VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

School of Mathematics, Statistics and Computer Science
Te Kura Tatau

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@mcs.vuw.ac.nz

Extending and Evaluating S.E.A.L.

Kunal Madhav

Supervisors: Dr Pavle Mogin, Dr Peter Andreae

October 16, 2009

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

Abstract

Structured Query Language (SQL) is a declarative language that is widely used for querying relational databases. SQL is poorly suited for entity association queries and queries against EAV structures. To construct these queries, users have to understand the details of the database implementation and require advanced SQL skills. Simple Entity Association Language (S.E.A.L.) is a simple query language that supports these kinds of queries and allows non database-expert users to construct queries easily. This report provides extensions to the S.E.A.L. syntax and its interpreter to support a wider range of SQL queries. In addition, we evaluate S.E.A.L. and provide evidence that it is easier to use than SQL.

Contents

Figures

Chapter 1

Introduction

1.1 The Problem

Structured Query Language (SQL) is a powerful declarative language that is widely used for querying relational databases. It can be used by technical and non-technical users with a modest level of SQL knowledge to retrieve relevant information from a database.

Most often we need to retrieve information from multiple entity types so we must associate entity types using an entity association query. An entity association query is a query that contains one or more join conditions which are used to associate one entity type with another. Users must know how to use these join conditions and what prerequisites apply in order to use a join.

There is a wide range in the complexity of entity association queries ranging from simple queries to queries that are too complex for users to construct. A simple entity association query is one that does not contain a conjunctive condition on a single attribute. An example of a complex entity association query is, retrieve all attributes from tableA and tableB where tableA.attributeC has the values ′value1′ and ′value2′. This is considered to be a complex query because the user needs to find all instances that satisfy the condition tableA.attributeC = ″value1″, find all instances that satisfy the condition tableA.attributeC = ″value2″ and then intersect the two sets.

Entity association queries become complex when we specify conjunctive conditions on a single attribute because SQL does not have a declarative syntax for expressing this class of queries in a simple way. Even more experienced database users can struggle with defining these types of queries in SQL. Therefore we claim that SQL is poorly suited for entity association queries with a conjunctive condition on a single attribute.

SQL's poor performance with defining entity association queries with conjunctive conditions are more evident when defining queries against Entity Attribute Value (EAV) structures. EAV structures are a technique used for storing sparse attributes efficiently. Whereas dense attributes are stored with the entity type itself using the conventional relational approach, sparse attributes are stored in a separate entity type.

There is a need to store sparse data in areas such as medicine, e-commerce and genetics because databases used in these areas usually contain a large number of sparse data. This data needs to be efficiently stored, easily accessible and needs to allow for real-time updates; requirements that a conventional relational database will not be able to support. Since we need to store sparse data we need EAV structures because they provide better support for handling sparse data than conventional relational databases.

Defining SQL queries for EAV structures involves the same problems as described before because the queries required are entity association queries that contain conjunctive conditions. Therefore we claim that SQL is poorly suited for queries against EAV structures.

We need a simple query language that is similar to SQL but reduces the complexity that SQL introduces when defining entity association queries with conjunctive conditions and queries against EAV structures. S.E.A.L. (Simple Entity Association Language) is a declarative language that addresses these problems because it enables users to easily define entity association queries with conjunctive conditions and queries against EAV structures.

1.2 Previous Work

An implementation of a S.E.A.L. interpreter has been created by a previous honours student, Edward Stanley[7]. The original S.E.A.L. interpreter defines the basic syntax for S.E.A.L. and many algorithms for translating S.E.A.L. queries into equivalent SQL queries. On start-up, the interpreter queries the PostgreSQL database to retrieve the meta-data which is used for constructing the SQL queries. The interpreter uses a parser that was created with the ANTLR[1] parser generator which takes a context free grammar and generates Java code. This generated code along with other Java code enables the interpreter to accept S.E.A.L. queries as input, parse the queries and then translate the queries into SQL queries.

The original S.E.A.L. interpreter had a number of limitations. Addressing these limitations will enable S.E.A.L. to cover a wider range of SQL queries making the language more useful. We have identified the following limitations:

- no support for comparison operators other than the equality operator in attribute constraints,
- no support for relationship cardinalities other than M:N,
- only base entity type attributes can appear in the attribute list within the SELECT clause,
- no support for relationship types implemented as EAV structures, and;
- no support for queries that contain logical combinations of associations

In addition to the limitations of S.E.A.L., Edward Stanley[7] has not provided any evidence for the claim that S.E.A.L. is easier to use than SQL. Functional and performance tests were conducted on the interpreter by Edward Stanley[7] however these tests were of a limited nature as well.

1.3 My Contributions

I am going to continue work on S.E.A.L. by addressing the limitations listed in section 1.2. I will also conduct an extensive usability test on S.E.A.L. and its interpreter using its current state and the extensions I will make to the interpreter. The goal of the usability test is to support our underlying claim that S.E.A.L. is easier to use than SQL for a class of database queries. This test will be performed with a group of 400 level computer science students playing the role of advanced database users and with a group of 300 level computer science students playing the role of novice users. After implementing the identified features (section 1.2) an analysis will be carried out comparing and contrasting SQL and S.E.A.L. Upon completing the implementation of the missing features, the S.E.A.L. interpreter will undergo functional and performance tests. The results from the usability, functional and performance tests may raise a number of missing features that perhaps I could implement or be left for future work.

Chapter 2

Running Example

We will use the following information about robbers as our example throughout the report.

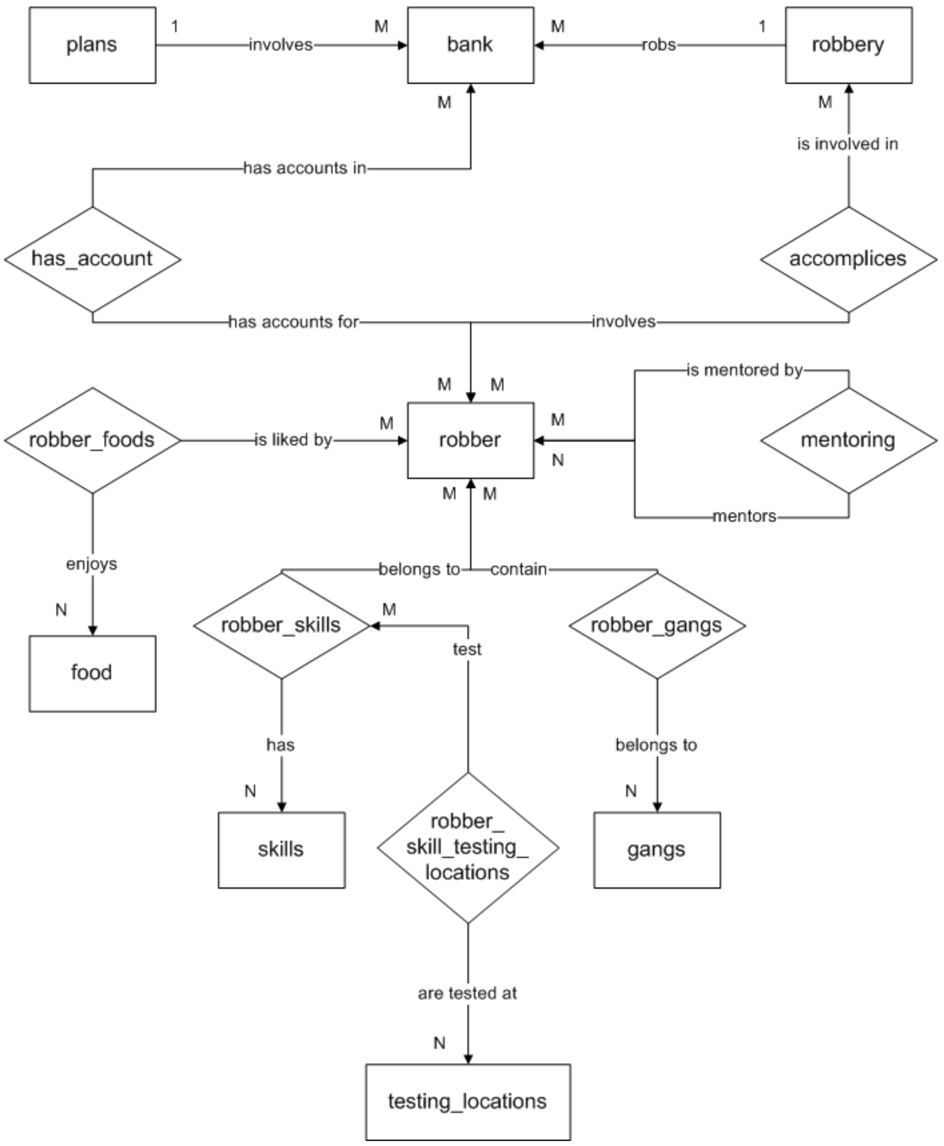


Figure 2.1: Database schema of our robbers example

2.1 Implementation Schema

accomplices (robberid, bankname, city, rdate, share)
bank (bankname, city, accounts, security)
foods (foodid, foodname)
gangs (gangid, gangname)
has_account (robberid, bankname, city)
mentoring (robberid2, robberid2)
plans (bankname, city, pdate, numrobbers)
robber (robberid, nickname, age, prisonyears, first_name, last_name)
robber_attributes (attributeid, attribute)
robber_eav (robberid, attributeid, value)
robber_foods (robberid, foodid)
robber_gangs (robberid, gangid)
robber_skill_testing_locations (robberid, skillid, locationid)
robber_skills (robberid, skillid, skillpreference, skilllevel)
robber_skills_attributes (attributeid, attribute)
robber_skills_eav (robberid, skillid, attributeid, value)
robbery (bankname, city, rdate, amount)
robbery_attributes (attributeid, attribute)
robbery_eav (bankname, city, rdate, attributeid, value)
skills (skillid, skilldescription)
testing_locations (locationid, locationname)

2.1.1 Referential Integrity Constraints

accomplices [robberid] references robber [robberid]
accomplices [bankname, city, rdate] references robbery [bankname, city, rdate]
has_account [robberid] references robber [robberid]
has_account [bankname, city] references bank [bankname, city]
mentoring [robberid1] references robber [robberid1]
mentoring [robberid2] references robber [robberid2]
plans [bankname, city] references bank [bankname, city]
robber_eav [robberid] references robber [robberid]
robber_eav [attributeid] references robber_attributes [attributeid]
robber_foods [robberid] references robber [robberid]
robber_foods [foodid] references foods [foodid]
robber_gangs [robberid] references robber [robberid]
robber_gangs [foodid] references gangs [gangid]
robber_skill_testing_locations [robberid] references robber [robberid]
robber_skill_testing_locations [robberid] references skills [skillid]
robber_skill_testing_locations [robberid] references testing_locations [locationid]
robber_skill [robberid] references robber [robberid]
robber_skill [robberid] references skills [skillid]
robber_skills_eav [robberid, skillid] references robber_skills [robberid, skillid]
robber_skills_eav [attributeid] references robber_skills_attributes [attributeid]
robbery [bankname, city] references bank [bankname, city]
robbery_eav [bankname, city, rdate] references robbery [bankname, city, rdate]
robbery_eav [attributeid] references robbery_attributes [attributeid]

Chapter 3

Definitions

To provide a clear understanding of the concepts used throughout this report we will provide the following definitions

Entity Type

The term entity type will be used for describing a concept that we model in a database. For example, robber, bank, skills, robber_skills and testing_locations from figure 2.1 are examples of an entity type.

Entity Instance

The term entity instance or instance will be used for describing an instantiation of an entity type. For example, Al Capone and Bugsy Malone are examples of a robber instance while Loanshark Bank is an example of a bank instance.

Relation Scheme

A relation scheme is a set of attribute names and a set of constraints that describe an entity type. For example, the robber relation scheme is Robber {robberid, nickname, age, prisonyears, first_name, last_name} and C; where C is a set of constraints that restrict the range of attribute values.

Relation

A relation is a set of tuples of a relation scheme.

Table

A table is a graphical representation of a relation. For example, below we have a table for the robber relation and each row represents a tuple:

robberid	nickname	age	prisionyears	first_name	last_name
1	A.C	45	12	Al	Capone
2	Bugsy	21	0	Bugsy	Malone

Tuple

A tuple t is defined as: t: R $\rightarrow \bigcup_{i \epsilon R}Dom(A_i)$ such that for each A_i ϵ R t(A_i) ϵ dom(A_i) and dom(A_i) is a set of values for A_i. A tuple is a set of pairs where each pair contains an attribute name and a value from dom(A_i). If we know the order of the attribute names and the attribute names are known then we can use this shorthand notation: (1, A.C, 45, 12, Al,Capone) a tuple from the robber relation

Chapter 4

Background

4.1 SQL

Structured Query Language (SQL) is declarative language that is widely used for querying relational databases. SQL has two classes Data Manipulation Language (DML) and Data Definition Language (DDL). DML is used for manipulating data in a database by retrieving, inserting or deleting data. DDL is used for defining new relations, altering existing relations and deleting either a whole relation or parts of the relation [9]. We will be focusing on the DML class of SQL because this is the SQL class that end users need to use to retrieve data; from now on when we say SQL we refer to DML.

SQL can be used by technical and non-technical users to retrieve information from databases. The SQL syntax allows for selecting one or many columns from one or many tables and to filter the information contained in these columns. Information can be clustered into informative groups and given an order. SQL also provides ways to perform aggregate functions on information which will provide valuable information to the user.

4.2 Entity Association Queries

An entity association query is a query that involves an association(s) between entity types. We need to associate entity types because in most cases a single entity type does not store all of the information that we require. When we associate entity types we aim to produce tuples that contain the required information. An example of an entity association query is;

retrieve all nicknames of robbers that have the 'Planning' skill.

Here we need to associate the Robber and Skills entity types because we want to retrieve data about Robbers based on an attribute constraint for a Skills attribute. We associate the entity types using the relationship type Robber_Skills as shown;

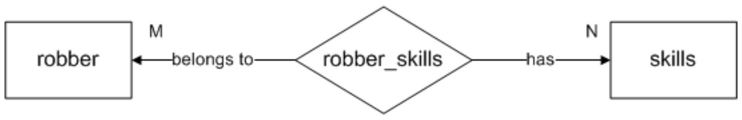


Figure 4.1: An entity relationship diagram of Robber, Robber_Skills and Skills entity types

```
SELECT nickname
FROM robber NATURAL JOIN robber_skills NATURAL JOIN skills
WHERE skilldescripion = 'Planning';
```

Defining this kind of entity association query in SQL is not hard and generally people can generate them without too many problems because this is a simple example. However the complexity of defining an entity association query becomes apparent when we have a conjunctive condition on a single attribute.

For example,

> retrieve all nicknames of robbers that have 'Planning', 'Gun-Shooting', 'Preaching' and 'Driving' skils.

Some naive SQL users would claim that they could solve this query by;

```
SELECT nickname
FROM robber NATURAL JOIN robber_skills NATURAL JOIN skills
WHERE skilldescripion = 'Planning' AND skilldescription = 'Gun-Shooting' AND
skilldescripion = 'Preaching' AND skilldescription = 'Driving';
```

It is important to note that the user's intention is correct because they are trying to retrieve robbers that have all skills. However the SQL query is incorrect because we cannot produce a tuple that contains multiple skilldescription attribute values.

We are aware of at least four methods to solve this query using; set theory, nested SELECT statements using the IN operator, nested SELECT statements using the EXISTS operator and nested SELECT statements using an equi-join. We will generate the correct results using the nested SELECT statements method by producing four sets of robbers; one set of robbers per skilldescription and return the robbers that appear in each set.

For example;

```
SELECT nickname FROM robber WHERE (
((robber.robberid) IN (SELECT robber_skills.robberid FROM robber_skills INNER JOIN skills ON robber_skills.skillid = skills.skillid
WHERE skills.skilldescription = 'Planning'))
AND
((robber.robberid) IN (SELECT robber_skills.robberid FROM robber_skills INNER JOIN skills ON robber_skills.skillid = skills.skillid
WHERE skills.skilldescription = 'Gun-Shooting'))
AND
((robber.robberid) IN (SELECT robber_skills.robberid FROM robber_skills INNER JOIN skills ON robber_skills.skillid = skills.skillid
WHERE skills.skilldescription = 'Preaching'))
AND
((robber.robberid) IN (SELECT robber_skills.robberid FROM robber_skills INNER JOIN skills ON robber_skills.skillid = skills.skillid
WHERE skills.skilldescription = 'Driving')));
```

In order to define a SQL query containing a conjunctive condition on a single attribute, the user needs to understand that they have to handle each condition individually. In addition, the user must have database implementation knowledge and advanced SQL skills to define such a query. It is this requirement that makes SQL complex and poorly suited for entity association queries that contain conjunctive conditions on a single attribute.

4.3 Entity Attribute Value Structures

Entity Attribute Value (EAV) structures store data in a different way to conventional relational databases by using two entity types and one relationship type; entityTypeName,

entityTypeName_attributes & entityTypeName_eav. EAV structures focus on the attribute types of an entity type; Strings, Floats, Integers [3] and whether an attribute is dense or sparse. The entityTypeName entity type stores all dense attributes of an entity type using the conventional relational approach for storing attributes (each attribute is a column). The entityTypeName_attributes entity type stores all sparse attribute names using a vertical approach [3] where each attribute name is actual data [5]. These entity types are associated by a relationship type (entityTypeName_eav) through a many-to-many relationship cardinality. When discussing EAV structures it is important to have a clear understanding of the different classifications of an attribute.

4.3.1 What is a sparse attribute?

Attributes are either single-valued or multi-valued and can be classified as either sparse or dense attributes. An attribute is considered to be sparse if we only know its value for a few entity instances out of many instances; regardless of whether we know all of the possible attribute values or just a few.

For example, a haircut attribute is a single-valued and sparse if we do not know the haircut values for every Robber instance. An example of a multi-valued sparse attribute is music because a Robber instance can like several music types and we do not know which music types every Robber likes.

4.3.2 How to handle sparse attributes?

A problem that arises with sparse attributes is how to efficiently store them. Suppose we would like to store the following attribute about the base entity type Robber: Nickname, Age, Date of birth, Haircut (single-valued sparse), Favourite types of music (multi-valued sparse) and Favourites car brands (multi-valued sparse).

In a conventional relational database we would store each of the (single-valued or multi-valued) sparse attributes in separate relations and associate them with a Robber instance through primary key-foreign key pairs. However we encounter many problems with the conventional design;

- If a new multi-valued sparse attribute like favourite country needed to be stored then we would need to create a new relation for the attribute. This involves accessing the database schema, creating a new relation and inserting the new instance(s)

- The number of relations increase with each new sparse attribute. If we had many sparse attributes say 100 each would need their own relation.

- Defining a query such as retrieve Robber names that have a 'spiked' haircut, like 'rock' and like the car brand 'Holden' would require a complex query. The more sparse attributes we require the greater the complexity of the query.

The best way to handle sparse attributes is with EAV structures because we could represent all of the information with just two entity types and one relationship type; Robber, Robber_Attributes and Robber_EAV.

Robber

robberid	nickname	age	dob
1	John	45	1/04/64
2	Jim	21	4/07/87
3	Joe	50	2/08/594

Robber_Attributes

attributeid	attribute
1	Haircut
2	Music
3	Car

Robber_EAV

robberid	attributeid	value
1	1	Mullet
1	2	Rock
1	2	Pop
1	3	Ford
2	1	Spiked
2	2	Rock
2	3	Subaru
2	3	Holden
3	3	Holden

Figure 4.2: The Robbers information represented in an EAV structure

The Robber entity type is the same as in the conventional relational database design.

The Robber_Attributes entity type stores all of the multi-valued sparse attributes that belong to any Robber instance. If we needed to add a new attribute like favourite countries then we would need to perform a simple insert into the Robber_Attributes relation. The fundamental principle of the Robber_Attributes entity type is that each attribute (Haircut, Music and Car) is not treated as attributes but as values of the attribute called 'attribute'. The value 'Haircut' in the first tuple is treated the same as the nickname 'John' from the Robbers relation. As the attributes are treated as values, users can insert, modify and delete these values as required with simple SQL INSERT, UPDATE and DELETE queries. Also valuable database space is not wasted with an entity type for each multi-valued sparse attribute.

The Robber_EAV is a relationship type that associates a Robber instance with some (possibly all) of the attributes from the Robber_Attributes entity type. The actual values of the attributes (Haircut, Music and Car) are stored in this relationship type along with the robberid and the attributeid. The combination of robberid, attributeid and a value is used as the primary key because a Robber instance can have many different values for an attribute.

As a result, we can conclude that EAV database structures provide us with a better approach for handling (single-valued or multi-valued) sparse attributes than the conventional relational database design.

4.3.3 Why do we need EAV structures?

There is a need for EAV structures in various areas such as medicine, e-commerce, neuroscience and genetics [3, 5, 8] because there is a large number of attributes which are sparse. For example in a medical database only a small number of the possibly thousands of symptoms can apply to any particular patient instance [3, 5]. We need to be able to store all of these symptoms, even if a single symptom is associated to a single patient. Using the conventional relational database design with the one attribute per column approach would result in an relation having thousands of columns. As relational databases have limits on the number of columns (1024 or less) per relation [4] this approach will not work. To add to the problem there is a tendency for these attributes to change [3] and these updates need to be completed in real-time [5]. Therefore we need to efficiently store the sparse attributes to allow efficient updates. The need to store sparse attributes leads to the conclusion that we need EAV structures because they provide clear advantages of handling sparse attributes over the conventional relational database approach.

4.3.4 How to query EAV databases?

We can represent the entityTypeName, entityTypeName_eav and entityTypeName_attributes types similar to the relations shown in figure 4.1 and query this representation in the same manner.

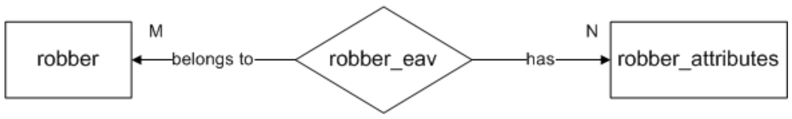


Figure 4.3: An entity relationship diagram of the Robber, Robber_EAV and Robber_Attributes entity types

A simple entity association query is required to query this structure which is manageable, however the pitfall of using SQL for querying EAV structures is illustrated in the following example.

> Retrieve nicknames of Robbers that have a 'mullet' haircut, like 'rock' music and like 'pop' music

In this query we have a conjunctive condition on a single attribute from the Robber_Attributes entity type and a conjunctive condition on a single attribute (music). The attribute called 'attribute' is part of a conjunctive condition because each haircut, music and car brand are just values for the attribute and the condition specifies different values for 'attribute'. Similarly the music attribute is part of a conjunctive condition because the condition specifics two values for music. To solve this query we need to find three sets of Robber instances; one with haircut = 'mullet', another with music = 'rock' and the last set with music = 'pop'. The required SQL query is very complex and is similar to the nested SELECT query shown in section 4.2.

We concluded from section 4.2 that SQL is poorly suited for entity association queries that contain conjunctive conditions on a single attribute. As EAV queries are a class of entity association queries that have a conjunctive condition on a single attribute we can also conclude that SQL is poorly suited for queries against EAV structures.

Chapter 5

Problem

We can conclude from subsection 4.3.4 that there is little difference between defining a simple entity association query in SQL and defining a simple EAV query in SQL. We can map the EAV structure to the conventional many-to-many relationship and query this relationship with a simple SQL query.

However as the EAV queries become more complex through conjunctive conditions the SQL syntax required to define these queries also becomes complex. In fact, the complexity increases dramatically depending on the number of entity types and conditions involved in the query. The example from subsection 4.3.4 demonstrates a more common type of an EAV query because this query will return more meaningful information. People often struggle when defining this kind of query because of the complexity involved and the SQL skills required to define the query.

One source of complexity is associating entity types, although using NATURAL JOIN seems to be an easy option, its use is limited because it relies on the primary and foreign key pairs having the same attribute names. On the other hand the nested SELECT statement using IN, EXISTS or equi-join does not rely on primary and foreign key pairs having the same name but it does rely on the user to specify exactly how to join the entity types. This makes things harder for the user because they need to know how to use the nested SELECT statements and what attributes to use for joining the entity types. Queries become increasingly more complex when we have a conjunctive condition on a single attribute. As a result the user must have sufficient knowledge regarding the database implementation and advanced SQL skills in order to successfully associate the entity types.

Even if the user has vast knowledge about the database implementation and knowledge of SQL they can still have trouble with these queries. Due to these complexities we conclude that SQL is poorly suited for entity association queries and queries against EAV structures. What is needed is a simple query language that supports these kinds of queries and allows non database-expert users to construct queries easily.

S.E.A.L. (Simple Entity Association Language) is a proposed language that addresses these kinds of queries. The language was developed by a previous honours student, Edward Stanley[7], but the implementation was incomplete and there was no evidence presented that S.E.A.L. was actually easier to use than SQL.

Chapter 6

Related Work

6.1 Dynamic tables: an alternative database design to EAV structures

Dynamic tables store sparse attributes using column-based storage; this design is also called decomposed storage [3]. The purpose is to exploit the knowledge that users have for horizontal storage, from conventional relational databases where each attribute is a column name. With a dynamic table the sparse attribute is the column name for the table and the values of the sparse attribute make up the rows in the dynamic table. The benefit of this approach is that it is easy to see what the sparse attributes are because each sparse attribute is stored separately unlike in the EAV structure where all sparse attributes become values of an attribute.

To demonstrate dynamic tables we would represent the sparse attributes of our robbers as such;

Car

robberid	**car**
1	Ford
2	Subaru
2	Holden
3	Holden

Haircut

robberid	**haircut**
1	Mullet
2	Spiked

Music

robberid	**music**
1	rock
1	pop
2	rock

Robber

robberid	**name**	**age**	**dob**
1	John	45	1/04/64
2	Jim	21	4/07/87
3	Joe	50	2/08/594

Figure 6.1: The robbers database represented with dynamic tables

These dynamic tables can be used like any conventional table so the user does not need to care about the differences between the dynamic (Car, Haircut, Music) and normal tables (Robber). The user does however need to know whether an attribute is dense or sparse, what attributes an entity can have and how to correctly associate the tables.

Although dynamic tables store sparse attributes separately they inhibit the same problems as discussed in subsection 4.3.2. We still need to manipulate sets for querying dynamic tables when dealing with conjunctive conditions on a single attribute. As a result, dynamic tables do not reduce the complexity involved when defining SQL queries that contain conjunctive conditions on a single attribute.

6.2 Query-by-Example

Query-by-example (QBE) is a query language proposed by IBM [6] and the main difference between QBE and other database query languages is that QBE has a graphical user interface (GUI). This GUI allows users to create example tables [6] through drag and drop features to make it easier for users to define SQL queries. QBE is implemented in many database applications such as Microsoft Access. The QBE interface in Microsoft Access provides the tables in the database from which the user can choose attributes, place conditions on attributes, perform grouping and aggregate functions.

To test the power of QBE in Microsoft Access we set up the robbers information with a relational database.

We used QBE in Microsoft Access to,

> retrieve nicknames of robbers that have a 'spiked' haircut, like 'rock' music and like the car brand 'Holden'

The SQL query produced by QBE yields the correct result;

```
SELECT Robber.nickname
FROM ((Robber INNER JOIN Haircut ON Robber.robberid = Haircut.robberid)
INNER JOIN Car ON Robber.robberid = Car.robberid) INNER JOIN Music ON
Robber.robberid = Music.robberid
WHERE (([Haircut]![Haircut]='Spiked') AND ([Music]![Music]='rock') AND ([Car]![Car]='Holden'));
```

We then tested QBE with the following query,

> retrieve nicknames of robbers that have a 'mullet' haircut and like 'rock' and 'pop' music

The SQL query produced by QBE was;

```
SELECT Robber.nickname
FROM (Robber INNER JOIN Haircut ON Robber.robberid = Haircut.robberid)
INNER JOIN Music ON Robber.robberid = Music.robberid
WHERE (([Haircut]![Haircut]='Mullet') AND ([Music]![Music]='rock') AND ([Mu-
sic]![Music]='pop'));
```

This query returns an empty record because the query generated is incorrect. QBE cannot create a single tuple that contains the correct Robber instance even though such an instance exists. The reason for this is that we have a conjunctive condition on a single attribute (music).

We can conclude that QBE is a good tool to use for simple SQL queries involving a few tables and non-conjunctive conditions. However QBE cannot create a query with a conjunctive condition on a single attribute in a single step. We can create a conjunctive query using a stepwise approach where we break down the query into views and then join the views to get the result. The problem with this approach is that we must use sophisticated tools in order to accomplish this task which violates the point of QBE.

In addition to QBE, Microsoft Access provides a SQL text editor for users to generate SQL queries that QBE cannot create. As a result we still rely on the users to manually define these complex queries, for that reason QBE does not reduce the complexity involved when defining SQL queries with conjunctive conditions on a single attribute.

6.3 Query Kernel

Query Kernel (QK) is a tool created with a GUI that allows users to define SQL queries against the ACT/DB database; a database that manages clinical trial data [5]. The aim of QK is to address problems associated with querying EAV structures. With QK the user does not need to know whether an attribute is stored in the entity or the EAV entity because the responsibility of creating the SQL query is on the tool [5].

QK generates the required SQL query by first creating a query for each selection condition the user specifies [5]. For example, suppose we want to select those patients that were used in a trial on date x AND on date y. The selection conditions are patients used on date x and patients used on date y; QK generates a SQL query for each condition and stores the patient Ids in a temporary table [5]. These tables are then combined using set theoretic operators based on the Boolean expression given by the user [5]; in our example the expression was AND.

These individual SQL queries are generated by predetermined SQL templates [5] that contain many place holders to hold the attributes and/or conditions the user specifies. For example; a template could be SELECT 1 FROM 2 WHERE 3 and using the example from above, QK would generate the following query for the first condition; SELECT patientID FROM patient WHERE date = 'x';. These templates are then combined to create the resulting SQL query.

The benefit of QK is that it removes the dependency on the user having knowledge about the database schema because it uses meta-data to generate the SQL queries dynamically [5]. QK also removes the necessity that the users have extensive SQL knowledge for creating complex SQL queries. However the main problem of QK is that it is specifically designed for the ACT/DB schema thus would not be of any use for our robbers database or any other database without considerable changes.

6.4 Querying the universal relation

The universal relation is a hypothetical relation [2] that exists in every database whereby the single relation contains all of the attributes that the relational database stores. For example, suppose we would like to store the following information about robbers; nickname, age and date of birth. Then our universal relation would be; Robber nickname, age, dob. This eliminates any entity associations created by integrity constraints [2] and removes the complexity involved with associating entities.

As a result, querying the universal relation is much simpler than querying relational databases because we only need to specify which attributes and conditions we want without specifying the association between relations [2]. Wisconsin data integration kit (Windik) is a tool created to display the universal relation to the users through a GUI who then select attributes and place conditions. This eliminates the need for the users to have knowledge of the database schema and SQL skills. Windik then translates the given information into a SQL query. The translation algorithm used by Windik is a graph based approach where an adjacency matrix is created for every relation in the relational database [2]. The cells of the matrix contain a '1' if there exists an association between the relations. This matrix is used for creating the SQL query where a '1' requires a join between the two relations [2].

There are limitations with the Windik tool in that conjunctive conditions on a single attribute are not handled nor are set theoretic operators. Also it is unclear how to handle sparse attributes. However this tool provides a good starting point for querying the universal relation and removing the complexity involved with defining SQL queries.

Chapter 7

Justification for the Project

There is an abundant need to store sparse data especially in healthcare databases and there is the need to retrieve that data at will. As we have seen relational databases cannot efficiently store sparse data and the design causes many problems for changing sparse data. EAV structures are a better alternative for handling sparse data because of the way sparse data is stored and the design allows data to be changed easily.

SQL is poorly suited for entity association queries and querying EAV structures because the queries required become very complex, the user requires vast knowledge about the database schema and the user must have good SQL skills. Simple Entity Association Language (S.E.A.L.) is a new language that aims to remove these problems associated with defining SQL queries. S.E.A.L. queries are not complex and the user does not need to have extensive knowledge about the database schema. Another advantage of S.E.A.L. is that it supports entity association queries as well as queries against EAV structures.

An implementation of the S.E.A.L. interpreter has been created by Edward Stanley[7] that translates S.E.A.L. queries into SQL queries. Similar to QK (section 6.3), S.E.A.L. uses meta-data for constructing the SQL queries and the interpreter contains many inference algorithms that infer which entity types to use in the SQL query. Therefore the user does not need to provide any join conditions or even care about the relationships between entity types, as they are inferred. The user also does not need to worry about where an attribute is stored; either in the entity type itself or an EAV entity type. By allowing the interpreter to do all of the hard work we reduce the dependency on the user and the complexity of defining entity association queries and queries against EAV structures.

Chapter 8

Extensions

The original S.E.A.L. interpreter defines the basic syntax for S.E.A.L. and many algorithms for translating S.E.A.L. queries into equivalent SQL queries. On start-up, the interpreter queries the PostgreSQL database to retrieve the meta-data which is used for constructing the SQL queries. The interpreter uses a parser that was created with the ANTLR[1] parser generator which takes a context free grammar and generates Java code. This generated code along with other Java code enables the interpreter to accept S.E.A.L. queries as input, parse the queries and then translate the queries into SQL queries.

The original S.E.A.L. interpreter had a number of limitations. We need to address these limitations so that S.E.A.L. can support a wider range of SQL queries thus making the language more useful. This chapter describes my extensions to the language and its interpreter.

8.1 S.E.A.L. Syntax

The S.E.A.L. syntax is a declarative syntax that allows the interpreter implementation to change without affecting the user. A declarative syntax is used to support our primary goal of making it easier for users to query databases. Below is the extended S.E.A.L. syntax.

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

An *associationClause_expr* has the following form:

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

The *baseEntityRestriction_expr* has the following form:
$[\ NOT\ ]\ attribute\ [\ > \mid < \mid >= \mid <= \mid != \mid =\ ]\ 'value'\ [\ AND \mid OR\ ]$

The *specification_expr* has the following form:
$<\ [\ NOT\ ]\ attribute\ [\ > \mid < \mid >= \mid <= \mid != \mid =\ ]\ 'value'\ >\ [\ AND \mid OR\ ]$

- attribute: this is a list of entity type attributes
- baseEntityType: this is a list of entity types
- baseEntityRestriction_expr: this is a restriction(s) placed on some or all of the base entity type attributes using Boolean expressions
- baseEntityRole: this is the role that the base entity type must play in the relationship
- associatedEntityType: the entity type that is associated (participating in a relationship) with the base entity type
- associatedEntityRole: the role that the associated entity plays in the relationship
- relationship: the name of the relationship entity type between the base entity and the associated entity
- specification_expr: this is a restriction(s) placed on some or all of the attributes belonging to the associated entity and/or relationship entity

8.2 Extension 1: support for various comparison operators in attribute constraints

The original interpreter only allows the equality operator (=) to be used for attribute constraints. Therefore if we wanted to select the robbers who are aged between 50 and 60 years old then we would need to define the following;

```
SELECT nickname, first_name, last_name
FROM robber [age=50 OR age=51 OR ... age=59 OR age=60]
```

The disadvantage of only supporting the equality operator is that we can not use any of the following operators >, <, >=, <= or ! = in a S.E.A.L. query. This is an important extension because SQL supports all of the listed operators but the original S.E.A.L. interpreter does not.

In order to implement this extension I;

- changed the context free grammar to allow the user to input one of >, <, >=, <=, ! = and = as a comparison operator for each attribute constraint
- changed the translation algorithm so that the generated SQL query includes the appropriate comparison operator for each attribute constraint

Now we can find the robbers who are aged between 50 and 60 years old by defining the following;

```
SELECT nickname, first_name, last_name
FROM robber [age>=50 AND age<=60]
```

We can also include different combinations of comparison operators;

```
SELECT nickname, first_name, last_name
FROM robber [age>=50 AND age<=60 AND robberid! =10]
```

8.3 Extension 2: support for relationship cardinalities other than M:N

The original S.E.A.L. interpreter has been implemented to handle many-to-many (M:N) relationship cardinalities only. This is a common relationship cardinality however there are other common relationship cardinalities too, such as one-to-many (1:M) and one-to-one (1:1). These two relationship cardinalities are also supported by SQL so S.E.A.L. too should support these, as it will enable the extended S.E.A.L. interpreter to translate a wider range of queries into SQL queries and thus become more useful.

In order to implement this extension I extended the inference algorithm to handle 1:M and 1:1 relationship cardinalities. The amount of inference performed by the extended S.E.A.L. interpreter depends on what information the user has omitted from the S.E.A.L. query.

My extension to the inference algorithm performs the following steps;

- if the user has omitted both the associated and the associating entity types;
  1. determine whether the query involves a M:N relationship cardinality,
  2. if not M:N, determine whether the query involves a 1:1/1:M relationship cardinality,
  3. if not 1:1/1:M, throw an error
- if the user has omitted the associating entity type only;
  1. determine whether the query involves a M:N relationship cardinality,
  2. if not M:N, determine whether the query involves a 1:1/1:M relationship cardinality,
  3. if not 1:1/1:M, throw an error
- if the user has not omitted any entity types;
  1. determine whether the query involves a M:N relationship cardinality,
  2. if not M:N, throw an error

The implementation of the check *determine whether the query involves a M:N relationship cardinality* was already implemented in the original S.E.A.L. interpreter and will not be discussed further.

The check *determine whether the query involves a 1:1/1:M relationship cardinality* was implemented using many checks. We would use the primary key and foreign key pairs of tables in order to determine whether we have a 1:1/1:M relationship cardinality. We do not differentiate between 1:1 and 1:M relationship cardinalities because there is no difference between them in the implementation. We take the base entity type from the inputted S.E.A.L. query and infer all entity types that are referenced by and reference the base entity type. Taking the inferred entity types we determine which pair of entity types (base + inferred entity type) contains all of the attributes given in the input query. This check returns a positive result if all attributes are contained within a single entity type pairing. If more than one entity type pairing contains all of the attributes then an ambiguous error is thrown.

During the implementation of this extension we discovered an association that occurs when a database has a sequence of tables that form a path of references. For example, tableA references tableB that references tableC that references tableD; tableA->tableB->tableC->tableD. We called this association a sequence of related tables and in our running example we have Accomplices that references Robbery that references Bank. For this association

S.E.A.L. needs to find a path from a given table to another table. For example a valid S.E.A.L. query is;

```
SELECT share
FROM accomplices ASSOCIATED_WITH(<security='Excellent'>)
```

The extended interpreter would infer that the associated entity type is Bank since security is an attribute of Bank. The interpreter would then infer that we have a sequence of related tables association because the checks for M:N and 1:1/1:M would fail. The interpreter performs a breadth-first traversal search through the database beginning at the given base entity type table, Accomplices. The search uses the primary key-foreign key pairs to traverse through the database until it reaches the associated entity type, Bank. The search determines which way the sequence of related tables flow. For example, the search algorithm determines whether the sequence is Accomplices->...->...->Bank or Bank->...->...->Accomplices. It is important to note that only a single path is selected by the algorithm even if multiple paths exist and the path selected may not be the most efficient or the path that the user wanted. A future extension would enable the user to choose which path of tables they would like to use if such a situation occurs during processing.

Another kind of association we discovered occurs when we have a common table that is being referenced by two other tables which we describe as X->?<-Y. Here we have two tables represented by X and Y which both reference an unknown table represented by '?'. For example, the Plans and Robbery tables from our running example can be the X and Y tables where as the unknown '?' table is the Bank table. For this association the extended S.E.A.L. interpreter needs to infer which table to use for the '?' table. A valid S.E.A.L. query for such an association is;

```
SELECT numrobbers
FROM plans ASSOCIATED_WITH(<amount=1000>)
```

Here the extended interpreter would infer that the associated entity type is Robbery and that the association is X->?<-Y since the checks for M:N, 1:1/1:M and sequence of related tables would fail. The interpreter will infer the '?' table by finding all tables that X (Plans) references and all tables that Y (Robbery) references; next it determines which table, called a common table, appears in both reference lists. If we have more than one common table, the interpreter throws an ambiguous error otherwise it will take the common table and use it to join the X and Y tables.

Upon testing this association we realised that a join of X->?<-Y will result in a lossy join; a join where some of the results are valid and some are invalid. We decided to leave this implementation in the extended S.E.A.L. interpreter because SQL would allow such a lossy join. We did make an improvement over SQL, in that the extended S.E.A.L. interpreter will give the user a warning and ask them if they really do want a lossy join when it discovers a X->?<-Y association. This will enable S.E.A.L. to be more useful and provide meaningful feedback for the user than SQL would in such a situation.

The sequence of related tables and X->?<-Y associations were implemented by further extending the inference algorithm. In the sequence of steps performed by the algorithm we replaced the step *if not 1:1/1:M, throw an error* with;

- if not 1:1/1:M, determine whether the query involves a sequence of related tables association,
- if not a sequence of related tables, determine whether the query involves a X->?<-Y association,
- if not X->?<-Y, throw an error

8.4 Extension 3: allowing the attribute list after the SELECT clause to contain attributes other than base entity type attributes

The original S.E.A.L. interpreter constrains the user to select attributes that belong to the base entity type only. Attributes of associated and associating entity types can only be used in attribute expressions but not in the SELECT clause. This is a major disadvantage of the original interpreter because it is very common to select attributes that belong to many entity types.

In order to implement this extension I;

- changed the context free grammar to allow the FROM clause to contain at least one table name,
- extended the query translation algorithm to;
  - check if the user inputs a '*' in the SELECT clause, in which case all of the attributes of each table given in the FROM clause are selected,
  - determine which attributes given in the SELECT clause belong to the table(s) given in the FROM clause and prefix those attributes,
  - if there attributes that are not prefixed, infer a table that is not given in the FROM clause, has no foreign keys and contains one or more of the attributes given
  - if there attributes that are not prefixed, infer a table that is not given in the FROM clause, has foreign keys and contains one or more of the attributes given
  - if there attributes that are not prefixed, throw an error
  - once all attributes are prefixed, determine the associations between the tables listed in the FROM clause and the inferred tables

We can allow users to omit tables in the FROM clause because one of the S.E.A.L. database design rules claims that no two attributes can have the same name unless they belong to a primary key - foreign key relationship. Therefore if an attribute in the SELECT list belongs to more than one table, it belongs to a referencing relationship and it does not matter which table prefixes the attribute.

In the extension we check tables that contain no foreign keys before tables that do contain foreign keys because if an attribute is in a primary key - foreign key relationship then we want to select the table that is being referenced. For example, a query selecting robberid will prefix the attribute with Robber rather than Robber_Skills or Accomplices or Robber_Foods because Robber is being referenced by Robber_Skills, Accomplices and Robber_Foods.

8.5 Extension 4: support for relationship types implemented as EAV structures

In our running example, the Robber and Robbery entity types have been implemented using an EAV structure. The original interpreter can successfully translate S.E.A.L. queries that contain either Robber or Robbery EAV attributes into SQL queries. The restriction that the original interpreter has is that the entity types that are implemented using EAV structures cannot be relationship types. Relationship types can be implemented using EAV structures and they are very common in databases because (M:N) relationships are common. Therefore S.E.A.L. should be able to handle queries involving relationship types that are implemented using EAV structures.

In order to implement this extension I;

- extended the inference algorithm to infer relationship types that are implemented using EAV structures
- extended an algorithm, that checks attributes in a relationship type, to also check for any EAV attributes of the relationship type

8.6 Extension 5: support for queries containing logical combinations of associations

A design decision was made during the initial stages of development for the original interpreter that each ASSOCIATED_WITH clause can only contain an attribute expression(s) for attributes that belong to a single pair of associating and associated entity types. The disadvantage of this decision is that it compromises the expressiveness of S.E.A.L. since we require separate ASSOCIATED_WITH clauses for each association. For example if we want to

> retrieve robbers that have either the 'Planning' or 'Explosives' skill and like both 'Mexican' and 'Pizza' and belong to either 'The Blue Gang' or 'The Red Gang'

We would need to define the following S.E.A.L. query;

```
SELECT nickname, first_name, last_name
FROM robber ASSOCIATED_WITH(<skilldescription='Planning'> OR <skilldescription='Explosives'>)
AND ASSOCIATED_WITH(<foodname='Mexican'> AND <foodname='Pizza'>)
AND ASSOCIATED_WITH(<gangname='The Blue Gang'> OR <gangname='The
Red Gang'>)
```

With the original design, users have to know which attributes belong to an associating and associated entity type pair and to group them in an ASSOCIATED_WITH clause. In addition, it is unclear why we need to have three separate ASSOCIATED_WITH clauses when all three relate to the base entity type, Robber.

We decided to alter the design to allow a single ASSOCIATED_WITH clause to contain attribute expressions that may belong to many pairs of associating and associated entity types. This will make the S.E.A.L. query easier to understand and more intuitive.

In order to implement this extension I;

- changed the translation algorithm to split the combination of associations into groups of associations based on associating and associated entity types,
- processed each group of associations to generate a sub-query
- built the resulting SQL query using each sub-query generated

We have allowed users to include brackets '(' & ')' around logical combinations of associations, similar to the way we use brackets in Mathematics, to make it easier to understand the query and to generate the correct results. As a result we can rewrite the query above as;

```
SELECT nickname, first_name, last_name
FROM robber ASSOCIATED_WITH(
(<skilldescription='Planning'> OR <skilldescription='Explosives'>) AND
(<foodname='Mexican'> AND <foodname='Pizza'>) AND
(<gangname='The Blue Gang'> OR <gangname='The Red Gang'>))
```

Chapter 9

Analysis Of SQL and S.E.A.L.

In this chapter we analyse S.E.A.L. and SQL. We aim to make the S.E.A.L. syntax SQL-like because we to take advantage of the SQL knowledge a user already has.

The S.E.A.L. syntax is given in section 8.1 and the SQL syntax is given in the appendix.

9.1 SELECT clause

Both the S.E.A.L. and SQL syntax require the SELECT clause to appear first in any query. The purpose of the SELECT clause is to allow the user to input which attributes they would like the result set to contain. This can be done by selecting all attributes using a '*' or by listing each attribute where each attribute is separated by a ','. The S.E.A.L. SELECT clause is the same as in SQL because this is easy to understand and usually does not cause problems when defining a SQL query.

9.2 FROM clause

The FROM clause in the S.E.A.L. syntax is similar to the FROM clause defined in the SQL syntax. The purpose of this clause is to allow the user to list the table(s) that contain the data that they want. The FROM clause is required for both S.E.A.L. and SQL queries and must follow the SELECT clause.

The difference between the two languages is that in SQL the user is required to list all tables that are referenced in the query. Therefore the FROM clause must list table names of the tables that stores the attribute(s) listed in the SELECT clause and any tables whose attributes are used in join conditions.

With S.E.A.L., the user is not forced to list all of the tables used in the query, only the base entity type is required while the remaining tables are inferred.

9.3 Joining Tables

The main difference between S.E.A.L. queries and SQL queries are that S.E.A.L. queries do not contain any join conditions. The reason for this is that S.E.A.L. makes assumptions about the database design and thus only relies on the equi-join. SQL does not make such assumptions because it supports a wider range of database designs and supports various join types like theta join. Therefore the user is forced to provide the join conditions. These conditions can be placed within a FROM clause or within a WHERE clause. The additional burden to this requirement is that the user must understand how join conditions work, which condition to use in a situation and how to define a join condition. For the advanced user this may be a simple task but for novice users this extra effort usually leads to errors. Errors usually occur when a user needs to define a query that contains a conjunctive condition on a single attribute or queries against EAV structures. The SQL query needs to join tables and also join sets of data. As a result, SQL queries rely on the user's expertise and experience in defining SQL queries, it is this dependence that leads to errors in defining queries.

The S.E.A.L. syntax aims to remove some of the SQL requirements described and allows the user to simply define what data they want. The S.E.A.L. syntax that does not allow join conditions because the interpreter will infer which joins to use. By removing the need to specify join conditions we have removed any dependence on the user's knowledge and experience. In addition, S.E.A.L. queries that contain conjunctive conditions on a single attribute or EAV attributes do not need any set operators.

9.4 Attribute Conditions

Attribute conditions are used in queries to specify that the result set contains instances that satisfy a given condition(s). Both S.E.A.L. and SQL allow for such conditions but the way they are defined differs. In SQL an attribute condition is placed within the WHERE clause. The attribute conditions require an attribute name followed by an operator such as $>$, $<$, $>=$, $<=$, $!=$ or $=$ and then the value or variable. Multiple conditions can be defined by using the logical operators such AND, OR & NOT.

For S.E.A.L. queries the attribute conditions are placed within '[]' or within an ASSOCIATED_WITH() clause depending on the attribute(s) used in the condition. If the attribute belongs to the base entity type (either the table itself stores the attributes or its EAV structure stores the attribute) then we use the following syntax *base entity type [attribute operator value]* where the operator could be any of $>$, $<$, $>=$, $<=$, $!=$ or $=$. We can combine many conditions on base entity type attributes by using AND, OR, & NOT. We define this condition in the FROM clause after the table name. If the attribute that is used in the condition belongs to an entity type that is associated with the base entity type then the condition is placed within the ASSOCIATED_WITH() clause using the following syntax *ASSOCIATED_WITH(<attribute operator value>)*, again we can combine many conditions using AND, OR, & NOT.

9.5 Application

The set of queries that can be expressed in S.E.A.L. is a proper subset of the set of queries that can be expressed in SQL. This is because the extended implementation of the S.E.A.L. interpreter only supports a limited functionality of the SQL language. It is important to note that the S.E.A.L. language and the interpreter is not intended to replace SQL or to provide an alternative database query language but its intention is to simply help users in querying their database. However, the extended implementation only supports the basic querying functionality of selecting data with optional conditions. The interpreter can be further extended to help users with a wider class of SQL queries. For example, queries that contain aggregate functions would be an important extension because these functions are common in SQL queries as well as in data warehouses and data marts.

Chapter 10

Testing

10.1 Test Machine Configuration

The machine that was used to perform all of the functional and performance tests had the following configuration:

- Memory: 2.00 GB
- PostgreSQL: version 8.2.13
- Processor: Intel(R) Core(TM)2 Duo CPU T7500 @ 2.20GHz
- Operating System: Windows Vista Ultimate Service Pack 2
- System Type: 32-bit Operating System

10.2 Functional Test

The original implementation of the S.E.A.L. interpreter successfully passed a set of functional tests [8]. The set of tests involved the following steps [8]:

- defining a query against the robbers database in English
- defining a S.E.A.L. query
- defining a SQL query using an expert's knowledge
- running the S.E.A.L. interpreter using the defined S.E.A.L. query
- comparing the SQL query output of the interpreter with the SQL query defined by the expert
- running both SQL queries against the robbers database and comparing the results

Many of the previous functional test cases were re-run to ensure that any new extensions did not break any of the previous functionality. In addition to the pre-defined set of tests, new functional test cases were defined to test the extended interpreter. In particular the following cases were tested:

- queries that used comparison operators, other than equality, in the attribute constraints and combinations or constraints,

- queries that involved 1:M, 1:1, X->?<-Y and queries that contain a sequence of related tables,
- queries that contained attributes in the SELECT clause that did not belong to the base entity type but belong to any associating or associated entity type,
- queries that returned data stored in an EAV structure for a relationship type entity,
- queries that contained logical combinations of associations and nesting.

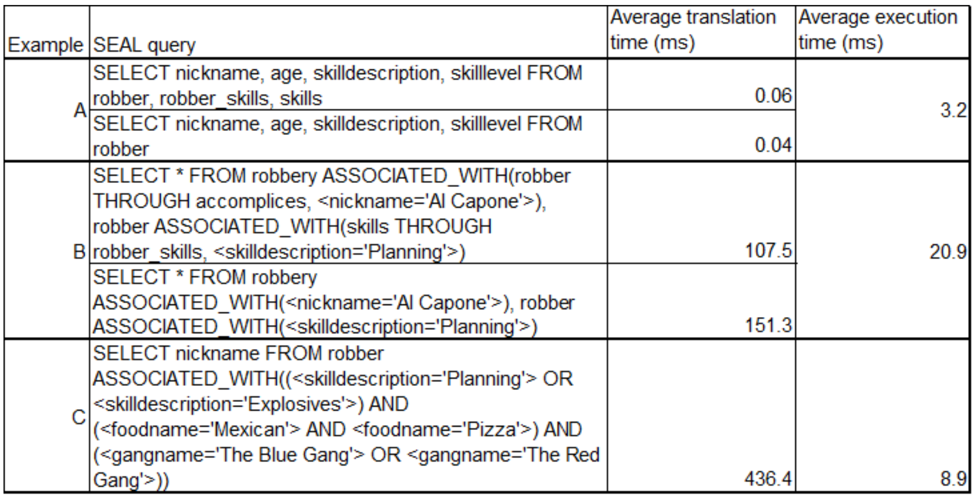
Many queries were defined to test each of the extensions in isolation and more complex queries were defined, that contained combinations of the extensions, to test the interpreter. The extended interpreter successfully passed all tests.

10.3 Performance Test

In order to test the performance of the extended S.E.A.L. interpreter, we chose a representative set of queries to use as the test cases. We created a test harness in the interpreter that would translate each S.E.A.L. query 10,000 times and output the average time it took. This translation time was then compared with the time it took for the resulting SQL query to execute in PostgreSQL.

The test queries ranged from simple queries to complex queries. Where possible, we tested a query two times; once with information omitted to force the interpreter to infer the missing information and once without inference.

Here is a representative set of the test queries and their results;



Example	SEAL query	Average translation time (ms)	Average execution time (ms)
A	SELECT nickname, age, skilldescription, skilllevel FROM robber, robber_skills, skills	0.06	3.2
	SELECT nickname, age, skilldescription, skilllevel FROM robber	0.04	
B	SELECT * FROM robbery ASSOCIATED_WITH(robber THROUGH accomplices, <nickname='Al Capone'>), robber ASSOCIATED_WITH(skills THROUGH robber_skills, <skilldescription='Planning'>)	107.5	20.9
	SELECT * FROM robbery ASSOCIATED_WITH(<nickname='Al Capone'>), robber ASSOCIATED_WITH(<skilldescription='Planning'>)	151.3	
C	SELECT nickname FROM robber ASSOCIATED_WITH((<skilldescription='Planning'> OR <skilldescription='Explosives'>) AND (<foodname='Mexican'> AND <foodname='Pizza'>) AND (<gangname='The Blue Gang'> OR <gangname='The Red Gang'>))	436.4	8.9

Figure 10.1: Performance test results for the extended S.E.A.L. interpreter

The test queries in example A are considered to be simple queries because we simply select attributes from tables. The average translation time for the first row in example A is slightly higher than the second row. A reason for this is because the interpreter uses each table from the FROM clause and checks whether the current table contains any of the attributes listed in the SELECT statement. Since there are more tables to check in the FROM clause for the first row then this step in the translation process takes slightly longer to complete.

The test queries in example B are more complex than the example A queries and as expected the average translation time was much greater. The main reason for such an increases in average translation time is that the queries contain multiple ASSOCIATED_WITH clauses. In order to translate a S.E.A.L. query containing an ASSOCIATED_WITH clause the interpreter must conduct many checks in the translation algorithm causing an increase in translation time.

The second query in example B has a greater average translation time than the first query because the interpreter is forced to infer the missing associating and associated entity types for each ASSOCIATED_WITH clause. Since the first query has given the associated entity types (robber, skills) and the associating entity types (accomplices, robber_skills) the interpreter does not need to do any inference so takes less time to translate the S.E.A.L. query. For the the second query the interpreter needs to infer the missing types which incurs a higher cost.

For the query in example C we expected that the average translation time was very high because the query contains an ASSOCIATED_WITH clause that contains many logical combinations of associations. The interpreter is forced to infer each associated and associating entity types for each constraint. In addition, we have logical operators (AND, OR) nested within each constraint which increases the translation time.

We can conclude that the average translation time of a S.E.A.L. query is dependent on the amount of inference required to translate the query, the number of ASSOCIATED_WITH clauses contained in the query and the number of associations within each ASSOCIATED_WITH clause.

The SQL query produced by the S.E.A.L. interpreter for the queries in example A had a small average execution time due to the simplicity of the SQL query. The query contained a simple SELECT, FROM, WHERE structure with the WHERE clause containing the join conditions. The average execution time for the SQL produced for example B was much higher than of example A because the query contained two nested SELECT statements each containing an INNER JOIN statement and an attribute constraint. An interesting result is the average execution time of the SQL query produced for example C. The SQL query was much more complex than the SQL query produced for example B yet the average execution time was much lower for example C. The SQL query for example C contained six nested SELECT statements each containing an INNER JOIN statement and an attribute constraint. A possible reason for this result is that one of the nested SELECT statements within the SQL query for example B was selecting many attributes from the largest table in the database. Therefore the SQL query for example B was handling a larger dataset than the SQL query for example C causing the average execution time of the SQL query for example B to be higher than example C.

10.4 Usability Test

As stated earlier, there was no evidence presented from previous work that S.E.A.L. was actually easier to use than SQL for a class of database queries. In order to verify our claim we conducted an usability test with database users.

The usability test involved two groups of computer science students classed as either novice database users or advanced database users. The novice database users were 300-level computer science students whom, at the time of testing, were half-way through an introductory database course and have recently completed a large project involving defining SQL queries. The advanced database users were 400-level computer science students whom, at the time of testing, were half-way through an advanced database course and had successfully completed a 300-level introductory database course or some equivalent database course. Each user group was tested individually due to the time in which the 300-level and 400-level courses were scheduled.

10.4.1 Test Methodology

Each of the user groups were:

- given a 25 minute presentation which described:
  - the project goals,
  - what S.E.A.L. is,
  - entity-association queries,
  - sparse attributes,
  - current problems with storing sparse attributes in a conventional relational database,
  - EAV structures and queries against EAV structures,
  - a comparison of SQL and S.E.A.L. queries,
  - examples of S.E.A.L. queries showing its syntax,
  - how to use the S.E.A.L. interpreter.
- given SQL and S.E.A.L. documentation, describing the language syntax and any required database schemas,
- given a set of practice questions to gain experience defining S.E.A.L. queries and using the interpreter,
- given a pre-test questionnaire,
- given a set of test questions in which they had to:
  - define a query in SQL,
  - define a query in S.E.A.L.,
  - record how long they took to define each query.
- given a post-test questionnaire.

The two questionnaires enabled us to gather various information about the students tested to see if these influenced their test performance. The pre-test questionnaire allowed us to gain background information of the student's database and SQL experience. Questions such as *Have you worked with EAV (Entity Attribute Value) structures before?* gave us insight into whether students would struggle defining queries against EAV structures because it is a new concept to them. The post-test questionnaire asked students to choose the query language they preferred to use and also gave students opportunity to present any feedback regarding the test.

10.4.2 Test Results

Each user group were further split into two groups (A & B) where;

- group A would define the SQL queries before the S.E.A.L. queries
- group B would define the S.E.A.L. queries before the SQL queries

We did this so that we can calculate and eliminate the time it took a student to understand the question so that we can measure the time it took a student to define a query in either SQL or S.E.A.L.

Group A Results

Novice Users

SQL results

Question	Correct	Incorrect	Incomplete	Average Time (mins)
1	0%	66%	33%	5
2	33%	33%	33%	4.1
3	33%	66%	0%	4.4
4	33%	0%	66%	9.5

SEAL results

Question	Correct	Incorrect	Incomplete	Average Time (mins)
1	66%	33%	0%	4.5
2	100%	0%	0%	3.2
3	100%	0%	0%	2.9
4	66%	33%	0%	5.3

Advanced Users

SQL results

Question	Correct	Incorrect	Incomplete	Average Time (mins)
1	33%	33%	33%	3.8
2	66%	33%	0%	3.0
3	66%	33%	0%	2.8
4	33%	66%	0%	4.5

SEAL results

Question	Correct	Incorrect	Incomplete	Average Time (mins)
1	66%	33%	0%	1.6
2	100%	0%	0%	2.4
3	100%	0%	0%	2.0
4	100%	0%	0%	2.2

Test students in group A generally performed better when defining the S.E.A.L. queries. Only a small percentage made mistakes when defining the S.E.A.L. queries but it should be noted that the mistakes made were very minor, such as forgetting to add a constraint or minor syntax errors. These errors would have been corrected if students had more time to review their work. The same cannot be said for the types of mistakes made when the students defined the SQL queries. The mistakes were usually caused by students not understanding which tables to join and which join conditions to use. The S.E.A.L. results are quite similar for the novice and advanced users however the advanced users performed slightly better when defining the SQL queries which was expected because the advanced users have more experience with SQL.

In addition to the higher accuracy when defining S.E.A.L. queries both user groups on average took less time to define the S.E.A.L. queries. We must note that since students in group A defined the SQL queries before S.E.A.L. queries some of the time was used in understanding the question and re-reading the question to define the query. Again the advanced users performed better than the novice users because on average they defined both the SQL and S.E.A.L. queries faster than the novice users, another expected result.

Group B Results

Novice Users

SEAL results

Question	Correct	Incorrect	Incomplete	Average Time (mins)
1	100%	0%	0%	2.6
2	100%	0%	0%	2.2
3	100%	0%	0%	1.8
4	100%	0%	0%	7.1

SQL results

Question	Correct	Incorrect	Incomplete	Average Time (mins)
1	50%	0%	50%	4.6
2	100%	0%	0%	3.6
3	0%	0%	100%	5
4	50%	50%	0%	8.8

Advanced Users

SEAL results

Question	Correct	Incorrect	Incomplete	Average Time (mins)
1	100%	0%	0%	1.0
2	100%	0%	0%	2.5
3	100%	0%	0%	1.2
4	100%	0%	0%	5.6

SQL results

Question	Correct	Incorrect	Incomplete	Average Time (mins)
1	33%	33%	33%	4.7
2	66%	33%	0%	4.4
3	66%	33%	0%	1.9
4	66%	33%	0%	6.1

Test students in group B were very accurate when defining the S.E.A.L. queries. The students did struggle with the SQL queries even though they had defined the S.E.A.L. queries first and had a good understanding of the question. Again the types of mistakes the students made when defining the SQL queries were not trivial, there were many cases in which students got confused when trying to join EAV tables and to extract attributes from these structures.

Students still spent on average more time defining the SQL queries than defining S.E.A.L. queries although the S.E.A.L. queries were defined first and their average times included understanding the question. There is not a major difference between the average times the novice and advanced users spent for defining the S.E.A.L. queries, the average times per question are quite close except for question 4 (a much more complex question).

Overall Results

It is clear to see that the test candidates in both groups (A and B) were more accurate when defining S.E.A.L. queries over SQL queries. There was no influence on accuracy when defining S.E.A.L. queries first (group B) or second (group A). Also there was no influence on accuracy when defining SQL queries first or second; students simply struggled when defining the SQL queries no matter the order. In addition, from the pre-test questionnaire no students in either of the groups tested had any previous EAV experience. Defining SQL queries for EAV structures was a difficult task for both user groups however they successfully defined the S.E.A.L. queries for EAV structures. These test results support our claim that S.E.A.L. is easier to use than SQL for a class of database queries.

Feedback

In addition to the above results, all students in both sets of user groups (novice and advanced) preferred to use S.E.A.L. over SQL. The collective reasoning for this was that students believed that S.E.A.L. is a more natural way of expressing queries. They enjoyed using S.E.A.L. more than SQL because they did not have to worry about any join conditions, the syntax was easy to understand and S.E.A.L. is intuitive.

Chapter 11

Conclusion

11.1 Contributions

This report describes entity association queries and the problems that SQL users have when defining entity association queries. We discuss the complexity involved with defining SQL queries for queries that contain conjunctive conditions on a single attribute. Accordingly we claimed that SQL is poorly suited for entity association queries that contain conjunctive conditions on a single attribute.

EAV structures are described and the need for these structures for effectively storing sparse attributes are examined. We mentioned that queries against EAV structures are a class of entity association queries that have a conjunctive condition on a single attribute. Therefore defining SQL queries against EAV structures incurs similar complexity as defining SQL queries for entity association queries with conjunctive conditions. Thus we also claimed that SQL is poorly suited for queries against EAV structures.

This report describes extensions to a simple query language (S.E.A.L.) whose aim is to reduce the complexity that arises from defining SQL queries for a class of database queries. We discussed the following extensions to S.E.A.L.;

1. support for various comparison operators in attribute constraints,
2. support for relationship cardinalities other than M:N,
3. allowing the attribute list after the SELECT clause to contain attributes other than base entity type attributes,
4. support for relationship types implemented as EAV structures, and
5. support for queries containing logical combinations of associations

With these extensions we have enabled S.E.A.L. to support a wider range of database queries.

An analysis was conducted on S.E.A.L. and SQL to compare and contrast the two languages in terms of syntax and functionality of the languages. With SQL a user is forced to provide all of the information necessary in a query to get the correct results because SQL does not make any assumptions about the database design. With S.E.A.L. a user is not forced to provide all information, a user can omit information and rely on the inference mechanism of S.E.A.L. to infer the omitted information. We can allow users to omit information because S.E.A.L. makes assumptions about the database design.

We conducted functional, performance and usability tests on S.E.A.L. The functional and performance tests assessed the language and the interpreter to ensure that the extensions made were valid and acceptable. The usability test provided positive evidence for our underlying claim that S.E.A.L. is easier to use than SQL. Students involved in the usability test found S.E.A.L. easy to use and understand. This was evident in the test results because the novice users achieved 90% accuracy when defining the S.E.A.L. queries and the advanced users achieved 96% accuracy. On the other hand, many of the students struggled to define the equivalent SQL queries because the novice users could only achieve 35% accuracy and the advanced users achieved 54% accuracy.

11.2 Future Work

The work described in this report has enabled S.E.A.L. to support a wider range of database queries thus become a more useful tool. However there are still areas in which S.E.A.L. can be expanded. For instance, one of the database design rules for S.E.A.L. states that no two attributes can have the same name unless they are involved in a primary key - foreign key pair [7]. This rule can be relaxed by extending the interpreter to find all tables that contain the duplicate attribute and then question the user as to which table to use in the translation algorithm.

The breadth-first traversal search implemented for the sequence of related tables association does not output the optimal solution. Currently, only a single path is selected by the search even if multiple paths exist in the database. The selected path may not be the most efficient or the path that the user wanted. A possible solution is to ask the user to choose the path of tables they would like to use or to output all possible solutions.

The current implementation of the S.E.A.L. interpreter does not provide informative error messages to the users. For example, in the case of an ambiguous error no meaningful information is outputted to tell the user how or why the error has occurred. Extending the interpreter to provide informative error messages will make S.E.A.L. more useful for the user. A further extension is to output a possible solution for the problem that occurred.

Appendix

Full ANTLR Grammar

```
grammar AssociatedWith;
selectStatement returns [SelectStmt s]
      :     selectKeyword fd=fieldList? fromKeyword fc=fromClause? {s =
new SelectStmt(fd,fc); };

selectKeyword
      :     'SELECT';

fromKeyword
      :     'FROM';

associatedWithKeyword
      :     'ASSOCIATED_WITH';

equalsKeyword
    :   '=';

lessThanOrEqualToKeyword
    :   '<=';

greaterThanOrEqualToKeyword
    :   '>=';

notEqualToKeyword
    :   '!=';

fromClause returns [FromList t]
      :     {t=new FromList();} tr=tableReference {t.addFromTable(tr);}
(FieldDelimiter tr=tableReference {t.addFromTable(tr);})*;

fieldList returns [ArrayList<String> a]
      :     '*' {a=new ArrayList<String>(); a.add("*");}
      |     {a=new ArrayList<String>();} fn=fieldName {a.add($fn.text);}
(FieldDelimiter  fn=fieldName {a.add($fn.text);})*;

fieldName
      :     (String)+;

tableReference returns [TableReference t]
      :     t1=tableRestriction {t=t1; String entityTableName = null;
String joinTableName=null; String entityRole=null;}(associatedWithKeyword
LP (et_name=tableName ('AS' entityRole {entityRole = $entityRole.text;
})? {entityTableName=$et_name.text;} ('THROUGH' jt_name=tableName
{joinTableName = $jt_name.text;})? FieldDelimiter)? exp=expression[t1]
RP {t.ASSOCIATED_WITH(entityTableName,entityRole, joinTableName,
exp);})?;


tableRestriction returns [TableReference t]
      :     table {t=$table.t;}
      |     table '[' res=tableAttributeExpression[$table.t] ']'
{t=$table.t; t.addAttributeRestriction(res);};

table returns [TableReference t]
```

```
      :     t1=tableName {t=new
TableReference(Main.metadata.get($t1.text));};

tableName
      :     (String)+;

tableAttributeExpression[TableReference t] returns [AttributeExpression
e]
      :     a=tableAttributeOrExpression[t] {e=a; };

tableAttributeOrExpression[TableReference t] returns [AttributeExpression
e]
      :     l=tableAttributeAndExpression[t] {e=l;} ('OR'
r=tableAttributeAndExpression[t] {e= new AttributeOrExpression(e,r); })*;

tableAttributeAndExpression[TableReference t] returns
[AttributeExpression e]
      :     l=tableAttributeNotExpression[t] {e=l;} ('AND'
r=tableAttributeNotExpression[t] {e= new AttributeAndExpression(e,r);
})*;

tableAttributeNotExpression[TableReference t] returns
[AttributeExpression e]
      :     'NOT' a=tableAttributeAtom[t] {e=new
AttributeNotExpression(a);}
      |     a=tableAttributeAtom[t] {e=a;};

tableAttributeAtom[TableReference t] returns [AttributeExpression e]
      :     c=condition {e = new AttributePredicate(c, t);}
      |     LP a=tableAttributeExpression[t] RP {e=a;};

expression[TableReference t] returns [Exp e]
      :     a=orexpression[t] {e=a; };

orexpression[TableReference t] returns [Exp e]
    : l=andexpression[t] {e=l;} ('OR' r=andexpression[t] {e=new
ORExp(e,r,t); })*;

andexpression[TableReference t] returns [Exp e]
    :   l=notexpression[t] {e=l;} ('AND' r=notexpression[t] {e=new
ANDExp(e,r,t); })*;

notexpression[TableReference t] returns [Exp e]
    : 'NOT' a=atom[t] {e = new NOTExp(a,t);}
    | a = atom[t] {e=a;};

atom[TableReference t] returns [Exp e]
      :     {String entityTableName = null; String joinTableName=null;
String entityRole=null;} pri_exp=simplePredicate[t]
(associatedWithKeyword LP (et_name=tableName ('AS' entityRole {entityRole
= $entityRole.text; })? {entityTableName = $et_name.text;} ('THROUGH'
jt_name=tableName {joinTableName = $jt_name.text;})? FieldDelimiter)?
sec_exp=expression[t] RP
{pri_exp.ASSOCIATED_WITH(entityTableName,entityRole, joinTableName,
sec_exp);})? {e=pri_exp;}
      |     {String entityTableName = null; String joinTableName=null;
String entityRole=null;} LP pri_exp=expression[t] RP
(associatedWithKeyword LP (et_name=tableName ('AS' entityRole {entityRole
= $entityRole.text; })? {entityTableName = $et_name.text;} ('THROUGH'
jt_name=tableName {joinTableName = $jt_name.text;})? FieldDelimiter)?
sec_exp=expression[t] RP
{pri_exp.ASSOCIATED_WITH(entityTableName,entityRole, joinTableName,
sec_exp);})? {e=pri_exp;};

condition returns [Condition returned]
    :    {String tName=null; } ((tn = tableName '.' {tName = $tn.text;
})? fieldName operator d=data) {returned = new Condition(tName,
$fieldName.text, $operator.text, d);};

simplePredicate[TableReference t] returns [Exp e]
      : LT c=condition {e = new PredicateExp(t);
((PredicateExp)e).addCondition(c);} (',' c=condition
{((PredicateExp)e).addCondition(c);})* GT;

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

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

entityRole
      :      String+;

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

numericalData
    :   Number+
    |      Number* DP Number+;

data returns [String s]
      : SQ stringData? SQ {s= new String("'"+$stringData.text+"'"); }
      | numericalData {s = new String($numericalData.text); };

operator
    : (equalsKeyword | lessThanOrEqualToKeyword |
greaterThanOrEqualToKeyword | notEqualToKeyword | LT | GT);

FieldDelimiter
      :     ',';

LP    :     '(';
RP    :     ')';
LSB   :     '[';
RSB   :     ']';
LT    :   '<';
GT    :   '>';
```

```
SQ    :   '\'';
DP    :   '.';

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

SQL Syntax

SELECT [ALL | DISTINCT [ON (*expression* [, ...])]] * | *expression* [AS *output_name*] [, ...]

[**FROM** *from_item* [, ...]]

[WHERE *condition*]

[UNION | INTERSECT | EXCEPT [ALL] *select*]

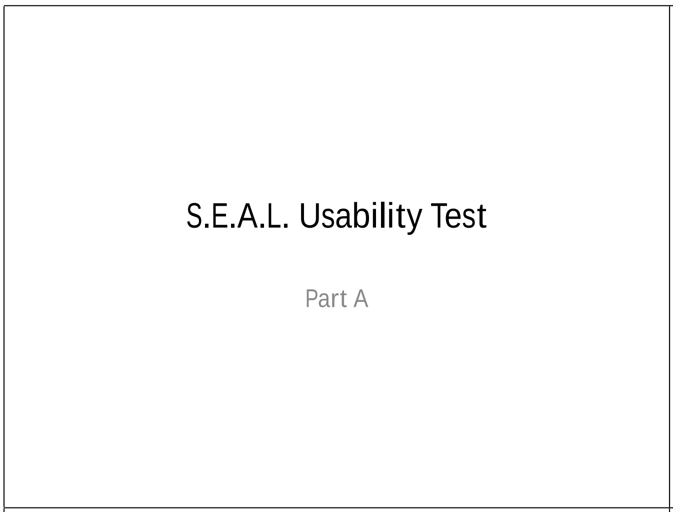
where *from_item* can be one of:

[ONLY] *table_name* [*] [[AS] *alias* [(*column_alias* [, ...])]]

(*select*) [AS] *alias* [(*column_alias* [, ...])]

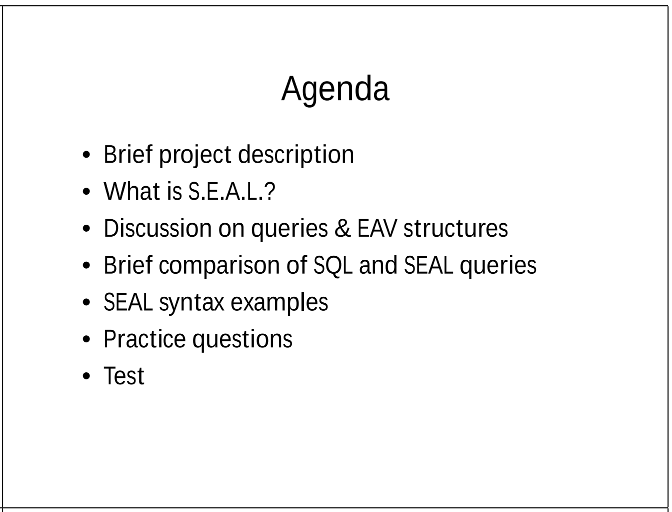
from_item [NATURAL] *join_type* *from_item* [ON *join_condition* | USING (*join_column* [, ...])]

Usability Test Documents



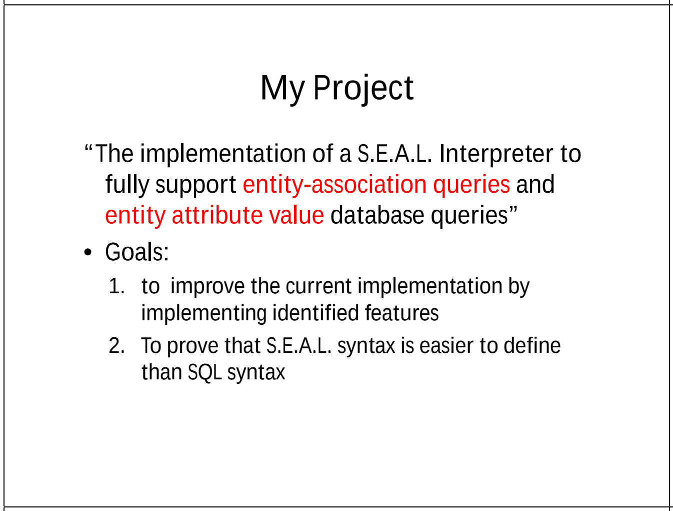
S.E.A.L. Usability Test

Part A



Agenda

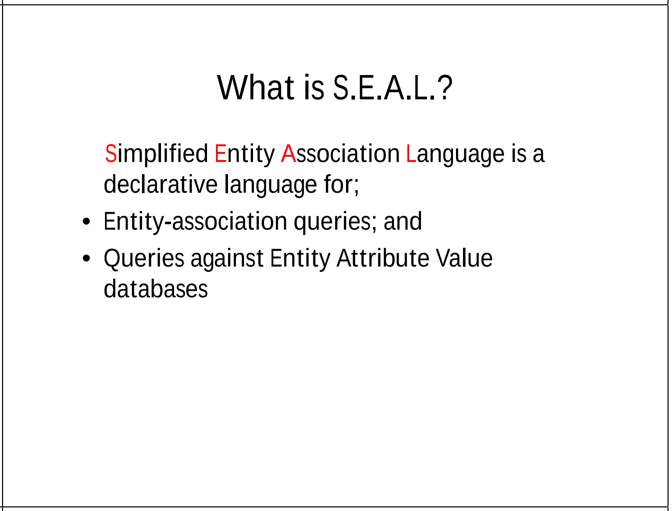
- Brief project description
- What is S.E.A.L.?
- Discussion on queries & EAV structures
- Brief comparison of SQL and SEAL queries
- SEAL syntax examples
- Practice questions
- Test



My Project

"The implementation of a S.E.A.L. Interpreter to fully support entity-association queries and entity attribute value database queries"

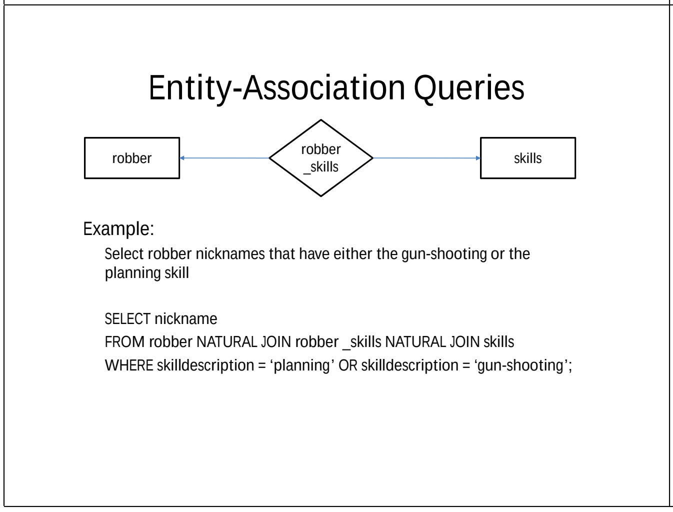
- Goals:
  1. to improve the current implementation by implementing identified features
  2. To prove that S.E.A.L. syntax is easier to define than SQL syntax



What is S.E.A.L.?

Simplified Entity Association Language is a declarative language for;

- Entity-association queries; and
- Queries against Entity Attribute Value databases



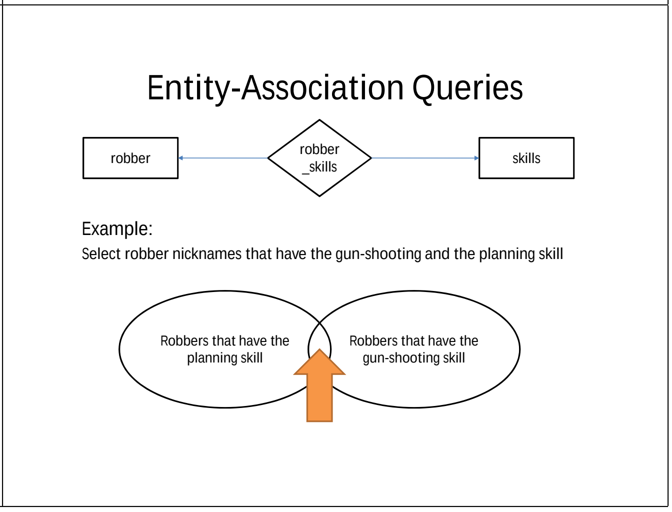
Entity-Association Queries

robber
robber _skills
skills

Example:

Select robber nicknames that have either the gun-shooting or the planning skill

```
SELECT nickname
FROM robber NATURAL JOIN robber _skills NATURAL JOIN skills
WHERE skilldescription = 'planning' OR skilldescription = 'gun-shooting';
```



Entity-Association Queries

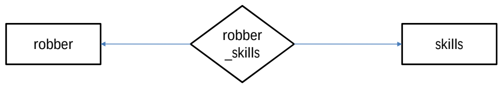
robber
robber _skills
skills

Example:

Select robber nicknames that have the gun-shooting and the planning skill

Robbers that have the planning skill
Robbers that have the gun-shooting skill

Entity-Association Queries



Example:
Select robber nicknames that have the gun-shooting and the planning skill

```
SELECT nickname
FROM robber NATURAL JOIN robber_skills NATURAL JOIN skills
WHERE skilldescription = ‘planning’
INTERSECT
SELECT nickname
FROM robber NATURAL JOIN robber_skills NATURAL JOIN skills
WHERE skilldescription = ‘gun-shooting’;
```

Sparse Attributes

Attributes:
Robber has an ID, Name, Age, only 1 type of haircut and may like many types of music

1. a robber can only have 1 type of haircut (single-valued)
2. a robber may like many types of music (multi-valued)

Sparse attributes:

1. Are either single-valued or multi-valued attributes; and
2. Is sparse because we only know its value for a few entities out of many; regardless whether we know all of the possible attribute values or just a few

Relational Database Structure

robber

ID	Name	Age	Haircut	Music1	Music2
1	Al Capone	33	Mullet	Rock	Pop
2	Bugsy	21	Spiked	R’n’b	null

How can we store a robber that likes 3 music types?

ID	Name	Age	Haircut	Music1	Music2	Music3
1	Al Capone	33	Mullet	Rock	Pop	null
2	Bugsy	21	Spiked	R’n’b	null	null
3	Sneaky	50	Mohawk	Blues	Jazz	Rock

Entity Attribute Value Structures

robber

ID	Name	Age
1	Al Capone	33
2	Bugsy	21

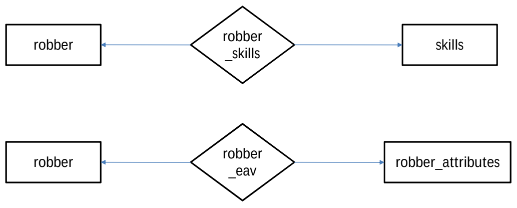
robber_attributes

AttributeID	Attribute
1	Haircut
2	Music

robber_eav

RobberID	AttributeID	Value
1	1	Mullet
1	2	Rock
1	2	Pop
2	1	Spiked
2	2	R’n’b
3	1	Mohawk
3	2	Blues
3	2	Jazz
3	2	Rock

Entity Attribute Value Queries



Comparing SQL and SEAL Queries

Entity-Association Query

Select robber nicknames that have both ‘Gun-shooting’ and ‘Planning’ skills

```
SELECT nickname
FROM robber NATURAL JOIN robber_skills NATURAL JOIN skills
WHERE skilldescription = ‘Planning’
INTERSECT
SELECT nickname
FROM robber NATURAL JOIN robber_skills NATURAL JOIN skills
WHERE skilldescription = ‘Gun-Shooting’;
```

```
SELECT nickname FROM robber ASSOCIATED_WITH (<skilldescription =
    ‘Gun-Shooting’> AND <skilldescription = ‘Planning’>)
```

Comparing SQL and SEAL Queries

EAV Query

Select robber nicknames that have a ‘Mullet’ haircut and ‘Pop’ music

```
SELECT nickname
FROM robber NATURAL JOIN robber_eav NATURAL JOIN
   robber_attributes
WHERE attribute='haircut' and value='Mullet'
INTERSECT
SELECT nickname
FROM robber NATURAL JOIN robber_eav NATURAL JOIN
   robber_attributes
WHERE attribute=’music’ and value='Pop'
```

```
SELECT nickname FROM robber [Haircut = ‘Mullet’ AND Music = ‘Pop’]
```

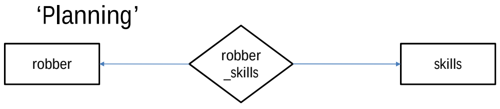
Example 1: base entity attributes

Retrieve all nicknames of robbers that have the value ‘Mullet’ for the ‘haircut’ attribute and the value ‘Pop’ for the ‘music’ attribute.

```
SELECT nickname FROM robber
   [haircut=’Mullet’ AND music=’Pop’]
```

Example 2: associating entities

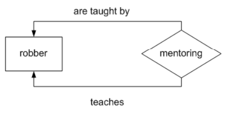
Retrieve all nicknames of robbers that have the skills with the description ‘Gun-Shooting’ and ‘Planning’



```
SELECT nickname FROM robber
  ASSOCIATED_WITH (
    <skilldescription = ‘Gun-Shooting’> AND
    <skilldescription = ‘Planning’>)
```

Example 3: recursive relationship

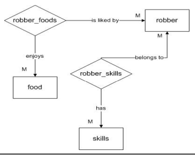
Retrieve all nicknames of robbers that have been mentored by the robber with the ID of 1



```
SELECT nickname FROM robber
   ASSOCIATED_WITH (robber AS teacher,
   <robberid=1>)
```

Example 4: logical associations

Retrieve all nicknames of robbers that have the skill with the description ‘Planning’ or ‘Explosives’ and like ‘mexican’ food



```
SELECT nickname FROM robber ASSOCIATED_WITH
   ((<skilldescription = ‘Planning’>  OR
   <skilldescription=‘Explosives’>) AND
   <foodname=‘mexican’>)
```

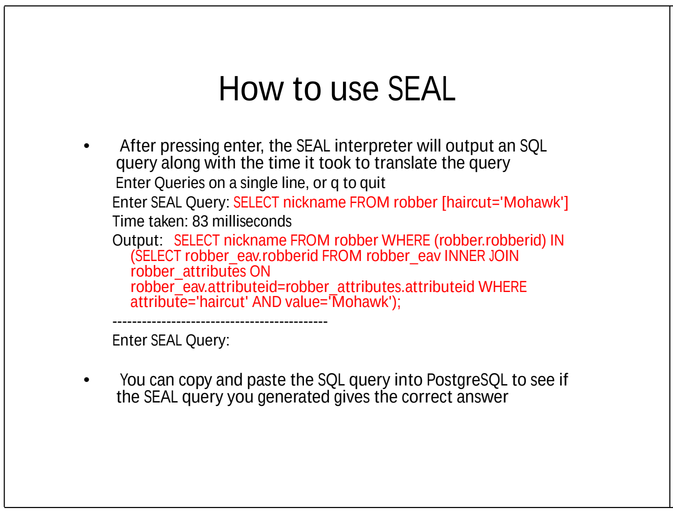
How to use SEAL

- You will need to use the command line to run SEAL
- Once SEAL has started you will see the following;

```
Enter Queries on a single line, or q to quit
Enter SEAL Query:
```

- You will enter your SEAL query on a single line

```
Enter Queries on a single line, or q to quit
Enter SEAL Query: SELECT nickname FROM robber
   [haircut='Mohawk']
```

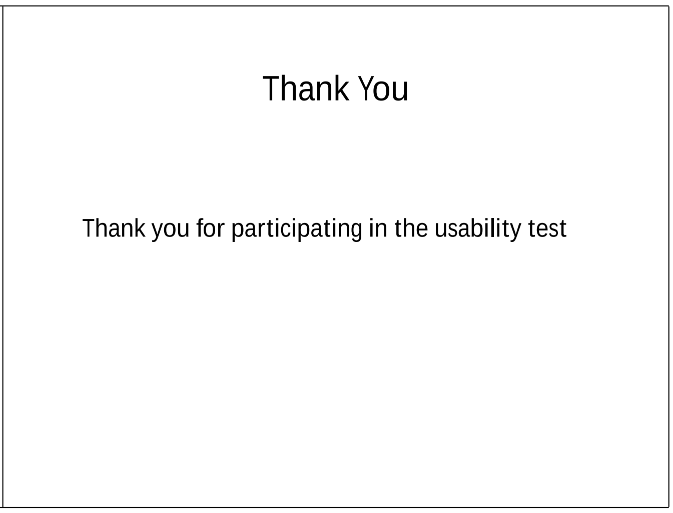


How to use SEAL

- After pressing enter, the SEAL interpreter will output an SQL query along with the time it took to translate the query

```
Enter Queries on a single line, or q to quit
Enter SEAL Query: SELECT nickname FROM robber [haircut='Mohawk']
Time taken: 83 milliseconds
Output:   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');
---------------------------------------------
Enter SEAL Query:
```

- You can copy and paste the SQL query into PostgreSQL to see if the SEAL query you generated gives the correct answer

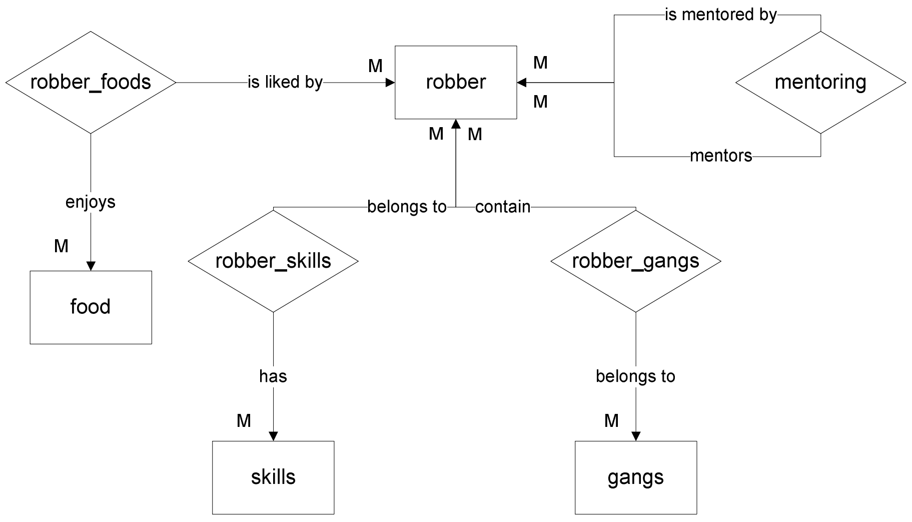


Thank You

Thank you for participating in the usability test

SEAL documentation

Conceptual Schema:



Implementation Schema:

Relations:
foods (<u>foodid</u>, foodname)
gangs (<u>gangid</u>, gangname)
mentoring (<u>robberid1</u>, <u>robberid2</u>)
robber (<u>robberid</u>, nickname, age, prisonyears, first_name, last_name)
robber_skills (<u>robberid</u>, <u>skillid</u>, skillpreference, skilllevel)
robber_foods (<u>robberid</u>, <u>foodid</u>)
robber_gangs (<u>robberid</u>, <u>gangid</u>)
skills (<u>skillid</u>, skilldescription)

Referential Integrity Constraints:
mentoring [robberid1] references robber [robberid1]
mentoring [robberid2] references robber [robberid2]
plans [bankname, city] references bank [bankname, city]
robber_foods [robberid] references robber [robberid]
robber_foods [foodid] references foods [foodid]
robber_gangs [robberid] references robber [robberid]
robber_gangs [foodid] references gangs [gangid]
robber_skill [robberid] references robber [robberid]
robber_skill [robberid] references skills [skillid]

SEAL Syntax:

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

An *associationClause_expr* has the following form:

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

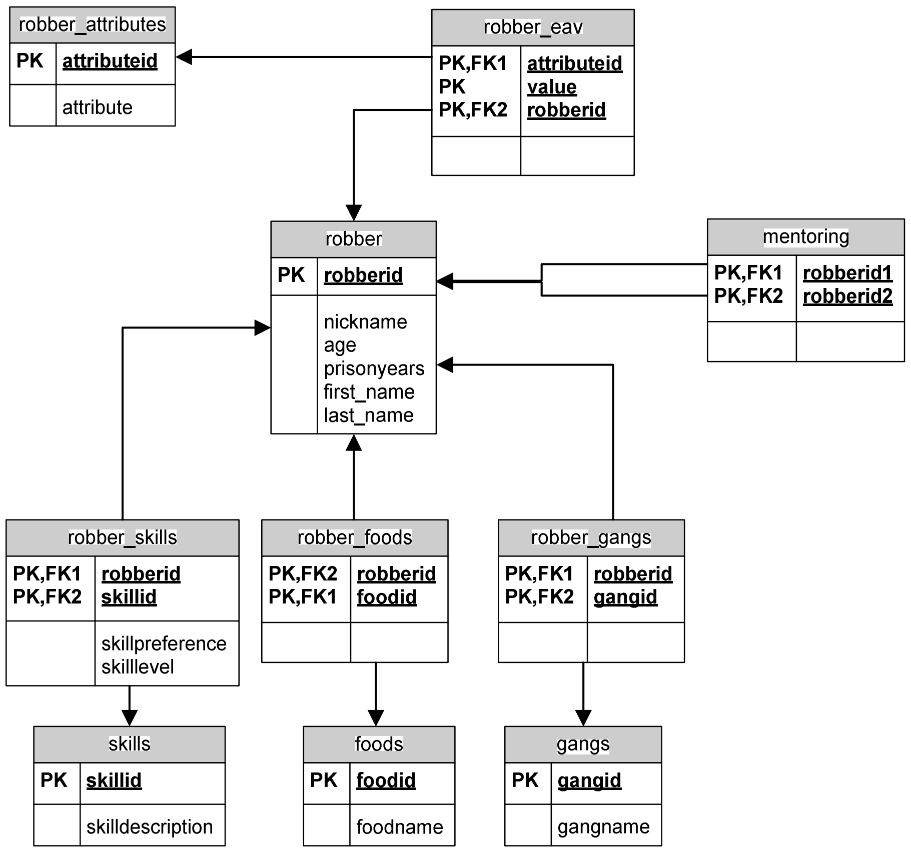
The *baseEntityRestriction_expr* has the following form:
[NOT] attribute [> | < | >= | <= | != | =] '*value*' [AND | OR]

The *specification_expr* has the following form:
< [NOT] attribute [> | < | >= | <= | != | =] '*value*'> [AND | OR]

- attribute: this is a list of entity type attributes
- baseEntityType: this is a list of entity types
- baseEntityRestriction_expr: this is a restriction(s) placed on some or all of the base entity type attributes using Boolean expressions
- baseEntityRole: this is the role that the base entity type must play in the relationship
- associatedEntityType: the entity type that is associated (participating in a relationship) with the base entity type
- associatedEntityRole: the role that the associated entity plays in the relationship
- relationship: the name of the relationship entity type between the base entity and the associated entity
- specification_expr: this is a restriction(s) placed on some or all of the attributes belonging to the associated entity and/or relationship entity

SQL documentation

Relational Database Schema:



The robber_attributes table contains the following data:

attributeid	attribute
1	haircut
2	music
3	callsign
4	origin

SQL Syntax:

SELECT [ALL | DISTINCT [ON (*expression* [, ...])]] * | *expression* [AS *output_name*] [, ...]
   [FROM *from_item* [, ...]]
   [WHERE *condition*]
   [{ UNION | INTERSECT | EXCEPT } [ALL] *select*]

where *from_item* can be one of:

   [ONLY] *table_name* [*] [[AS] *alias* [(*column_alias* [, ...])]]
   (*select*) [AS] *alias* [(*column_alias* [, ...])]
   from_item [NATURAL] *join_type* *from_item* [ON *join_condition* | USING (*join_column* [,
...])]

Practice Questions:

How to use SEAL

1. You will need to use the command line to run SEAL
2. Once SEAL has started you will see the following;

```
Enter Queries on a single line, or q to quit
Enter SEAL Query:
```

3. You will enter your SEAL query on a single line

```
Enter Queries on a single line, or q to quit
Enter SEAL Query: SELECT nickname FROM robber [haircut='Mohawk']
```

4. After pressing enter, the SEAL interpreter will output an SQL query along with the time it took to translate the query

```
Enter Queries on a single line, or q to quit
Enter SEAL Query: SELECT nickname FROM robber [haircut='Mohawk']
Time taken: 83 milliseconds
Output:   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');
--------------------------------------------
   Enter SEAL Query:
```

5. You can copy and paste the SQL query into PostgreSQL to see if the SEAL query you generated gives the correct answer

Question 1:
Retrieve all of the nicknames of robbers that have a value of ‘China’ for the ‘origin’ attribute or a value of ‘Latin’ for the ‘music’ attribute

Expected Output:

```
  nickname
--------------
 Al Capone
 Bugsy Malone
 Anastazia
 Dutch Schulz
(4 rows)
```

Question 2:
Retrieve all of the nicknames of robbers that have the skill with the description ‘Scouting’ and ‘Planning’

Expected Output:

```
nickname
----------
 Clyde
(1 row)
```

Question 3:
Retrieve all of the nicknames of robbers that have been taught by the robber with ‘robberid’ of 3

Expected Output:

```
   nickname
--------------
 Al Capone
 Bugsy Malone
(2 rows)
```

Question 4:
Retrieve all of the nicknames of robbers that have the skill with the description ‘Planning’ or ‘Explosives’ and like ‘mexican’ and ‘pizza’ (use ‘foodname’ as the attribute)

Expected Output:

```
   nickname
--------------
 Al Capone
 King Solomon
(2 rows)
```

Pre-test Questionnaire

Code Number: ______________

Please circle 1:

1. Have you successfully completed COMP 302 or an equivalent database course?

   Yes No

2. Are you familiar with the robbers database?

   Yes No

3. Have you completed any other course using SQL or one of its derivatives?

   Yes No

   If yes, please specify:

   ____
   ____
   ____

4. Have you or are you currently working in a job that requires SQL knowledge?

   Yes No

5. Have you ever worked with EAV (Entity Attribute Value) databases?

   Yes No

6. How would you rate your SQL knowledge?

   Minimal Limited Average Extensive

Test

Instructions:

- There are 4 questions that you need to answer
- You will need to complete each question twice; once in SQL and once in SEAL
- Each question has a time limit and you have this amount of time to answer the question in SQL and the same amount of time to answer the question in SEAL. For example; question 1 has a time limit of 5 minutes so you have 5 minutes to answer the query in SQL and then another 5 minutes to answer the question in SEAL.
- Please start the timer after you have read the question
- You will complete your SQL query using PostgreSQL
- You will complete your SEAL query using the SEAL interpreter. To check that your SEAL query is correct you can copy and paste the SQL query generated by SEAL into PostgreSQL
- You will need to time yourself when answering each question using the Timer.htm file. Each question has the time limit built in. Start the timer then answer the question, when you think you have finished stop the timer and record the time in your text file.
- Once you are happy with your query (SQL or SEAL) then paste your query into a text file along with the time it took you to create the query. For example, you could format the text file like:
  Question 1:

  SQL query: SELECT * FROM robber;

  Time Taken: 1 minute

  SEAL query: SELECT * FROM robber

  Time Taken: 1 minute
- Please paste your SEAL query in the text file and not the generated SQL query
- If you cannot complete a question because you run out of time then please write in the text file that you did not have enough time. Please also include whatever answer you have got so far, even incomplete queries will do

Question 1:
Time limit: 5 minutes
Retrieve all of the nicknames of robbers that have a value of 'Skull' for the 'callsign' attribute and have either a value of 'Latin' or a value of 'Heavy Metal' for the 'music' attribute

Expected Output:

```
   nickname
--------------
 Bugsy Malone
 Anastazia
(2 rows)
```

Question 2:
Time limit: 5 minutes
Retrieve all of the nicknames of robbers that have the skill with the description 'Preaching' and 'Safe-Cracking' and 'Planning'

Expected Output:

```
nickname
-----------
 Al Capone
(1 row)
```

Question 3:
Time limit: 5 minutes
Retrieve all of the nicknames of robbers that have been taught by the robber with 'nickname' 'Mimmy The Mau Mau'

Expected Output:

```
nickname
-----------
 Al Capone
(1 row)
```

Question 4:
Time limit: 10 minutes
Retrieve all of the nicknames of robbers that have the skill with the description 'Planning' or 'Explosives' and like 'mexican' and 'pizza' (use 'foodname' as the attribute) and belong to either 'The Boyd Gang' or 'Bannot' (use 'gangname' as the attribute)

Expected Output:

```
   nickname
--------------
 Al Capone
 King Solomon
(2 rows)
```

****

Usability Test Timer

Question 1

Start Timer | Stop Timer | Reset Timer

Time remaining:

Your time is ______ minutes

Question 2

Start Timer | Stop Timer | Reset Timer

Time remaining:

Your time is ______ minutes

Question 3

Start Timer | Stop Timer | Reset Timer

Time remaining:

Your time is ______ minutes

Question 4

Start Timer | Stop Timer | Reset Timer

Time remaining:

Your time is ______ minutes

Post-test Questionnaire

Code Number: ______________

Please circle 1:

1. Did you attempt each question in the test?

   Yes No

2. Which query language did you prefer?

   SQL SEAL

   Why?

3. Did you find SEAL easy to understand?

   Yes No

   Please explain:

4. Did you find SEAL easy to use?

   Yes No

   Please explain:

5. Please provide any other feedback on the test, SEAL or anything else

Bibliography

1. ANTLR: Another tool for Language Recognition
   http://www.antlr.org/. Accessed 27 April 2009.

2. Applicability of Universal Relation to Data Integration
   http://pages.cs.wisc.edu/ vshree/cs764/Windik.pdf. Accessed 16 April 2009

3. Corwan, J., Silberschatz, A., Miller, P.L. and Marenco, L. (2007): Dynamic tables: An architecture for managing evolving, heterogeneous biomedical data in relational database management systems. *International Journal of the American Medical Informatics Association* **14**(1):86-93.

4. Dinu, V. And Nadkarni, P. (2006): Guidelines for the effective use of entity-attribute-value modelling for biomedical databases. *International Journal of Medical Informatics* **76**(11-12):769-779.

5. Prakash, M.N. And Cynthia, B. (1998): Data extraction and ad hoc query of an entity-attribute-value database. *International Journal of the American Medical Informatics Association* **5**(6):511-527.

6. QBE: Query-By-Example
   http://pages.cs.wisc.edu/ dbbook/openAccess/thirdEdition/qbe.pdf. Accessed 15 April 2009.

7. Stanley, E.: S.E.A.L: Simple entity-association query language project (2007).

8. Stanley, E., Mogin, P. And Andreae, P.: S.E.A.L- A query language for entity-association queries.

9. What is SQL?
   http://cplus.about.com/od/thebusinessofsoftware/a/whatissql.htm. Accessed 12 April 2009.