WEB DATABASE (WDB): A JAVA SEMANTIC DATABASE

by

Bo Li

A thesis submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science In Computer Sciences: Turing Scholars Honors

University of Texas at Austin

Spring, 2006

UNIVERSITY OF TEXAS AT AUSTIN

ABSTRACT

Building a Semantic Database in Java

By Bo Li

With the widespread use of relational database systems such as MySQL and Oracle, the flaws of such systems become more apparent. While the relational model presents the database designer with a great degree of flexibility, it captures little meaning of the stored data and offers limited data integrity capabilities. This paper describes a database based on the semantic data model to capture the meaning of data so that the schema better represents its corresponding real world objects. A subset of the data definition and manipulation language is also explained. Furthermore, a detailed examination of implementing such a database management system in Java with the SleepyCat database engine is presented along with possible ways of utilizing dynamically compiling Java objects for data storage. By using the semantic data model, it is demonstrated how querying for entries in a hierarchic structure or with entity relationships is much simpler when compared to the equivalent SQL query.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

The author wishes to express sincere appreciation to Professor Phil Cannata for all his help and guidance on the project. In addition, special thanks goes to Dr. Don Batory and Dr. Greg Lavender for serving on the project committee and providing valuable input. Last but not least, thanks to Doug Tolbert of Unisys for developing the SDM and the SIM framework.

INTRODUCTION

Database Management Systems (DBMS) have gained tremendous popularity with the increase in demand for rich, data-driven web applications. With almost all websites using some type of database backend, database systems are no longer confined to storing and organizing business critical data for large corporations. Today, database systems are used to power anything from personal websites such as myspace.com to mobile applications found in cell phones. Although there are many different database systems with different data models, the most commonly used database system utilizes the relational data model. While the relational model offers many advantages such as flexibility and scalability, it does not capture the meaning of the data and has limited data integrity constraints. With the current relational model, it is very difficult for users to manipulate the data without prior knowledge of the semantics and the model constructs utilized to design the schema. Furthermore, with the widespread use of complex object oriented programming languages such as Java and C# to develop modern data driven applications, it is very difficult for developers to translate their application objects to a relational schema for storage on disk. This results in the developer creating objects with complex relationships and rich constraints that accurately represent their real world objects, but with no way to store and run queries against them in relational database systems.

The semantic data model (SDM) is designed to overcome some of the shortcomings of the traditional relational models. Developed by Hammer and McLeod, SDM shares some concepts of object-oriented programming such as attributes, classes, hierarchies, and inheritance. It also incorporates structural semantic concepts such as bidirectional relationships, multi-valued attributes, and integrity constraints. Unisys developed the initial implementation of a database based on SDM in the 80s for A series machines. Their product, called the Semantic Information Manager (SIM) was never released for x86 PCs due to business reasons. This project aims to implement the fundamental features of SIM

entirely in Java for platform interoperability. Web Database (WDB) utilizes a subset of the SIM data definition and manipulation languages for schema definitions and queries. However, the implementation of WDB is completely unrelated to that of SIM. Although the current version is nothing more then a prototype used for testing and research, we hope it will contribute to furthering the knowledge of implementing semantic database systems in modern programming languages.

Chapter 1

BASIC CONCEPTS

The Web Database (WDB) is an implementation of the Semantic Data Model (SDM) developed by Hammer and McLeod. It is aimed to overcome the limitation of hierarchical, network, and relational models by including more information about the meaning of the data and what they represent in the real world. Semantic modeling allows for more complex data relationships while maintaining data integrity as well as the flexibility to model the schema in both a hierarchical or relational fashion.

1.1 Concepts

Entity

Entities in the SDM represent the real objects being modeled such as people, places, or things. These are analogous to tuples in a table of a relational data model or objects in object-oriented programming.

Attribute

Attributes are the characteristics of an SDM entity. In an entity that represents people for example, an attribute would be name, height, weight, etc. The collection of attributes defines an entity. WDB offers two types of attributes: data-valued attributes (DVAs) and entity-valued attributes (EVAs). A DVA defines a displayable value of an entity that is characteristic of that entity. For example, an entity that represents person would have DVAs such as name, age, etc. WDB uses three data types for DVAs: Boolean, Integer, and String. An EVA establishes a bidirectional relationship between the entity in its owning class and the entities of a target class. The target class can also be the owning class itself. The relationship implied by EVAs also requires WDB to maintain referential integrity of the link between the entities. WDB guarantees that an EVA cannot be assigned an invalid reference

when it's inserted or modified. In addition, upon the deletion of an entity, all EVAs referencing that entity are automatically updated as long as the deletion does not violate other integrity constraints. EVAs can either be defined as single or multi-valued. Single valued EVAs can only establish a one-to-one or one-to-many relationship depending on the inverse EVA in the target class. On the other had, multi-valued EVAs can "point" to many entities and establish many-to-many or many-to-one relationships.

Class

Classes represent a collection of similar entities or entity types. For example, entities that represent "France", "New York", and "San Francisco" are all members of the City class. Each class must have a unique name and all entities of that class share the same attributes.

Subclass and Superclass

Each base class can also have a subclass that inherits all the attributes of that base class. A subclass extends a class with attributes that are only specific to the subclass. Subclasses are often used to define a subtype of an object in the real world. For example, class employee can be a subclass of the class person. An employee is a subtype of person. In this case, the parent of subclass employee is called the superclass. The class that does not have any superclasses is called the base class. A SDM class can have many subclasses. With this strategy, a generalization hierarchy is much easier to model in WDB than other relational database systems where there is no built in support for hierarchy data structures. In addition, since the subclass contains all the attributes of the superclass, EVAs can also reference entities of the subclasses of the EVA's target class. This is often referred to as polymorphism in object oriented programming languages. All hierarchies in WDB must represent a valid directed acyclic graph (DAG). This means that all hierarchies must stem from a single base class. In other words, all superclass paths in a hierarchy must meet at the same base class. In addition, cycles where a subclass is a superclass of itself is not allowed.

Index

An index is a special internal construct used by WDB to optimize query performance. By default, WDB assigns a unique global identifier to each entity upon creation. WDB uses this identifier to access and retrieve a particular entity from the SleepyCat database engine. Since this identifier provides no information about the values of attributes in a particular entity, the only way WDB can search for an entity that satisfies a particular condition is to perform a linear search over all entities. The index provides an auxiliary path that can be used to access these entities by using the values of attributes in an entity as its key. Much like an index of a book, when a query searches for an entity that contains a value for an indexed attribute, WDB can quickly construct a key based on the conditional expression to access that entity. If no such entry is found in the index, an entity that satisfies the conditions does not exist. This dramatically decreases the amount of time required to search for entities since WDB does not have to search through all the entities. When a query is performed, the WDB query optimizer automatically chooses any indexes that can be used to increase the speed of the query. Due to limitations of indexes in the SleepyCat database engine, WDB only supports queries with equality conditions (ie. *`dva_name = value`*) and complete key matches (all attributes in the index must be used). In addition to optimizing query performance, indexes can also be used to enforce uniqueness constraints. When a unique index is declared, WDB ensures that the key represented by its DVA values exists only once.

1.2 Object Definition Language

The Object Definition Language (ODL) is one of the two main languages used to communicate with WDB. ODL is mainly used to define the data structures and their behaviors in the WDB database. This is analogous to the Data Definition Language (DDL) used by other database systems to define the schema. ODL is a declarative language which, unlike programming languages, does not include any executable statements. Since WDB aims to capture the core functionalities of SIM, only a subset of the ODL is utilized. The implemented declarations include base class, subclass, and index elements of the schema.

Base Class Declaration Syntax:

```
CLASS bass_class_name [comment] ([class_attributes,…]);
```

base_class_name	Unique string that identifies the base class being defined. A base class name must be a valid WDB identifier, namely, any combination of letters, digits, hyphens, or underscores, starting with a letter and ending with a letter or digit. The WDB identifier is not case sensitive and does not distinguish between underscores (_) and hyphens (-).
comment	Optional remarks about the base class such as a brief description of its purpose. The comment must be enclosed in quotation marks (" ").
class_attributes	Specifies all class attributes associated with the base class being defined. The syntax for defining class attributes is presented under later in this section.

Subclass Declaration Syntax

```
SUBCLASS subclass_name [comment] OF superclass_name
([class_attributes,…]);
```

subclass_name	Unique string that identifies the base class being defined. A base class name must be a valid WDB identifier, namely, any combination of letters, digits, hyphens, or underscores, starting with a letter and ending with a letter or digit. The WDB identifier is not case sensitive and does not distinguish between underscores (_) and hyphens (-).
comment	Optional remarks about the subclass such as a brief description of its purpose. The comment must be enclosed in quotation marks (" ").
superclass_name	The name of the base class or subclass to be extended by the subclass being defined.
class_attributes	Specifies all class attributes associated with the base class being defined. The syntax for defining class attributes is presented under later in this section.

Class Attribute Declaration Syntax

The attributes in a base class declaration or subclass declaration can be data-valued attributes (DVAs) or entity-valued attributes (EVAs).

```
data_valued_class_attributes:
dva_name [comment] : data_type [dva_options,...];

dva_options:
  REQUIRED
| INITIALVALUE initial_value

data_type:
  INTEGER
| STRING
| BOOLEAN
```

dva_name

Unique string that identifies the base class being defined. A base class name must be a valid WDB identifier, namely, any combination of letters, digits, hyphens, or underscores, starting with a letter and ending with a letter or digit. The WDB identifier is not case sensitive and does not distinguish between underscores (_) and hyphens (-).

comment

Optional remarks about the DVA such as a brief description of its purpose. The comment must be enclosed in quotation marks (" ").

dva_options

Special characteristics or integrity constraints for the DVA.

REQUIRED
This option ensures that this DVA never has a null value. By default, DVAs are not required to have values and will have a special "NULL" value if no value is assigned with the OML.

INITIALVALUE *initial_value*
The INITIALVALUE option allows a default value to be assigned to the DVA when an entity is inserted without an explicit value. The value must be valid values for the data type of the DVA. These specifications are the same with explicit DVA value assignments used in the insert statement of the OML.

data_type

Specifies the data type of the class attribute. For the current

implementation, only integers, strings, and booleans are supported. These types are directly mapped to the corresponding Java types in the query driver. As an effect, all the rules and restrictions for those Java types also apply to these data types.

In the SDM, a pair of EVAs defines the relationship between entities. The cardinality of the EVA in the class that owns that EVA (perspective class) and the inverse EVA in the target class defines the type of relationship formed. Table 1.1 shows the various relationship types that can be constructed based on the cardinality of the EVA pair.

Perspective Class EVA	**Target Class Inverse EVA**	**Relationship Type**
SV	SV	One-to-one relationship
MV	SV	One-to-many relationship
SV	MV	Many-to-one relationship
MV	MV	Many-to-Many relationship (Duplicate instances are allowed)
MV DISTICT	MV DISTICT	Many-to-many relationship (duplicate relationship instances not allowed)

Table 1.1: Relationships Between EVA Pairs

```
entity_valued_class_attributes:
eva_name [comment] target_class_name [eva_options,…];

eva_options:
| REQUIRED
| SV | SINGLEVALUED
| MV | MULTIVALUED [(DISTINCT [,MAX limit])]
| INVERSE IS eva_name
```

eva_name — Unique string that identifies the base class being defined. A base class name must be a valid WDB identifier, namely, any combination of letters, digits, hyphens, or underscores, starting with a letter and ending with a letter or digit. The WDB identifier is not case sensitive and does not distinguish

between underscores (_) and hyphens (-).

comment — Optional remarks about the EVA such as a brief description of its purpose. The comment must be enclosed in quotation marks (" ").

target_class_name — Identifies the target base class or subclass of the entities that can form a relationship with the EVA being defined. The referenced entities must either be an entity of the target class or a subclass of the target class.

eva_options — Special characteristics or integrity constraints for the EVA.

REQUIRED
This option ensures that this EVA always points to one or more entities in the target class depending on the cardinality of this EVA. By default, EVAs do have any relationships with other entities of the target class.

SV or SINGLEVALUED
Defines that this EVA can only form a relationship with only one entity in the target class. By default, all explicitly declared EVAs are single valued (SV).

MV or MULTIVALUED
Defines that this EVA can form a relationship with one or more entities in the target class.

DISTINCT
This option is only allowed for EVAs that are multivalued. It ensures this EVA never references the same entity in the target class twice for any entity of the class that owns this EVA. By default, SIM allows duplicate instances when the relationship type formed by the EVA pair is many-to-many.

MAX *limit*
This option is only allowed for EVAs that are multivalued. This option limits the number of entities this EVA can reference for each entity of the class that owns this EVA. If this EVA is required, the limit must be bigger then 1.

INVERSE IS *eva_name*
This required option specifies the EVA in the target class as the inverse of the EVA currently being defined. Be defining

the inverse EVA, it allows the database user to update the relationship from the perspective of either EVA while ensuring referential integrity.

Index Declaration Syntax

```
INDEX index_name [comment] ON target_class_name (dva_name,…)
[UNIQUE]
```

index_name	Unique string that identifies the base class being defined. A base class name must be a valid WDB identifier, namely, any combination of letters, digits, hyphens, or underscores, starting with a letter and ending with a letter or digit. The WDB identifier is not case sensitive and does not distinguish between underscores (_) and hyphens (-).
comment	Optional remarks about the base class such as a brief description of its purpose. The comment must be enclosed in quotation marks (“ “).
target_class_name	Identifies the target base class or subclass that the index spans. The index applies only to entities that participate in its target class.
UNIQUE	Ensures that each value corresponding to the index key specification is never used twice. In addition, the uniqueness constraint is enforced for entities with null attribute values.
dva_name	The immediate data-valued attributes of the target class that are used to construct the index key. Since WDB only support complete index key matches, the number of data-valued attributes an index uses is important. For example, if an index uses the DVA named “address” and “phone_number”, a query that only specifies the attribute “address” in its equality condition will not be able to take advantage of this index. Due to this fact, It is important to define indexes for commonly searched attribute combinations.

1.3 Object Manipulation Language

The Object Manipulation Language (OML) is the second of two languages used to communicate with WDB. OML is a high-level language used to construct update and retrieve queries.

Retrieval Query Syntax

```
FROM perspective_class RETRIEVE attribute_definition,… WHERE
global_filter_expression;

attribute_definition:
* | dva_name [[OF eva_name]…]
```

perspective_class — Unique string that identifies the base class being defined. A base class name must be a valid WDB identifier, namely, any combination of letters, digits, hyphens, or underscores, starting with a letter and ending with a letter or digit. The WDB identifier is not case sensitive and does not distinguish between underscores (_) and hyphens (-).

attribute definition — This element specifies the data to be retrieve from the entities. This information includes immediate or inherited DVAs or target attribute expressions that traverse the EVA relationships. Use an asterisk (*) to indicate all the immediate and inherited DVAs of the perspective class to be returned.

dva_name — The DVA attribute to output from the retrieve query.

eva_name — Specifies the EVA of the extended attributes to retrieve from. A DVA from the EVA' target class entities must be specified before specifying the EVAs. Multiple levels of extended attributes could be retrieved with multiple OF *eva_name* clauses.

global_filter_expression — The last element of the retrieve query specifies conditions limiting the entities to be retrieved. Only one WHERE clause can be included with the retrieve query. This implementation of SIM supports both equality (=) and inequality (<, >, <=, >=, <>) conditions. In addition, the expression also supports the following Boolean operators along with their order of operation: AND, OR, NOT. The expression must evaluate to a Boolean value. Literal Boolean values can also be used. For example, to retrieve all entities of a perspective class, just specify WHERE TRUE as the global filter expression.

The filter can apply to immediate, inherited, and extended attributes of the perspective class. For single-valued extended attributes, the behavior is similar to that of

DVAs. However, if the extended is multi-value, if one value of the EVA satisfies the condition, a true value is returned.

Output from Retrieve Queries

Output from retrieve queries are represented in tabular form. Each row of the table represents one entity of the perspective class. Table 1.2 illustrates the tabular output from the example query above where only single-valued attributes (DVAs or single valued EVAs) are requested.

Project Number	**Project Title**
101	Camelot
102	Excalibur
103	Galahad

Table 1.2: Tabular output for single-valued retrieve query

When values from multi-valued target expressions are requested in the query, the outputted table represents a tree structure. There are two types of multi-valued target expressions.

Dependent target expressions are all single-valued or extended attributes in the same class and are connected to the perspective class by the same multi-valued EVA. In the example retrieve query:

```
FROM PROJECT_EMPLOYEE
RETRIEVE LAST_NAME, PROJECT_NUMBER OF CURRENT_PROJECTS,
PROJECT_NAME OF CURRENT_PROJECTS, FIRST_NAME OF CHILDREN WHERE
TRUE;
```

`PROJECT_NUMBER OF CURRENT_PROJECTS` and `PROJECT_NAME OF CURRENT_PROJECTS` are dependent multi-valued target expressions since they all belong to the class PROJECTS and are connected to the perspective class PROJECT_EMPLOYEE by the EVA

CURRENT_PROJECTS. Please refer to Appendix A for the example schema used by the example query above.

Multi-valued target expressions are independent if they are single-valued or extended attributes that are connected to the perspective class by different multi-valued EVAs. In the example above, `LAST_NAME OF CHILDREN` and `PROJECT_NAME OF CURRENT_PROJECTS` are independent multi-valued target expressions since they are connected to the perspective class by two different multi-valued EVAs: CHILDREN and CURRET_PROJECTS.

In the structured tabular output of queries containing multi-valued target expressions, each single-valued attribute of each entity in the perspective class is only shown on the first row for that entity. Blank values for each single-valued attribute are displayed for all other occurrences of multi-valued target expressions in subsequent rows. Table 1.3 illustrates an example output for the example query described above. The query returns each value of the single-valued attribute Last Name once for all the values in the dependent multi-valued target attributes Project Number, Project Name, and First Name. Dependent multi-valued target expressions, Project Number and Project Name, for each connected entity of class Project are shown in each subsequent row. Since First Name is an independent entity of Project Number and Project Name, each row of First Name is not correlated in any way to the Project Number and Project Name rows.

Last Name	Project Number	Project Name	First Name
Carlin	101	Camelot	Billy
	102	Excalibur	Ashley
	103	Galahad	
Aquino	102	Camelot	Corin
	103	Galahad	Kirsten
Reinholtz	101	Camelot	Charles
			Jeff

Table 1.3: Structured tabular output for a multi-valued retrieve query

Insert Query Syntax

The Insert query is used to create new entities of a target class. Values and relationships to the DVAs and EVAs respectively can also be assigned in the insert query.

```
INSERT perspective_class [FROM super_class WHERE
transferred_entity_filter_expression] (
assignment_expression,… );

assignment_expression:
  dva_name := dva_value
| eva_name := INCLUDE eva_target_class WITH
              ( eva_filter_expression )
```

perspective_class — The class of the entity to be created.

super_class — The optional from clause is used to extend a pre-existing entity of a super class into the perspective class, which must be a sub class of the super class. Extending a pre-existing entity will preserve all the existing attributes from the

super class along with their values and add the attributes of the sub class. The super class needs not to be an immediate super class of the target sub class. WDB will create any new entities of intermediary classes between the super class and the perspective sub class. If the target class has more then one super classes, it will create any required entities for those super classes.

transferred_entity_filter_expression

Use this expression to specify which entities from the super class to extend to the perspective class. The syntax of this expression is the same as the global filter expression of the WHERE clause used in the RETRIEVE query.

assignment_expression

The assignment expression is used to assign values to the different attributes in the entity to be created. Values can be assigned to both immediate and extended attributes if the perspective class is a sub class.

The expression syntax for DVAs are fairly straight forward. The name of the immediate or extended DVA . *dva_name*, is on the left side of the assignment operator (:=) while the value to be assignment , *dva_value*, is on the right side. Similar to the filter expressions used in WHERE clauses, the values must be formatted according to the value type of the attribute.

For EVA assignments, the name of the EVA, *eva_name*, is on the left side of the assignment operator like the DVA. On the right side, the *eva_target_class* is the target class from which WDB will search for entities to establish a relationship with the entity being created. The EVA target class must either be the target class of the EVA specified during its declaration or a subclass of that target class. The *eva_filter_expression* is used to qualify the entities of the EVA target class for establishing the relationship. The syntax for this filter expression is the same as the filter expressions used in the

WHERE clauses of the RETRIEVE query.

Modify Query Syntax

The modify query alters attribute values in existing entities. The modify query can alter the values of DVAs as well as add, remove, or replace the relationships in an EVA.

```
MODIFY [LIMIT = ALL | limit_number] perspective_class (
assignment_expression,… ) WHERE global_filter_expression

assignment_expression:
  dva_name := dva_value
| eva_name := [INCLUDE | EXCLUDE] eva_target_class WITH
              ( eva_filter_expression )
```

limit_number — The optional LIMIT clause is used to limit the maximum number of elements that will be altered by a MODIFY query. If that number is exceeded, the query is rejected and the database will be left unchanged. By default, the number of 1 is assigned to the LIMIT clause. Assigning the value ALL will allow alterations to all the entities that satisfies the global filter expression of the WHERE clause.

perspective_class — The class of the entity to be created.

super_class — The optional from clause is used to extend a pre-existing entity of a super class into the perspective class, which must be a sub class of the super class. Extending a pre-existing entity will preserve all the existing attributes from the super class along with their values and add the attributes of the sub class. The super class needs not to be an immediate super class of the target sub class. WDB will create any new entities of intermediary classes between the super class and the perspective sub class. If the target class has more then one super classes, it will create any required entities for those super classes.

assignment_expression — The assignment expression is almost identical to the assignment expressions in the insert query

with the exception of the EVA assignment expression.

By default, specifying new target entities for an EVA with the EVA assignment expression will replace any existing relationships. To add more relationships to a multi-valued EVA, use the INCLUDE keyword right after the assignment operator. Any entities that match the EVA filter expression will be added to the relationships formed by that EVA. To remove relationships from a multi-valued EVA, use the EXCLUDE keyword. Any entities matching the EVA filter expression will be removed from any existing relationships formed by the EVA.

global_filter_expression

This filter expression is identical to the filter expression used the in WHERE clause of the RETRIEVE query. Only entities matching the filter will be altered.

Chapter 2

USING WDB

This chapter will present some examples of using WDB for an example organization database schema. The complete ODL and ER diagram of the schema is located in the appendix for reference.

2.1 Defining Classes

This example defines the base class "Person" for the organization database.

Example 1

```
CLASS Person "Persons related to the company"
(
  person-id : INTEGER, REQUIRED;
  first-name : STRING, REQUIRED;
  last-name : STRING, REQUIRED;
  home_address : STRING;
  zipcode : INTEGER;
  home-phone "Home phone number (optional)" : INTEGER;
  us-citizen "U.S. citizenship status"  : BOOLEAN, REQUIRED;

  spouse "Person's spouse if married" : Person, INVERSE IS
   spouse;
  children "Person's children (optional)" : Person, MV
   (DISTINCT), INVERSE IS parents;
  parents "Person's parents (optional)" : Person, MV (DISTINCT,
   MAX 2), INVERSE IS children;
);
```

The first seven attributes are DVAs that contain basic information about the person. For example, the class attribute "person-id" is a DVA that stores integer values. The REQUIRED keyword means a valid value is required for each entity of this class. The last three attributes are EVAs that reference other entities. The first EVA, "spouse", is a single-valued reflexive EVA that references other entities of its own class. Notice the inverse EVA

for "spouse" is also identified in this EVA definition. The last two EVAs, "children" and "parents" are both multi-valued EVAs as indicated by the MV keyword. Notice that extra integrity constraints, DISTINCT and MAX are also used to limit the entities those EVAs can reference. Lastly, all comments are contained by double quotes.

The next example defines a sub-class "Employee" of the organization database.

Example 2

```
SUBCLASS Employee "Current employees of the company" OF Person
(
  employee-id "Unique employee identification" : INTEGER,
   REQUIRED;
  salary "Current yearly salary" : INTEGER, REQUIRED;
  salary-exception "TRUE if salary can exceed maximum" : BOOLEAN;

  employee-manager "Employee's current manager" : Manager,
   INVERSE IS employees-managing;
);
```

This sub-class definition extends a previously defined "Person" class. The "Employee" sub-class will inherit all the attributes from the "Person" class. This sub-class defines three more DVAs and one more EVA. The "employee-manager" EVA references entities of the "Manager" class. Note that the target class, in this case "Manager," does not need to be defined before a referring EVA, in this case, "employee-manager", is defined.

2.2 Inserting Entities

The following example adds a person into the organization database.

Example 1

```
INSERT Person ( person-id := 1 , first-name :=  "Bill" , last-
   name := "Dawer" ,  home_address:= "432 Hill Rd", zipcode :=
   78705, home-phone := 7891903 ,  us-citizen :=  TRUE  );
```

This basic INSERT statement adds an entity to the "Person" base class with attribute assignments listed in the parentheses. Each of the assignment statements assigns a value on the right hand side to the DVA identified on the left hand side of the assignment operator

(:=). Notice string values for STRING typed DVAs are enclosed in double quotes while INTEGER and BOOLEAN values are not.

Example 2

```
INSERT Employee ( person-id := 6 , first-name :=  "Susan" , last-
   name  := "Petro" , home_address:= "323 Country Lane", zipcode
   := 73421, home-phone := 6541238 ,  us-citizen :=  TRUE ,
   employee-id:= 106,salary:= 70210, employee-manager := Manager
   WITH (employee-id = 106));
```

This is another example of using the INSERT statement to add entities. However, this example inserts an entity to the sub-class "Employee." When adding new entities to a sub-class, WDB will automatically create new entities for all the super-class of the inserted sub-class. Values for both immediate and extended attributes can be supplied in the assignment list when creating new sub-class entities. In this example, a new entity of the "Employee" sub-class will be created along with the new entity for the "Person" super-class. Each entity will be supplied with the appropriate values from the assignment list. Notice that the employee-manager EVA will reference a Manager entity with employee-id 106.

Example 3

```
INSERT Employee FROM Person WHERE first-name = "Bill" AND last-
   name = "Dawer" ( employee-id:= 101,salary:= 70200, salary-
   exception := TRUE );
```

In this example, an entity of the "Employee" sub-class is inserted from the super-class "Person". This statement will promote pre-existing entities of the class "Person" that satisfy the conditions in the WHERE clause to entities of the "Employee" sub-class. The immediate attributes in the newly promoted entity of the "Employee" sub-class will contain values identified in the assignment list. Values can only be supplied for the immediate attributes of the new "Employee" entity.

The super-class from which the new entity is inserted does not need to be an immediate super-class of the inserted sub-class. If the entity being promoted does not exist in the levels between the super-class in the FROM clause and the inserted sub-class, new entities will be

created automatically as part of the insert operation. In addition, if the inserted target sub-class extends multiple super-classes, SIM will also create the entities for the other super-class where appropriate.

2.3 Modifying Entities

The MODIFY statement can be used to add, alter, or delete attribute values from pre-existing entities.

Example 1

```
MODIFY LIMIT = 1 Person ( spouse :=  Person WITH (first-name =
   "Bill" AND last-name = "Dawer")  ) WHERE first-name =  "Alice"
   AND last-name = "Dawer";
```

This example replaces the value of the spouse EVA to an entity from the "Person" base class that satisfies the conditions in the WITH clause. The WHERE clause specifies which entity will be altered, in this case, an entity of the "Person" class with first name of "Alice" and last name of "Dawer." The LIMIT clause is used to ensure that only one entity will be altered by this statement even if more the one entity satisfies the WHERE clause.

Example 2

```
MODIFY Person ( children := INCLUDE Person WITH((first-name =
   "Bill" AND last-name = "Dawer") OR (first-name =  "Alice" AND
   last-name = "Dawer"))) WHERE first-name =  "Mike" AND last-
   name = "Dawer";
```

This MODIFY statement differs slightly than the one in Example 1 in that it's modifying a multi-valued EVA. The INCLUDE keyword in this example adds entities that satisfie the condition in the WITH clause to any existing ones. If the INCLUDE keyword is not used, the entities satisfying the WITH clause conditions will replace any existing referenced entities. Using the INCLUDE statement when modifying single-valued EVAs will not alter the modification process. Lastly, since a LIMIT clause is not specified, the default value of one will be applied, meaning at most one entity will be altered by this statement.

Example 3

```
MODIFY Manager (spouse := Employee WITH (employee-id = 106))
   WHERE first-name = "Henry" AND last-name = "Silverstone";
```

Lastly, this example illustrates the class hierarchy features of WDB. This MODIFY statement alters the value of the inherited attribute "spouse" of an entity of class "Manager." The value of the "spouse" EVA is replaced by entities from the "Employee" class that satisfy the conditions in the WITH clause. Since the class "Employee" is a sub-class of the target class "Person" identified in the EVA definition, an "Employee" entity can take the place of the corresponding "Person" entity. SIM checks that all entities being assigned to an EVA are either an entity or a sub-class entity of the target class identified during EVA definition.

2.4 Retrieving Entities

The RETRIEVE statement can be used to query the database for entities that satisfy certain conditions.

Example 1

```
FROM Person RETRIEVE first-name, last-name WHERE TRUE;
```

This query will retrieve all entities of the "Person" base class and display the values of the DVAs "first-name" and "last-name." Other attribute values can also be displayed by adding to the target list. The target list can include any immediate or inherited DVA. In addition, an asterisk(*) can be used in the target list to display all immediate and inherited attribute values of the perspective class. The output from the following two examples is the same.

Example 2

```
FROM Project RETRIEVE * WHERE TRUE;
```

Example 3

```
FROM Project RETRIEVE project-no, project-title WHERE TRUE;
```

The target list can also include extended attributes values from EVAs. However, they must be qualified with the EVA name to show their relationship to the perspective class.

Example 4

```
FROM Person RETRIEVE *, first-name OF spouse OF children WHERE
   TRUE;
```

This example retrieves all of the DVA values for all the "Person" entities along with the first name of their children's spouses. EVAs are traversed from right to left. In this case, the EVA "children" is an immediate EVA of the class "Person". The EVA "spouse" of all the entities referenced by the "children" EVA are then traversed to retrieve the value of the DVA "first-name." When retriving extended values, only immediate and inherited DVA values of the target class specified during the EVA definition can be retrieved. In Example 5, an error is produced because the DVA "salary" is not a valid DVA for the target class "Person" specified by the EVA definition in the "Person" class definition.

Example 5

```
FROM Person RETRIEVE *, salary OF spouse WHERE first_name =
   "Henry" AND last_name = "Silverstone";
```

In addition, an asterisk(*) can also be used to retrieve all the immediate and inherited values of an EVA traversed entity. In this example, the WHERE clause is used to filter for a person named Henry Silverstone. In addition to immediate attributes like the ones used in this example, inherited and extended attributes can also be used as filter criteria.

Example 6

```
FROM Manager RETRIEVE *, * of projects_managing WHERE TRUE;
```

Finally, the WHERE clause can also be used to filter the entities that will be returned by the retrieve query. An entity will only be returned and displayed if the expression in the WHERE clause evaluates to a Boolean true value. Example 7 shows an example that uses immediate and inherited single-valued attributes.

Example 7

```
FROM Manager RETRIEVE * WHERE bonus = 200000 AND last-name =
   "Silverstone";
```

Since the DVAs "bonus" and "last-name" are immediate and inherited DVAs respectively, no qualifications are needed.

Filters can also be based on extended EVA values. Qualifying extended attribute values are similar to syntax used in target lists. If the EVA is single valued, then the behavior is similar to filtering with DVAs. Example 8 illustrates filtering with single-valued EVAs. It displays all entities of "Person" who has a spouse with the first name "Bill."

Example 8

```
FROM Person RETRIEVE * WHERE first-name OF spouse = "Bill";
```

For multi-valued EVAs, the condition is applied to all values of the EVA. If any one value of the EVA satisfies the condition, a true value will be returned.

Example 9

```
FROM Person RETRIEVE * WHERE first-name of children = "Bill";
```

This query will return any "Person" entity if the first name of some of their children is "Bill."

Example 10

```
FROM Person RETRIEVE * WHERE first-name OF children <> "Alice";
```

Example 11

```
FROM Person RETRIEVE * WHERE NOT first-name OF children =
   "Alice";
```

Example 10 and 11 are equivalent queries that will return any "Person" entity if the first name of some of their children is not "Alice."

Chapter 3

IMPLEMENTATION

The choice of implementing WDB in Java is mainly based on the object oriented nature of Java, which is very similar to the Semantic Data Model (SDM). Many of the concepts in SDM such as classes and entities can be directly translated to class and objects in Java. In addition, since the SleepyCat database engine used in this project is also written in pure Java, WDB could run on multiple platforms without any porting efforts.

The choice for using SleepyCat over Java Data Objects (JDO) as the persistence mechanism lies in SleepyCat's support for indexes and ACID transactions (ACID is defined later in this chapter). The lack of support for indexes in JDO means that they must be implemented with B-Trees explicitly in WDB. In addition, since most queries require modifications to multiple Java objects, it is important to employ the ACID model so that the query is executed reliably. Both of these requirements will add large level of complexity to WDB if they are not available in the database engine. While JDO provided a query language called JDOQL that makes retrieving objects easier then SleepyCat, it still required translation between SIM's OML and JDOQL.

This project considered two implementation approaches that utilize some unique features of Java in very different ways.

3.1 Project Design

First Approach

In the first approach to design, we wish to take advantage of as much object oriented features of Java as possible. Since the SleepyCat database engine directly stores serialized Java objects, this approach aims to translate SDM classes into Java classes on the fly and

dynamically load them into WDB. WDB could then create new objects of these translated classes and store them directly in the SleepyCat database engine. SDM attributes are translated into Java properties in these generated classes and their values could be altered using the reflection class in Java. This approach decreases the amount of storage overhead needed by translating SDM classes to Java classes using a custom object generated on the fly to store object data. In addition, the dynamically compiled Java classes generated based on their SDM counterparts will implicitly perform type checking tasks without explicit coding. Since SleepyCat does not allow retrieving all objects that match a partial key, an array stored with a known permanent key will serve as the master index of defined classes in SleepyCat. This allows WDB to randomly access a known class object as well as search through existing classes. In addition, each transplanted class will maintain an array of keys to its instances so they can be recalled when a RETRIEVE query is requested. The basic steps taken by this approach are outlined below.

SDM Class Definition

1 Translate the SDM class into a Java class and write it out to a temporary Java file on disk. DVAs in the SDM class are translated to public properties in the Java class with the same type. EVAs in the SDM are translated into an array along with other properties to store metadata information such as cardinality, inverse EVA, and other options.

2 Call the Java compiler on the local machine to generate a temporary class file.

3 Load the compiled class into the running program and store the class object in the SleepyCat database with a key derived from the class name for quick retrieval. In addition, update the master index with the key at which the class object is stored.

SDM Entity Creation

1 Load the compiled Java class object from SleepyCat and create a new instance of that class.

2 Use the reflection class to assign values to the translated public properties.

3 Store the new object back to SleepyCat with a key derived from the class name and an unique identifier. Update the instance list of the class with the key used to store the new object.

Although this method offers many advantages in theory such as low overhead and implicit type checking of attributes, actual implementation revealed many limitations:

- **Performance Issues:** Since each translated Java class needed to be compiled in the background while running WDB, the user experiences a delay when defining new classes for the first time. The delay could be substantial depending on the size of the class being defined and the efficiency of the Java compiler installed. While caching mechanisms could be used to speed up access to the compiled Java class after the initial compilation, a compiling process is still required when the class definition is updated such as the removal or addition of attributes. Additionally, since the objects of these dynamically generated classes are stored in a serialized format in SleepyCat, an updated class definition might not be compatible with the previous serialized objects.

- **Security Issues:** Another side effect of dynamic compiling is security vulnerabilities. A user could potentially modify the temporary generated Java file and insert malicious code in the constructor that could lead to program crashes or deletion of user data.

- **Compatibility Issues:** Lastly, the SDM offers some unique features that are not part of the Java language. For example, SDM supports multiple inheritance where a subclass could have multiple super classes. One way to simulate this unsupported

feature in Java is to wrap the dynamically generated objects inside of custom pre-defined objects in SIM or add more metadata information inside of dynamic Java classes. This approach will not take advantage of the inheritance and type casting features in Java at all and adds additional overhead. On the other hand, the Java language also contains certain limitations that will make the translation process difficult and complex. For example, there is a hard limit for how many properties a class can contain set by the Java compiler. However, SDM does not set a limit on the number of attributes in a class to allow for scalability to large data sets. This approach will thus hinder the scalability of WDB.

All these limitations and side effects prompted us to look for an alternative design that will provide improved flexibility and scalability while reducing the security flaws and performance issues.

Second Approach

Despite the high utilization of object oriented features of Java to implicitly implement many analogous features of SDM, the first approach lacked the flexibility needed to fully implement all the characteristics of SDM. In addition, the use of reflection classes and dynamic class loaders added complexity to the project that could cause problems in future debugging. The second approach is much simpler than the first since all classes are defined statically at compile time instead of using dynamic compiles and loads. The generated Java classes used in the first approach are replaced by a custom Java class that describes all aspects of the SDM class such as name, attributes, comments, instance keys, etc. (more details about this class is described in later sections). An instance of this class is created and stored in SleepyCat for every new SDM class definition. This improves on overall query processing performance by removing the translation step needed in the previous approach. SDM entities are represented by another custom Java class that stores all the attribute values and relationship connections. Like before, each object is assigned a unique identifier that is stored in an array of keys to all instances of a particular SDM class. Additionally, inheritance

features of SDM are explicitly maintained through child key and parent key properties in the custom SDM entity Java class.

This approach solves many problems encountered by the first approach. However, it accomplishes this with a heavy toll on storage overhead. Since each Java object that represents an SDM entity or class must contain properties to store metadata and methods to manipulate those properties, the serialized size of these Java objects are much larger, especially for simple SDM classes with few attributes. While this might become a problem for production level databases, it should not pose a problem for this version of WDB since it's targeted for small prototype databases used for research and testing purposes.

3.2 WDB Architecture

WDB is built from many different modules made up by Java packages. Each module performs a specific function in the overall data flow of the WDB database. The modularization and encapsulation characteristics of WDB's architecture provide the flexibility for future expandability. For example, the SleepyCat module made up of data adapter and database abstraction objects could be replaced with a relational SQL module to use a relational database for data storage. Figure 2.2 illustrates the overall design of WDB's architecture. While the actual implementation resembles the model below, the separate modules are not always so clearly separated.

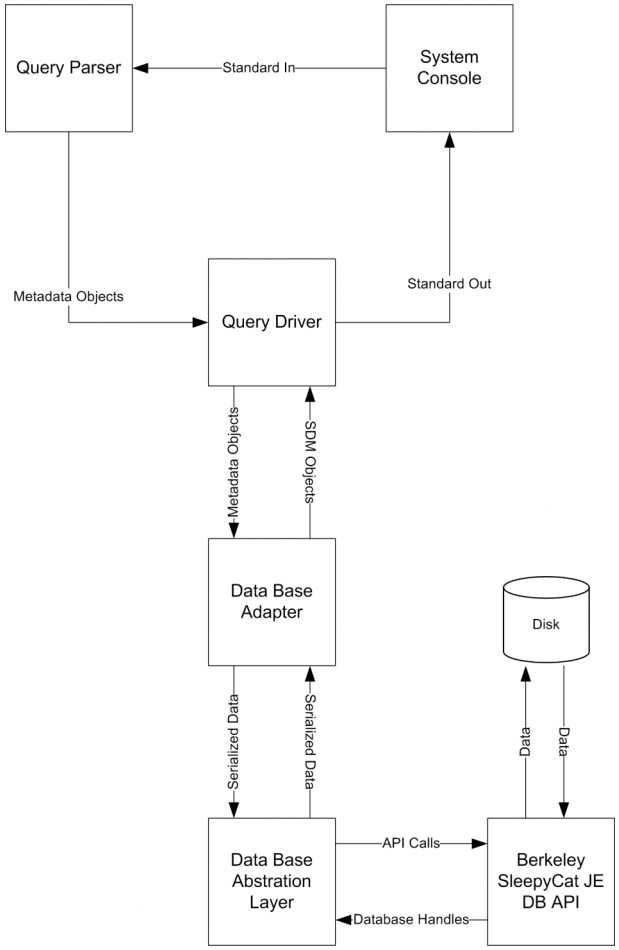

Figure 3.1 SIM Architecture

Query Parser

To parse the Object Definition and Object Manipulation Languages described in the previous chapter, we built a WDB parser using the Java parser generator, JavaCC. JavaCC produces a top-down parser based on extended BNF grammar and regular expression lexical specifications. The BNF specification is included in the appendix for reference. All ODL and OML keywords such as INSERT and WHERE, as well as WDB identifiers such as class and attribute names, are defined as case insensitive tokens in the parser. Only strings enclosed in double quotes or single quotes will remain case sensitive. In addition, underscores and hyphens in WDB identifiers are treated identically by converting all dashes

to underscores using lexical actions in the parser. Some points of the ODL such as DVA and EVA assignment statements produced shift-shift ambiguities in the parser since both statements appear the same until the fifth token. We were able to resolve this issue with the syntactic and semantic look-ahead feature of JavaCC.

Expression trees are generated using the parse tree preprocessor for JavaCC called JJTree. JJTree inserts parse tree building actions into the JavaCC source for a specific set of non-terminals in the language. This implementation only uses JJTree to generate tree nodes for the filter expressions used in the WHERE clauses. The expression tree produced by the grammar specifications will ensure the correct order of operations when traversed.

The package "wdb.parser" makes up the parser module. The QueryParser object generated by JavaCC communicates with the query driver with metadata objects created during the parsing process. Each metadata object represents an ODL or OML statement and contains all the information requested in the query. Most metadata objects that represent ODL statements such as class definitions are often directly stored into the SleepyCat database without any further processing. This preserves the original contents of the query so that it can be reconstructed later. The main QueryParser object, along with the token manager and expression tree objects, make up the parser module of SIM. Please refer to the appendix for a complete listing of these parser classes.

Metadata Objects

As mentioned before, most metadata objects are used to communicate between the different modules in WDB. Most of these classes such as RetrieveQuery, InsertQuery, and ModifyQuery in the "wdb.metadata" package are nothing but public properties and getter and setter methods used to pass information between the parser and the driver. Hence forth, they do not persist across consecutive query statements. However, four classes serve a more vital role in WDB's architecture and are stored in SleepyCat for persistence: ClassDef, SubclassDef, IndexDef, and WDBObject.

The ClassDef class represents SDM class and captures all the information about the class such as name, comments, attribute names, types, indexes, etc. However, it also houses many important methods that operate on all entity instances of that class. One important such method is the search method. The search method first examines the RETRIEVE query against the list of defined indexes for that class. If it found a useable index, it will use them to access the entities in the SleepyCat database. Similarly, the SubclassDef class extends ClassDef and represents SDM sub class definitions. It adds properties and methods for handling super class information. These classes also maintain an array of identifiers to all the entities that are instances of the SDM class they represent. This allows for iterations through all entities of a particular SDM class for searches. Both of these classes are often stored directly into the SleepyCat database after their creation by the parser.

Another data definition class, IndexDef, represents WDB index statements. An instance of the IndexDef stores all information about the index statement including its name, target class, attributes to index, etc. Each instance of IndexDef is stored in the SleepyCat database for later retrieval during searches. In addition, just after the IndexDef object is stored, secondary SleepyCat databases and key generators are also created that correspond to the attributes to index. This allows WDB to take advantage of the inherent indexing features present in SleepyCat.

Finally, the WDBObject class represents SDM entities. They use array properties to store the attribute values as well as methods to set and retrieve the attributes for each SDM entity. In addition, it also maintains referential integrity of any EVA relationships. Each WDBObject instance contains a unique identifier that is used to reference other instances in EVA and hierarchy relationships. The WDBObject class represents all SDM entities, even if they are for different SDM classes. While the WDBObject class contains information about what SDM class a given instance belongs to, it does not contain any information about the structure of the SDM class itself. That information must be obtained from ClassDef or SubclassDef objects that represents the SDM class.

Query Driver

The query driver is the heart of the WDB database system. The driver class, wdb, is not only the entry point for the database program, it also sets up the SleepyCat database and initializes the parser for input from standard in. When metadata objects are received from the parser after a successful parse, the driver is responsible for interpreting the metadata objects and executing the appropriate actions against the SleepyCat database adapter. In addition, it is also responsible for building the entity trees used for output to standard out from entities returned by the RETRIEVE query. In the current implementation, the query driver merely executes queries and passes on the appropriate metadata object without any optimization of the original queries other then using indexes where possible. While this is not a problem for the small databases encountered in this research project, a query optimizer is essential for efficient use of indexes and RETRIEVE queries. Lastly, the driver also maintains an array of all objects stored in the SleepyCat database as a master index of all the SDM classes, entities, and indexes. This array is also stored in SleepyCat with a well-defined method for retrieval and updating.

Database Adapter

The SleepyCat database adapter class, SleepyCatDataAdapter, is mainly used to abstract the serialization and de-serialization procedures when storing and retrieving metadata objects from the SleepyCat database. In addition, each instance of the adapter object also represents an ACID transaction against the SleepyCat database. Characteristics of ACID are as follows:

- Atomicity: All modifications made though an adapter will either all be executed or none will be executed.

- Consistency: If any operation violates consistency checks or produces an error while using the adapter, all previous modifications will be rolled back to their states before the transaction.

- Isolation: Operations against different adapters will be independent from each other. Although this is not a problem for this version of WDB since it is still a single user, single threaded application, it allows for simple implementation of multi-threaded capabilities in the future.

- Durability: Once the adapter is committed after all necessary operations, it ensures that all changes will not be lost even if software or hardware failures are encountered.

The database adapter is also responsible for generating keys for all the objects stored in the SleepyCat database. The keys are just strings that contain the name, such as in the case of ClassDef or IndexDef objects or the SDM class name and the unique identifier in the case of WDBObject objects. The database adapter uses these keys to retrieve these objects back from the SleepyCat database when requested by the query driver.

Database Abstraction Layer

Lastly, the database abstraction layer made up by the SleepyCatDatabase and SleepyCatEnv classes provide handles for the SleepyCat database and transaction objects to the adapter object. In addition, it maintains any secondary databases and keys used by indexes as well as class catalogs to perform serialization tasks.

Chapter 4

COMPARISON AND CONCLUSION

By capturing more information about the meaning of data, WDB promises to improve on some of the weaknesses of the relational data model. This chapter will look at some comparative example queries in WDB and in SQL and conclude if WDB alleviated some of the problems with the relational data model.

4.1 Comparing WDB against SQL

WDB holds two major advantages over relational models. Inheritance features available in WDB avoid multiple tables and complicated foreign keys required in SQL. For instance, in the example organization schema, the subclass "manager" extends subclass "employee" which also extends the base class "person." To build this hierarchical structure in WDB, one only needs to define these three classes with the appropriate superclasses with the OF clause. The ODL required to define these classes is shown below. The equivalent SQL DDL for the Adjacency List Model for storing hierarchical data in relational databases is also included for comparison. Note the attribute definitions are omitted for length.

Define a basic hierarchical structure with three classes

WDB:

```
CLASS Person "Persons related to the company"
( … );
SUBCLASS Employee "Current employees of the company" OF Person
( … );
SUBCLASS Manager "Managers of the company" OF Employee
( … );
```

SQL:

```
CREATE TABLE Person
```

```
( id INTEGER PRIMARY KEY, … );
CREATE TABLE Employee
( id INTEGER PRIMARY KEY,
  person_id INTEGER PRIMARY KEY, … );
CREATE TABLE Manager
( id INTEGER PRIMARY KEY,
  employee_id INTEGER PRIMARY KEY, … );
```

To maintain the hierarchical structure in the flat tables of the relational model, foreign keys and extra primary keys must be defined in the SQL DDL. These extra requirements make the SQL schema much more complicated then the WDB schema while not adding any addition information about the data. In addition, the relationship between these tables is not evident unless the user creates meaningful foreign key names. Lastly, referential integrity of the foreign keys must be specified explicitly if the relational DBMS supports such features. This increases the possibility for update anomalies when foreign keys no longer reference a valid primary key.

Due to the use of foreign keys in relational databases, inserting data is much more complicated and less efficient as WDB. Since the relational models use separate tables to represent an object with inherited attributes, multiple insert statements are required for each of the inherited tables. The example below shows the insert statements for WDB and SQL.

Insert a new manager "Henry Silverstone"

WDB:

```
INSERT Manager ( person-id := 8, first-name := "Henry", last-name
:= "Silverstone", salary :=570201, bonus := 200000, … );
```

SQL:

```
INSERT INTO Person (id, first-name, last-name, … ) VALUES (8,
'Henry', 'Silverstone', … );
INSERT INTO Employee (id, person_id, salary, … ) VALUES (2, 8,
570201, … );
INSERT INTO Manager (id, employee_id, bonus, … ) VALUES (1, 2,
200000, … );
```

As the example shows, not only are three separate SQL insert statements required, the primary keys for each table must also be correlated between the insert statements. In addition, if the SQL server instead of the application generates the primary keys, separate select statements are also required to retrieve the primary keys for the newly inserted tuples. Another benefit of WDB worthy of noting is that when inserting a sub class entity, both immediate and inherited attribute values may be assigned with no knowledge of the class those attribute actually reside. In the case of the SQL insert statements, the user must know in which table a specific attribute is defined.

Another case where WDB is much more efficient than relational models is promoting an entity to another sub class. To accomplish the same task in SQL, a select statement is required to look up the primary key of the existing tuple before inserting the new tuple. The example shows such a case.

Promoting employee "Bill Dawer" to be a manager

WDB:

```
INSERT Manager FROM Employee WHERE  first_name = "Bill" AND
last_name = "Dawer" ( bonus:= 10000 );
```

SQL:

```
SELECT id FROM Employee LEFT JOIN Person ON Employee.person_id =
Person.id WHERE Person.first_name = "Bill" AND Person.last_name =
"Dawer";

INSERT INTO Manager (id, employee_id, bonus) VALUES (3,
$ID_FROM_SELECT, 10000 );
```

For retrieving hierarchical data, WDB's ability to better capture the meaning of the data avoids lengthy SQL joins required by relational DMBSs. This not only streamlines the queries but also increases query efficiency. The example queries below illustrate the length of

each query. The example SQL query assumes the SQL schema defined in the example above.

Print all information about the president

WDB:

```
FROM president RETRIEVE * WHERE TRUE;
```

SQL:

```
SELECT * FROM president
LEFT JOIN manager ON president.manager_id = manager.id
LEFT JOIN employee ON manager.employee_id = employee.id
LEFT JOIN person ON employee.person_id = person.id
WHERE TRUE;
```

Another advantage WDB holds over SQL is its ability to form relationships between entities while maintaining referential integrity. This is often done in relational models with joins on foreign keys and additional constraints. Both of these requirements add to the length of queries and management complexity. The example queries below illustrate this point.

Print all departments, and all the projects the department managers are managing if the department manager has the last name "Dawer".

WDB:

```
FROM department RETRIEVE *, project_title OF projects_managing
OF dept_managers WHERE last_name OF dept_managers = "Dawer";
```

SQL:

```
SELECT department.*, projects.title FROM department
INNER JOIN manager ON department.id = manager.manager_dept
INNER JOIN project ON manager.id = project.project_manager
INNER JOIN employee ON manager.employee_id = employee.id
INNER JOIN person ON employee.person_id = person.id
WHERE person.last_name = "Dawer";
```

4.2 Conclusion

This paper presents an overview of SDM in Chapter 1. It explains the essence of SDM and how it captures the meaning of data. In addition, it also explains how WDB uses the features of SDM such as class definitions and hierarchies, entities, relationships between the entities, and referential integrity constraints. In addition, the syntax of the object definition and object manipulation languages used by WDB are also outlined for easy reference. In chapter 2, the use of WDB is illustrated with examples from the organization schema. Chapter 3 provide details on the implementation of WDB in Java along with the design, data structures, and the architecture within WDB. Lastly, Chapter 4 compares WDB to the SQL relational database model.

This project shows that WDB could be very useful for any application where complex relationships and constraints exist between data entities. Other uses for WDB would be in applications where flexible data models are needed since WDB provides both hierarchical and relational modeling support.

REFERENCES

Boyed Saurabh.
A Semantic Database Management System: SIM.
University of Texas at Austin, 2003.

Hammer, Michael and McLeod, Dennis
The Semantic Data Model: A Modeling Mechanism for Data Base Applications
ACM Press, 1978

InfoExec Semantic Information Manager (SIM) Object Definition Language (ODL) Programming Guide.
Unisys, 1998.

InfoExec Semantic Information Manager (SIM) Object Manipulation Language (OML).
Unisys, 1998.

Kruszelnicki, Jacek
Persist Data With Java Data Objects
Java World, 2002

Tolbert, Doug.
SIM: Origins and Evolution.
Unisys, 1998.

APPENDIX A: SAMPLE ORGANIZATION SCHEMA

```
// Organization Schema

//Person

CLASS Person "Persons related to the company"
(
  person-id    : INTEGER, REQUIRED;
  first-name   : STRING, REQUIRED;
  last-name   : STRING, REQUIRED;
  home_address : STRING;
  zipcode      : INTEGER;
  home-phone "Home phone number (optional)" : INTEGER;
  us-citizen "U.S. citizenship status"  : BOOLEAN, REQUIRED;

  spouse "Person's spouse if married" : Person, INVERSE IS
spouse;
  children "Person's children (optional)" : Person, MV(DISTINCT),
INVERSE IS parents;
  parents "Person's parents (optional)" : Person, MV (DISTINCT,
MAX 2), INVERSE IS children;
);

// Persons with person-id 1 to 5 created

INSERT Person ( person-id := 1 , first-name :=  "Bill" , last-
name := "Dawer" ,  home_address:= "432 Hill Rd", zipcode :=
78705, home-phone := 7891903 ,  us-citizen :=  TRUE  );

INSERT Person ( person-id := 2 , first-name :=  "Diane" , last-
name := "Wall" ,  home_address:= "32 Cannon Dr", zipcode :=
78705, home-phone := 7891903 ,  us-citizen :=  TRUE  );

INSERT Person ( person-id := 3 , first-name :=  "Jennifer" ,
last-name := "Brown" , home_address:= "35 Palm Lane", zipcode :=
73014, home-phone := 2360884 ,  us-citizen :=  TRUE  );

INSERT Person ( person-id := 4, first-name :=  "Alice" , last-
name  := "Dawer" , home_address:= "432 Hill Rd", zipcode :=
78021, home-phone := 6541658 ,  us-citizen :=  FALSE );

INSERT Person ( person-id := 5 , first-name :=  "George" , last-
name := "Layton" ,  home_address:= "347 Nueces St", zipcode :=
78705, home-phone := 8798798 ,  us-citizen :=  TRUE  );
```

```
INSERT Person ( person-id := 9 , first-name :=  "Mike" , last-
name := "Dawer" ,  home_address:= "432 Hill Rd", zipcode :=
78705, home-phone := 7891903 ,  us-citizen :=  TRUE  );

//Finally person-id 1 to 9 People

//Employee

SUBCLASS Employee "Current employees of the company" OF Person

(
  employee-id "Unique employee identification" : INTEGER,
REQUIRED;
  salary "Current yearly salary" : INTEGER, REQUIRED;
  salary-exception "TRUE if salary can exceed maximum" : BOOLEAN;

  employee-manager "Employee's current manager" : Manager,
INVERSE IS employees-managing;
);

// Person with person-id 1, 2, and 5 made employee

INSERT Employee FROM Person WHERE first-name = "Bill" AND  last-
name = "Dawer" ( employee-id:= 101,salary:= 70200, salary-
exception := TRUE );

INSERT Employee FROM Person WHERE person-id = 2 ( employee-id:=
102,salary:= 80210, salary-exception := FALSE );

INSERT Employee FROM Person WHERE person-id = 5 ( employee-id:=
105,salary:= 70201, salary-exception := FALSE );

// Persons with person-id 6 to 7 created and made Employee

INSERT Employee ( person-id := 6 , first-name :=  "Susan" , last-
name  := "Petro" , home_address:= "323 Country Lane", zipcode :=
73421, home-phone := 6541238 ,  us-citizen :=  TRUE , employee-
id:= 106,salary:= 70210);

INSERT Employee ( person-id := 7 , first-name :=  "Steven" ,
last-name := "Williams" ,  home_address:= "3 Seton St", zipcode
:= 78705, home-phone := 8798712 ,  us-citizen := FALSE  ,
employee-id:= 107,salary:= 70210);

// Finally person-id 1, 2, 3, 5, 6 , 7 and 8 are Employee

// Project-Employee
```

```
SUBCLASS Project-Employee "Employees who are project team
members" OF Employee
(
  current-projects "currentproject of employee" : Current-
Project, MV (DISTINCT, MAX 6), INVERSE IS project-members;
);

// Person with Person-id 1, 2, 5, 6 and 7 made  Project-Employee

INSERT Project-Employee FROM Employee WHERE  employee-id = 101 ()
;

INSERT Project-Employee FROM Employee WHERE  employee-id = 102 ()
;

INSERT Project-Employee FROM Person WHERE  person-id =  3
(employee-id:= 103,salary:= 80210) ;

INSERT Project-Employee FROM Employee WHERE  employee-id = 106 ()
;

INSERT Project-Employee FROM Employee WHERE  employee-id = 107 ()
;

// Finally person-id  1, 2, 3, 6 and 7  are Project-Employee

//Manager

SUBCLASS Manager "Managers of the company" OF Employee
(
  bonus "Yearly bonus, if any" : INTEGER;
  employees-managing "Employees reporting to manager" : Employee,
MV, INVERSE IS employee-manager;

 projects-managing "Projects responsible for" : Project, MV, INVERSE IS project-manag
  manager-dept "Department to which manager belong" : Department,
INVERSE IS dept-managers;
);

// Persons with person-id 8 created and made Employee and Manager

INSERT Manager ( person-id := 8 , first-name :=  "Henry" , last-
name := "Silverstone" ,  home_address:= "100 Gates St", zipcode
:= 70007, home-phone := 4565404 ,  us-citizen :=  TRUE ,employee-
id:= 108,salary:= 570201 , bonus:= 200000 );

// Persons with person-id 1  made Manager

INSERT Manager FROM Employee WHERE  employee-id = 101 ( bonus:=
10000 );
```

```
// Finally person-id 1 , 7 and  8 are Project-Employee

// Interim-Manager

SUBCLASS Interim-Manager "Employees temporarily acting as a
project employee and a manager" OF Manager AND Project-
Employee();

// Person with Person-id 1 and 7 made  Interim-Manager.
// Note 7 will be automatically made manager

INSERT Interim-Manager FROM Manager WHERE  employee-id = 101 ();

INSERT Interim-Manager FROM Employee WHERE  employee-id = 107 ();

// Finally person-id 1 and 7

// President

SUBCLASS President "Current president of the company" OF
Manager();

// Persons with person-id 8 made President

INSERT President FROM Person WHERE first-name =  "Henry" AND
last-name = "Silverstone" ();

// Finally person-id 8

// Previous-Employee

SUBCLASS Previous-Employee "Past employees of the company" OF
Person
(
  IsFired  : BOOLEAN ;
  salary "Salary as of termination" : INTEGER, REQUIRED;
);


// Persons with person-id 4 created and made Previous-Employee


INSERT Previous-Employee FROM Person WHERE  person-id = 4 (
salary:= 50500 ) ;
```

```
// Project


CLASS Project "Current and completed Projects"
(
  project-no "Unique project identification" : INTEGER, REQUIRED;
  project-title "Code name for project" : STRING [20], REQUIRED;
  project-manager "Current project manager" : Manager, INVERSE IS
projects-managing;
  dept-assigned "Responsible department" : Department, SV,
INVERSE IS project-at;
  sub-projects "Component projects, if any" : Project, MV,
INVERSE IS sub-project-of;
  sub-project-of "Master project, if any" : Project, INVERSE IS
sub-projects;
);


INSERT Project( project-no:= 701 ,project-title := "Mission
Impossible");

INSERT Project( project-no:= 702 ,project-title := "Code Red");

INSERT Project( project-no:= 703 ,project-title := "Desert
Rose");

INSERT Project( project-no:= 704 ,project-title := "Hallo");

INSERT Project( project-no:= 705 ,project-title := "Stick And
Fly");

INSERT Project( project-no:= 706 ,project-title := "Night
Rider");

// Current-Project

SUBCLASS Current-Project "Projects currently in progress" OF
Project
(
  project-active "Whether project has been started" : BOOLEAN,
REQUIRED;
  project-members "Current employees on project" : Project-
Employee, MV (DISTINCT, MAX 20), INVERSE IS current-projects;
);
```

```
INSERT Current-Project FROM Project WHERE project-title =
"Mission Impossible"( project-active := TRUE );

INSERT Current-Project FROM Project WHERE project-title =
"Hallo"( project-active := FALSE );

INSERT Current-Project FROM Project WHERE project-title = "Stick
And Fly"( project-active := TRUE );

INSERT Current-Project FROM Project WHERE project-title = "Night
Rider"( project-active := TRUE );

// Previous-Project

SUBCLASS Previous-Project "Completed Projects" OF Project
(
  end-date-month "Date project completed month" : INTEGER;
  end-date-day   "Date project completed day" : INTEGER;
  end-date-year  "Date project completed year" : INTEGER;
  est-person-hours "Estimated hours to complete"  : INTEGER;
);


INSERT Previous-Project FROM Project WHERE project-title = "Code
Red"( est-person-hours := 2000,end-date-month := 1, end-date-day
:= 6 , end-date-year := 1999);

INSERT Previous-Project FROM Project WHERE project-title =
"Desert Rose"( est-person-hours := 1300,end-date-month := 5, end-
date-day := 3 , end-date-year := 1997);


// Department


CLASS Department "Departments within the company"
(
  dept-no "Corporate department number" : INTEGER, REQUIRED;
  dept-name "Corporate department name" : STRING [20], REQUIRED;
  project-at "Projects worked on at this department" : Project ,
INVERSE IS dept-assigned, MV (DISTINCT);
  dept-managers "Managers for this department"  : Manager, MV,
INVERSE IS manager-dept;
);


INSERT Department( dept-no:= 501 ,dept-name := "Purchasing");

INSERT Department( dept-no:= 502 ,dept-name := "Sales");
```

```
INSERT Department( dept-no:= 503 ,dept-name := "Marketing");

INSERT Department( dept-no:= 504 ,dept-name := "R&D");

INSERT Department( dept-no:= 505 ,dept-name := "Accounting");


// EVA Relationship

MODIFY LIMIT = 1 Person ( spouse :=  Person WITH (first-name =
"Bill" AND last-name = "Dawer")  ) WHERE first-name =  "Alice"
AND last-name = "Dawer";


MODIFY Person ( children := INCLUDE Person WITH((first-name =
"Bill" AND last-name = "Dawer") WHERE first-name =  "Mike" AND
last-name = "Dawer"  ;


MODIFY LIMIT = ALL Employee (employee-manager := Manager
WITH(first-name =  "Bill" AND last-name = "Dawer")) WHERE
employee-id = 102 OR employee-id = 106;

MODIFY LIMIT = ALL Employee (employee-manager := Manager
WITH(first-name =  "Steven" AND last-name = "Williams")) WHERE
employee-id = 103 OR employee-id = 105;

MODIFY LIMIT = ALL Employee ( employee-manager := Manager
WITH(first-name =  "Henry" AND last-name = "Silverstone")) WHERE
employee-id = 101 OR employee-id = 107;

MODIFY LIMIT = ALL Employee ( employee-manager := Manager
WITH(first-name =  "Henry" AND last-name = "Silverstone")) WHERE
employee-id = 101 OR employee-id = 107;

MODIFY LIMIT = ALL Project-Employee( current-projects := INCLUDE
Current-Project WITH ( project-title = "Mission Impossible" ))
WHERE person-id = 7 OR person-id = 3 OR person-id = 2 OR
employee-id = 106 OR person-id = 1;

MODIFY LIMIT = ALL Project-Employee( current-projects := INCLUDE
Current-Project WITH ( project-title = "Stick And Fly" )) WHERE
person-id = 3 OR person-id = 7 OR person-id = 106;

MODIFY LIMIT = ALL Project-Employee( current-projects := INCLUDE
Current-Project WITH ( project-title = "Night Rider" )) WHERE
person-id = 2 OR person-id = 1 OR person-id = 7;

MODIFY Manager (projects-managing := INCLUDE Project WITH(
project-title  = "Mission Impossible" OR project-title  = "Night
```

```
Rider"), manager-dept := Department WITH ( dept-name = "Sales" ))
WHERE employee-id = 101;

MODIFY Manager (projects-managing := INCLUDE Project WITH(
project-title  = "Stick And Fly" OR project-title  = "Code Red"
OR project-title  = "Desert Rose" OR project-title  = "Hallo"),
manager-dept := Department WITH ( dept-name = "R&D")) WHERE employee-id = 107;

MODIFY Manager (manager-dept := Department WITH ( dept-name =
"Sales")) WHERE employee-id = 108;

MODIFY Department ( project-at := INCLUDE Project WITH ( project-
title  = "Mission Impossible"  OR project-title  = "Night
Rider")) WHERE dept-name = "Sales";

MODIFY Department ( project-at := INCLUDE Project WITH (project-
title  = "Stick And Fly" OR project-title  = "Code Red" OR
project-title  = "Desert Rose" OR project-title  = "Hallo"))WHERE
dept-name = "R&D";

MODIFY Project ( sub-projects := INCLUDE Project WITH (project-
title  = "Stick And Fly" OR project-title  = "Desert Rose"))
WHERE project-title  = "Code Red";
```

APPENDIX B: BNF FOR SIM PARSER

NON-TERMINALS

```
Start          ::= QueryText
                 | <EOF>
                 | <QUIT>
QueryText      ::= Class
                 | Subclass
                 | Insert
                 | Retrieve
                 | Source
                 | Index
                 | Modify
Class          ::= <CLASS> ( getIdentifier | getLString ) (
                       getQString )* <LP> ( Attrs )* <RP> <SC>
Subclass       ::= <SUBCLASS> ( getIdentifier | getLString ) (
                       getQString )* Parents <LP> ( Attrs )* <RP>
                       <SC>
Insert         ::= <INSERT> ( getIdentifier | getLString ) (
                       <FROM> getIdentifier <WHERE>
                       getExpression )? <LP> ( Assignments (
                       <COMMA> Assignments )* )? <RP> <SC>
Retrieve       ::= <FROM> ( getIdentifier | getLString )
                       <RETRIEVE> AttributePath ( <COMMA>
                       AttributePath )* <WHERE> getExpression
                       <SC>
Source         ::= <SOURCE> ( getQString ) <SC>
Index          ::= <INDEX> ( getIdentifier ) ( getQString )? <ON>
                       ( getIdentifier ) <LP> ( getIdentifier (
                       <COMMA> getIdentifier )* ) <RP> (
                       <UNIQUE> )? <SC>
Modify         ::= <MODIFY> ( <LIMIT> <EQ> ( getInteger |
                       <ALL> ) )? ( getIdentifier | getLString ) <LP>
                       ( Assignments ( <COMMA> Assignments )* )?
                       <RP> <WHERE> getExpression <SC>
AttributePath  ::= ( getIdentifier ( <LB> getInteger <RB> )? |
                       <ASTERISK> ) ( <OF> getIdentifier ( <OF>
                       getIdentifier )* )?
Assignments    ::= DvaAssign
```

```
                        |  EvaAssign
DvaAssign               ::= getIdentifier ( <LB> getInteger <RB> )? <ASSN>
                            ( getBoolean | getQString | getInteger )
EvaAssign               ::= getIdentifier <ASSN> ( <INCLUDE> |
                            <EXCLUDE> )? ( getIdentifier | getLString )
                            <WITH> <LP> getExpression <RP>
Parents                 ::= <OF> ( getIdentifier | getLString ) ( <AND> (
                            getIdentifier | getLString ) )*
Attrs                   ::= Dva
                        |  Eva
Dva                     ::= getIdentifier ( getQString )? <COLON> getType (
                            <LB> getInteger <RB> )? ( ( <COMMA> )?
                            DvaOptions )* <SC>
DvaOptions              ::= <INITIALVALUE> ( getQString | getInteger |
                            getBoolean )
                        |  <REQUIRED>
Eva                     ::= getIdentifier ( getQString )? <COLON> (
                            getIdentifier | getLString ) ( ( <COMMA> )?
                            EvaOptions )* <SC>
EvaOptions              ::= <SV>
                        |  <MV> ( <LP> EvaMultivaluedOptions (
                            <COMMA> EvaMultivaluedOptions )* <RP>
                            )?
                        |  <REQUIRED>
                        |  <INVERSE> getIdentifier
EvaMultivaluedOptions   ::= <DISTINCT>
                        |  <MAX> getInteger
getExpression           ::= OrExpression
OrExpression            ::= AndExpression ( <OR> OrExpression )?
AndExpression           ::= UnaryExpression ( <AND> AndExpression )?
UnaryExpression         ::= BoolExpression
                        |  NotExpression
NotExpression           ::= <NOT> BoolExpression
BoolExpression          ::= <LP> OrExpression <RP>
                        |  CondExpression
                        |  TrueExpression
                        |  FalseExpression
CondExpression          ::= getAbsoluteAttributePath getQuantifier (
                            getBoolean | getQString | getInteger )
getAbsoluteAttributePath ::= ( getIdentifier ( <LB> getInteger <RB> )? ) (
                            <OF> getIdentifier ( <COMMA> <OF>
                            getIdentifier )* )?
```

```
TrueExpression      ::= <TRUE>
FalseExpression     ::= <FALSE>
getQuantifier       ::= <EQ>
                    |   <NEQ>
                    |   <GT>
                    |   <LT>
                    |   <GTE>
                    |   <LTE>
getType             ::= <INT>
                    |   <REAL>
                    |   <CHAR>
                    |   <BOOLEAN>
                    |   <STRING>
getIdentifier       ::= <IDENTIFIER>
getLString          ::= <LSTRING>
getQString          ::= <QSTRING>
getInteger          ::= <INTEGER>
getBoolean          ::= <TRUE>
                    |   <FALSE>
```

APPENDIX C: JAVA CLASS STRUCTURE

Class Hierarchy

- java.lang.Object
    - wdb.metadata.Assignment (implements java.io.Serializable)
        - wdb.metadata.DvaAssignment
        - wdb.metadata.EvaAssignment
    - wdb.metadata.Attribute (implements java.io.Serializable)
        - wdb.metadata.DVA
        - wdb.metadata.EVA
    - wdb.metadata.AttributePath (implements java.io.Serializable)
    - wdb.metadata.IndexSelectResult
    - wdb.parser.JJTQueryParserState
    - wdb.metadata.PrintCell
    - wdb.metadata.PrintNode
    - wdb.metadata.Query
        - wdb.metadata.ClassDef (implements java.io.Serializable)
            - wdb.metadata.SubclassDef
        - wdb.metadata.IndexDef (implements java.io.Serializable)
        - wdb.metadata.RetrieveQuery
        - wdb.metadata.SourceQuery (implements java.io.Serializable)
        - wdb.metadata.UpdateQuery
            - wdb.metadata.InsertQuery
            - wdb.metadata.ModifyQuery
    - wdb.parser.QueryParser (implements wdb.parser.QueryParserConstants, wdb.parser.QueryParserTreeConstants)
    - wdb.parser.QueryParser.JJCalls
    - wdb.parser.QueryParserTokenManager (implements wdb.parser.QueryParserConstants)
    - wdb.parser.SimpleCharStream
    - wdb.parser.SimpleNode (implements wdb.parser.Node, java.io.Serializable)
        - wdb.parser.And
        - wdb.parser.Cond
        - wdb.parser.False
        - wdb.parser.Not
        - wdb.parser.Or
        - wdb.parser.Root
        - wdb.parser.True

- wdb.SleepyCatDataAdapter
- wdb.SleepyCatDataBase
- wdb.SleepyCatDbEnv
- wdb.SleepyCatKeyCreater (implements com.sleepycat.je.SecondaryKeyCreator)
- java.lang.Throwable (implements java.io.Serializable)
    - java.lang.Error
        - wdb.parser.QueryParser.LookaheadSuccess
        - wdb.parser.TokenMgrError
    - java.lang.Exception
        - wdb.parser.ParseException
- wdb.parser.Token
- wdb.WDB
- wdb.metadata.WDBObject (implements java.io.Serializable)

Interface Hierarchy

- wdb.parser.Node
- wdb.parser.QueryParserConstants
- wdb.parser.QueryParserTreeConstants