"}
and *arrival_airport* **within** {"National", "Baltimore", "Dulles"}
and *fare* **between** <301, 400>

select * from **AA_flight**
where *departure_airport* **within** {"LAX", "Burbank", "Long_Beach"}
and *arrival_airport* **within** {"National", "Baltimore", "Dulles"}
and *fare* **between** <301, 400>

select * from **NW_flight**
where *departure_airport* **within** {"LAX", "Burbank", "Long_Beach"}
and *arrival_airport* **within** {"National", "Baltimore", "Dulles"}
and *fare* **between** <301, 400>

Further Query Refinement From CC_train

select * from **santa_fe_train**
where *departure_train_station* **within** {"Los Angeles", "Long Beach"}
and *arrival_train_station* **within** {"DC", "Fairfax"}
and *fare* **between** <301, 400>

Further Query Refinement From CC_bus

select * from **greyhound_bus**
where *departure_bus_station* **within** {"LA Downtown", "Hollywood", "Long Beach"}
and *arrival_bus_station* **within** {"DC Downtown", "Rockville", "Fairfax"}
and *fare* **between** <301, 350>

2.2 Conceptual Query Answering

In addition to neighborhood query answering, the type abstraction hierarchy supports conceptual query answering. Often, a user has a question in mind but does not know *exactly* how to formulate the query. For example, if a user wants to get information about traveling from Los Angeles to Washington D.C., but is unfamiliar with the airline, bus and train schedules, he cannot expect to ac-

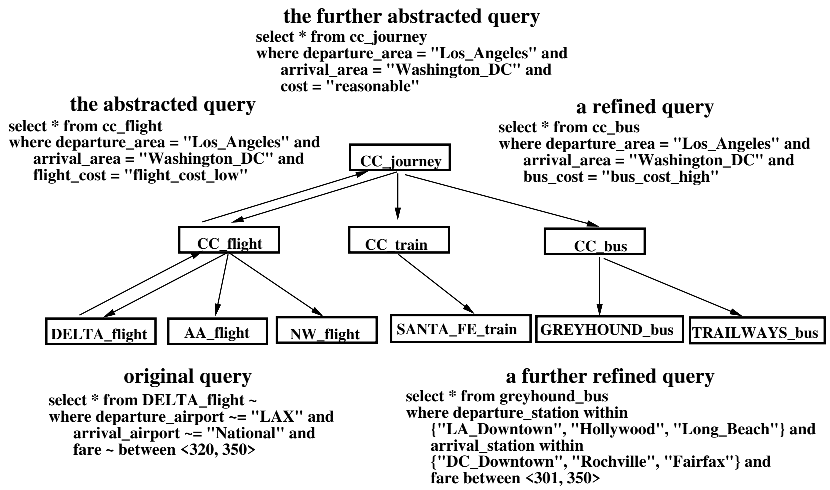

Figure 4: Example of Neighborhood Query Answering with Two–Level Abstraction/Refinement

curately phrase his queries. The type abstraction mechanism is able to answer conceptual queries. With this approach, the user may ask a more general question about "How to travel from Los Angeles to Washington D.C. at a reasonable cost",

select * from **CC_journey**
where *departure_area* = "Los Angeles"
  and *arrival_area* = "Washington"
  and *cost* = "reasonable"

Using the *hierarchy* in Figure 5, such a query can be automatically refined to the following queries:

select * from **CC_flight**
where *departure_area* = "Los Angeles"
  and *arrival_area* = "Washington"
  and *flight_cost* = "flight_cost_low"

select * from **CC_train**
where *departure_area* = "Los Angeles"
  and *arrival_area* = "Washington"
  and *train_cost* **within** {"train_cost_high", "train_cost_medium"}

select * from **CC_bus**
where *departure_area* = "Los Angeles"
  and *arrival_area* = "Washington"

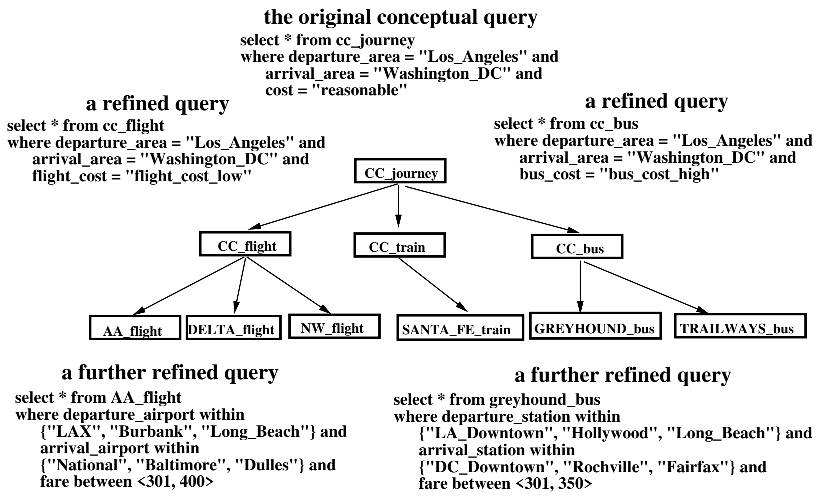

Figure 5: Refining a Conceptual Query

and *bus_cost* = "bus_cost_high"

Note that in the above refined queries, **CC_journey** is "refined" to **CC_flight**, **CC_bus** and **CC_train**. These three queries can be further refined to

select * from **AA_flight**
where *departure_airport* **within** {"LAX", "Burbank", "Long Beach"}
and *arrival_airport* **within** {"National", "Baltimore", "Dulles"}
and *fare* **between** <301, 400>

select * from **santa_fe_train**
where *departure_train_station* **within** {"LA", "Long Beach"}
and *arrival_train_station* **within** {"DC", "Fairfax"}
and *fare* **between** <301, 400>

select * from **greyhound_bus**
where *departure_bus_station* **within** {"LA Downtown", "Hollywood", "Long Beach"}
and *arrival_bus_station* **within** {"DC Downtown", "Rockville", "Fairfax"}
and *fare* **between** <301, 350>

and so on.

Based on the hierarchy of knowledge, a conceptual query can be processed to derive a set of more specific queries which can be answered by a conventional query processing system.

3 CSQL – A Language for Cooperative Query Answering

CSQL is an extension of the SQL query language containing cooperative operations to support cooperative query answering. *CSQL* has general purpose syntactic constructs but domain-oriented interpretation. We provide, on top of an existing data schema, a type hierarchy specification based on the proposed abstraction notion and store the corresponding super-type and sub-type domain values in a table. This information is used to associate various concepts at different knowledge levels and to interpret *CSQL*. Based on the users interest, the set of typical queries, and the expected answers, we can construct the desired type abstraction hierarchies in selected problem domains to provide a structured organization for concepts. The system can then provide cooperative query answering in those domains.

3.1 Foundations of CSQL Semantics

The semantics of *CSQL* is based on type abstraction hierarchy, match relaxation and query pattern relaxation as discussed in this section.

3.1.1 Type Abstraction Hierarchy

The type abstraction hierarchy (TAH) combines the notions of subsumption (is_a) and composition (part_of) to form an integrated hierarchy. In the TAH, *abstraction* is achieved by different representations of an object at different levels of the hierarchy; *subsumption* is a special case of abstraction where the abstract representation of an object under a super-type is the same as its representation under a sub-type; and *composition* is introduced by defining the structural subtype relationship. Therefore our extension also takes into account complex objects. In general, type T is a subtype of T' means:

1. The domain of T' is abstract over the domain of T,
2. The domain of T' subsumes the domain of T (as a special case of the above),
3. The type structure of T' is contained in the type structure of T.

Formally, a type can be an atomic type, a tuple-type $T : (T_1, \ldots, T_n)$, or a set-type, $T : \{S\}$, where $T_1, \ldots, T_n$ and S are types. Based on the type abstraction notion, a type T is a sub-type of a type T', denoted as $T \leq T'$, if,

Subtyping

1. For atomic types T and T', if each value in $dom(T')$ represents a single or multiple values in $dom(T)$, then $T \leq T'$, where $dom(T)$ denotes the domain of type T.

2. For tuple-types $T : (T_1, \ldots, T_n)$ and $T' : (T'_1, \ldots, T'_m)$ where $n \geq m, T \leq T'$ iff $\forall i \in \{1, \ldots, m\} T_i \leq T'_i$.

3. For set-types $T : \{S\}$ and $T' : \{S'\}, T \leq T'$ iff $S \leq S'$.

The above definition integrates the subsumption composition views of type hierarchy with subsumption as a special case of abstraction. For a given class, multiple type hierarchies may be formed based on different views.

Let $\Rightarrow$ denote logical implication. By combining object instances and variables of the basic types, the notion of terms can be defined as follows.

Term

The set of terms of type T, Γ_T, is defined recursively as follows:

1. For a constant value c and an atomic type T, $c \in dom(T) \Rightarrow c \in \Gamma_T$.

2. For a variable v of type T, $v \in \Gamma_T$.

3. For a tuple-type $T : (T_1, \ldots, T_n), \forall i \in \{1, \ldots, n\} t_i \in \Gamma_{T_i} \Rightarrow T(t_1, \ldots, t_n) \in \Gamma_T$ (tuple-term).

4. For a set-type $T : \{T_0\}, \forall i \in \{1, \ldots, n\} t_i \in \Gamma_{T_0} \Rightarrow T\{t_1, \ldots, t_n\} \in \Gamma_T$ (set-term).

Query abstraction and refinement are based on the mechanisms of **generalization rewrites** and the **specialization rewrites** of types and terms for converting object representations between different abstract levels. The type generalization rewrite converts a type to a more abstract type, and the term generalization rewrite deals with both types and domain values for transforming low level object representations to higher level ones. Their definitions are given below where $\rightarrow$ denotes rewrite or mapping, *type(t)* denotes the type of a term t, and *gen* represents the relation between values of atomic types and their subtypes such as "Los Angeles" and "LAX".

Type Generalization Rewrite

The set of *generalization rewrite* of type T, Ω_T, is defined as :

- For atomic types T and S, $T \leq S \Rightarrow T \rightarrow S \in \Omega_T$.

- For tuple-types $T : (T_1, \ldots, T_n, \ldots)$ and $S : (S_1, \ldots, S_n)$ where $T \leq S$, $T_1 \rightarrow S_1 \in \Omega_{T_1}, \ldots, T_n \rightarrow S_n \in \Omega_{T_n} \Rightarrow T : (T_1, \ldots, T_n, \ldots) \rightarrow S : (S_1, \ldots, S_n) \in \Omega_T$.

- For set-types $T : \{T_0\}$ and $S : \{S_0\}$ where $T \leq S$, $T_0 \rightarrow S_0 \in \Omega_{T_0} \rightarrow T : T_0 \Rightarrow S : S_0 \in \Omega_T$.

Term Generalization Rewrite

The set of *generalization rewrites* of term t, ω_t, are defined as :

- For atomic types T and S, if $T \rightarrow S \in \Omega_T, t, s$ are instances, $type(t) = T$ and $type(s) = S$ and $s = gen(t)$, then $t \rightarrow s \in \omega_t$.
- If $T \rightarrow S \in \Omega_T$, t is a variable of T and s a variable of S, then $t \rightarrow s \in \omega_t$.
- For $t = T(t_1, \ldots, t_n, \ldots), s = S(s_1, \ldots, s_n)$,
  $T \rightarrow S \in \Omega_T$ and $t_1 \rightarrow s_1 \in \omega_{t_1}, \ldots, t_n \rightarrow s_n \in \omega_{t_n} \Rightarrow t \rightarrow s \in \omega_t$.
- For t $= T\{t_1, \ldots, t_n\}, s = S\{s_1, \ldots, s_m\}$,
  $T \rightarrow S \in \Omega_T$ and $\forall i \in \{1, \ldots, n\}(\exists j \in \{1, \ldots, m\} t_i \rightarrow s_j \in \omega_{t_i}) \Rightarrow t \rightarrow s \in \omega_t$.

The notions of type specialization rewrite and term specialization rewrite can also be defined similarly. For a type abstraction hierarchy, the specialization rewrite from an abstract type or term may yield a set of refined types or terms which provide refined information for searching for neighborhood objects. In general, cooperative query answering is based on query modification via term rewriting.

In general, the proposed cooperative query answering is based on query modification which is just a term rewrite. For instance, a query modification from

select * from **AA_flight** ~
where *departure_airport* ~= "LAX"
  and *arrival_airport* ~= "Dulles"
  and *departure_time* ~= "10am"
  and *arrival_time* ~= "6pm"

to a more abstract one

select * from **CC_flight**
where *departure_area* = "Los Angeles" and *arrival_area* = "Washington"
  and *departure_period* ~= "morning"
  and *arrival_period* ~= "afternoon"

is a term rewrite from the term

AA_flight(LAX, Dulles, 10am, 6pm, X)

to the following more abstract term

CC_flight(Los Angeles, Washington, morning, afternoon, X)

where X is a variable for the flight identifier. Since a more abstract term has a wider coverage than that of a more specialized one, when a query represented by a specialized term is modified to a query represented by an abstract term, the scope of the query is enlarged. When a query represented by an abstract term is refined to multiple queries represented by specialized terms, the query is split into multiple goals for database access.

Query abstraction/refinement offers the operational view to cooperative query processing. The nature of cooperative query processing can also be described denotationally. Below we shall discuss it from *query matching* and *query pattern* point of views.

3.1.2 Match Relaxation

Traditional query answering aims at finding the objects that match the query condition exactly. Cooperative query answering aims at finding the objects that only match the query condition at an abstract level, namely, the abstract representation of the query condition. In other words, objects returned in a cooperative query answer have the same representations as the objects returned by an exact query answer, but at a higher level of abstraction.

For example, while the object being queried

AA_flight(LAX, Dulles, 10am, 6pm, #076)

is abstracted to a more generic representation

CC_flight(Los Angeles, Washington, morning, afternoon)

and then refined to a set of objects which might include

DELTA_flight(LAX, National, 11am, 5pm, #024).

There is an approximate match between the queried object of type AA_flight and the object of type DELTA_flight in the answer. This relaxed match characterizes the semantics of *CSQL*.

Under conventional typing, a precondition for two objects to match is that the types of their component terms at respective argument positions be identical. In a subsumption-based hierarchical typing system, two objects can also match if they are in a subsumption relationship. Thus,

AA_flight(LAX, Dulles, 10am, 6pm, #076)

matches

CC_flight(LAX, Dulles, 10am, 6pm, #076).

since CC_flight subsumes AA_flight. But

AA_flight(LAX, Dulles, 10am, 6pm, #076)

does not match

DELTA_flight(LAX, National, 11am, 5pm, #024),

because there is neither a match, nor a subsumption relationship. However, the type abstraction notion underlying *CSQL* has extended this concept one step further. The fact that type T_2 is a super-type of type T_1 implies the existence of abstract representations of objects of type T_1 under type T_2, so that "Dulles" on attribute *departure_airport* has the abstract representation "Washington" on attribute *departure_area*. The goal reformulation facility realizes the relaxable match between an object of T_1 and its abstraction representation of T_2 where $T_1 < T_2$. Therefore,

AA_flight(LAX, Dulles, 10am, 6pm, #076)

and

DELTA_flight(LAX, National, 11am, 5pm, #024)

match each other approximately via

CC_flight(Los Angeles, Washington, morning, afternoon).

3.2 Query Pattern Relaxation

The proposed cooperative query answering mechanism can also be viewed as a *query pattern relaxation mechanism* [12]. To show this we introduce the notions of pattern and pattern instance.

A pattern is defined on a type by specifying a *condition* [3]. For example, given the following schema definition,

AA_flight(*departure_airport*, *arrival_airport*, *departure_time*, *arrival_time*, *flight#*)

conditions such as

departure_airport = "LAX", or
departure_time = "9am"

define patterns on the type AA_flight. The objects of the type that satisfy the pattern are said to *match the pattern*, and the identifiers of those objects form a **Pattern Instance (PI)**. Since it is formed of conditions on attributes, a query can be viewed as a pattern called a **query pattern**. The set of identifiers of the objects matching a query pattern is the **Query Pattern Instance (QPI)** for that pattern, while the set of identifiers of objects that are actually included in the query answer is called the **Answer Pattern Instance (API)**. Exact query answering means that the API for a query is identical to its QPI.

Exact match is a special case of cooperative query answering. Very often the API of a relaxed query contains the QPI of the original query. Thus a

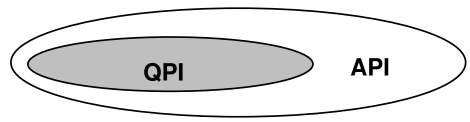

Figure 6: Relaxed API subsumes QPI

cooperative query answering is made through **relaxing the query pattern**. For example, when "morning" is used as the abstract representation of "9am", the query pattern with condition "$departure_time = 9$am" is relaxed to the query pattern with condition "*departure_time* between <8am, 11am>" and "morning" may be viewed as the *name of the relaxed range* <8am, 11am>. This offers a quantitative view to the type abstraction based cooperative query answering mechanism.

The query pattern relaxation view provides a general way to express an abstract object by a range. It is typical for those attributes whose values form a *total ordering*, such as attributes on time, space or numerical numbers, but their ranges are often *difficult to name*. For instance, it is hard to find an *abstract representation* for "between 10am to 1pm". Note that using type abstraction representations may not be suitable for relaxing a query on condition *departure_time*

between <10am, 1pm>,

since 10am falls into the abstract time period *morning* which can be refined to (7am, 11am), and 1pm falls into the abstract time period *afternoon* which can be refined to (1pm, 6pm). Obviously we do not prefer relaxing it to

between <7am, 6pm>

since 10am nears the right limit of (7am, 11am) and 1pm is the left limit of (1pm, 6pm). Instead the system should take into account the position of 10am within (7am, 11am) and the position of 1pm within (1pm, 6pm) and yields a closer range

between <9am, 2pm>

as the default range.

By viewing the query abstraction/refinement function as a black-box and only concerning ourselves with its input and output ranges, the type abstraction approach to cooperative query answering is a special kind of range relaxation. Using a type abstraction hierarchy to guide range relaxation is necessary when the range can only be expressed conceptually, and not quantitatively.

3.3 Language Constructs

CSQL provides the following extended query language constructs to allow query relaxation:

- Relaxation symbol '$\sim$' for marking the relaxable relations, attributes and values for which an approximate query answer is tolerable. For example,

  select * from *AA_flight*
  where *departure-airport* = 'Dulles'
      and *arrival-airport* $\sim=$ 'LAX'
      and *departure-time* $\sim=$ 10am

  means arrival-airport and departure-time may be relaxed which rewrites the query to,

  select * from *AA_flight*
  where *departure-airport* = 'Dulles'
      and *arrival-airport* **within** { 'LAX', 'Long Beach', 'Burbank' }
      and 9am $<$ *departure-time* $<$ 11am

- **Relaxation order** specifying the ordered preference of attribute relaxation. For example, after the above query,

  relaxation-order (*arrival-airport, departure-airport*)

  indicates that if no exact answer is found, *arrival-airport* is relaxed. If an answer is still not found, *departure-airport* is relaxed. If no single-attribute approximate answer is found, both attributes are relaxed.

- Syntactic relaxable predicates **between** and **within**. **between** indicates a range of values and **within** indicates set membership. Both operations can include the $\sim$ operator, as specified below:

  arrival-time **between** $<$7am, $\sim$11am$>$ becomes
      7am $<$ X $<$ $\sim$11am, which is
      7am $<$ X $<$ Noon

  within $\sim$ {'LAX', 'Burbank'} becomes
      within {$\sim$'LAX', $\sim$'Burbank'} which is,
      within {'LAX', 'Burbank', 'Long Beach'}

- Context Sensitive predicates which are relaxable based on given contexts, such as

near-to ('Redondo Beach')
similar-to ('McDonalds') **based-on** (cost, w_1; type-of-food, w_2)

Near-to maps to Euclidean distance less than a threshold based on the query context. In general, this predicate returns values with a context-based semantic distance less than the specified threshold.

similar-to requests a set of objects semantically similar to the given object. Similarity is a multiple attribute operator with a weight assigned to each attribute in accordance with the relative importance of that attribute. Each object fulfilling the query conditions is evaluated against the given object by calculating the weighted mean-squared error [19] over the list of attributes. The similar objects are listed in ranked order, along with their measures.

- Control commands such as **nearer** and **further** which control the relaxation scope of **near-to** interactively. **nearer** reduces the threshold by a prespecified percentage, while **further** increases the threshold value by that percentage.

3.4 Context Based Interpretation of CSQL

The relationship between the semantic distance and the actual distance of objects is context sensitive. For example, scales for flight distances and for urban commuting distances should be assigned differently. The nearness relations also vary from case to case. For example, **near-to** has different meaning in different situations as shown by Table 3. Such domain knowledge is stored in the knowledge base for supporting cooperative query answering.

attribute	**nearness measure**
departure_time	1 hour
flight_time	10 minutes
airport_distance	50 miles
restaurant_distance	2 miles

Table 3: Context Sensitive Nearness Measure for Different Attributes

Unlike SQL interpretation, *CSQL* interpretation considerably depends on the database content. Language constructs for *CSQL* allow the user to specify the context of type abstraction hierarchy, e.g. "business trip", and the context dependent nearness measure, e.g. "relaxation range for *departure_time* is 1 hour". Such a specification is then converted to the domain specific nearness measure used by *CSQL*.

3.5 Relaxation Control

When query is not answerable, some of its attribute values are relaxed to provide a wider search scope. As a result, approximate answers are derived. However, without control over relaxation, we may generate too many approximate answers for the user. Further, if the user fails to sufficiently constrain the query, too many answers may be generated. For example, if the user asks for flights from Los Angeles to Washington on Thursday, then he will be inundated with hundreds of answers. Suppose we know in addition that the user is a frequent flier on United Airlines and that he probably has an early Friday morning meeting. With that extra information we can further constrain the query search to United flights arriving before 9 PM. If it is known that he has many friends in Washington, and that his company will pay for an extra night in the hotel if it leads to a lower airfare *, his return may be postponed to Sunday morning to reduce cost. He can, of course, override this modification if he needs to arrive back before then.

The *Relaxation Manager* combines such rules to restrict the search for approximate answers. The rules are provided from several contexts, including the user profile and query context. After the rules are applied, relaxation can proceed with only a fraction of the possible tuples under consideration. In addition to constraints on the search, the user can specify the number of answers as well as rules that combine restrictions on multiple attributes.

The RM has three functions. As presented above, it applies rules to constrain the relaxation of attributes by trimming the type abstraction hierarchy as necessary. Next, the RM determines how to direct the search through those hierarchies. For example, there still may be many flights with low fare from LAX and Burbank to New York. Based on application– and user–specified demands, the RM selects the order to relax the attributes. Further, if the user over-constrains the system and no tuples are available after full constrained relaxation, the RM may decide which rules to ignore so that some answer can be obtained. A trimmed type abstraction hierarchy for the coast-to-coast travel example is shown in Figure 7.

---

*staying over Saturday night usually yields much lower airfares

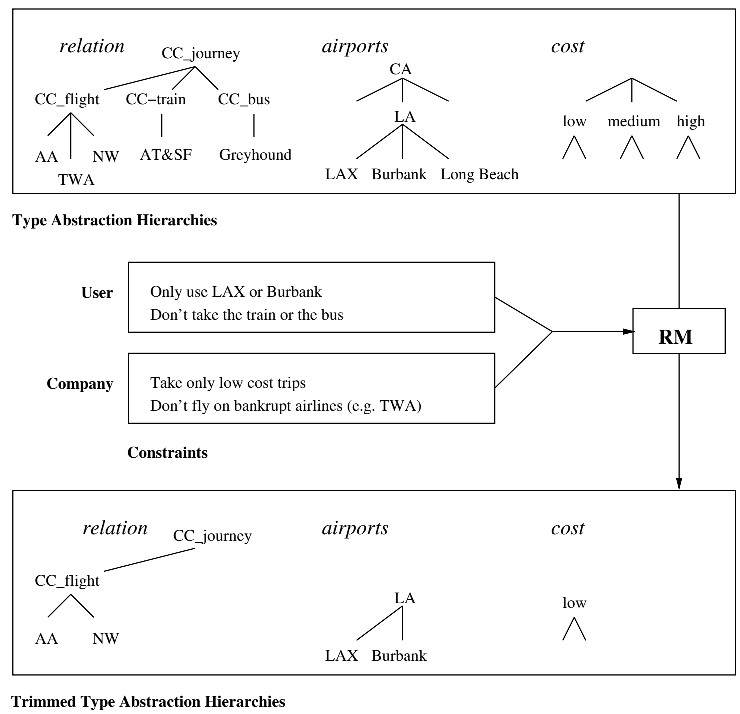

Figure 7: An Example of the Relaxation Manager trimming a Type Abstraction Hierarchy

3.5.1 Relaxation Rules

There are several types of relaxation specifications. Although more general rules give the user more flexibility of relaxation specification, using general rules may severely limit the amount of trimming possible on the type abstraction hierarchies. Less trimming means more tuples retrieved for evaluation and a slower system. Thus we recommend against using general type rules and present several special rule classes, below:

Attribute Relaxation Order A list of attributes in the order which they should be relaxed, as described in the explicit **relaxation-order** operation in *CSQL*. Multiple attributes can be relaxed simultaneously by specifying them together in the list. For example, *Food, Fare, Food and Fare, Departure Time, Departure Airport, Food and Departure Time, ...*

Relaxation Level The schedule of successive relaxations for each attribute. For example, *Fare: (10%, 30%, 50%), Departure Time: (20%, 30%), ...*

indicates that the first time Fare is relaxed, it should only be by 10%. If no approximate answers are found, the next relaxation of Fare is by 30%. Relaxation Level only specifies how far to relax each attribute; the order of relaxation is specified by the attribute relaxation order.

Desired Answer Set Size The desired size of the answer set.

Meta-Rules Rules that hold in general, rather than being about specific attributes, such as *Relax no more than two attributes at a time*, and *Don't relax any attributes more than 50%.*

The query language may restrict which categories of rules are possible. For example, the current implementation of *CSQL* allows only the relaxation order to be specified.

3.5.2 Contexts

Relaxation rules come from several sources, called contexts. There may be one set of rules for the user's company, another for the user himself, and still another one for the particular application domain. In the example of Figure 7, the user context has two rules and the company context has two rules. For a particular query, the user may decide to override some of the rules if, for example, he wishes to take the train instead of flying this time.

The relaxation manager gathers rules from the different applicable contexts. There are four context classes:

Explicit Specification
Constraints specified within the current query, such as "Find a non-stop flight to Tokyo, and *do not relax the non-stop*".

User Profile
Preferences of a particular user, such as, *United is preferred over American.* The User Profile may actually consist of several sub-contexts. For example, a user may have personal preferences, as well as rules from the company where he works and other outside constraints.

Domain Specific
Rules related to a particular application domain, such as, *Flight cost is more important than the in-flight meal*

Default Strategy
A general set of rules to guide relaxation when none of the other constraints are appropriate. For example, a default rule says to relax attributes until an answer is found. Further, the default rule to relax multiple values simultaneously is to minimize the sum of the semantic distances between the values in the approximate and exact answers.

3.5.3 Using Multiple Contexts in Relaxation

The relaxation manager can combine rules from multiple contexts, but the rules in the contexts may interfere with one another. To handle this problem, the user must select one of the following policies for selecting which rule to apply:

Default Order Policy

Use the following precedence to relax the query:
Query Specification > User Profile > Domain Specific > Default Strategy
Thus, restrictions specified in the query will override other any other rules. This is the strategy used in the current CoBase implementation.

User Specified Policy

Instead of using the default order policy, the user may specify a different order to apply the rules.

Separate Policy

Apply each context separately to derive different answer sets for each context. This policy will offer answers which satisfy one or more of the contexts, but not necessarily all of them.

Apply-All Policy

Apply all rules simultaneously. When there are no conflicting rules, stronger rules will supersede weaker rules. Therefore, this is the most restrictive policy. If no answers are found, then the RM switches to one of the other policies to remove certain rules until a solution is found.

4 Implementation

We have implemented a prototype cooperative database system at UCLA (see Figure 8) to validate the concepts presented in this chapter. Our modular design divides the control section of the relaxation manager from the individual cooperative operator modules. There is a module for each cooperative operator, as well as separate modules to handle general query modification and join queries. The join module coordinates the simultaneous relaxation in multiple hierarchies by alternating between the two (or more) hierarchies. CoBase uses LOOM [15, 16] as knowledge representation and inference system and supports relational data bases (e.g. Oracle and Sybase) and LIM (the Loom Interface Module) [17, 10]. Access to distributed databases is made possible through SIMS, a transparent multi-database access layer [1]. The CoBase engine, written in LISP, controls query relaxation and modification based on the user-provided relaxation constructs, the type abstraction hierarchy, and direction from the relaxation manager. The engine uses the RM to trim the type abstraction hierarchies for query relaxation as well as translating the relaxation constructs included explicitly in the query. The cooperative primitives presented have been implemented both in *CSQL* and in *CLOOM*, a cooperative extension to the LOOM query language.

When a query is presented to CoBase (see Figure 9), the system first relaxes any explicit cooperative operators in the query. The modified query is then presented to the underlying database system for execution. If no answers are returned, then CoBase, under the direction of the Relaxation Manager, relaxes the queries by query modification using the trimmed type abstraction hierarchy. The relaxed query is executed, and, if there is no answer due to over-trimming the type abstraction hierarchy, the relaxation manager will deactivate certain relaxation rules, restoring part of the trimmed TAH to broaden the search scope until an answer is found.

When an approximate answer is returned, the user may wish for an explanation of how the answer was derived. An explanation system is included in CoBase to present an annotated relaxation path to the user. When possible, the semantic nearness to the exact answer will be given for each approximation and the approximations will be ranked by their nearness to the exact answer.

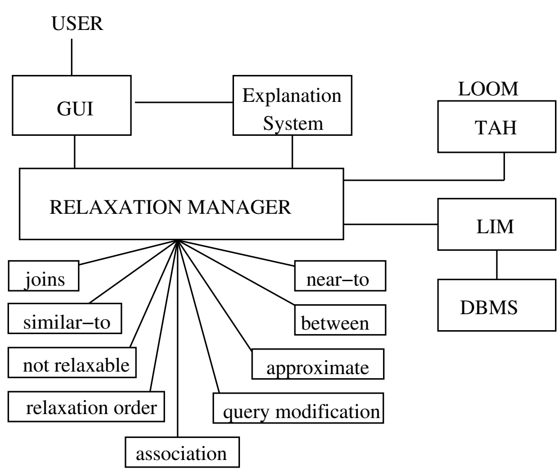

Figure 8: CoBase System Architecture

A graphical user interface (GUI) allows interactive control over system parameters and displays the TAHs and the relaxation path. The GUI is menu-based, providing control to allow explanations of the relaxation choices at the desired level of detail.

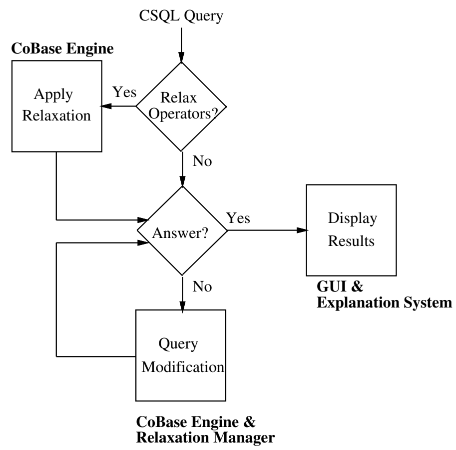

Figure 9: Data Flow in CoBase

5 Technology Transfer

We have demonstrated the feasibility and functionality of CoBase in several different domains, including a Transportation Planning Database [8] and a Medical Imaging Database System [2].

The transportation planning system includes information about military cargo, vehicles and locations throughout the world. CoBase can answer cooperative queries on this database, such as

Q_1 : List the airports with the parking capacity *approximately* equal to 200,000 square feet, and

Q_1 : Find the airports in Tunisia *similar to* the Bizerte airport. Use the attributes *runway_length_ft* (with weight 2.0) and *runway_width_ft* (with weight 1.0) as the criteria for similarity. Provide the best n answers.

In medical databases that store X–ray and MR images, images need to be retrieved by object feature or contents rather than patient ID. The queries asked are often conceptual and not precisely defined. We need to use knowledge about the application (e.g. age class, ethnic class, disease class, bone age etc.), user profile and query context to answer such queries. Further, exact matching of features is very difficult, if not impossible. For example, if the query "Find the treatment methods used for tumor x on 12-year-old Korean males," cannot be answered, based on the TAHs for tumors, age, and ethnic groups, we can relax "tumor x" to "tumor class X", and "12-year-old Korean male" to "pre-teen Asian," which results in the relaxed query, "Find the treatment methods used for the tumor class X on pre-teen Asians." Further, we can obtain such relevant information as the *success rate, side effects,* and *cost of the treatment* from the association operations.

The number and size of the type abstraction hierarchies for the transportation planning database are too large to be constructed manually. Thus we have devised methods for automatically constructing TAHs from the database directly. Pattern–Based Knowledge Inference [18] is a bottom-up method that uses rules derived from the database instance to cluster attribute values into a hierarchy. DISC [7] is a method for building hierarchies for numeric domains based on the distribution and frequency of data values. Together, these methods can generate the type abstraction hierarchies from a given database. Further, both of these methods provide a quantitative measure of the nearness for each node in the hierarchy. This measure can also be combined to derive the nearness distance for multiple hierarchies.

The CoBase engine is currently over 8,000 lines of LISP code, excluding user-interface code. The GUI consists of an additional 8,000 lines of LISP code. To measure performance, we have a sample test–suite of nineteen queries using different aspects of CoBase. The response time varies with the queries. On a Sun SPARCstation 10, response time ranges from a few seconds to nearly twenty seconds, with an average time of ten seconds per query. However, over 80% of that time is spent calling the underlying relational database and presenting the results. The overhead added by CoBase averages about 20% which is less than two seconds per query in the curent environment.

6 Conclusions

In this chapter, we proposed a cooperative query answering system that is structurally organized on the basis of the type abstraction hierarchy and functionally represented by the extended query language *CSQL*. Based on such a framework, type abstraction hierarchies express concepts at different knowledge levels and in various domains, and *CSQL* introduces additional language constructs to SQL that use these concepts for cooperative query answering. The interpretation of *CSQL* is based on the knowledge of type abstraction hierarchy and is transparent to the users. Thus, *CSQL* provides cooperative query answering not only by extending the functionality of *SQL*, but also by modifying the query evaluation process to include logic inferencing, rewriting and heuristic searching. These extensions are far beyond the ones in object oriented and relational databases.

A relaxation manager controls the system using user profile and query context to trim the type abstraction hierarchy and restrict the search space. An explanation system provides the user with reasoning for the relaxation path taken and the nearness of the approximate answers provided.

With the proposed cooperative query answering capability, a database system can provide approximate and conceptual query answering, tolerate imprecisely specified queries, and support object association. This approach can also be used to improve the availability of distributed databases during network partitions by providing relevant information when the exact information is inaccessible, and be used to support negotiation between multi-agents by determining the joint requirements that are relaxed from the the original ones made by these agents.

We have implemented a prototype cooperative database system, CoBase, and the extended query language *CSQL* at UCLA to validate the proposed concept. We have also applied CoBase to two domains – transportation planning and medical imaging. The type abstraction hierarchies for these domains are generated automatically from the databases. We find it is an essential stage for applying CoBase to large–scale problems. The explanation system is useful for informing the user about the relaxation process as well as providing the nearness measure of the approximate answer. Our experimental results reveal that our structured approach provides an efficient and scalable methodology for cooperative query answering.

References

[1] Y. Arens and C. Knoblock. Planning and reformulating queries for semantically–modelled multidatabase systems. In *Proceedings First International Conference on Information and Knowledge Management (CIKM)*, pp. 92–101, Baltimore, Maryland, 1992.

[2] W. W. Chu, A. F. Cardenas, and R. K. Taira. A knowledge–based multimedia medical distributed database system — KMeD. Technical Report 93–005, UCLA Computer Science Department, 1993.

[3] W. W. Chu and Q. Chen. Neighborhood and associative query answering. *Journal of Intelligent Information Systems*, 1(3/4):355–382, 1992.

[4] W. W. Chu and Q. Chen. A structured approach for cooperative query answering. to appear in IEEE Transactions on Knowledge and Data Engineering, 1994.

[5] W. W. Chu, Q. Chen, and A. Hwang. Query answering via cooperative data inference. *To appear in Journal of Intelligent Information Systems*, 1994.

[6] W. W. Chu, Q. Chen, and R. Lee. Cooperative query answering via type abstraction hierarchy. In S.M. Deen, editor, *Cooperating Knowledge Based Systems*, pp. 271–292. North-Holland, Elsevier Science Publishing Co., Inc., 1991.

[7] W. W. Chu and K. Chiang. A distribution sensitive clustering method for numerical values. Technical Report 93–0006, UCLA Computer Science Department, 1993.

[8] W. W. Chu, M. Merzbacher, and L. Berkovich. The design and implementation of CoBase. *SIGMOD '93*, pp. 517–522, May 1993.

[9] F. Cuppens and R. Demolombe. Cooperative answering: a methodology to provide intelligent access to databases. In *Proc. of the 2nd international conference on expert database systems*, 1989.

[10] T. W. Finin, D. P. McKay, and A O'Hare. The intelligent database interface. *Proceedings of the 7th National Conference on Artificial Intelligence (AAAI)*, 1990.

[11] T. Gaasterland, P. Godfrey, and J. Minker. An overview of cooperative answering. *Journal of Intelligent Information Systems*, 1(2):123–157, 1992.

[12] T. Gaasterland, P. Godfrey, and J. Minker. Relaxation as a platform for cooperative answering. *Journal of Intelligent Information Systems*, 1(3/4):293–321, 1992.

[13] A. Hemerly, M. Casanova, and A. Furtado. Cooperative behavior through request modification. Technical report, IBM Brasil, Brazil, May 1991.

[14] T. Imielinski. Intelligent query answering in rule based systems. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pp. 275–312, Washington, D.C., 1988. Morgan Kaufman.

[15] R. MacGregor. A deductive pattner matcher. In *Proceedings of the National Conference on Artificial Intelligence (AAAI 88)*, pp. 403–408, Cambridge, Mass, 1988. MIT Press.

[16] R. MacGregor. The evolving technology of classification-based knowledge representation systems. in Principles of Semantic Networks: Explorations in the Representation of Knowledge, 1991. J. Sowa, ed.

[17] D. P. McKay, J. Pastor, and T. W. Finin. View–concepts: Knowledge–based access to databases. In *Proceedings First International Conference on Information and Knowledge Management (CIKM)*, pp. 84–91, Baltimore, Maryland, 1992.

[18] M. Merzbacher and W. W. Chu. Pattern–based clustering for database attribute values. In G. Piatetsky-Schapiro, editor, *AAAI Workshop on Knowledge Discovery in Databases*, Washington, D.C., July 1993. AAAI Press.

[19] A. Motro. VAGUE: A user interface to relational databases that permits vague queries. *ACM Journal Transactions on Office Information Systems*, 6(3):187–214, July 1988.

[20] A. Motro. FLEX: A tolerant and cooperative user interface to databases. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):231–245, 1990.

[21] S. Su. Sam*: A semantic association model for corporate and scientific-statistical databases. *Information Sciences*, 29, 1983.