

S.E.A.L. – A Query Language for Entity-Association Queries

Edward Stanley*, Pavle Mogin[†], Peter Andreae[†]

* Kakapo Technologies

32a Salamanca Road, Wellington 6140, New Zealand

[†] School of Mathematics, Statistics, and Computer Science

Victoria University of Wellington

PO Box 600, Wellington 6140, New Zealand

eddiwould@paradise.net.nz, Pavle.Mogin/Peter.Andreae@mcs.vuw.ac.nz

Abstract

The paper presents the S.E.A.L. query language and interpreter for entity-association queries that allows such queries to be expressed in a much simpler way than in SQL. S.E.A.L. (Simplified Entity Association Language) also supports Entity-Attribute-Value (EAV) data structures, enabling users to write queries without having to know whether a particular attribute is a regular attribute or an EAV attribute. The language allows parts of a query to be omitted if they are implied by the rest of the query, and the interpreter will infer the missing requirements, or ask the user to resolve the ambiguity. S.E.A.L. makes it easier for users of databases, particularly science and eCommerce databases, to make use of the valuable information in their databases.

Keywords: Databases, Query Language, EAV, Query Inference

1 Introduction

Many database users in fields such as Medicine, Biology, and Genetics have large collections of data which contain important information. This information would be valuable to their owners, if only it could be extracted from the data. A similar situation exists in the case of many eCommerce databases. However, many users of these databases have only modest database expertise and use very simple SQL or query-by-example (QBE) tools such as Microsoft Access. But QBE tools are limited in the kinds of queries they support, and therefore users are unable to exploit all the information hidden in their data. This paper presents S.E.A.L. (the Simplified Entity Association Language) – an extension to SQL to enable such users to extract information more easily.

A common class of queries involves finding all instances of an entity-type which satisfy some constraints involving participation in relationships with other entities. An example of this kind of query would be finding all the patients in a medical database with a set of heart related symptoms who were closely related to someone who had been diagnosed with a particular disease. These queries, called entity-association queries, are common but are

difficult to express in SQL. The difficulty arises because users must know the implementation schema in detail. Particularly, users need to know table names, the location of attributes, and table primary and foreign keys; simply knowing the conceptual schema (problem domain) is insufficient. Additionally, users need advanced specialist knowledge of SQL in order to declare explicit joins and constraints on entity properties in terms of conditional expressions.

Entity-Association queries become even more complex if a query contains a conjunctive condition on two different values of the same attribute, or if some of the query attributes are stored in Entity-Attribute-Value (EAV) database structures. In both cases, the number of nesting levels may increase, and nesting makes queries difficult to construct and harder to debug. (Such queries may prove to be complex even for database professionals.)

If an entity-association query contains a conjunctive constraint on two different values of the same attribute, e.g. “retrieve students who passed both Database Systems and Operating Systems courses”, expressing it in SQL requires finding the intersection of entity sets satisfying each of the constraints alone. Such queries are nested by default. Increasing the number of entity and relationship types involved in the query may also increase the number of nesting levels in the SQL statement.

Entity-Attribute-Value database structures are used for efficiently storing sparse data and for allowing user-defined attributes. There are a number of variants of EAV database structures, but they all share a common characteristic: meta-data regarding attribute names is stored in database tables like other common data. Syntactically, EAV structures can be considered as entity-association structures, and therefore even conceptually simple queries involving EAV database structures share all the complexity of other entity-association structures, requiring detailed knowledge of the database schema, and often requiring complex nested query structures. The need for EAV structures is very common in Medicine, Biology, Genetics, Chemistry, and generic e-Commerce web database applications. For example, in the medical database above, most patient symptoms would need to be stored in an EAV structure because there would be too many sparsely populated possible symptoms to use regular attributes.

As a solution to these problems, S.E.A.L. offers a highly declarative syntax for expressing entity-association queries in a natural way that is closer to the conceptual than to the implementation level of the

database abstraction. Additionally, S.E.A.L. possesses inference mechanisms that allow information about some of the database structural concepts to be omitted from the query. S.E.A.L.'s inference mechanism relies on a disciplined database schema approach. Thanks to this design approach, S.E.A.L. accepts queries having only one entity type name, a list of result attributes, and conditional expressions on attribute values, and produces a corresponding SQL expression or warns the user that the query specification is insufficient for an unambiguous translation of the query. In the course of the query translation, S.E.A.L. infers all necessary associated entity types, relationship types, and entity type roles, and generates a SQL query in terms of the underlying relational implementation structure.

The paper describes:

- A set of rules and guidelines for designing databases compatible with S.E.A.L.,
- The S.E.A.L. declarative language for specifying entity-association queries, which also supports EAV structures,
- A prototype interpreter for S.E.A.L. queries.

Section two of the paper reviews related work. Sections three and four introduce an example database schema, and briefly discuss EAV database structures. Section five analyses a simple entity-association query and justifies the claim that such queries are complex to define. Sections six and seven introduce the S.E.A.L. syntax and briefly describe the design of the S.E.A.L. interpreter. Section eight reports on the results of a limited number of performance measurements and the final section presents conclusions and ideas for future work.

2 Related Work

Because of its inference abilities, S.E.A.L belongs to the class of the database query languages with a Universal Relation Schema Interface. According to the Universal Relation Assumption (Ullman 1982), there exists a hypothetical relation schema (URS) which contains all attributes of a universe of discourse (problem domain). An actual database schema is produced by decomposing the URS. A URS query interface allows a user to define queries solely on attributes without having to care about real database objects like tables.

One of the first research projects on a query language with URS user interface was developed within an experimental database management system called System/U (Ullman 1982). Ullman describes a query interpretation algorithm and two ways to cope with ambiguities induced by database cyclic structures, but does not discuss interpretation of queries having conjunctive conditional expressions on different values of an attribute.

The Query-By-Example (QBE) language is a graphical query language developed by IBM Research and is available as a part of the Query Management Facility (Elmasri and Navathe 2006). It is also embedded into Microsoft Access. A query is formulated in QBE by filling in table templates that are displayed on the screen. Users drag and drop tables and set predicate conditions to construct a query. The QBE engine takes care of issues

such as aliasing and generating join conditions. QBE relieves a user from having to know the structure of the underlying database, but it is still not a URS interface language, since it builds queries using tables. Also, QBE engines that we have tested have been unable to produce queries involving conjunctive conditional expressions on different values of an attribute (except by modifying the database structure).

Entity Attribute Value (EAV) database structures are important for this paper since queries against EAV database structures belong to the class of entity-association queries. EAV structures are described in Nadkarni and Brandt (1998), Dinu and Nadkarni (2006), and Corwin *et al* (2007).

We identified two systems which abstract EAV attribute representation from a database user. The first, ACT/DB (Nadkarni *et al.* 1998), is a database tool for managing clinical trial data. It uses a client/server architecture with Oracle7 at the backend. Users construct queries with a GUI-based tool written in Microsoft Access. The tool uses Visual Basic code to handle the abstraction of EAV attributes in queries and translation into SQL to be executed at the backend. The tool relies upon the specific schema for which it was designed. The schema includes conventional tables as well as six general purpose EAV tables for the various data types supported. ACT/DB supports a number of comparison operators as well as aggregate functions such as average and standard deviation.

The second is QAV (Nadkarni 1996) which is a GUI-based tool which allows users to perform queries against the Columbia MED dataset, a large medical metadata repository. QAV uses a special schema in which all data is represented in EAV form. QAV is also based on client-server architecture.

Both of these systems are tied to a particular schema. A significant advantage of S.E.A.L is that it can be used with any schema which conforms to the rules and guidelines given in section 6.3 of this paper.

During the literature search, we found no previous attempts to classify entity-association queries or implement a general solution for allowing users to express these queries in an easier way.

3 Example Database Schema

To illustrate entity-association queries in a way that requires no specialized domain knowledge and to explain the proposed language and interpreter, the paper uses a running example of a fictional 'robbers' database. The 'robbers' database contains a variety of structures that require defining entity-association queries.

The *robbers* database describes robbers and banks they have robbed. Robbers have certain skills that are tested at special testing locations and some robbers may have special features such as haircuts, or like particular kinds of music. Figure 1 contains the conceptual 'robbers' database schema in the form of an entity-relationship diagram, which, for simplicity, is presented with no attributes.

Figure 2 contains a relational schema that corresponds to the ER diagram in Figure 1, giving the schema name and the set of attributes, with the primary key underlined.

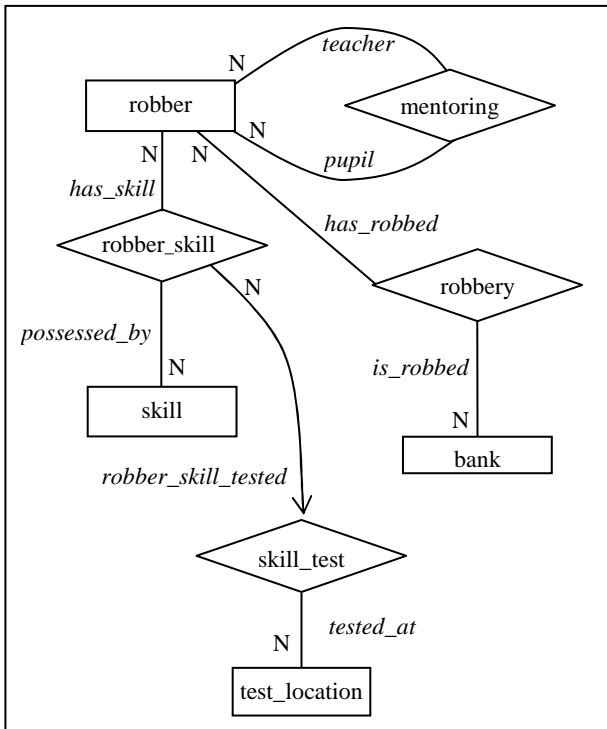


Figure 1: ER diagram of the *Robbers* database.

Relation Schemes:

```

robber{robberid, nickname, age},
robber_attributes{attributeid, attribute},
robber_eav{robberid, attributeid, value},
mentoring{robberid1, robberid2}
robber_skill{robberid, skillid, skilllevel},
skill{skillid, skillname},
skill_test{robberid, skillid, locationid },
test_location{locationid, locationname},
bank{bankid, bankname},
robbery{robberid, bankid, date, amount}

```

Referential Integrity Constraints:

```

robber_eav[robberid] ⊆
  robber[robberid] (eav_robber),
robber_eav[attributeid] ⊆
  robber_attributes [attributeid] (eav_skill),
mentoring[robberid1] ⊆ robber [robberid] (teacher),
mentoring[robberid2] ⊆ robber [robberid] (pupil),
robber_skill[robberid] ⊆
  robber [robberid] (has_skill),
robber_skill[skillid] ⊆ skill [skillid] (possessed_by),
skill_test[(robberid, skillid)] ⊆
  robber_skill[(robberid, skillid)] (rob_skill_tested),
skill_test[locationid] ⊆
  test_location[locationid] (tested_at),
robbery[robberid] ⊆ robber [robberid] (has_robbed),
robbery[bankid] ⊆ bank [bankid] (is_robbed)

```

Figure 2: Implementation Schema

For simplicity, all relation schemes have only one key — the primary key — and a small number of attributes

The referential integrity constraints have the form $N_1[FK] \subseteq N_2[PK]$ (*role*), where *FK* is the foreign key corresponding to the primary key *PK* of N_2 , and *role* is the role of the entity type N_2 in the relationship type N_1 .

Sparse robber attributes like *haircut*, *music*, and *call-sign* are stored in EAV database structures, represented by *robber_attributes* and *robber_eav* relation schemes. EAV database structures are described in the next section.

4 EAV Database Structures

Under a conventional relational database design, each entity instance is represented by a single row in the table representing that entity type. This row has a fixed number of columns and each column stores an attribute value. Each attribute is described by the name of the column and the data type. As a consequence of this, all entity instances of the same entity-type contain values of the same set of attributes. Metadata on the number, names and types of attributes an entity-type stores information on is defined by the table structure.

An Entity Attribute Value (EAV) database structure is an alternative method for representing the attributes belonging to an entity type. EAV represents each entity instance as a set of (*entityid*, *attribute*, *value*) triples (Nadkarni and Brandt 1998). The *entity* describes the entity, the *attribute* value carries the information about the attribute name, while the *value* assigns data to the attribute name. Under EAV, metadata is represented as data: not only are the values of attributes data (as they were in the conventional relational model) but the attribute names are also data.

EAV tables are generally not shared between entity types: that is, each entity type making use of EAV storage is assigned a separate EAV table. A useful variation to the described structure is to also have a separate lookup table to record the names of attributes stored in EAV form. Often the attribute lookup table will have a surrogate integer primary key (*attributeid*) and the EAV table stores a reference to this field.

EAV database structures are used in applications that handle sparse data, or have a need for user defined attributes. They can also be used to store multivalued attributes.

4.1 Sparse Attributes

Conventional “one fact per column” table designs are unsuitable for extremely sparse data. A common example of sparse data is a patient record in a medical database. While there may be thousands of possible facts that can apply to a single patient record, the number that typically applies may be only a few dozen (Nadkarni and Brandt 1998). Unlike conventional relational tables which set aside space for each attribute whether it is null or not, EAV only represents facts which apply to the given entity instance. EAV can be used in combination with conventional storage: data which applies to every instance can be stored in a conventional table and the sparse data can be represented in an EAV table (Dinu and Nadkarni 2006).

4.2 User Defined Attributes

In certain applications, as in generic e-commerce databases, there is a need for users to be able to define (and destroy) attributes as part of a normal usage. Under a conventional schema this would involve acquiring an exclusive lock on the table and then issuing data definition language (DDL) instructions to alter the table structure. In a busy high-volume database, it may not be possible to acquire an exclusive lock or it may be too detrimental to performance. Allowing the user/application to perform DDL is also a significant security risk.

Using an EAV design, the attributes for an entity-type are described as regular data, and therefore only regular row INSERT/DELETE operations are required to add and remove attributes respectively. This design removes the security risk associated with user-defined attributes and also removes the need to acquire an exclusive lock on the whole table, making user-defined attributes feasible even in a high-volume database.

4.3 Multivalued Attributes

In the conventional normalized relational databases, multivalued attributes are stored using separate tables. For example, the multivalued attribute robber's *skills* is represented this way in the schema of Figure 2. An EAV database structure is another possible solution for representing multi-valued attributes, if the primary key of the EAV table is set to be (entityid + attributeid + value).

Example 1. In the example *robbers* database, dense robber attributes *nickname* and *age* are stored in the conventional table *robber*, and sparse attributes like *haircut*, *music*, and *call-sign* are stored in EAV database structures. Figure 3 presents instances of the *robber_attributes* and *robber_eav* EAV relation schemes. The *robber_attributes* instance lists the sparse attributes *haircut*, *music*, and *callsign*, and the *robber_eav* instance specifies the association between robbers and attribute values.

<i>robberid</i>	<i>nickname</i>	<i>Age</i>
1	Al Capone	31
2	Bugsy Malone	23
3	Lucky Luchiano	55
4	Anastazia	47
5	Dutch Schulz	63

<i>attributeid</i>	<i>attribute</i>
1	haircut
2	music
3	callsign

<i>robberid</i>	<i>attributeid</i>	<i>value</i>
1	1	Mohawk
4	1	Mohawk
4	2	Latin
4	2	Classic

Figure 3

Using the combination of the following two SQL statements, a user may insert a new sparse attribute *food* and assign a value 'Pizza' to the *food* attribute of a robber:

```
INSERT INTO robber_attributes VALUES
(4, 'food');
INSERT INTO robber_eav VALUES (2, 4,
'Pizza');
```

The structure composed of tables *has_skill* and *skill* is another EAV structure that is used to represent the multivalued attribute *skill* in the *robbers* database. Here, the attribute *skilllevel* in the *robber_skill* table associates values of the attribute *skill* to robbers.

A significant problem with EAV structures is that the user has to know whether an attribute is stored as a regular column or in an EAV table (Nadkarni and Brandt 1998). This problem adds to the complexity of writing SQL queries against EAV database structures. As it will be shown in the following sections, thanks to its inference capabilities S.E.A.L. hides this complexity from users.

5 Analysis of an Entity Association Query

If an entity association query contains two or more conjunctively bound constraints on the same attribute, SQL does not have a declarative syntax for expressing this class of queries in a simple way. In principle, such queries require finding the intersection or set difference of entity sets satisfying each of the constraints alone. We are aware of four main strategies for expressing such entity-association queries in SQL. These are:

1. Multiple SELECT statements combined with the set operators,
2. Nested SELECT queries using the IN operator,
3. Correlated nested SELECT queries using EXISTS operators, and
4. Multiple nested SELECT queries where the intersection is performed using an equi-join.

Example 2. The following is a relatively simple entity-association query for the *robbers* database:

"Find the robbers who have the skill 'Gun Shooting' but do not have the skill 'Money Counting', and the robbers who have the skill 'Explosives'".

Using the first approach, this query could be broken down into three SELECT queries and then combined with the set operators UNION, INTERSECT and EXCEPT:

```
SELECT nickname FROM robber NATURAL
JOIN
((SELECT robberid FROM robber_skill
NATURAL JOIN skill WHERE
skillname='Gun Shooting'
EXCEPT
SELECT robberid FROM robber_skill
NATURAL JOIN skill WHERE
skillname='Money Counting')
UNION
SELECT robberid FROM robber_skill
NATURAL JOIN skill WHERE
skillname='Explosives') AS foo;
```

Using the second approach, the query could be expressed using nested IN statements:

```

SELECT nickname FROM robber WHERE
((robberid IN (SELECT robberid FROM
  robber_skill NATURAL JOIN skill
  WHERE skillname='Gun Shooting'))
AND
(NOT (robberid IN (SELECT robberid
  FROM robber_skill NATURAL JOIN
  skill WHERE skillname='Money
  Counting'))))
OR
(robberid IN (SELECT robberid FROM
  robber_skill NATURAL JOIN skill
  WHERE skillname = 'Explosives'));

```

Using the third and fourth strategies results in similarly complex nested SQL expressions.

We argue however that a query constructed using either of these strategies poorly reflects the logical intent of the query and because of this is difficult to write. Given the intent of the query, it will not be clear to a user why they would need to issue multiple select statements. From the user's perspective, the relationships in which an entity participates can be viewed as a property of the entity itself. The following relation schema probably better represents the user's understanding of the domain:

```

robber{robberid, nickname,
  has_gun_shooting_skill,
  has_money_counting_skill,
  has_explosives_skill}

```

Example 3. With this schema, the query could be expressed straightforwardly as:

```

SELECT nickname FROM robber WHERE
(has_gun_shooting_skill AND NOT
has_money_counting_skill )
OR has_explosives_skill;

```

While this database design is flawed (it will not scale to hundreds of possible skills), it is easy to see that this SQL reflects the user's intention more simply and more directly than the complex queries above.

Furthermore, all of the strategies described lead to queries which are unnecessarily difficult to write. Factors contributing to this difficulty include the following:

1. Users need to know where an attribute is stored. Is it stored in a regular column or as an EAV attribute?
2. Users need to construct join conditions explicitly. For this trivial example, NATURAL JOIN suffices but, in the case of joins involving a recursive relationship type, INNER JOIN has to be used.
3. When using INNER JOIN, a user needs to know the primary keys of tables representing the entities involved in the relationship as well as the primary and foreign keys of the tables which represent the relationship. In more complex examples, these keys may be composite, involving three or more attributes.
4. With nested associations, multiple table aliases must be used. These aliases must be uniquely named and knowing which alias to refer to is a source of confusion.
5. Finally, if the query contains a conjunctive constraint on the same attribute, the query has to find the intersection of entity sets satisfying each of the

constraints alone. Such queries are complex: hard to define and even harder to debug.

This analysis applies equally to queries on EAV structures, since the *entity_eav* relation schema corresponds to a relationship type and the *entity_attributes* relation schema corresponds to an entity type.

Example 4. Consider the following query against the *robbers* database:

“Find robbers who like ‘Latin’ and ‘Classic’ music.”

The query constraints ask for two values of the multivalued sparse attribute *music*, which is stored in the EAV structure. The following SQL query uses the SELECT ... IN ... approach:

```

SELECT nickname FROM robber WHERE
(robberid IN (SELECT robberid FROM
  robber_eav NATURAL JOIN
  robber_attributes WHERE attribute
= 'music' AND value = 'Latin'))
AND
(robberid IN (SELECT robberid FROM
  robber_eav NATURAL JOIN
  robber_attributes WHERE attribute
= 'music' AND value = 'Classic'));

```

6 S.E.A.L. Design

This section describes the design of the S.E.A.L. system, including the syntax of the S.E.A.L. query language, the requirements on the design of the database, and the S.E.A.L. interpreter, particularly the inference algorithms S.E.A.L. uses to infer entity and relationship types left implicit in a query.

6.1 S.E.A.L. Syntax

The syntax of the S.E.A.L. query language is inspired by the SQL SELECT syntax, but is specialised for entity-association queries. A S.E.A.L. query specifies a set of attributes and a base entity type, along with two kinds of constraints: constraints on the attributes of the base entity (regular or EAV attributes), and constraints on entities associated with the base entity. A S.E.A.L. query has the following form:

```

Query ::=
SELECT
  attribute [...] | *
FROM
  baseEntityType
  [ '['
    AttributeConstraintExpression
  ']' ]
  [ AssociationExpression ]

```

An *AttributeConstraintExpression* is a logical expression (which may contain conjunction, disjunction, negation, and parentheses) involving *AttributeConstraint(s)* which specify constraints on regular or EAV attributes of the base entity type:

```

AttributeConstraint ::=
attribute ( '=' | '!=' | '<' | '>' |
  '>=' | '<=' )value

```

An AssociationExpression is a logical expression involving Association(s), each of which specifies an associated entity type, the relationship type by which the entity type is associated with the base entity type, the roles in the relationship type of the base entity type (“VIA”) and associated entity type (“AS”), followed by constraints on the relationship and associated entity types. The entity type, relationship type and roles are all optional if they can be unambiguously inferred from the rest of the query.

```
Association ::=
  'ASSOCIATED_WITH' '('
    ['VIA' EntityRole ]
    [AssociatedEntityType
      ['AS' AssociatedEntityRole ]
      ['THROUGH' AssociatingRelationship ]
    ;']
    AssociationConstraintExpression
  ')'
```

An AssociationConstraintExpression is a logical expression involving Association Constraints, each of which specifies constraints on the associating relationship type and/or the associated entity type, possibly including nested Association Expressions to constrain the entity or the relationship type.

```
AssociationConstraint ::=
  '<' AttributeConstraint [, ...] '>'
  [AssociationExpression ]
```

The attribute constraints in an association constraint can refer to any attribute (regular or EAV) of either the associating relationship type or the associated entity type of the Association.

6.2 Example Queries in S.E.A.L. Syntax

The following examples use the *robbers* schema given in Section 3. All examples follow the same structure: the aim of the example is given first, it is followed by the query, then the S.E.A.L. syntax, and finally by an optional comment.

Example 5. The following demonstrates uniform access to attributes (whether EAV or regular ones):

“Find robbers who are 39 years old and have a ‘Mohawk’ haircut.”

```
SELECT nickname FROM robber [haircut
= 'Mohawk' AND age = 39]
```

The user does not need to know that *haircut* is represented in EAV form, while *age* is a regular column.

Example 6. The following demonstrates how a simple entity-association query is composed in S.E.A.L.:

“Find robbers who have the skill ‘Lock-Picking’ and who have the skill ‘Planning’.”

```
SELECT nickname
FROM robber ASSOCIATED_WITH(
  skill THROUGH robber_skill,
```

```
<skillname = 'Lock-picking'> AND
<skillname = 'Planning'>)
```

The base entity-type is restricted based on the participation constraint specified in the ASSOCIATED_WITH clause. The associated entity-type, *skill*, is specified explicitly as is the relationship type *robber_skill*. The specification of roles is not needed, since the relationship type *robber_skill* associates only the *robber* and *skill* entity types. Two specifications of attribute constraints are used in the query which are combined into an expression with the AND operator.

Example 7. The following demonstrates the role of inference in S.E.A.L.:

“Find robbers who robbed the ‘Loan Shark’ bank.”

```
SELECT nickname FROM robber
ASSOCIATED_WITH(<bankname =
'Loan Shark'>)
```

In this query, the associated entity type, the relationship type, and the roles of the both entity types were all omitted. With regard to the *robbers* schema, given in Section 3, S.E.A.L. is able to infer that the specification *<bankname = 'Loan Shark'>* refers to the associated entity type *bank* through the relationship type *robbery*. The corresponding roles are also inferred.

Example 8. The following shows how an entity-association query, where the relationship type has a participation constraint, is constructed in S.E.A.L.:

“Find robbers with the skill ‘Guarding’ who had it tested at a testing location called ‘Harvard’.”

```
SELECT nickname FROM robber
ASSOCIATED_WITH(skill,
<skillname = 'Guarding'>
ASSOCIATED_WITH(test_location,
<locationname = 'Harvard'>))
```

S.E.A.L. interprets the query through the following two main steps. First, it infers the association between *robber* and *skill* tables through the *has_skill* table using the attribute *skillname*. Next, it infers the association between *has_skill* and *test_location* tables through the *skill_test* table using the attribute *locationname*. As a result, S.E.A.L. constructs a SQL statement with two levels of nesting.

Example 9. The final example is a query where roles cannot be inferred and must be specified:

“Find robbers who have been taught by ‘Bugsy Malone’.”

```
SELECT nickname FROM robber
ASSOCIATED_WITH(robber
AS teacher THROUGH mentoring,
<nickname= 'Bugsy Malone'>)
```

In this example, the role of the associated entity type (also a *robber*) was specified as a *teacher*. It was necessary to specify at least one role, because the *mentoring* relationship involves two entities of the same type (a *robber* acting as a *teacher* and a *robber* acting as a *pupil*).

6.3 Rules and Guidelines for Implementing S.E.A.L. Compatible Databases

The S.E.A.L. interpreter has a set of rules and conventions to which a schema must adhere if it is to be used with S.E.A.L. These rules and conventions provide a consistent way to describe metadata as well as simplifying the interpreter code by reducing the number of special cases needed. These rules follow a common disciplined database design approach and are illustrated by the *robbers* schema in Figure 2. The most significant guidelines are the following:

- All attributes that have different meaning (whether stored conventionally or in EAV form) must have distinct names.
- EAV database structures should be implemented using an attribute lookup table, named $\langle entity \rangle_attributes$ and a relationship table named $\langle entity \rangle_eav$ with the structure $(entityid, attributeid, value)$, where $\langle entity \rangle$ is the name of the entity type the attributes belong to.
- The referential integrity (foreign key) constraints have to be named after the name of the role the referenced table plays in the relationship type.

These, and the other requirements, all represent good database design practices, and are not difficult to satisfy. We note that the first requirement is needed to avoid ambiguities induced by cyclic database structures (Ullman 1982).

6.4 Inference Algorithms

S.E.A.L. gives users considerable flexibility when specifying a participation constraint via an ASSOCIATED_WITH clause. While the base entity-type and an expression constraining attribute values are required, the associated entity type, the relationship type, and the roles of both the base and associated entity types are optional.

If any of these types or roles is omitted, S.E.A.L. will attempt to infer them based on the attributes used in the specification expression. To be able to perform translation without ambiguity, S.E.A.L. needs to find exactly one valid $(relationship\ type, associated\ entity\ type, base\ entity\ role, associated\ entity\ role)$ combination for the base entity-type and attributes used.

6.4.1 Inferring possible combinations

The first step is to infer all the possible combinations of relationship types, associated entity types, and roles that are consistent with the specified base entity type. Any types or roles specified in the query constrain the search. Subject to these constraints, S.E.A.L. searches the metadata for all tables in the database that have at least two foreign keys, at least one of which references the base entity type. Any such table is a candidate relationship type, and all tables referred to by the other foreign keys of the table are candidate associated entity types. Names of the foreign keys involved are the roles.

S.E.A.L. then attempts to reduce the set of candidate combinations by examining the attributes specified in the query (using the attribute coverage algorithm described in 6.4.2) and eliminating combinations that are not consistent with the attributes. If the set of combinations is

not reduced to a single combination, then the query contains ambiguity which the user must resolve.

One kind of ambiguity arises if there are two possible relationships in the database between the base entity type and an associated entity with the specified attributes. In this case, S.E.A.L. can report the alternative relationships to the user so that the user can specify the relationship they intended.

Another kind of ambiguity arises if the relationship type of a combination involves the base or associated entity type in more than one possible role (*i.e.*, the table has two or more foreign keys referring to the same entity type) and the role is not specified in the query. In this case, the algorithm reports the ambiguity to the user, along with the possible roles, so that the user can refine the query.

Once there is an unambiguous consistent combination of $(relationship\ type, associated\ entity\ type, base\ entity\ role, associated\ entity\ role)$, this information is passed to the translation algorithms (6.5) to construct the SQL query.

Note that a S.E.A.L. query may contain multiple ASSOCIATED_WITH clauses and that the inference process must be applied to each clause. For nested ASSOCIATED_WITH clauses, the process must be applied recursively.

6.4.2 Attribute coverage algorithm

For a given $(relationship\ type, associated\ entity\ type, base\ entity\ role, associated\ entity\ role)$ combination to be valid with respect to a query, the associated entity type together with the relationship type must contain (in either EAV or regular column form) all attributes referred to in an ASSOCIATED_WITH clause. A combination with this property is said to “cover” the ASSOCIATED_WITH clause. The attribute coverage algorithm checks a $(relationship\ type, associated\ entity\ type)$ pair to determine if it covers the query.

Regular attributes can be checked from the metadata, but checking for the existence of an EAV attribute requires a query against the database. For efficiency, attributes are grouped into sets corresponding to their expected location and all EAV attributes used in a query are checked at once rather than individually.

6.5 Translation Algorithms

Once a query has been parsed and all necessary inferences are completed, the translation into SQL is performed by a bottom-up, recursive traversal of the query tree. In this design, the various parts of the query tree are responsible for their own conversion into SQL with relatively little interdependence.

At an abstract level, the translation of a S.E.A.L. statement produces SQL blocks of the form

```
base_entityid IN (SELECT
base_entityid FROM relationship INNER
JOIN associated_entity ON ... WHERE
AssociationConstraint)
```

for each Association Constraint in an ASSOCIATED_WITH clause. These SQL blocks are connected by logical operators in accordance with the structure of the

Association Constraint Expression and placed in a SQL WHERE clause in the following way

```
SELECT attribute_list FROM
base_entity_type WHERE
(SQL_block_expr);
```

The form of the generated SQL statement has a form very close to the second SQL query in Example 2.

7 Implementation of the S.E.A.L. Interpreter

When the S.E.A.L. interpreter first starts, it connects to the PostgreSQL database and collects metadata about the database that it needs for translating queries. When a user issues a query, the interpreter parses and translates the query. If it identifies any ambiguity or errors in the query, the interpreter not only reports the problems, but also gives suggestions to the user on how to resolve them. Once the query is successfully translated, it outputs the SQL, which can then be run against the database.

The SEAL prototype interpreter is implemented in a Java, JDBC, and PostgreSQL environment. After retrieving metadata, translating a query comprises three main stages: parsing, inference, and translation.

Since the principles of the inference and translation process were described in section 6.4, this section briefly describes only the metadata retrieval and parsing.

7.1 Retrieval of metadata

PostgreSQL provides an 'Information Schema' that is a collection of views and tables describing structures of databases contained in a cluster. These tables and views can be queried through regular SQL.

S.E.A.L connects to PostgreSQL through JDBC and issues custom queries against the PostgreSQL Information Schema to collect metadata on attribute names, table names, table primary keys, table foreign keys, and foreign key names. Metadata, which carries information about role names, is stored in an internal data structure.

EAV attributes are not collected during the metadata acquisition phase, since an entity type may have many thousands of EAV attributes, and many are most probably not needed by a query. Any EAV attributes used in a query are checked at the translation time (which also allows for updates to the EAV attributes during the interaction with the interpreter).

7.2 Parsing

The parser for the interpreter was created with the ANTLR (ANTLR <http://www.antlr.org>) parser generator. ANTLR allows the programmer to write a context free grammar along with Java code to describe actions to perform on rule/token matches. In the case of the S.E.A.L interpreter, the rules defined are relatively simple: the parser simply generates an OO representation of the query tree substituting base and associated entity-types, relationship types and roles with table, join-table and foreign key names, respectively, contained in the metadata structure.

8 Testing

We performed three kinds of testing. One was testing the claim that entity-association queries are complex. The

other was comprised of an extensive sequence of functional tests on the S.E.A.L prototype interpreter. A limited number of performance tests was our third kind of tests. The section briefly describes the tests.

8.1 Testing complexity of entity-association queries

To prove the claim that entity-association queries with a conjunctive condition on two different values of the same attribute are hard to define, we asked a group of 47 students to produce SQL statements for the following entity-association queries within given time limits:

- 1 Find robbers who have the 'Planning' skill. [5 minutes],
- 2 Find robbers who have been taught by 'Bugsy Malone'. [6 minutes], and
- 3 Find robbers who have the 'Planning' and 'Lock-Picking' skills. [10 minutes].

The students had just completed an intensive training in SQL and were familiar with the structure of the *robbers* database (which excludes any effect of surprise on their performance). The aim of the first two queries was to test students' SQL expertise with simple entity-association queries, while the third query was an entity-association query with a conjunctive condition on two values of an attribute. The outcomes of the test are given in the table below.

Query	Correct answers [%]
1	92
2	56
3	32

The fact that 92% and 56% of students defined correctly the first two queries within the given time limits shows that they possessed a fair level of SQL expertise. The fact that almost 70% of them failed to give a correct answer to the third question supports the claim that entity-association queries with a conjunctive condition on two values of an attribute are hard to define.

8.2 Functional testing

The goal of the functional testing was to check that S.E.A.L meets the expected functional requirements. These tests comprised the following steps:

- Defining a query against the *robbers* database in English,
- Expressing the query in S.E.A.L syntax,
- Expressing the query in SQL using an expert's knowledge,
- Running the S.E.A.L interpreter to produce a SQL query,
- Comparing the S.E.A.L generated SQL query and the SQL query produced by the expert, and
- Running both SQL queries against the *robbers* database and comparing results.

The examples listed in previous sections of the paper are a representative set of the test queries that were run. The actual testing comprised numerous variants of these queries and a number of queries that went beyond the initial requirements of the S.E.A.L. project. The query variants included more complex conditional expressions on attributes, multiple associations of the base entity type

with associated entity types, multiple associations of relationship types, and conditional expressions on associated entity type EAV attributes. The SEAL prototype interpreter passed all these tests successfully.

The queries that went beyond the project requirements identified some limitations of the current implementation and are discussed in the Conclusion as a part of the future work.

8.3 Performance testing

There were two parts to the performance testing: comparing the performance of Nested-In and Set theoretic operator approaches to produce entity-association queries and comparing the interpreter overhead to the query execution time.

All performance tests were performed on an Intel Pentium 4 processor at 3.2 GHz, with 1.5 GB DDR2 memory at 533 MHz, and a Seagate 80Gb SATA disk, using Net-BSD 4.99.9, and PostgreSQL Version 8.2.4.

Two databases were used: the *robbers* database and an EAV database consisting of a table representing an entity type and two other tables representing its EAV data structure. The *robbers* database was small; the smallest table contained just 5 tuples, while the largest table contained 40 tuples. The EAV database was considerably larger. It was populated by randomly generated data. The entity table contained 3,000 tuples, the *entity_attributes* table contained 2,500 tuples, and the *entity_eav* table contained 1,000,000 tuples.

8.3.1 Comparing the performance of Nested-IN with Set theoretic operator approach

A set of experiments was conducted to compare the performance of the ‘nested IN’, ‘nested equi-join’, and ‘set theoretic operator’ approaches described at the start of Section 5. The goal of these experiments was to help in deciding which of the three approaches to adopt for the implementation of the S.E.A.L interpreter. The performance of the ‘nested EXISTS’ approach was not tested, since the use of the SQL EXISTS operator results in correlated nested queries, which are known to have worse query costs than equivalent nested non correlated queries (ElMasri&Navathe 2006).

A number of queries involving conjunctive conditional expressions on randomly chosen (*attribute = attribute_name, value = value_data*) pairs were executed on the EAV database. Such queries are indicative of queries involving EAV attributes. The difference in performance was negligible with all three SQL query implementations taking approximately 3 ms to execute.

Another set of experiments involving conjunctive conditional expressions on randomly chosen (*attribute = attribute_name*) pairs were executed on the EAV database. Such queries are indicative of entity-association queries involving conventional relational mapping of entity and relationship type structures. The ‘nested IN’ implementation took approximately 12.3 ms, outperforming the ‘nested equi-join’ implementation which took 13.75 ms, and the ‘set theoretic’ implementation which took 15.8 ms. The outcomes of these experiments led to the decision to implement the SEAL interpreter using the ‘nested IN’ approach.

8.3.2 Comparing interpreter overhead with the query execution time

Multiple experiments were performed on the robbers schema to compare the time the S.E.A.L interpreter took to translate S.E.A.L queries with the time the queries took to execute against a small database. The following table presents the measurements for three characteristic entity-association queries.

	S.E.A.L. Syntax	Tran	Exec
Ex7	SELECT nickname FROM robber ASSOCIATED_WITH(bank THROUGH robbery, <bankname = 'Loan Shark'>)	0.2	0.4
	SELECT nickname FROM robber ASSOCIATED_WITH (<bankname = 'Loan Shark'>)	3.6	
Ex8	SELECT nickname FROM robber ASSOCIATED_WITH (skill THROUGH robber_skill, <skillname = 'Guarding'> ASSOCIATED_WITH (test_location THROUGH skill_test, <name = 'Harvard'>))	0.4	0.7
	SELECT nickname FROM robber ASSOCIATED_WITH (<skillname = 'Guarding'> ASSOCIATED_WITH (test_location, <name = 'Harvard'>))	3.7	
Ex4	SELECT nickname FROM robber ASSOCIATED_WITH (robber_attributes THROUGH robber_eav, <attribute = 'music'> AND <value = 'Latin'> AND <attribute = 'haircut'> AND <value = 'Mohawk'>)	1.4	1.2
	SELECT nickname FROM robber ASSOCIATED_WITH (<attribute = 'music'> AND <value = 'Latin'> AND <attribute = 'haircut'> AND <value = 'Mohawk'>)	4.1	
	SELECT nickname FROM robber[haircut = 'Mohawk' AND music = 'Latin']	0.6	0.7

The first column specifies the example where the semantics of queries are defined. The second column contains the S.E.A.L. syntax of the query. The third and fourth columns contain the average query translation time and the average query execution time in milliseconds. For each query semantics, the first table row contains a S.E.A.L. expression where only entity type roles are omitted requiring only a small amount of inference. The second row contains a S.E.A.L. expression where both the associated entity and the relationship types are omitted requiring a considerable amount of inference.

The first query (Ex7) is a simple entity association query. The second query (Ex8) requires nested S.E.A.L. expressions, produces a SQL statement with two levels of nesting, and is a more complex query than the first one. The third query (Ex4) involves a conjunctive condition on EAV attributes of the base entity type. The first two S.E.A.L. expressions treated the EAV structure as a conventional entity-association structure; the third treated EAV attributes as base entity type attributes, forcing SEAL to find their real source by inference.

The table shows that in the case of nearly full syntax expressions, S.E.A.L. translation time is of the same order of magnitude as the query execution time against a small database. The inference generally incurs an overhead that is just a few times greater than in the case of expressions that require practically no inference. The last table row highlights an interesting S.E.A.L. feature. Although the query requires inference, the translation time is lower than in the case of practically no inference. This is because the SEAL expression in the last row does not contain an ASSOCIATED_WITH clause, which

incurs many checks. Also, S.E.A.L. produced a simpler and more efficient SQL code resulting in a faster execution time against the robber database.

The S.E.A.L. interpreter was then used against the EAV database to perform the queries retrieving entities based solely on EAV attribute conditions. As an indicative result, a query on a conjunctive attribute condition took 1 ms to translate and 12.3 ms to execute.

The experiments showed that the S.E.A.L. interpreter incurs an overhead that is negligible in interactive use.

9 Conclusions and Future Work

This paper describes S.E.A.L., a highly declarative extension to SQL for the specification of entity-association queries in an easy and natural way. Entity-association queries constitute a class of queries often needed by users of databases in Medicine, Biology, Chemistry, Genetics and similar fields of science. These queries ask for data about entities based on their participation in relationships with other entities. The complexity of defining entity-association queries in SQL prevents users from exploiting the full potential of information contained in their databases. S.E.A.L. is intended to alleviate the problem.

9.1 Contribution

The paper contains the following contributions:

- The definition of the S.E.A.L. language syntax that allows association queries to be expressed concisely and simply.
- A description of the S.E.A.L. prototype interpreter, including the inference and translation algorithms.
- A set of rules for defining S.E.A.L. compatible database schemas, mainly consisting of good database design practice,
- A description of the Entity Attribute Value (EAV) database structure, since these database structures are particularly suited for databases in many fields of science and in generic eCommerce web database applications, for which queries constitute a subclass of entity-association queries and are hard to define.

With S.E.A.L., end-users can issue entity-association queries without a high level of database proficiency and without detailed knowledge of the database implementation. Users need to know only a part of the conceptual schema, and because of its inference capability, even this requirement is alleviated, since S.E.A.L. possesses features of query tools with a Universal Relation Interface. In many cases, a user needs to declare only an entity type and conditions on a number of attributes and can ignore their location.

Another feature of S.E.A.L. is that it abstracts EAV storage. Attributes stored in EAV form and in conventional relational form are treated uniformly in queries. Finally, S.E.A.L. is not bound to a particular database schema since it is able to use the metadata about the schema extracted from the database system.

Functional and performance testing performed have shown that S.E.A.L. represents a good proof of concepts and that it incurs an insignificant overhead.

9.2 Future work

While this work provides a general solution to supporting entity-association queries including queries on EAV structures, there are a number of areas which can be expanded on. Desirable functional features currently not supported by S.E.A.L. which need to be addressed in future work include the following:

- Comparison operators other than equality to use in Attribute Constraints,
- Support for relationship cardinalities other M:N,
- Allowing the attribute list after the SELECT clause to contain attributes other than base entity type attributes,
- Extending the support for EAV structures to relationship type EAV attributes, and
- Support of queries with nesting based on associated types of an associated entity type and queries containing logical combinations of Associations.

The S.E.A.L. interpreter is currently a standalone program that interacts with a database. This means that users must decide which query system to use for a given task. A very desirable improvement would be to integrate the interpreter into PostgreSQL, as an extension of the SELECT syntax, which would make S.E.A.L. queries very much more accessible and useful to most users.

10 References

- ANTLR: ANOther tool for Language Recognition <http://www.antlr.org/>. Accessed 6 Sep 2008.
- Corwin, J., Silberschatz A., Miller, P.L. and Marenco, L. (2007): Dynamic tables: An architecture for managing evolving, heterogeneous biomedical data in relational database management systems. *J of the American Medical Informatics Association* **14**(1):86-93.
- Dinu, V. and Nadkarni, P. (2006): Guidelines for the effective use of entity-attribute-value modeling for biomedical databases. *International Journal of Medical Informatics* **76**(11-12):769-779.
- Elmasri, R. and Navathe, S. (2007): *Fundamentals of Database Systems; 5th edition* Addison-Wesley, Inc.
- Nadkarni, P. M. (1996): QAV: Querying entity-attribute-value metadata in a biomedical database. *Computer Methods and Programs in Biomedicine* **53**(2):93-103.
- Nadkarni, P. M. and Brandt, C. (1998): Data extraction and ad hoc query of an entity-attribute-value database. *Journal of the American Medical Informatics* **5**(6):511-527.
- Nadkarni, P. M., Brandt, C., Frawley, S., Sayward, F., Einbinder, R., Zelterman, D., Schacter, L., and Miller, P. (1998): Managing attribute-value clinical trials data using the ACT/DB client-server database system. *Journal of the American Medical Informatics Association* **5**(2):139-151.
- Elmasri, R. and Navathe S. B. (2006): *Fundamentals of Database Systems*. Fifth Edition, Pearson-Addison Wesley.
- Ullman, J. D. (1982): *Principles of Database Systems*. Rockville, Maryland, Computer Science Press, Inc.