QUICK: A System that Uses Conceptual Design Knowledge for Query Formulation

R. D. Semmel
The Johns Hopkins University Applied Physics Laboratory
Laurel, MD 20723
rds@aplcen.apl.jhu.edu

Abstract

This paper describes how database design knowledge can be used to formulate queries in response to high-level requests. A prototype system, QUICK, is discussed that employs a knowledge-rich conceptual schema to identify semantically reasonable subgraphs that correspond to user intent. QUICK manipulates conceptual schema subgraphs to produce database queries. Furthermore, by treating the subgraphs as knowledge constructs, QUICK can be used for interactive design evaluation. The paper concludes with a description of QUICK's functional architecture.

1. Introduction

Designing a complex database is a knowledge-based activity that requires significant human effort. In particular, substantial resources must be expended in the creation of a high-level conceptual schema that is independent of the implementation data model. For example, an Entity-Relationship (ER) diagram often is used to represent high-level design knowledge that can be mapped to a relational database. However, once the mapping is achieved, the conceptual schema often is used only for documentation. Unfortunately, not all of the knowledge in the conceptual schema is represented in the implementation data model. Consequently, activities such as query formulation cannot be supported by high-level design knowledge that has been discarded in the transformation to the logical schema.

In this paper, a system that exploits conceptual schema knowledge is discussed. In addition to query formulation support, the system uses knowledge in the conceptual schema to provide feedback on the validity of a database design, for dynamic conceptual schema reinterpretation, and for semantic query optimization. In the next section, some related work is discussed. Then, in the third section, the notion of conceptual schema segmentation through the use of inferred contexts is described. In the fourth section, it is demonstrated how the segmented conceptual schema can be used to simplify query formulation. This leads to a discussion of database design feedback mechanisms in the fifth section. Finally, a functional architecture of the system is presented.

2. Related Work

Early in the database life cycle, a knowledge-rich conceptual schema is created using semantic data modeling techniques. The resulting schema then is mapped to an implementation-dependent logical schema. However, the logical schema is a sparse representation of the real world, and the burden for formulating queries over the database lies primarily with users and, for complex requests, with analysts. However, by using knowledge available in the conceptual schema, many queries previously requiring the support of analysts can be generated automatically.

Recognizing the complexity inherent in database design, a number of semantic data models have been proposed that provide intuitive means for representing design knowledge. The most popular of these models is the ER model [3], which represents classes of objects as entity types that can be related via relationship types. The relationship types express various integrity constraints such as mandatory or optional entity participation and cardinality ratios (e.g., many-to-many associations) [1]. While more expressive than the relational model, the basic ER model is not rich enough to model complex domains, such as those requiring inheritance. Thus, extended ER models have been proposed that incorporate knowledge representation constructs such as generalization and aggregation [12, 13].

Unfortunately, the ER model has been used primarily for database design. In those cases where the model has been used as the basis for a user interface, the emphasis has been on graphical representation rather than on how
the expressed ER design knowledge can be used to infer appropriate navigational paths \[14,15\]. Similarly, there has been little effort to use the model as the basis for interactive design evaluation.

Alternative semantic data models have been proposed that alleviate some of the deficiencies of the ER model and that integrate data manipulation languages \[6\]. However, these models often (a) rely upon constructs not often employed by database designers, or (b) do not map in a straightforward way to the relational model. Consequently, the alternative models have achieved only limited success in real-world applications.

From a database perspective, research has been conducted in simplifying the database interface by presenting a universal relation (UR) view of a normalized logical database design \[7,14\]. Such systems reduce the need for the user to be aware of the underlying logical schema. However, these systems also have relied upon abstractions (e.g., hypergraphs and maximal objects) that are not typically employed by designers. Moreover, the systems tend to impose strict design criteria that real-world systems cannot satisfy \[9\].

3. Inferring Context from a Conceptual Schema

To cope with the shortcomings identified in the previous section, a system has been constructed that focuses on automating query formulation by exploiting ER conceptual schema design knowledge. QUICK ("QUICK is a Universal Interface with Conceptual Knowledge") accepts high-level requests consisting only of attribute names, and generates relational queries. For example, consider the ER conceptual schema for a simple banking database shown in Figure 1. In this example, customers may have many accounts or loans, and accounts and loans may be joint. Similarly, an account or loan may be located at only one branch, though branches may have many accounts and loans.

The logical schema corresponding to the conceptual schema is shown in Figure 2. Notice that in the logical schema, attributes are duplicated to provide the value-based matching required for natural joins in the relational model. Furthermore, note that the semantics associated with whether a relation corresponds to an entity type, a relationship type, or both cannot be inferred from the logical schema. Similarly, the cardinality and participation constraints cannot be inferred. Such knowledge is used by QUICK to formulate queries.

Figure 1. Banking example conceptual schema.
To illustrate how QUICK works, consider a high-level request to list the account IDs and balances associated with a particular customer (the following notation is referred to as USQL because of its resemblance to SQL and its reliance upon a universal relation view of the logical schema):

```sql
SELECT Account-ID, Balance
WHERE Customer-Name = "E. Glazer"
```

In the above request, no relations or natural join criteria are specified. Thus, navigational paths that associate the requested attributes must be found. Rather than trying to do this at the relational level, QUICK attempts to identify semantically reasonable paths in the ER graph. For the example, QUICK deems the path from `ACCOUNT` to `CUSTOMER` through `CUSTOMER-ACCOUNT` reasonable and deems the alternative path through `LOAN` unreasonable; the method for achieving this is discussed below. Mapping the ER subgraph to the logical schema in Figure 2 results in the following SQL query:

```sql
SELECT account.account-id, account.balance
FROM account, customer-account, customer
WHERE customer.customer-name = "E. Glazer" AND account.account-id = customer-account.account-id AND customer-account.ssn = customer.ssn
```

When determining whether a path is reasonable, QUICK attempts to infer intent from a high-level request and map it to a corresponding ER subgraph. To achieve this, QUICK preprocesses the ER model to establish maximal subgraphs that correspond to strongly associated objects. These maximal subgraphs are referred to as contexts. Formally, a context is a connected, acyclic subgraph whose corresponding relations can be natural joined in a lossless way. The context diagram for the banking example is shown in Figure 3.
While contexts can be identified manually, QUICK can preprocess the ER conceptual schema to identify contexts automatically. The process proceeds inductively, where each relationship type and its participating entity types initially constitute a context. Then, each context is extended by following localized functional dependency chains (i.e., N:1 associations) while avoiding cycles. An M:N relationship type is included if it is not involved in a cycle; in the banking example, both M:N relationship types are involved in cycles. When no context can be extended further, the process terminates, and subsumed contexts (i.e., contexts that are subsets of other contexts) are eliminated.

To clarify the context generation process, the derivation of contexts for the banking example will be considered in greater detail. Initially, there are four contexts:

1. (ACCOUNT-BRANCH, ACCOUNT, BRANCH)
2. (LOAN-BRANCH, LOAN, BRANCH)
3. (CUSTOMER-ACCOUNT, CUSTOMER, ACCOUNT)
4. (CUSTOMER-LOAN, CUSTOMER, LOAN)

Neither Context 1 nor Context 2 can be extended, as this would require pushing through an M:N relationship type that is involved in a cycle. On the other hand, both Context 3 and Context 4 can be extended:

3'. (CUSTOMER-ACCOUNT, CUSTOMER, ACCOUNT, ACCOUNT-BRANCH, BRANCH)
4'. (CUSTOMER-LOAN, CUSTOMER, LOAN, LOAN-BRANCH, BRANCH)

At this point, neither Context 3' nor Context 4' can be extended. Moreover, as Context 3' subsumes Context 1, and Context 4' subsumes Context 2, Contexts 1 and 2 can be eliminated. Note that the eliminated contexts can be reconstituted by pruning the remaining contexts. Thus, the final set of contexts contains Context 3' and Context 4', as illustrated in Figure 3.

4. Query Formulation with Contexts

Given a set of contexts, query formulation is straightforward. First, all contexts that cover the requested attributes are identified. The found contexts then are iteratively pruned until all leaves in each context accounts for requested attributes. Resulting duplicate contexts are eliminated, and the explicit links in each ER subgraph are used to identify a relational natural join order. ER objects then are mapped to relation schemas, duplicate relation schemas are eliminated, and the final query is formulated. Note that if more than one pruned context remains, the final query may include a union of subqueries.

An example will clarify the query formulation process. Consider a request to list all customer name and branch name associations:

```sql
SELECT Customer-Name, Branch-Name
FROM branch, account, customer-account, customer
WHERE branch.branch-name = account.branch-name AND
      account.account-id = customer-account.account-id AND
      customer-account.ssn = customer.ssn
```

Both contexts in Figure 3 account for the requested attributes. As each leaf in both contexts accounts for requested attributes, no pruning is necessary. Suppose that after interacting with the user that both contexts are desired; intuitively, this is reasonable as the request was for all associations. Then, using the explicit ER links to produce a natural join order and mapping to relation schemas results in the following relation sequences:

R1: (BRANCH, ACCOUNT, ACCOUNT, CUSTOMER-ACCOUNT, CUSTOMER)
R2: (BRANCH, LOAN, LOAN, CUSTOMER-LOAN, CUSTOMER)

Upon eliminating duplicate relation schemas, the following sequences are produced:

R1': (BRANCH, ACCOUNT, CUSTOMER-ACCOUNT, CUSTOMER)
R2': (BRANCH, LOAN, CUSTOMER-LOAN, CUSTOMER)

The last two sequences are used to formulate the final query:

```sql
SELECT customer.customer-name, branch.branch-name
FROM branch, account, customer-account, customer
WHERE branch.branch-name = account.branch-name AND
      account.account-id = customer-account.account-id AND
      customer-account.ssn = customer.ssn
```
UNION

SELECT customer.customer-name, branch.branch-name
FROM branch, loan, customer-loan, customer
WHERE branch.branch-name = loan.branch-name AND
loan.loan-id = customer-loan.loan-id AND
customer-loan.ssn = customer.ssn

As a second example, consider a request to list all customer SSNs:

Select SSN

As in the previous example, both contexts account for the requested attributes. After pruning however, both contexts are identical, consisting only of the ER object CUSTOMER. Thus, one context is eliminated, the remaining context is mapped to a relation schema sequence, and the final query is produced:

SELECT customer.ssn,
FROM customer

Several UR approaches have difficulty with this last request because of the ambiguity associated with the duplication of the attribute SSN in several relation schemas (i.e., CUSTOMER, CUSTOMER-ACCOUNT, and CUSTOMER-LOAN) [8]. However, by exploiting the fact that attributes are not repeated in an ER conceptual schema, the query is generated in a straightforward manner by QUICK.

While formulating contexts automatically is expensive (in the worst case it is an exponential function of the number of entity types), it need be done only when the ER conceptual schema is designed or modified. In addition, heuristic methods have been devised that typically allow processing of large ER graphs (i.e., graphs with more than 50 ER objects) to occur in minutes or, sometimes, seconds [10]. Furthermore, given a set of contexts, query formulation is efficient. Specifically, query formulation typically takes on the order of one to two seconds over complex databases.

5. Database Design Feedback

Experience with QUICK has indicated that the context generation algorithm usually finds conceptual schema subgraphs that are consistent with a designer's intent. However, there are cases in which the contexts produced do not coincide with intent. In such cases, it is possible to handcraft a set of contexts. This can be done from scratch or, more often, by modifying the original generated set. Before handcrafting, however, it is worthwhile to ensure that the conceptual schema is consistent. Unfortunately, this may not be an easy task.

The difficulty in evaluating and validating a conceptual schema stems from the potentially large number of ER objects in the schema as well as the potentially high degree of connectivity among objects. Contexts abstract the ER conceptual schema into overlapping subgraphs that correspond to sets of relations that can be natural joined in lossless ways. Thus, rather than having to explore ER objects and links individually to evaluate a schema, it is possible to focus on a smaller number of individual contexts. Often, inconsistencies in a context indicate inconsistencies in the underlying conceptual schema.

Unfortunately, contexts, themselves, may be large and complex, and, therefore, difficult to evaluate. Furthermore, it is difficult to quantify designer intent; thus, automatically discovering all inconsistencies in either contexts or the conceptual schema is not feasible. However, by interacting with QUICK, it is possible for a designer to evaluate the system's interpretation of a conceptual schema. One way to achieve this is by entering high-level requests and determining if the generated queries correspond to likely intent. If they do, then the design probably is acceptable. On the other hand, if the queries generated do not correspond to a designer's intentions, then there may be problems in the underlying design that need to be corrected.

Unfortunately, contexts, themselves, may be large and complex, and, therefore, difficult to evaluate. Furthermore, it is difficult to quantify designer intent; thus, automatically discovering all inconsistencies in either contexts or the conceptual schema is not feasible. However, by interacting with QUICK, it is possible for a designer to evaluate the system's interpretation of a conceptual schema. One way to achieve this is by entering high-level requests and determining if the generated queries correspond to likely intent. If they do, then the design probably is acceptable. On the other hand, if the queries generated do not correspond to a designer's intentions, then there may be problems in the underlying design that need to be corrected.

The query evaluation strategy described above was used with the Data Archive and Distribution System (DADS), which is a large database system being developed by NASA to support space scientists using Hubble Space Telescope data [5]. DADS is partitioned into three components, of which the primary component is the Catalog section. The Catalog section contains more than 70 ER objects that map to more than 30 relation schemas.

Queries over DADS tend to be long and complex. For example, it is not uncommon to pose requests that require joins of more than five relations. Consequently,
relatively few queries were considered in the evaluation of the DADS design. With QUICK, formulating queries is simplified. Specifically, queries that required on the order of 50-100 lines of SQL typically required less than five to ten lines of USQL. As a result, more queries can be formulated, and the designers are able to focus their analysis on specific portions of the conceptual schema.

For DADS, most requests made by analysts resulted in the generation of semantically reasonable queries by QUICK. However, some requests resulted in queries that were inconsistent with intent. For example, one request resulted in an SQL query that was several hundred lines in length and that contained five subqueries. Looked at individually, each subquery made sense; however, as a union they did not. The problem stemmed from the fact that the conceptual design linked data engineering data with scientific data. The generated query highlighted this fact, and allowed the designers to identify necessary modifications to the conceptual schema. When implemented, these changes affected the generated set of contexts and resulted in an appropriate query.

6. QUICK Architecture

Figure 4 shows the functional architecture of QUICK. As described above, query formulation begins with the input of a high-level request that specifies the attributes of interest. This request can assume many forms (e.g., USQL, natural language, direct manipulation), though in all cases the output of the request processor will be the essential terms and conditions characterizing the request. The terms and conditions then are passed to the conceptual schema processor, which generates a query based on the specified requirements.

Figure 4. QUICK architecture.
schema processor for subgraph selection and query construction.

The conceptual schema processor performs several tasks. First, the logical schema is generated, and mappings are established among ER and relational objects; this is done in a preprocessing mode prior to accepting query requests. Then, from the semantic integrity constraints inherent in the conceptual schema as well as from logical integrity constraints that can be inferred from the generated logical schema, contexts are created and adjustments are made to the ER conceptual schema to create an internal representation. As with logical schema generation, this is done in a preprocessing mode.

Generating final queries requires knowledge-based processing that entails interaction among several modules. First the internal representation is used to identify contexts, relation and attribute objects, and special-case name transformations. The contexts then are reduced to essential conceptual schema subgraphs that cover the set of requested attributes in semantically reasonable ways. From these inferred semantic query components, specific term syntax is derived from the logical schema and corresponding relation and attribute objects. This information, in conjunction with target query language knowledge, is provided to the query generator. The query generator transforms its inputs into an internal representation (e.g., augmented relational algebra), combines query components as appropriate, and composes the final target language query.

7. Summary and Conclusions

Formulating queries is a difficult problem that requires knowledge of the application domain as well as knowledge of the underlying database design. This paper has provided an overview of QUICK, which is a system that uses ER conceptual schema knowledge to formulate queries in response to high-level requests. QUICK accepts requests that specify only attributes of interest; join criteria and relation qualifiers are inferred by the system. By employing the ER conceptual schema, QUICK reuses abstractions that typically would have been employed only early in the database life cycle. Moreover, by aggregating ER objects into contexts, QUICK supports an enriched ER model that facilitates the evaluation of complex designs. Consequently, through an iterative process of conceptual schema modification and context adaptation, an appropriate high-level design can be developed.

QUICK also supports features that extend basic query formulation capabilities. For example, QUICK semantically optimizes ER subgraphs so that resultant queries use the minimum number of joins possible. Significantly, only a small amount of processing time is required for this optimization. In addition, QUICK allows the user to specify ER objects to ignore in a request, thus enabling dynamic reconfiguration of the ER conceptual schema. However, difficult interface issues must be resolved before this strategy can be used in an ad hoc manner.

Currently, there are several extensions to QUICK being explored. For example, extended ER constructs now are supported by the conceptual schema processor. These extended constructs enable QUICK to model more complex application domains. However, issues in automatic context generation must be resolved before the constructs can be fully exploited. Similarly, knowledge-based extensions to the context abstraction are being explored. For example, contexts are being extended to support arbitrary predicates. Such predicates can be used by QUICK to constrain the set of contexts considered for selected requests. These enhancements will facilitate the development of more powerful query formulation and design capabilities.

8. References


