S.E.A.L: Simple entity-association query language

Eddie Stanley

Supervisors: Dr Pavle Mogin, Dr Peter Andreae

Submitted in partial fulfilment of the requirements for
Bachelor of Information Technology.

Abstract

Entity-association queries are an important class of database query. These queries return instances of an entity-type which satisfy some constraint involving participation in relationships with other entities. EAV (Entity-Attribute-Value) storage is a data modeling technique to cope with sparse, multi-valued or user-defined data. SQL is poorly suited to expressing entity-association queries and queries involving EAV attributes because the queries must be nested and involve complex join conditions. This report provides a declarative language and prototype interpreter for specifying entity-association queries, with seamless support for EAV attributes.

Contents

Figures

Chapter 1

Introduction

This research presents a query language with an interface to SQL that permits end users with a modest knowledge of SQL to express an important category of complex queries against a relational database in an easy and natural way.

1.1 The Problem

Relational databases store data in rows (tuples) belonging to tables (relation instances). Under the relational model, all tuples belonging to a given relation instance have a fixed number of columns (attributes). Relational databases are commonly used to represent a part of the real world, storing information on *entities* and their *relationships* with other entities. In general, a separate table is used to record data on each entity-type (class of entity), where each row in the table represents an instance of the entity-type. An entity-type should have a column or set of columns which can be used to uniquely identify a given entity instance, referred to as its *primary key*. Foreign keys are used to represent a reference from one entity to another. A *foreign key* is a set of columns in the referencing entity which correspond to the *primary key* columns of the referenced table. If however, an entity can have multiple references to another entity-type, the standard solution employed is a *join table* also known as an associative entity. A row in the join table represents a relationship between two (or more) entities participating in the relationship. At a minimum, the join table should include *foreign keys* which reference each entity involved in the relationship.

The most common method for querying relational databases is through Structured Query Language (SQL). While SQL works well for many classes of queries, it is poorly suited to an important class of query described as *Entity Association* queries throughout this report.

1.1.1 Entity-Association queries

Entity-Association queries are a broad class of database query which involve finding all instances of an entity-type which satisfy some constraint involving participation in relationships with other entities. These constraints can specify mandatory participation: *"The entity must participate in this relationship"*, as well as forbidden participation: *"The entity must not participate in this relationship"*. Participation constraints can be combined into complex expressions using the the Boolean operators AND, OR & NOT. These queries are common but are difficult to express in SQL. The difficulty arises because users must know the physical database schema (structure) in detail: simply knowing the *conceptual schema* (problem domain) is insufficient. The user needs to know the location of attributes as well as which attributes are stored in Entity Attribute Value (EAV) format (EAV storage is used to cope with sparse or user-defined attributes). If an attribute is stored in EAV format it must be extracted from EAV representation through a special subquery. Entity-Association queries make heavy use of explicit joins, requiring the user to know the primary and *foreign keys* of all tables in the schema. The nesting of SQL required for entity-association queries makes them difficult to construct and harder to debug. A final issue facing the user is ensuring that subqueries resulting from this nesting are aliased correctly.

1.2 Contributions

There are three main contributions of the project:

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

By adopting the S.E.A.L language, domain experts will be able to issue an important class of query without requiring technical database knowledge. The S.E.A.L interpreter, when used with databases adhering to the rules and guidelines given in section 6.2, gives database administrators and software developers a general solution to the problem of allowing users to execute these queries. A significant strength of S.E.A.L (and the S.E.A.L interpreter) is that it is independent of the database used.

1.3 Outline of this report

- Chapter 2 defines key terms and concepts used throughout this report
- Chapter 3 reviews previous work on EAV storage
- Chapter 4 motivates the need for S.E.A.L through a worked example
- Chapter 5 examines the design of the S.E.A.L language and the algorithms used in the interpreter
- Chapter 6 discusses the problems encountered implementing the S.E.A.L interpreter and the steps taken to resolve them
- Chapter 7 summarizes results from testing
- Chapter 8 reviews the contribution of this research and identifies future work

Throughout this report, many of the examples refer to a fictional `'robbers'` database which stores information on `robbers`, the `skills` they possess and their involvement with other `robbers`. The schema for this database as well as sample data is included in Chapter 4.

Chapter 2

Terms and definitions

To avoid ambiguity, this Chapter defines key terms and concepts used throughout this report.

S.E.A.L

S.E.A.L stands for Simplified Entity Association Language, a declarative language for expressing entity-association queries. The report will also occasionally use 'S.E.A.L' to refer to the prototype interpreter for S.E.A.L..

Entity

An Entity (or entity instance) is something that has a distinct, separate existence (though it need not be a material existence) [9], belonging to exactly one entity-type. For example, in the `robbers` schema, `banks`, `robbers`, `skills` and `gangs` are all examples of entity-types, while `'Al Capone'` (a famous robber) is an example of an entity *instance* (or simply, entity).

Relationship

A relationship is an association between two or more entities which conveys at a minimum the nature of the involvement of the entities but may also carry other information. For example, in the `robbers` schema, the relationship `'robber_has_skills'` between a `'robber'` entity and a `'skill'` entity carries information on how proficient the `robber` is at that `skill` (the `'skilllevel'` attribute). Relationships are classified as either *asymmetric* or *symmetric*; in practice, asymmetric relationships are much more prevalent. It is also possible for a relationship to behave as an entity: a relationship instance may itself be involved in relationships with other entities.

Attribute

An attribute is an atomic property of an entity. Sometimes the distinction between an attribute and an entity-type is blurred: It is possible to represent properties of entities through a relationship with another entity-type representing the property. In general, if the property is atomic then it is represented as an attribute, whereas if information is stored about the property then it mapped to a separate entity-type. For example: for a person the entity-type, 'date_of_birth' would probably be mapped to an attribute whereas 'place_of_birth' would be mapped to a relationship with another entity-type, Country. This would allow extra information to be stored about the property, for example currency, capital city etc.

Role

A role is defined as the behaviour and/or obligations of an entity participating in a relationship. For example in a `'mentoring'` relationship between two `robber` entities, the role of `'teacher'` might be assigned to one `robber` entity and the role of `'pupil'` to the other.

Relationship participation notation

For a relationship RS, entities E1 & E2 and roles R1 & R2, the following notation expresses the fact that entity E1 & E2 participate in relationship RS where E1 acts as R1 and E2 acts as R2:

$$E1 \overset{R1}{\Rightarrow} RS \overset{R2}{\Leftarrow} E2$$

Asymmetric relationship

In an asymmetric relationship, distinct roles are assigned to each entity participating. For entities E1 & E2, roles R1 & R2 and an asymmetric relationship RS_{asym}, if E1 participates in RS_{asym} with E2 where E1 acts as R1 and E2 acts as R2, we cannot infer that E2 participates in RS_{asym} with E1 where E2 acts as R1 and E1 acts as R2.

Symmetric relationship

In a symmetric relationship, all entities participating are of the same entity-type and have equivalent roles. Because roles in a symmetric relationship are identical, they may be omitted. For entities E1, E2 and a symmetric relationship RS_{sym}, if E1 participates in RS_{sym} with E2 then it is also the case that E2 participates in RS_{sym} with E1.

More formally, $E1 \Rightarrow RS_{sym} \Leftarrow E2 \textit{ implies } E2 \Rightarrow RS_{sym} \Leftarrow E1$

Consider a `friendship` relationship between two `people` entities, `Peter` & `John`. If we were to declare this relationship symmetric, then proclaiming `Peter` is friends with `John` implies that `John` is also friends with `Peter`. This would not be the case had we declared the relationship as asymmetric.

Base entity-type

The base entity-type of an entity-association query is the entity-type that the query (or subquery) returns. In S.E.A.L, the base entity-type precedes the `ASSOCIATED_WITH` clause (see section 6.1).

Associated entity-type

An entity-type involved with the base entity-type through some relationship. In S.E.A.L, each `ASSOCIATED_WITH` clause specifies a participation constraint for exactly one associated entity-type (see section 6.1).

EAV Table

A table of triples (EntityID, AttributeID, Value) used to store sparse, multi-valued or user-defined attributes for a particular entity-type. EAV is used as an alternative to adding extra columns to the table representing the entity type. EAV storage is described in Chapter 3.

EAV Attribute lookup table

A table which stores a list of all attributes stored in EAV form for a given entity-type.

Chapter 3

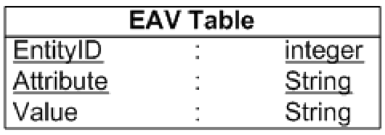
Background/ Literature review

3.1 EAV Storage

3.1.1 Description

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 one attribute or property of an entity instance. Each attribute is described by the name of the column. As a consequence of this, all entity instances of the same entity-type store 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.

Entity Attribute Value (EAV) storage is an alternative method for representing the attributes belonging to an entity. EAV represents each entity instance as a set of (key, value) pairs[7]. The key describes the attribute, while the value assigns data to the fact description. Under EAV, metadata is represented as data: not only are the values of attributes data (as they were in the relational model) but the attribute `descriptions` are also data. The simplest implementation of EAV involves just a single relational table with three attributes: Entity-Identifier, Attribute, Value. Figure 3.1 shows this simplest implementation:



EAV Table		
EntityID	:	integer
Attribute	:	String
Value	:	String

Figure 3.1: A simple EAV table

The Entity-Identifier groups the (key, value) pairs into entities and is optionally used to join to conventionally stored data.

Not all data in a production system will lend itself to EAV storage, therefore it is common for schemas to involve a mixture of conventionally stored data and data represented in EAV form [3]. EAV tables are generally not shared between entity-types: that is, each entity-type making use of EAV storage will be assigned a separate EAV table. For example, in the `robber` schema given in Chapter 4, the entity-type `robber` has an EAV table assigned. All rows in this table represent facts about `robber` instances, and all rows with the same RobberID belong to the same robber instance. For example, `Anastazia` likes both `'Heavy`

`Metal'` and `'Latin'` music. Because the information about a single entity can span many rows, EAV is sometimes referred to as *row modelling* [7].

3.1.2 Applications

Sparse data

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 which typically apply may be only a few dozen [7]. 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. For sparse data, using EAV has two key advantages:

1. Many database management systems impose a limit on the number of columns a table may have (a common limit is 255)[6]. Using EAV circumvents this problem because attributes are represented in rows rather than columns.
2. EAV only stores values for attributes which apply to a given entity instance, thus saving space over a conventional representation which stores a NULL value when an attribute does not apply.

User-defined fields

In certain applications, there is a need for users to be able to define (and destroy) attributes as part of 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. Because the attributes for an entity-type are described as regular data under EAV, only regular row INSERT/DELETE operations are required to add and remove attributes respectively. This also removes the need to acquire an exclusive lock on the table.

Multi-valued attributes

Under the normalization process for relational database described by E.F Codd[2], first normal form states that a relation may have no repeating groups and that all attributes must be atomic. However in the real world, multi-valued attributes are common. There are a number of poor solutions to representing multi-valued attributes which are used in industry. Two such solutions are :

- The delimited non-atomic attribute solution
- The numbered attribute solution

The 'delimited non-atomic attribute' solution uses some delimiter such as a comma `','` to delimit instances of a multi-valued attribute. For example, suppose the user wishes to store the `haircuts` a `robber` has been seen with. Under the 'delimited non-atomic attribute' solution, this would be represented as follows:

RobberID	Nickname	Haircuts
1	Al Capone	Mohawk, Pony Tail
2	Bugsy Malone	Crew Cut

This complicates queries: to find `robbers` who have been seen with a `Mohawk` and have been seen with a `Crew Cut`, a query similar to the following would need to be issued:

```
SELECT nickname FROM robber WHERE haircuts LIKE '%Mohawk%' AND
    haircuts LIKE '%Crew Cut%';
```

The biggest issue with this representation is that LIKE comparisons with a wildcard on either side of the pattern are expensive to issue because an index cannot be used.

This design also suffers for updates: to add a `haircut` to a `robber`, the user must first examine whether the delimited string already contains the `haircut`, in which case it should not be added. To remove a `haircut` from a `robber`, the user needs to find where it occurs in the string and splice it out. This can be achieved with string manipulation functions such as `substring()` and `indexOf()`.

Under the 'numbered attribute' solution, an estimate is made on the upper bound of occurrences of the attribute. For example the user might decide that no robber will ever be seen with more than three haircuts. A separate field is created for each possible occurrence. Under the 'numbered attribute' solution, the multi-valued haircut attribute would be represented as follows:

RobberID	Nickname	Haircut_1	Haircut_2	Haircut_3
1	Al Capone	Mohawk	Pony Tail	
2	Bugsy Malone	Crew Cut		

To find `robbers` who have been seen with a `Mohawk` and who have been seen with a `Crew Cut`, a query similar to the following would need to be issued:

```
SELECT nickname FROM robber WHERE haircut_1='Mohawk' AND
    haircut_2='Crew Cut'
UNION
SELECT nickname FROM robber WHERE haircut_1='Mohawk' AND
    haircut_3='Crew Cut'
UNION
SELECT nickname FROM robber WHERE haircut_2='Mohawk' AND
    haircut_1='Crew Cut'
UNION
SELECT nickname FROM robber WHERE haircut_2='Mohawk' AND
    haircut_3='Crew Cut'
UNION
SELECT nickname FROM robber WHERE haircut_3='Mohawk' AND
    haircut_2='Crew Cut'
```

```
UNION
SELECT nickname FROM robber WHERE haircut_3='Mohawk' AND
   haircut_1='Crew Cut'
```

The query was complicated by the fact that there is no guarantee which column an particular value will appear in. This design also suffers from update problems: If the user comes across a `robber` who has been seen with more than three `haircuts`, they can only record the first three. If the user wishes to add a `haircut` for a `robber`, the user must first check all the columns to ensure it does not already exist (and find a free 'slot' for it). If the user wishes to remove a `haircut` from a `robber`, the user must find which column the `haircut` is currently stored in and set it to NULL.

While both of these solutions work to some extent, they both have serious problems with queries and updates. EAV is one possible solution for representing multi-valued attributes: if the primary key of the EAV table is set on (`EntityID` + `Attribute` + `Value`) rather than just (`EntityID` + `Attribute`) then for a given {EntityID, Attribute}, multi-valued attributes can be stored. If related information is stored on the multi-valued attribute, or the same attribute values apply to many entity instances, it is more appropriate to represent the multi-valued attribute as a relationship with another entity-type.

3.1.3 Variations

Key on E+A or E+A+V

In the EAV structure described above, the primary key for the relation was on {EntityID, Attribute}. As a consequence of this, only a single value is permitted per attribute per entity instance. By defining the key on {EntityID, Attribute, Value} multi-valued attributes can be supported.

Attribute lookup table

In this variation of EAV, a separate table (termed the lookup table) is used to record the names of attributes stored in EAV form. The EAV table references the lookup table through a foreign key constraint. Often the attribute lookup table will have a surrogate integer primary key (AttributeID) and the EAV table will store a reference to this field. This variation has two subtle advantages:

- The domain of 'attribute' is restricted: before an attribute can be used, a corresponding row must exist in the lookup table. This makes administering attributes easier.
- Attribute names are stored only once: if an attribute name needs to be changed, it need only be changed in one place. This also results in a modest space saving.

However, this approach introduces an additional join per attribute which increases the complexity of queries.

Table per datatype

Instead representing all values as strings in a single EAV table, a separate EAV table is used for each datatype. For example: attributes of type `string` are stored in `entity_eav_string`, attributes of type `boolean` are stored in `entity_eav_boolean` etc[3]. This approach has a number of advantages:

- Aggregate functions can be performed on numerical data without typecasting
- Value comparisons work more efficiently as typecasts are not required
- In some circumstances data may be stored more efficiently in its native representation rather than as strings

Column per datatype with indicator

In this approach, a single EAV table is used which has a value column for each datatype and an 'indicator column' which indicates which of the value columns contains a valid value for a given row [3]. The purpose of the indicator column is to avoid having to compare the value of every value column with NULL. While the 'column per datatype with indicator' approach has the same advantages as the 'table per datatype' approach, it also has a couple of drawbacks:

- If *n* datatypes are stored in EAV format then for every row in the EAV table there will be *n-1* NULL values
- There is additional complexity involved in examining the indicator to determine which value column contains valid data.

3.1.4 Entity associations

While not strictly a variation on EAV, entity associations can be used to represent multi-valued and/or sparse attributes. With this approach, an attribute belonging to an entity-type E is represented as an entity-type of its own, X. A relationship between E and X is defined. Instances of entity-type X represent distinct *values* of attribute X. The advantages of this approach are:

- An attribute *type* can be shared between multiple entity-types
- An appropriate datatype can be used
- If a need arises to store additional information on the attribute (in effect, the attribute becomes an entity) then extra columns can be added to the table representing X.

Some disadvantages of this approach:

- Queries become more complex (all queries become entity-association queries)
- This approach is less suitable for user-defined attributes because it requires alteration of the database structure (new tables need to be created)
- It can result in confusion for the user: the user needs to know that a particular attribute is actually an entity-type and will need to write queries involving the attribute as entity-association queries
- It results in a large number of tables which can make managing the database schema difficult.

Dynamic tables

John Corwin et al.[4] describe an approach which uses one table per attribute called dynamic tables. Under this approach, an entity-type is represented as:

1. A table of OIDs which provide a handle for individual entity instances; and
2. A table for each attribute belonging to the entity-type with two columns: an OID reference and a value, typed appropriately for the attribute.

Dynamic tables is a similar approach to the entity associations approach described above, apart from the distinction that it does not involve an intermediate relationship table. As a consequence of this, this approach does not allow attribute types to be shared between different entity-types. Like the approach using entity associations, this will result in a large number of tables and without an interpreter would require complex queries. John Corwin et al.[4] describes an extension to PostgreSQL for supporting dynamic tables in user queries.

3.1.5 Problems relating to EAV

Metadata

A significant problem with EAV for the user is knowing where an attribute is located [7]. The attribute may be stored as a regular column or in an EAV table. If multiple EAV tables are employed, such as the table per datatype solution, then this becomes even harder. The user must also be aware of the datatype of the attribute, the set of permitted values and whether the attribute is required or optional [7, 3]. If some attributes are multi-valued and others are not, the user needs to know this also. A final problem is finding the set of all attributes which exist for an entity-type.

Bulk retrieval

A common use-case for a database storing information on real world objects (entities) is to retrieve all the data on a given entity instance. In a conventionally structured database, this involves a "SELECT * FROM ..." on the table representing the entity as well as joins to any related tables. The field list implied by '*' is populated by the DBMS: the DBMS records a list of all attributes contained in each table.

Retrieving complete entity data from a database utilizing EAV attributes is more difficult and more computationally expensive. The difficulty arises because (with the most basic form of EAV) there is no list of all attributes belonging to an entity-type. Some solutions to this problem are:

- A variant of EAV which employs an attribute lookup table. The lookup table records all attributes which are stored in EAV form [3]
- Use of a global metadata structure (often a table) which records information (location, datatype etc) about all attributes in the domain [7]

Assuming appropriate indices are in place, the actual *extraction* of all (EAV) data relating to an entity instance is not prohibitively expensive:

```
SELECT * FROM entity WHERE entityID = x;
SELECT * FROM entity_EAV WHERE entityID = x;
```

The first query returns a single tuple containing the conventionally stored part of the entity while the second query returns a tuple for every (attribute,value) pair stored in EAV form. The problem is however that this data is not in a form which is particularly useful to the user. The vertically represented EAV attributes must be transformed or 'pivoted' [8] into a horizontal one-column-per-attribute format, consistent with the conventionally stored data. This pivoting operation is extremely expensive when performed in SQL (using an outer join/alias per attribute) and still relatively expensive when performed in-memory using an external program [8].

Current solutions to improve the performance of complete entity retrieval all involve some aspect of redundancy to reduce the number of joins. One possibility is to perform extraction in advance into a materialized view [3]. A solution by 3M corporation uses a non-relational, ASN.1 representation for storing complete patient records for efficient retrieval [7]

3.1.6 Previous work on abstracting EAV storage

We identified two systems which abstract EAV attribute representation from the database user.

ACT/DB[7, 5] is a database/tool for managing clinical trial data. It uses a client/server architecture with Oracle 7 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 datatypes supported: integer, real, date, short string, long string & binary. ACT/DB supports a number of comparison operators as well as aggregate functions such as average and standard deviation.

QAV[6] 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 a client-server architecture.

3.2 Entity-association queries

While there is an abundance of information on the design of relational schemas storing M:N relationships between entities, no previous work was found regarding performing entity-association queries on such databases.

Chapter 4

Analysis of an example Entity-Association query

4.1 Example schema

To illustrate entity-association queries and to explain the proposed language and interpreter, the report uses a running example of a fictional `'robbers'` database. The structure of the database is given, followed by sample data. Particular attention should be paid to the following:

- Which attributes form the primary key of each table
- Which attributes belong to a foreign key
- Which table a foreign key references
- For each attribute belonging to a foreign key, the corresponding attribute in the referenced table
- The name of foreign keys

```
CREATE TABLE robber (
    robberid integer NOT NULL,
    nickname varchar NOT NULL,
    age smallint,
    prisonyears smallint,
    CONSTRAINT robber_pk PRIMARY KEY(robberid)
);

CREATE TABLE skill (
    skillid integer NOT NULL,
    skilldescription varchar NOT NULL,
    difficulty integer,
    CONSTRAINT skill_pk PRIMARY KEY(skillid)
);

CREATE TABLE robber_has_skill (
    robberid integer NOT NULL,
    skillid integer NOT NULL,
    skillpreference integer,
    skilllevel character(2),
    CONSTRAINT robber_has_skill_pk PRIMARY KEY(robberid, skillid),
    CONSTRAINT robber_with_skill FOREIGN KEY(robberid) REFERENCES
       robber(robberid),
    CONSTRAINT skill_possessed_by_robber FOREIGN KEY(skillid)
       REFERENCES skill(skillid)
);

CREATE TABLE food (
    foodid integer NOT NULL,
    name varchar,
    CONSTRAINT food_pk PRIMARY KEY (foodid)
);

CREATE TABLE robber_likes_food (
    robberid integer NOT NULL,
    foodid integer NOT NULL,
    CONSTRAINT robber_likes_food_pk PRIMARY KEY (robberid, foodid),
    CONSTRAINT robber_who_enjoys_food FOREIGN KEY (robberid)
       REFERENCES robber(robberid),
    CONSTRAINT food_enjoyed_by_robber FOREIGN KEY (foodid)
       REFERENCES food(foodid)
);

CREATE TABLE testing_location (
    locationid integer NOT NULL,
    name varchar NOT NULL,
    CONSTRAINT testing_location_pk PRIMARY KEY(locationid)
);

CREATE TABLE robber_has_skill_tested (
    robberid integer NOT NULL,
    skillid integer NOT NULL,
    locationid integer NOT NULL,
    CONSTRAINT robber_has_skill_tested_pk PRIMARY KEY(robberid,
       skillid, locationid),
    CONSTRAINT tested_robber_skill FOREIGN KEY(robberid, skillid)
       REFERENCES robber_has_skill(robberid, skillid),
    CONSTRAINT location_tested_at FOREIGN KEY(locationid)
       REFERENCES testing_location(locationid)
);

CREATE TABLE mentoring (
    robberid1 integer,
    robberid2 integer,
    CONSTRAINT mentoring_pk PRIMARY KEY(robberid1, robberid2),
    CONSTRAINT teacher FOREIGN KEY(robberid1) REFERENCES
       robber(robberid),
    CONSTRAINT pupil FOREIGN KEY(robberid2) REFERENCES
       robber(robberid)
);

CREATE TABLE friendship (
    robberid1 integer,
    robberid2 integer,
    CONSTRAINT friendship_pk PRIMARY KEY(robberid1, robberid2),
    CONSTRAINT friendship_to_robber1 FOREIGN KEY(robberid1)
       REFERENCES robber(robberid),
    CONSTRAINT friendship_to_robber2 FOREIGN KEY(robberid2)
       REFERENCES robber(robberid)
);

CREATE TABLE robber_attributes (
    attributeid integer NOT NULL,
    attribute varchar,
    CONSTRAINT robber_attributes_pk PRIMARY KEY (attributeid)
);

CREATE TABLE robber_eav (
    robberid integer NOT NULL,
    attributeid integer NOT NULL,
    value varchar NOT NULL,
    CONSTRAINT robber_eav_pk PRIMARY KEY (robberid, attributeid,
       value),
    CONSTRAINT robber_eav_to_robber FOREIGN KEY(robberid)
       REFERENCES robber(robberid),
    CONSTRAINT robber_eav_to_robber_attributes FOREIGN
       KEY(attributeid) REFERENCES robber_attributes(attribute_id)
);
```

robber

robberid	nickname	age	prisonyears
1	Al Capone	31	2
2	Bugsy Malone	42	15
3	Lucky Luchiano	42	15
4	Anastazia	48	15
5	Mimmy	18	0
6	Dutch Schulz	64	31

skill

skillid	skilldescription	difficulty
2	Explosives	5
3	Guarding	1
4	Gun-Shooting	3
5	Lock-Picking	2
6	Money Counting	4
7	Planning	5

robber_eav

robberid	attributeid	value
1	1	Mohawk
2	3	Skull & Crossbone
4	2	Heavy Metal
4	2	Latin

robber_eav

attributeid	attribute
1	haircut
2	music
3	callsign

robber_has_skill

robberid	skillid	skillpreference	skilllevel
1	7	1	A+
1	8	3	A+
1	9	2	C+
2	2	1	A
3	1	2	B+
3	5	1	B+
4	3	1	A
5	1	2	C
5	7	1	A+
6	1	2	C+
6	5	1	A+

robber_likes_food

robberid	foodid
1	1
2	2
3	3
4	4
5	5
5	3

food

foodid	name
1	pizza
2	Mexican
3	Chinese

symmetric_foreign_keys

table_name	fk1_name	fk2_name
friendship	friendship_to_robber1	friendship_to_robber2

mentoring

robberid1	robberid2
1	2
2	3
3	1
3	2
5	1

4.2 A simple Entity-Association query

A simple example of an entity-association query for the `robbers` database is the following:

> "Find the `robbers` who have `skill` '`Gun shooting`' and do not have the `skill` '`Money Counting`', or the `robbers` who have the skill '`Explosives`'"

Figure 4.1: A simple example of an entity-association query

4.3 Difficulty of expressing entity-association queries in SQL

SQL does not have a declarative syntax for expressing this class of query directly. We are aware of two main strategies for expressing entity-association queries in SQL:

1. Multiple SELECT statements combined with the set operators (UNION, INTERSECT & EXCEPT)

2. Nested SELECT queries using the IN operator

4.3.1 Representation using set operators

Using the first approach, the query given in figure 4.1 can be broken down into three SELECT queries and then combined with the SET operators UNION, INTERSECT & EXCEPT:

```
SELECT nickname FROM robber INNER JOIN ((SELECT robberid FROM
    robber_has_skill INNER JOIN skill ON skill.skillid =
    robber_has_skill.skillid WHERE skilldescription = 'Gun shooting'
EXCEPT
SELECT robberid FROM robber_has_skill INNER JOIN skill ON
    skill.skillid = robber_has_skill.skillid WHERE skilldescription
    = 'Money Counting')
UNION
SELECT robberid FROM robber_has_skill INNER JOIN skill ON
    skill.skillid = robber_has_skill.skillid WHERE skilldescription
    = 'Explosives') as foo ON foo.robberid = robber.robberid;
```

4.3.2 Representation using nested selects with IN operator

Using the second approach, the query given in figure 4.1 can be expressed using nested IN statements:

```
SELECT nickname FROM robber WHERE
(((robber.robberid) IN (SELECT robber_has_skill.robberid FROM
    robber_has_skill INNER JOIN skill ON
    robber_has_skill.skillid=skill.skillid WHERE
    skill.skilldescription='Gun Shooting'))
AND
(NOT ((robber.robberid) IN (SELECT robber_has_skill.robberid FROM
    robber_has_skill INNER JOIN skill ON
    robber_has_skill.skillid=skill.skillid WHERE
    skill.skilldescription='Money Counting'))))
OR
((robber.robberid) IN (SELECT robber_has_skill.robberid FROM
    robber_has_skill INNER JOIN skill ON
    robber_has_skill.skillid=skill.skillid WHERE
    skill.skilldescription='Explosives'));
```

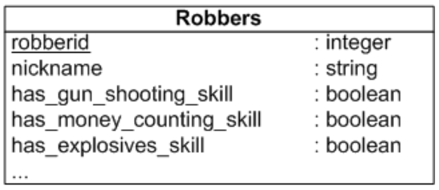
Chapter 8 will show that the nested-select implementation (above) offers marginally better performance in PostgreSQL than the set-operator implementation.

We argue however that queries constructed using either of these strategies poorly reflect the logical intention of the query and because of this, are difficult to write.

4.3.3 Why both of these representations are flawed

Given the intent of the query, it is not clear why we need to issue multiple select statements. From the user's perspective, *the relationships in which an entity participates can be viewed as a properties of the entity itself.*

To test whether an entity satisfies a participation constraint, the average user would see no reason why it is necessary to check for the presence of the entity's primary key in another table. The schema given in figure 4.2 probably better represents the user's understanding of the domain:



Robbers	
robberid	: integer
nickname	: string
has_gun_shooting_skill	: boolean
has_money_counting_skill	: boolean
has_explosives_skill	: boolean
...	

Figure 4.2: A possible (poor) design for storing the skills of robbers

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 thousands of possible skills), it is easy to see that this SQL reflects the user's intention more simply than the complex queries above.

A perhaps more pragmatic concern is that both of the described strategies (set operator and nested-in) lead to queries which are unnecessarily difficult to write. Factors contributing to this difficulty are:

1. Users need to know where an attribute is stored. Does it belong to the relationship, or to the associated-entity? 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 probably would suffice but often this is not possible: consider a relationship which involves two entities of the same type, for example: two robbers. The relationship table cannot have two columns labelled `robberid` so a natural join will not work.
3. Users need 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.

4.4 Problems arising from EAV attribute storage

Two problems with EAV storage are the difficulty of using EAV attributes in queries, and the requirement that users know whether an attribute is represented in EAV form or not. The difficulty arises because EAV attributes must be transformed from their vertical representation to their horizontal *logical* representation. As an example, suppose the attribute `'haircut'` belonging to `robber` has been designated as an EAV attribute. Perhaps it is multi-valued: *"a robber may have been seen with different haircuts"*. Another possibility is that it is sparse: *"we only have haircut information on a very small fraction of robbers"*. For the query

“Find `robbers` with a `Mohawk` `haircut` or who like `Heavy Metal` `music`”

A reasonable *conceptual* query would be:

```
SELECT nickname FROM robber WHERE haircut='Mohawk' OR music='Heavy
    Metal';
```

Because the attribute was stored in EAV format, the user must write SQL similar to the following:

```
SELECT nickname FROM robber WHERE (robber.robberid) IN (SELECT
    robber_eav.robberid FROM robber_eav INNER JOIN robber_attributes
     ON robber_eav.attributeid=robber_attributes.attributeid WHERE
    attribute='haircut' AND value='Mohawk') OR (robber.robberid) IN
    (SELECT robber_eav.robberid FROM robber_eav INNER JOIN
    robber_attributes ON
    robber_eav.attributeid=robber_attributes.attributeid WHERE
    attribute='music' AND value='Heavy Metal');
```

The SQL tests the expression `robber.haircut = 'Mohawk'` for a given `robber` by examining the `robber_eav` table to check for the presence of a row which references the given `robber`, has `attribute='haircut'` and `value='Mohawk'`. The same method is used to test for the music condition. Aside from the difficulty of constructing this query, a prominent issue is that the user needs to be *aware* that these attributes are represented in EAV format in order to construct this query.

4.5 Why this is a problem

The need for entity-association queries is most common in fields such as Biology, Genetics & Chemistry, where users issuing the queries generally only have modest experience using databases. Often, the extent of their knowledge is limited to using query-by-example (QBE) tools such as Microsoft Access. QBE tools allow the user to drag and drop tables and set predicate conditions to construct a query. The QBE engine will take care of issues such as aliasing and generating join conditions. We are not aware of any mainstream QBE tools which will handle entity-association queries. The literature review (Chapter 3) described a couple of QBE tools (QAV, ACT/DB) which assist users in writing queries involving EAV attributes. However, all of these tools are significantly limited in that they are tied to a particular database schema.

Chapter 5

Evolution of the project

The initial focus of this project was providing better support for EAV attributes. Specifically:

- Making the use of EAV attributes as seamless as possible
- Improving the performance of queries involving EAV attributes

During the course of the project, an observation was made that EAV attributes can be viewed as associated entities. As a consequence of this, queries involving EAV attributes can be viewed as entity-association queries.

For an entity-type E:

- The attribute name, A, can be viewed as the associated entity-type
- The EAV table can be viewed as the relationship between E & the associated entity-type
- The value of the attribute can be viewed as a property of the relationship between E and A

It follows that a condition, C, of the form *attribute=value* on an (EAV) attribute belonging to E can be expressed as:

$$E \overset{entity}{\Rightarrow} eav\_table \overset{property}{\Leftarrow} A : A.attribute = C.attribute \wedge eav\_table.value = C.value$$

While the focus of the project shifted to entity-association queries, support for EAV attributes was included in the language and the interpreter. However, a number of features are missing from the EAV support in S.E.A.L, namely:

- Support for attribute types other than String: Numeric, Date etc
- Support for aggregate functions: MAX, SUM, COUNT, AVERAGE etc
- Support for comparison operators other than = : '!=', BETWEEN, LIKE, '$>$', '$<$' etc

Chapter 6

Design

6.1 Language design

The primary goal of S.E.A.L was to relieve the user from the need to understand how their query would be executed. To realise this goal, the syntax was designed to be as declarative as possible: the syntax only describes the conditions the result set should meet, rather than describing the algorithm used to perform the query. With a declarative syntax, the underlying implementation can change without affecting the user.

6.1.1 Basic syntax:

The following gives the S.E.A.L syntax for an entity-association query. For the full grammar, see the appendix. In the following definition, productions of the form *production*_expr are a Boolean expression using parentheses and the keywords 'AND', 'OR' & 'NOT' on *production*. For example, a `baseEntityRestriction` is of the form:

```
attribute='value'
```

Therefore

```
(haircut='Mohawk' AND music='Heavy Metal') OR
   haircut='Crew Cut'
```

is a valid `baseEntityRestriction_expr`.

Syntax:

```
SELECT
  attribute [ , ... ] | *
FROM
  baseEntityType
  [ [baseEntityRestriction_expr] ]
  [ [ AS baseEntityRole ] associationClause_expr ]
```

An `associationClause` is of the form:

```
ASSOCIATED_WITH( [ [ associatedEntityType
                     [ AS associatedEntityRole ]
                     [ THROUGH relationship ]
                   ],
                 ] specification_expr
               )
```

A `specification` is of the form:

```
<attribute='value' [, ...]> [ associationClause_expr ]
```

Figure 6.1: S.E.A.L syntax

- `baseEntityType`: The base entity-type the query returns
- `attribute`: The name of an attribute
- `baseEntityRole`: The role the base entity-type must play in the relationship
- `baseEntityRestriction_expr`: A Boolean expression on attributes belonging to the base entity-type, restricting the set of the base entity-type. These attributes may be stored as regular columns or they may be EAV attributes.
- `associatedEntityType`: An entity-type associated (participating in a relationship) with the base entity-type
- `associatedEntityRole`: The role the associated entity-type must play in the relationship
- `relationship`: The name of a relationship (possibly one of many) between the base entity-type and the associated-entity type
- `specification_expr`: A Boolean expression composed of one or more *specifications*. A specification is a set of predicate conditions on attributes belonging to the associated entity-type and/or the relationship.

Each `associationClause` refers to exactly one (associated entity-type, relationship) pair whether explicitly stated or inferred. If the relationship is omitted, S.E.A.L will attempt to infer a relationship between the associated entity-type and the base entity-type. If both the relationship and the associated entity-type are omitted, S.E.A.L will attempt to infer both, using the algorithms described in section 6.3.

Example queries

The following examples use the robber schema given in Chapter 4.

Example Query 1: Uniform access to attributes

This example demonstrates uniform access to attributes (whether EAV or regular columns) in S.E.A.L:

"Find robbers who are 39 years old and have a Mohawk haircut"

S.E.A.L syntax:

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

This query demonstrates restricting the base entity-type on its attributes. The attributes are used in a uniform manner: the user does not need to know that haircut is represented in EAV form while age is a regular column. S.E.A.L allows a simple representation of this query although the implementation in the database involves multiple tables and complex SQL.

Example Query 2: A simple entity-association query

This example demonstrates how a simple entity-association query is composed in S.E.A.L.

"Find robbers who have skill 'Lock-picking' or who have the skill 'Planning'"

S.E.A.L syntax:

```
SELECT nickname FROM robber ASSOCIATED_WITH(skill THROUGH
  robber_has_skill, <skilldescription='Lock-picking'> OR
  <skilldescription='Planning'>)
```

In this example 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 to the base entity-type, robber_has_skill. It is not necessary to specify either of the roles because a 'robber_has_skill' relationship involves exactly one entity of type robber and one entity of type skill. Two specifications are used in the query which are combined into an expression with the OR operator. Each specification uses a single attribute from the associated entity-type, skilldescription. Note that a specification need not specify a single instance: if there were more than one skill with skilldescription='Planning' then any of these skills would satisfy the specification.

Example Query 3: Inference

This example demonstrates inference in S.E.A.L.

"Find robbers who like pizza"

S.E.A.L syntax:

```
SELECT nickname FROM robber ASSOCIATED_WITH(<name='pizza'>)
```

In this example, the following were omitted:

- The associated entity-type
- The relationship between the base entity-type and the associated entity-type
- The role of the associated entity-type & the role of the base entity-type

With regard to the robber schema (given in Chapter 4), S.E.A.L is able to infer that the specification <name='pizza'> refers to the associated entity-type 'food' through the relationship 'robber_likes_food'. The roles robber_who_enjoys_food & food_enjoyed_by_robber are also inferred.

Example Query 4: A nested entity-association query

This example shows how an entity-association query where the *relationship* has a participation constraint is constructed in S.E.A.L.

> "Find robbers with the skill 'Guarding' who have had it tested at a testing location called 'Harvard'.

S.E.A.L syntax:

```
SELECT nickname FROM robber ASSOCIATED_WITH(skill,
   <skilldescription='Guarding'>  ASSOCIATED_WITH(testing_location,
    <name='Harvard'>))
```

In this example, a participation constraint has been placed on the base entity-type robber: the robber must possess the skill 'Guarding'. Additionally, a participation constraint has been placed on the specification <skilldescription='Guarding'>: not only must the robber possess the skill, but they must have it tested at a testing_location with name='Harvard'.

Example Query 5: An entity-association query involving roles

This example shows a query where roles cannot be inferred and must be specified.

> "Find robbers who have been taught by 'Bugsy Malone'"

S.E.A.L syntax:

```
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 teacher. It was necessary to specify a 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). Note that because the mentoring relationship involves only two entities of type robber, the query could have also been written with the role of the base entity-type, pupil, specified and the role of the associated entity-type, teacher, omitted.

Example Query 6: A query involving a symmetric relationship

This example demonstrates the support for symmetric relationships in S.E.A.L. The relationship `‘friendship’` between two `robbers` has been declared as symmetric in the database specification: if `robber A` is friends with `robber B` then it follows that `robber B` is also friends with `robber A`.

Find robbers who are friends with 'Bugsy Malone'

Syntax:

```
SELECT nickname FROM robber ASSOCIATED_WITH(robber THROUGH
    friendship, <nickname='Bugsy Malone'>
```

Because the relationship used is symmetric it is not necessary (and would not make sense) to specify roles for the entities involved.

While S.E.A.L would still be useful without special support for symmetric relationships, the included support makes S.E.A.L more user friendly. The symmetric relationship `‘friendship’` is stored in a table similar to that shown in figure 6.2:

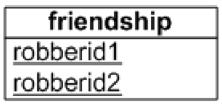


Figure 6.2: Representation of `friendship` relationship

- `robberid1` + `robberid2` forms the primary key of this table
- `robberid1` references `‘robber’` through a foreign-key referential integrity constraint named `‘friendship_to_robber1’`
- `robberid2` also references `‘robber’`, through a foreign-key named `‘friendship_to_robber2’`

Because the relationship `‘friendship’` is symmetric, for two `robbers`, `‘Al Capone’` and `‘Bugsy Malone’`, only one row in the table is required to store the facts:

1. `Al Capone` is friends with `Bugsy Malone`
2. `Bugsy Malone` is friends with `Al Capone`

For asymmetric relationships, the foreign keys must be named after the role the entity plays in the relationship (see section 6.2). However, because this relationship is symmetric, the foreign keys have been named arbitrarily.

Without support for symmetric relationships, all relationships would be treated as asymmetric. For the friendship example:

- Surrogate roles would need to be added to the relationship. For example, 'friend' and 'friendee'. This is undesirable as such roles do not exist in the real world
- A user wishing to represent a friendship relationship between `Bugsy Malone` & `Al Capone` would need to insert two rows into the friendship table: One with `Al Capone` as the `‘friend’`, the other with `Al Capone` as the `‘friendee’`

Assuming users always enter data correctly (inserting mirroring rows to represent symmetric relationships), the only modification to example query 6 required would be the inclusion of either of the surrogate roles:

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

It is only safe to arbitrarily choose *one* of the surrogate roles on the assumption that users always enter the two mirroring rows. If a user forgets to add one of the rows then the data will be inconsistent. To deal with this inconsistency the query would need to be rewritten as:

```
SELECT nickname FROM robber ASSOCIATED_WITH(robber AS friendee
   THROUGH friendship, <nickname='Bugsy Malone'> OR
   ASSOCIATED_WITH(robber AS friend THROUGH friendship,
   <nickname='Bugsy Malone'>
```

The benefits of language and database support for symmetric relationships are:

- Improved usability: It is not necessary to account for both halves of the symmetric relationship in queries. In addition, there is no need to refer to surrogate roles which have no real-world meaning.
- Storage is saved: Only one row is required to store two facts
- Integrity: The `‘friendship’` table is automatically kept consistent

6.1.2 Syntax for attribute predicate conditions inside specification

During the design of S.E.A.L, a positional syntax for giving attribute predicate conditions in a specification was considered. Under this design, the user would first list all attributes used in the specification expression. Specifications would then assign values to the attributes in the order the attributes were listed.

For example, the query:

"Find robbers with the skill 'Guarding' OR with any skill of difficulty '5'"

Would be written as

```
SELECT nickname FROM robber ASSOCIATED_WITH(skill,
   [skilldescription, difficulty], <'Guarding', ?> OR <?, '5'>)
```

(? represents a 'wildcard')

In the specification `‘‘<?, ‘5’>’’`, the value `‘5’` refers to the `‘difficulty’` attribute because it is the second attribute in the specification and the second attribute in `[skilldescription, difficulty]` is `‘difficulty’`. The original rationale for this syntax was that it will often be shorter: For a given query most of the specifications will involve the same attributes. However, this syntax was dropped in favour of the current syntax to improve usability. Because the attribute labels are next to attribute values (rather than implied by their position), queries in the new syntax are easier to understand, if a little more verbose:

```
SELECT nickname FROM robber ASSOCIATED_WITH(skill,
   <skilldescription='Guarding'> OR <difficulty='5'>)
```

6.1.3 Specifying or inferring the join table

The facility for inferring the relationship (or 'join table') in an entity-association query was not part of S.E.A.L from the onset: earlier versions required users to always explicitly specify both the associated entity-type and the relationship. The reasoning was that ambiguity is undesirable in most languages. However, queries are more difficult to write if the relationship must always be specified, since the user must know the name of the relationship. Supporting inference of the relationship when it is unambiguous makes the language much easier to use. The solution to the problem of ambiguity is for the inference mechanism to refuse to guess in ambiguous cases and simply report the ambiguity (and possible options) to the user.

6.1.4 Built-in support for EAV attributes

As described in Chapter 5, it was observed that queries involving EAV attributes can be implemented as entity-association queries. Without explicit support for EAV, S.E.A.L could still be used to answer queries involving EAV attributes. For example, example query 1 could be written as:

```
SELECT nickname FROM robber[age=39]
    ASSOCIATED_WITH(robber_attributes THROUGH robber_eav,
    <attribute='haircut', value='Mohawk'>)
```

However, while queries using EAV attributes can be considered entity-association queries from an implementation perspective, the semantics of the queries are different. The tables `robber_attributes` and `robber_eav` don't map to any real world entity-types/relationships. Additionally, the attributes `'attribute'` and `'value'` do not exist in the domain. EAV attributes have real-world applications, so special support was added so that they can be used seamlessly in place of regular attributes.

6.1.5 One association per ASSOCIATED_WITH clause

To make the syntax easier to understand as well as simplifying parsing and inference, a design decision was made that within an `ASSOCIATED_WITH` clause, all specifications must refer to the same associated entity-type & relationship. The only required part of the `ASSOCIATED_WITH` clause is the expression on specifications. The associated entity-type and the relationship are optional.

The following example (shown in figure 6.3) is illegal because it refers to two different associated entity-types (`'skill'` & `'food'`) within the one `ASSOCIATED_WITH` clause:

```
SELECT nickname FROM robber
    ASSOCIATED_WITH(<skilldescription='Explosives'> AND
    <name='Pizza'>)
```

Figure 6.3: An `ASSOCIATED_WITH` clause involving two different (associated entity-type, relationship) combinations

However, the requirement that an `ASSOCIATED_WITH` clause refers to exactly one associated entity-type and one relationship does not limit the expressiveness of S.E.A.L: the above query should be written as:

```
SELECT nickname FROM robber
    ASSOCIATED_WITH(<skilldescription='Explosives'>) AND
    ASSOCIATED_WITH(<name='Pizza'>)
```

Figure 6.4: Query 6.3 rewritten using only one (associated entity-type, relationship) combination per `ASSOCIATED_WITH` clause

6.2 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. With extra work, it would be possible to relax some of these expectations, however this was not done in this prototype.

1. Single-valued information that applies to all or most instances of an entity type should be stored as attributes in a table named after the entity-type. Singular rather than plural names are used for entity-type table names.

2. Sparse, multi-valued and user-defined attributes should be stored vertically using EAV & attribute lookup tables named after the entity to which the data pertains

3. All attributes (whether stored conventionally or in EAV form) belonging to an entity-type must have distinct labels. If an attribute is stored in EAV form then an attribute with the same label may not exist in conventional form and vice-versa.

4. EAV attribute lookup tables should be named *entity_type*_attributes and should contain two columns: `attributeid::integer` and `attribute::character varying`. A primary key should be defined on `attributeid`.

5. EAV tables should be named *entity_type*_eav and should contain columns `attributeid::integer`, `value::character varying`, `attributeid` should reference *entity_type*_attributes through a foreign key constraint. A foreign key referencing the primary key of the entity-type should be defined. A primary key should be defined on all attributes in the EAV table.

6. M:N relationships between entities must be modeled in 'join tables'. These tables must contain a foreign key referencing the primary key of each entity-type involved. For asymmetric relationships, *this foreign key must be named after the role of the entity-type in the relationship*. Join tables may also include additional columns storing information on the nature of the relationship/association. The key for the join table should include at a minimum the columns belonging to the foreign keys of participating entity-types.

7. By default, M:N relationships are assumed to be asymmetric, i.e. role based. The DBA needs to explicitly declare symmetric relationships by inserting a row into the system table *symmetric_foreign_keys*, specifying a pair of symmetric foreign keys.

6.3 Inference algorithms

S.E.A.L gives users considerable flexibility when specifying a participation constraint (`ASSOCIATED_WITH` clause). While the base entity-type and an expression on specifications is required, the following are optional:

1. The associated entity-type
2. The relationship between the base entity-type and the associated entity-type
3. The role of the base entity-type
4. The role of the associated entity-type

If any of the above are 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, assocatedEntityType, baseEntityRole, associatedEntityRole) combination for the base entity-type and attributes used.

6.3.1 Inferring the roles (foreign keys) to use when both the associated-entity and the relationship have been specified

If entity roles are omitted in a query, these must be inferred. Roles specify which foreign key in a relationship should be used to reference an entity-type participating in the relationship. The translation stage needs to know which foreign keys and attributes to use to join the relationship table to the base & associated entity tables.

The role inference algorithm (given in figure 6.5) takes a base entity-type, an associated entity-type and a relationship between them as input. If the relationship is asymmetric and involves more than one instance of the base entity-type or more than one instance of the associated entity-type then the algorithm reports the query as ambiguous. Otherwise it determines the foreign keys to reference the base entity-type and the associated-entity type. If the relationship is symmetric, the algorithm arbitrarily chooses suitable foreign keys to reference the base entity and the associated entity.

```
foreignKey baseEntityReference = null;
foreignKey asscEntityReference = null;

for foreign key $f \in relationship\_table$
  if (f.references(baseEntity))
    if (baseEntityReference == null)
      baseEntityReference = f;
    else if (f.references(asscEntity) && f.symmetricWith(baseEntityReference))
        asscEntityReference = f;
        break;
    else
        // Ambiguous, roles must be specified
    end
  else if (f.references(asscEntity))
    if (asscEntityReference == null)
      asscEntityReference = f;
    else
      // Ambiguous, roles must be specified
  end
end
```

Figure 6.5: The role inference algorithm

6.3.2 Inferring possible join tables between two entities

If a query is given with the relationship between the base entity-type and the associated-entity type omitted, S.E.A.L will attempt to infer the relationship (join table) to use. The following algorithm takes two entity types as input (the base entity-type and the associated entity-type) and returns a set of relationships involving both of these entity-types. A relationship is suitable if it has a foreign key f which references the base entity-type and has another foreign key f' which references the associated entity-type. This algorithm does not take the attributes used in the query into account.

For entity-types E_1 and E_2:

```
joinTableSet = {}
for table $t \in metadata \mid t.foreignKeySet.size \geq 2$
  if ($\exists f \in t.foreignKeySet \mid f.references(E_1) \bigwedge \exists f' \in t.foreignKeySet \mid f'.references(E_2) \bigwedge f' \neq f$)
    joinTableSet.add(t);
  end
end
```

Figure 6.6: Algorithm for inferring possible join tables between two entities

6.3.3 Inferring everything from the base entity-type and the attributes used

If a query is given with only the base entity-type and a specification expression, to be able to translate the query into SQL S.E.A.L must infer:

For a base entity-type, E:

```
combos<joinTable, fkToBaseEntity, fkToAssociatedEntity, associatedEntity> = {}

for table $t \in metadata | t.foreignKeySet.size \geq 2$
  for ($foreign\_key\ f \in t.foreignKeySet \mid f.references(E)$)
    for ($foreign\_key\ f' \in t.foreignKeySet \mid f' \neq f$)
      associatedEntityType = $f'$.referencedEntityType();
      combos.add(t, f, $f'$, associatedEntityType);
    end
  end
end
```

Figure 6.7: Combination enumeration algorithm

- The associated entity-type and relationship that the specifications refer to implicitly
- The roles of both the base entity-type and the associated entity-type.

The combination enumeration algorithm (shown in figure 6.7) takes the base entity-type as an input and returns a set of combinations of the form (*relationship*, *assocatedEntityType*, *baseEntityRole*, *associatedEntityRole*) representing relationships the base entity-type could possibly participate in. The algorithm works by looking for tables which could act as relationship (join) tables, then filters these for the ones which reference the base entity-type. It then enumerates all of the possible combinations.

The combinations are checked for attribute satisfaction by the attribute coverage algorithm (shown in section 6.3.4). If the set of combinations is not reduced to a single combination then the query contains ambiguity which the user must resolve.

6.3.4 Attribute coverage algorithm

For a given (`relationship`, `associatedEntityType`, `baseEntityRole`, `associatedEntityRole`) combination to be valid with respect to a query, the associated entity-type together with the relationship must contain (in either EAV or regular column form) all attributes referred to in the query. A combination with this property is said to 'cover' the query. The attribute coverage algorithm (shown below) checks a (`relationship`, `associatedEntityType`) pair to determine if it covers the query.

It is possible to qualify an attribute in a query by prefixing it with the location of the attribute (either an entity-type or a relationship) and a period '.'. For qualified attributes, the algorithm simply checks that the attribute exists where it was specified. For unqualified attributes, the algorithm checks that:

1. The attribute exists in either the associated entity-type OR the relationship
2. The attribute exists as either a regular column OR as an EAV attribute

Because checking for the existence of an EAV attribute requires a query against the database, attributes are grouped into sets corresponding to their expected location. All EAV attributes used in a query are checked at once rather than individually.

```
// Attributes expected to be found in the relationship's EAV table
```

```
rel_eav_atts = {}

// Attributes expected to be found in the associated-entity's EAV table
ae_eav_atts = {}

// Attributes expected to be found in EAV form, location not specified
eav_unspecified  = {}

for each attribute_name, a, used in expression
  if a.isQualified() // Has a location specified
    if a.location = relationship
      if relationship.hasAttribute(a)
        Mark a as non—EAV
      else
        Add a to rel_eav_atts
        Mark a as EAV
      end
    end
    if a.location = associated_entity
      if associatedEntity.hasAttribute(a)
        Mark a as non—EAV
      else
        Add a to ae_eav_atts
        Mark a as EAV
      end
    else
      Discard Combination
    end
  else // unqualified attribute (location omitted)
    if relationship.hasAttribute(a) && associatedEntity.hasAttribute(a)
      if associated_entity.primaryKey.contains(a)
        a.location = associatedEntity
        Mark a as non—EAV
      else
        Alert user to ambiguity
      end
    else if relationship.hasAttribute(a)
      a.location = relationship
      Mark a as non—EAV
    else associatedEntity.hasAttribute(a)
      a.location = associatedEntity
      Mark a as non—EAV
    else
      Add attribute to eav_unspecified
      Mark a as EAV
    end
  end
end

atts_found_in_relationship_EAV = relationship.checkEAV(rel_eav_atts)
```

```
atts_found_in_associatedEntity_EAV = associatedEntity.checkEAV(ae_eav_atts)

if (rel_eav_atts \ atts_found_in_relationship_EAV) != {}
  Discard Combination

if (ae_eav_atts \ atts_found_in_associatedEntity_EAV) != {}
  Discard Combination

eav_found_in_rel  = joinTable.checkForEAVAttributes(eav_unspecified)
eav_found_in_ae   = associatedEntity.checkForEAVAttributes(eav_unspecified)

// Check {eav_found_in_rel, eav_found_in_ae} is a partition on eav_unspecified

if !eav_unspecified ⊆ (eav_found_in_rel ⋃ eav_found_in_ae)
  Alter user that some EAV attributes could not be found

if eav_found_in_rel ⋃ eav_found_in_ae != {}
  Alter user that some EAV attributes were ambiguous — appear in both rel & ae

return combination
```

6.4 Translation algorithms

Once a query has been parsed and all necessary inference is completed, translation into SQL is performed by a recursive toString() call on 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.

6.4.1 Checking satisfaction of a participation constraint for an asymmetric relationship

The following algorithm is used to generate the SQL to determine whether an entity satisfies a specification. The algorithm takes a base entity-type, an associated entity-type, a relationship between the base entity-type and the associated entity-type, roles for both entity-types and a set of predicate conditions in the specification. The algorithm generates SQL which checks for the presence or absence of a row in the relationship table which satisfies the specification. A row in the relationship table satisfies the specification if it references the base entity-type, references an instance of the associated entity-type and all predicate conditions are satisfied.

Given:

- A base entity `'E'`, E_{table}
- An associated entity-type `'A'`, represented in table A_{table} with primary key A_{PK}
- An EAV table belonging to entity-type `'A'`, A_{eav_table}
- A relationship between `'E'` and `'A'`, `'RS'` represented in table RS_{table}

- An EAV table belonging to relationship RS, represented in table RS_{eav_table}
- 'A' acts as role 'R' in relationship 'RS', represented by foreign key RS_{R_FK}
- A set of predicate conditions of the form (att1=value, att2=value...), P.

1. During the inference stage all attributes used in P are located in exactly one of (A_{table}, A_{eav_table}, RS_{table}, RS_{eav_table})
2. An inner join is performed between RS_{table} and A_{table} on foreign key RS_{R_FK}=A_{PK}. From this join a select is performed using all non-EAV conditions in P. The result set is further restricted by checking the EAV attribute conditions [6.4.5]. The result is the set $S_{a\_in\_rs\_sat\_p}$: instances of 'A' involved in a relationship 'RS' satisfying all conditions in P.

If a role R_E was specified for E, E satisfies the mandatory participation constraint if there exists a row in RS_{table} such that the foreign key R_E references E and RS_{R_FK} references one of $S_{a\_in\_rs\_sat\_p}$.

If no role was specified for E, E satisfies the mandatory participation constraint if there exists a row in RS_{table} such that there is a foreign key other than RS_{R_FK} which references E and RS_{R_FK} references one of $S_{a\_in\_rs\_sat\_p}$.

If the participation constraint was a forbidden participation constraint rather than mandatory, then E would satisfy it if no such row exists.

6.4.2 Checking satisfaction of a participation constraint for a symmetric relationship

The process for determining whether an entity satisfies a symmetric relationship participation constraint is similar to that for checking an asymmetric relationship constraint. The key difference is that a symmetric relationship (symmetric relationships are defined in Chapter 2) involves entities of the same type acting in the same role, therefore it is not significant which foreign key references the base entity-type and which foreign key references the associated entity-type, only that they are *distinct*.

Given:

- A base entity-type 'E' represented in table E_{table}
- An associated entity-type which is also E
- An EAV table belonging to entity-type 'E', E_{eav_table}
- A symmetric relationship between two or more entities of entity-type 'E', 'RS' represented in table RS_{table}
- An EAV table belonging to relationship RS, RS_{eav_table}
- A set of predicate conditions of the form (att1=value, att2=value...), P.

1. During the inference stage all attributes used in P are located in exactly one of (E_{table}, E_{eav_table}, RS_{table}, RS_{eav_table})
2. An arbitrary foreign key, f, belonging to RS_{table} which references E is selected
3. An inner join is performed between RS_{table} and E_{table} on foreign key RS_{R_FK}=E_{PK}. From this join a select is performed using all non-eav conditions in P. The result set is further

restricted by checking the EAV attribute conditions using the algorithm in section 6.4.5. The result is the set $S_{e_i n_r s_s a t_p}$: instances of 'E' involved in a relationship 'RS' satisfying all conditions in P.

E satisfies the mandatory participation constraint if there exists a row in RS_{table} such that there exists a foreign key f which references E and another foreign key f' such that $f' \neq$ f, which references one of $S_{e\_in\_rs\_sat\_p}$.

If the participation constraint was a forbidden participation constraint rather than mandatory, then E would satisfy it if no such row exists.

6.4.3 Ensuring primary/foreign keys for IN are correct

The S.E.A.L database design guidelines (given in 6.2) do not assume that attributes belonging to a foreign key will have the same label as the attributes in the corresponding primary key. In fact, when a relationship involves two or more entities of the same entity-type, this is not possible due to the fact that column names must be unique within a table.

For example in the `robber` schema, the primary key of the `robber` entity-type is `'robberid'` and the `mentoring` relationship has two foreign keys referencing the `robber` entity-type on columns named `'robberid1'` & `'robberid2'`.

A secondary problem is that if a foreign key contains more than one attribute, then the ordering of these attributes is significant when used in an IN clause. For example, the first SQL fragment given directly below is semantically correct while the second is not:

```
SELECT * FROM robber_has_skill WHERE (robber_has_skill.robberid,
   robber_has_skill.skillid) IN (SELECT
   robber_has_skill_tested.robberid,
   robber_has_skill_tested.skillid FROM robber_has_skill_tested
   ...)
```

```
SELECT * FROM robber_has_skill WHERE (robber_has_skill.skillid,
   robber_has_skill.robberid) IN (SELECT
   robber_has_skill_tested.robberid,
   robber_has_skill_tested.skillid FROM robber_has_skill_tested
   ...)
```

During the metadata collection phase, for every attribute in each foreign key the corresponding attribute belonging to the primary key of the referenced table is recorded. This allows the S.E.A.L to refer to the attributes in the correct order and using the correct labels.

6.4.4 Nested associations

The approach to dealing with nested associations is consistent with the recursive translation of the tree: if an association has a nested association, instead of selecting tuples from the table representing the relationship, tuples are selected from the subquery generated by the nested association. These subqueries require an alias:

```
... FROM (SELECT ...) as <alias_name>
```

Because alias names are of little significance in automatically generated SQL, S.E.A.L simply appends a global post-incremented integer onto the name of the table for alias names.

6.4.5 Translation of EAV attributes

As mentioned in Chapter 3, the use of EAV attributes can (at least from an implementation standpoint) be considered an entity-association query. While the actual implementation differs slightly, this pseudocode illustrates the concept:

Given: v

- An entity-type or relationship, 'E' represented in table E_{table}
- An EAV table belonging to 'E', represented in table E_{EAV_table}
- An EAV attribute lookup table belonging to E, represented in table $E_{EAV\_attributes\_table}$
- A set of EAV attribute predicate conditions of which apply to E of the form (attribute='value'), C

Note: In the following pseudocode, text between % and % in strings is evaluated.

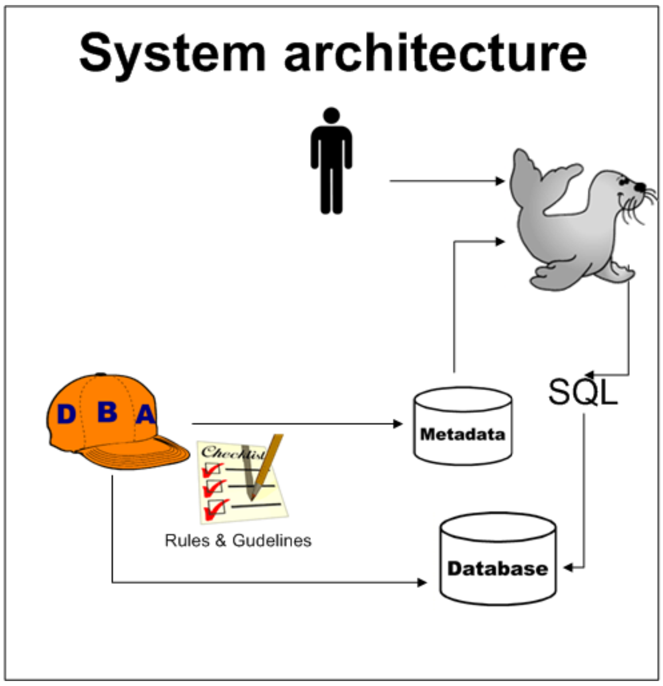
```
for condition $c \in C$
  E.add("ASSOCIATED_WITH($E_{EAV\_attributes\_table}$ THROUGH $E_{EAV\_table}$, <attribute='%c.attribute%'
end
```

Chapter 7

Implementation

7.1 System architecture

The following diagram gives an overview of how the S.E.A.L interpreter interacts with the user and the PostgreSQL database:



On startup, the S.E.A.L interpreter connects to the PostgreSQL database and collects metadata (described in section 7.2.1). User queries are then parsed and translated. If any ambiguity or errors are found in the query, S.E.A.L gives suggestions on how to resolve these. Otherwise, S.E.A.L outputs SQL which can be run against the database.

7.2 S.E.A.L interpreter prototype

Originally the intention was to implement the S.E.A.L interpreter directly in PostgreSQL. While this would result in a more efficient system overall, this was not feasible under the time constraints of an undergraduate honours project. The prototype (written in Object-Oriented Java code with JDBC database connectivity) demonstrates the principles of the translation and provides a solid foundation for future implementation at the DBMS level.

After metadata has been retrieved, the translation of a query is comprised of three main stages:

1. Parsing
2. Inference
3. Rewriting

7.2.1 Retrieval of metadata

One of the reasons for choosing PostgreSQL to implement the prototype S.E.A.L interpreter is the ease of accessing metadata. PostgreSQL provides an '`information schema`': a collection of views and tables describing the structure of databases contained in the 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 the following:

1. Table names
2. Attribute names
3. Table primary keys
4. Table foreign keys (constraint name, referencing attributes, referenced table, referenced attributes)
5. For each foreign key, a set of foreign keys with which it is symmetric

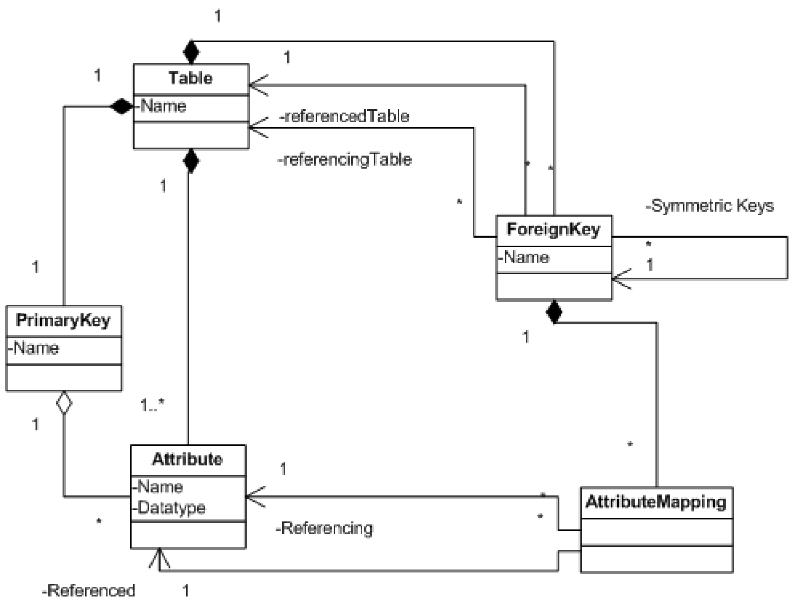


Figure 7.1: Metadata representation in the S.E.A.L Interpreter

All metadata is stored in an OO datastructure (shown in figure 7.1) which is globally accessible throughout the interpreter. EAV attributes *are not* collected during the metadata acquisition phase. As mentioned in Chapter 3, one of the key applications for EAV is representing extremely sparse datasets. An entity-type may have many thoussands of attributes: it is simply not feasible nor worthwhile to retrieve all of these on startup. Any EAV attributes used in a query are checked at translation time (described in section 6.3.4)

7.2.2 Parsing

The parser for the interpreter was created with the ANTLR [1] parser generator. ANTLR allows the programmer to write a context free grammar (CFG) 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, relationships and roles with tables, join-tables and foreign key names (respectively) contained in the global metadata structure. Instead of creating *Association* objects, the parser generates *TemporaryAssociation* objects due to the nature of the inference algorithms, described in section 7.2.3.

For example, query:

```
SELECT nickname FROM robber ASSOCIATED_WITH(skills THROUGH
   robber_has_skill, <skillid=5> OR <skillid=3>)
```

Is represented by the following tree (after inference):

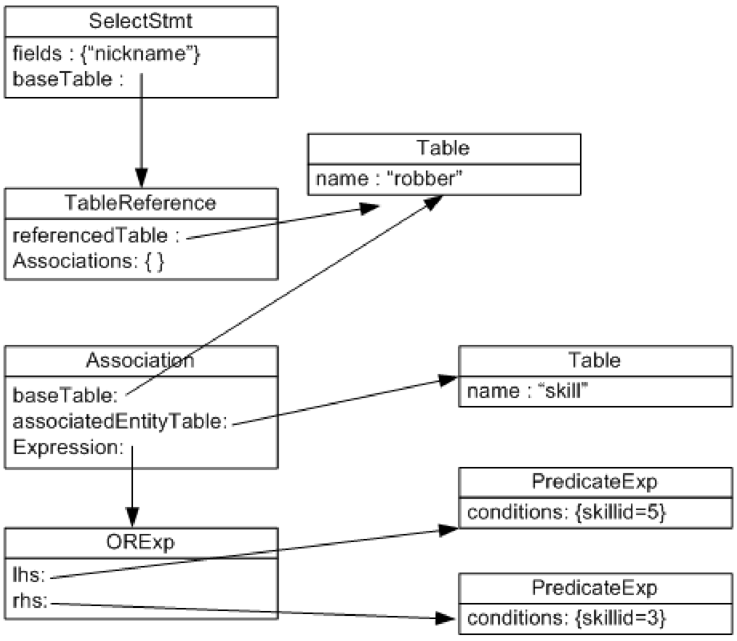


Figure 7.2: OO Representation of the query tree after inference

7.2.3 Inference

As described in section 6.3, inference in S.E.A.L works from left to right. The general case of inference in S.E.A.L can be approximated as "Given this base entity-type (left hand side) and the set of attributes used in the query, what are the possible associated entity-types, relationships and roles which would cover the attributes used?". Because the inference works from left to right, the entire query tree must be parsed before inference can take place: the base entity-type must be known before the relationships it is involved in can be deduced.

An example query involving nested associations:

```
SELECT nickname FROM robber
    ASSOCIATED_WITH(<skilldescription='Explosives'>
    ASSOCIATED_WITH(<name='Harvard'>));
```

In this example, the interpreter is expected to infer that the `‘robber’` entity-type is associated with a `‘skill’` entity-type through the `‘robber_has_skill’` relationship and also that the `‘robber_has_skill’` entity-type is associated with the `‘testing_location’` entity-type.

If the interpreter attempted to perform the inference during parsing (bottom up), it would parse the predicate expression `‘‘name=‘Harvard’’’` first, followed by the enclosing `ASSOCIATED_WITH` clause. At this point, the interpreter would try to find an associated entity A and relationship R such that $attributesinquery \subseteq attributesfromA \cup attributesfromB$. However, as described in section 6.3, to find the possible relationships (and therefore possible associated entity-types), the interpreter would need to know the base entity-type. Because the base entity-type has not yet been parsed, the interpreter is unable to look for relationships and associated entity-types.

To solve this problem, the interpreter performs an initial parse which stores for each table reference and each expression a set of TemporaryAssociations. After this initial parse, the interpreter executes a recursive call on the query tree which performs all necessary inference from left to right, converting the TemporaryAssociations to Associations in the process. Figure 7.3 shows the steps involved in translating a user query.

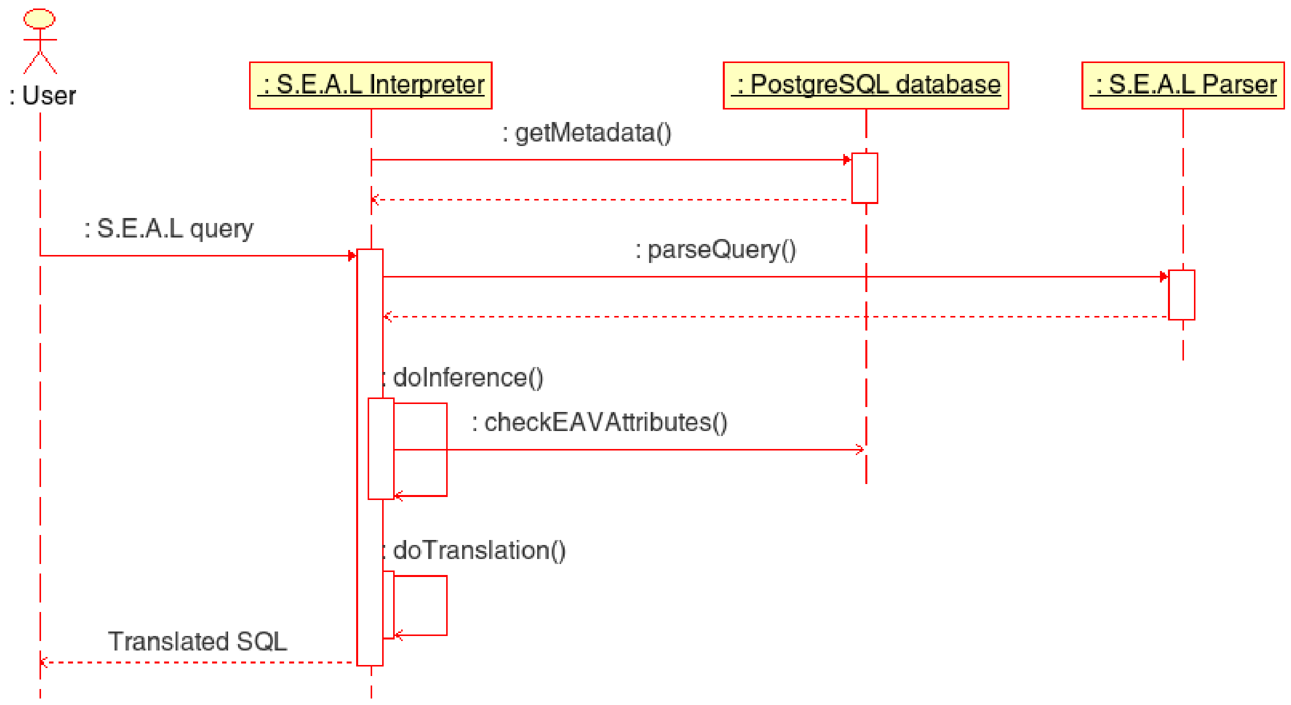


Figure 7.3: Steps involved in translating a user query

Chapter 8

Testing

Test system configuration

The following system was used for all tests:

- Processor: Intel Pentium 4 @ 3.2GHz
- Memory: 1.5Gb DDR2 @ 533 MHz
- Disk: Seagate 80Gb ST380819AS (SATA)
- Operating system: Net-BSD 4.99.9
- PostgreSQL: Version 8.2.4

8.1 Performance of Nested-IN strategy compared with Set theoretic operator strategy

An experiment was conducted to compare the performance of the 'nested IN' and 'set theoretic operator' strategies described in Chapter 4.

8.1.1 Experiment setup

For the experiment, a new schema '`eav_performance`' with the following tables was defined:

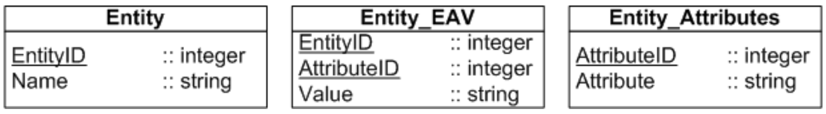


Figure 8.1: Schema used for testing subquery performance

A Java program was then used to populate the tables with random string data:

- 3,000 unique entities were created
- 2,500 unique attributes were created

- 1,000,000 rows were inserted into the table `entity_eav comprised of random combinations of an entityid, an attributeid and one of 2500 randomly generated string values.`

8.1.2 Experiment 1: EAV performance

A number of queries involving randomly chosen (attribute, value) pairs (indicative of queries involving EAV attributes) were executed. An example of one such query with the corresponding implementations is given below:

Find entities with dmhxphx='gpmbc' AND gynizct='pjvau'

Query written using the nested-IN strategy:

```
SELECT name from ENTITY where entityID in (SELECT entityID from
   entity_eav inner join entity_attributes on
   entity_eav.attributeid = entity_attributes.attributeid where
   attribute='dmhxphx' AND value='gpmbc') AND entityID in (SELECT
   entityID from entity_eav inner join entity_attributes on
   entity_eav.attributeid = entity_attributes.attributeid WHERE
   attribute='gynizct' AND value='pjvau');
```

Query written using the set theory operator strategy:

```
SELECT name FROM entity INNER JOIN (SELECT entityID from entity_eav
    inner join entity_attributes on entity_eav.attributeid =
   entity_attributes.attributeid where attribute='dmhxphx' AND
   value='gpmbc' INTERSECT SELECT entityID from entity_eav inner
   join entity_attributes on entity_eav.attributeid =
   entity_attributes.attributeid WHERE attribute='gynizct' AND
   value='pjvau') as foo ON entity.entityid = foo.entityid;
```

On the test system, the difference in performance was negligible with both implementations taking approximately 3ms to execute.

8.1.3 Experiment 2: Entity-association query performance

The SQL from Experiment 1 was modified so that it did not include the condition on the value attribute (belonging to the entity_eav table). The justification for this was that many entity-association queries will only involve attributes from the associated entity-type.

On the test system, the nested-IN implementation took approximately 12.3ms, marginally outperforming the set theory implementation which took 15.8ms.

8.2 Interpreter overhead as a function of query execution time

Multiple experiments were performed on the robber 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. The robber schema has very little data: the smallest table has 5 rows while the largest table has 40 rows.

- A basic association query with no inference took approximately 1ms to translate and 0.4ms to execute. The same query with inference on the associated attribute took 4ms to translate.
- An association query involving nesting (with no inference) took 4ms to translate and 0.3ms to execute.
- A query involving nested associations and EAV attributes belonging to the associated entity-type took only 2ms to translate and 0.4ms to execute.

The S.E.A.L interpreter was then used against the 'eav_performance' schema described in the previous section to perform the queries retrieving entities based solely on EAV attribute conditions.

As an indicative result, the query

> Find entities with dmhxphx='gpmbc' AND gynizct='pjvau'

Took 1ms to translate and 12.3ms to execute.

Chapter 9

Conclusions

9.1 Contributions

This report describes entity-association queries: an important class of query which restrict entities based on their participation in relationships with other entities.

- EAV storage is described and the use of EAV is justified. The flexibility and practicality of various EAV storage implementations are compared. It is shown that from an implementation perspective, queries involving EAV attributes may be considered a special case of an entity-association query.
- The report describes S.E.A.L, a declarative language for expressing entity-association queries. With S.E.A.L, end-users can issue this important class of query without a high level of database proficiency and without detailed knowledge of the database implementation: only knowledge of the *conceptual schema* is required. A significant feature of S.E.A.L is that it abstracts EAV storage: in queries, there is no difference between attributes stored in EAV form and attributes represented in regular columns. To make writing queries easier still, S.E.A.L allows parts of the query to be omitted and will perform inference where the query given is unambiguous.
- The syntax of S.E.A.L is given, along with example queries which motivate language features of S.E.A.L such as symmetric relationships, roles and EAV attribute support.
- A set of recommendations and rules for designing databases compatible with S.E.A.L are provided. These describe how to model entities and their relationships with other entities, as well as how to cope with sparse, multi-valued or user-defined attributes.
- A prototype interpreter for S.E.A.L is described along with problems encountered during implementation. The algorithms used by the interpreter for inference an translation are described.

9.2 Comparison with previous work

During the literature search, no previous attempts to classify entity-association queries or implement a general solution for allowing novice users to express these queries were found. Previous works on EAV attribute abstraction have all been GUI-driven and 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 this report. However, while S.E.A.L provides generic support for EAV attributes, it is not as flexible as some of the other implementations. In particular, all attributes are represented as strings and the only comparison operator supported is the equality operator, `'='`.

9.3 Future work

While this work provides a general solution to supporting entity-association queries including basic support for EAV attributes, there are a number of areas which can be expanded on. In particular, the EAV storage solution could be extended to support datatypes other than string and support for comparison operators other than equality could be implemented.

Further development of EAV functionality in S.E.A.L will necessitate EAV specific DDL. By introducing DDL with EAV support, it will be possible to specify whether an attribute is single or multi-valued and the type of the attribute (string, integer, date etc). DDL should also be extended to allow the database administrator to declare a relationship as symmetric: the current solution of inserting a row into a system table is somewhat inelegant.

Extending DML to cope with EAV is also a top priority for future work. Currently, updating and inserting data involving EAV attributes is complex and requires knowledge of the database implementation.

The problem of efficiently retrieving and displaying complete entity data remains open. It is not clear what should be included in 'complete entity data'. Should this be limited simply to the attributes of the entity, or should it include relationships the entity is involved in? Because entity relationships form a graph, there is no obvious way to display this data to the user in a concise manner.

Appendix

Full ANTLR grammar for S.E.A.L

```
grammar AssociatedWith;
selectStatement
  : selectKeyword fieldList fromClause? ;

selectKeyword
  : 'SELECT' ;

fromKeyword
  :   'FROM' ;

associatedWithKeyword
  : 'ASSOCIATED_WITH' ;

equalsKeyword
  :   '=' ;

fromClause
  : fromKeyword tableReference ;

fieldList
  : (fieldName)(FieldDelimiter fieldName)* | '*' ;

fieldName
  : (String)+ ;

tableReference
  : tableRestriction ('AS' entityRole)? (associatedWithKeyword LP
     (tableName ('AS' entityRole )? ('THROUGH' tableName)?
     FieldDelimiter)? expression RP )? ;

tableRestriction
  : table
  | table '[' tableAttributeExpression ']' ;

table
  : tableName ;
```

```
tableName
  : (String)+ ;

tableAttributeExpression
  : tableAttributeOrExpression ;

tableAttributeOrExpression
  : tableAttributeAndExpression
    ('OR' tableAttributeAndExpression)* ;

tableAttributeAndExpression
  : tableAttributeNotExpression ('AND'
     tableAttributeNotExpression)* ;

tableAttributeNotExpression
  : 'NOT' tableAttributeAtom
  | tableAttributeAtom ;

tableAttributeAtom
  : condition
  | LP tableAttributeExpression RP ;

expression
  : orexpression ;

orexpression
  : andexpression ('OR' andexpression)* ;

andexpression
  : notexpression ('AND' notexpression)* ;

notexpression
    : 'NOT' atom
    | atom ;

atom
  :  simplePredicate (associatedWithKeyword LP (tableName ('AS'
     entityRole )? ('THROUGH' tableName)? FieldDelimiter)?
     expression RP )?
  |  LP expression RP (associatedWithKeyword LP (tableName ('AS'
     entityRole )? ('THROUGH' tableName )? FieldDelimiter)?
     expression RP )? ;

condition
  : (tableName '.')? fieldName equalsKeyword data ;

simplePredicate
  : LT condition (',' condition)* GT ;

String
```

```
  : ('a'..'z' | 'A'..'Z' | '_') ;

Number
  : ('0'..'9') ;

entityRole
  : String+ ;

stringData
  :   (String |  DP | '+' | '!' | '?' | Number | '-')+ ;

numericalData
     :   Number+
     |   Number* DP Number+ ;

data
  : SQ stringData? SQ
  | numericalData ;


FieldDelimiter
  : ',' ;

LP  : '(' ;
RP  : ')' ;
LSB : '[' ;
RSB : ']' ;
LT  :    '<' ;
GT  :    '>' ;
SQ  :    '\' ';
DP  :    '.' ;

WS : (' '|'\t'|'\u000C') { $channel=HIDDEN; };
```

Bibliography

[1] ANTLR: ANother tool for Language Recognition (http://www.antlr.org).

[2] Codd, E. F. Normalized data structure: A brief tutorial. In *Proceedings of 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, California, November 11-12, 1971* (1971), E. F. Codd and A. L. Dean, Eds., ACM, pp. 1–17.

[3] Dinu, V., and Nadkarni, P. Guidelines for the effective use of entity-attribute-value modeling for biomedical databases. *International Journal of Medical Informatics In Press, Corrected Proof* (2006), 1056.

[4] John Corwin, Avi Silberschatz PhD, P. L. M. M. P. L. M. M. Dynamic tables: An architecture for managing evolving, heterogeneous biomedical data in relational database management systems. *Journal of the American Medical Informatics Association* (2007).

[5] Nadkarni, P., Brandt, C., Frawley, S., Sayward, F., Einbinder, R., Zelterman, D., Schacter, L., and Miller, P. Managing attribute–value clinical trials data using the ACT/DB client-server database system. *J Am Med Inform Assoc 5*, 2.

[6] Nadkarni, P. M. QAV: querying entity-attribute-value metadata in a biomedical database. *Computer Methods and Programs in Biomedicine* (1996).

[7] Prakash M. Nadkarni, M., and Cynthia Brandt, MD, M. Data extraction and ad hoc query of an entity-attribute-value database. *Journal of the American Medical Informatics Association* (1998).

[8] Valentin Dinu, Prakash Nadkarni, C. B. Pivoting approaches for bulk extraction of entity-attribute-value data. *Computer Methods and Programs in Biomedicine 82* (2006).

[9] Wikipedia. Entity — wikipedia, the free encyclopedia, 2007. [Online; accessed 15-September-2007].