Matisse
The Object Developer's Database

Managing a Class Hierarchy with SQL in Matisse

A Technical Brief

Matisse Software Inc.

Introduction

The SQL standard, which has been constrained for the past 30 years to query and manipulate relational tables, has now been morphed by Matisse Software into a powerful language to operate on a hierarchy of classes. This technical brief explores the management of a hierarchy of classes with standard SQL-99, and shows the benefits of using SQL on an object database schema in terms of simplicity, extensibility and performance.

Object developers will discover that it is now possible to develop component-based business logic with “stored methods”. While relational database developers will discover that classes, attributes, stored methods and objects can easily be mapped into familiar relational concepts respectively tables, columns, stored procedures and rows.

Defining a Class Hierarchy

In response to the market demand, relational vendors have extended their relational model to support the definition of class hierarchy. The object-relational model was born. However database developers are not taking advantage of these object features, simply because it is (1) complex to design, (2) the class hierarchy is too rigid to be extended and (3) the runtime performance degrades substantially compared to plain relational modeling.
Consequently, what database developers really need is a true object model accessible via SQL.

For example, consider the case of a simple class hierarchy that models clients in a stock management system as described by the UML diagram in Figure 1. Ideally, you would like to define categories of clients that share properties from the parent `Customer` class while maintaining properties of their own. This category tree lends itself naturally into a hierarchy of classes.

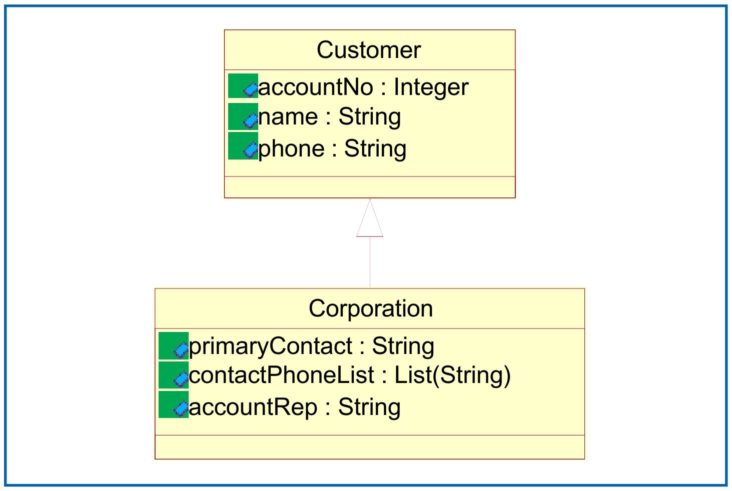

Figure 1: The class Customer and its subclass Corporation in UML

Matisse Native Object Model matches naturally with the UML class hierarchy, resulting in a two classes database schema as shown in figure 2.

```
CREATE CLASS Customer (
  accountNo INTEGER NOT NULL,
  name STRING,
  phone STRING
 );

 CREATE CLASS Corporation UNDER Customer (
  primaryContact STRING,
  contactPhoneList LIST(STRING),
  accountRep STRING
 );
```

Figure 2: The class Customer and its subclass Corporation in DDL with Matisse

By comparison, with the object-relational model, you need to define types and tables (and possibly views) to map into a class hierarchy, as shown in figure 3. This gives you the illusion to deal with an object model while it all ends up into relational tables. The underlying model of object-relational databases remains the same: rows and columns. These mappings between types and tables or tables and views are usually complex to define, to maintain and error prone, so database developers are rarely using these features

```
CREATE TYPE Customer_objtyp AS OBJECT (
  accountNo NUMBER,
  name VARCHAR(50),
  phone VARCHAR(20)
 ) NOT FINAL;

 CREATE TYPE contactPhones_vartyp
     AS VARRAY(10) OF VARCHAR(20);

 CREATE TYPE Corporation_objtyp
        UNDER  Customer_objtyp (
  primaryContact VARCHAR(50);
  contactPhones contactPhones_vartyp,
  accountRep VARCHAR(50);
 ) NOT FINAL;

 CREATE TABLE Customer_objtab
    OF Customer_objtyp (accountNo PRIMARY KEY)
  OBJECT IDENTIFIER IS PRIMARY KEY;

 CREATE TABLE Corporation_objtab
    OF Corporation_objtyp
  (accountNo PRIMARY KEY)
  OBJECT IDENTIFIER IS PRIMARY KEY;
```

Figure 3: The type Customer and its subtype Corporation in DDL with an Object-Relational DBMS

The definition of both `Customer_objtab` and `Corporation_objtab` tables, in figure 3, prevents the application from accessing the corporate customers when selecting data from the `Customer_objtab` table, which is expected when programming with objects.

Accessing Objects in the Hierarchy

A key benefit of modeling a class hierarchy is to filter data by class and sub-classes within the hierarchy. Another important advantage is the generic access to objects, which lets you extend the class hierarchy without modifying the access methods already in place. However with object-relational technology, accessing objects in a class hierarchy is quite challenging, which often leads developers to flatten the class hierarchy into a single table.

In Figure 4, Matisse `SELECT` statement returns all the instances of the class `Customer` and its subclass `Corporation` in a table format. The columns of the resulting table are the attributes of the Customer class.

```
> SELECT * FROM Customer;

accountNo name         phone
--------- ------------ ------------
 33450001 John Doe      415-596-1042 <- Customer
 33450008 Mary Smith    650-222-2222 <- Customer
102000448 Matisse Inc   650-548-2581 <- Corporation
```

Figure 4: Selecting all customer types with Matisse

Matisse also supports the syntax to select `Customer` objects while excluding objects of its sub-classes by using the keyword `ONLY` to filter objects as shown in figure 5.

```
> SELECT * FROM ONLY Customer;

accountNo name         phone
--------- ------------ ------------
 33450001 John Doe      415-596-1042 <- Customer
 33450008 Mary Smith    650-222-2222 <- Customer
```

Figure 5: Selecting customers without corporate customers with Matisse

On the other hand, when you are using the object-relational model defined in figure 3, retrieving all customers requires to combine the result of two `SELECT` statements from the tables for `Customer_objtyp` and `Corporation_objtyp` (figure 6).

```
> SELECT accountNo, name, phone
        FROM Customer_objtab
   UNION
  SELECT accountNo, name, phone
        FROM Corporation_objtab;
```

Figure 6: Selecting two types of customers with an Object-Relational DBMS

In order to avoid using `UNION` to select both kinds of customers, you need to create a single table for both `Customer_objtyp` and

Corporation_objtyp, where you insert both types of objects into the table. The problem with this approach is the complexity for accessing sub-table columns. Figure 7 illustrates the type-casting mandatory to filter objects of type Corporation and to access the primaryContact column, which is defined in Corporation_objtyp, and not in Customer_objtyp.

```
> SELECT TREAT(VALUE(c)
         AS Corporation_objtyp).primaryContact
    FROM Customer_objtab c
   WHERE VALUE(c) IS OF (Corporation_objtyp);
```

Figure 7: Selecting a column from the Corporation class with an Object-Relational DBMS

In figure 8, Matisse demonstrates the simplicity of selecting an attribute from a sub-class.

```
> SELECT primaryContact FROM Corporation;
```

Figure 8: Selecting an attribute from Corporation with Matisse

Today, The vast majority of applications and services are developed using an object programming language such as Java, C# or C++. Ideally you would expect to retrieve objects directly from the database, avoiding the O-R mapping layer. To avoid O-R mapping, Matisse provides the REF() function, when used in SELECT statement, returns objects rather than atomic values as shown in the following Java example:

```
String query = "SELECT REF(c) FROM Customer c
                ORDER BY c.accountNo";
stmt = conn.createStatement();
resultSet = stmt.executeQuery(query);
while (resultSet.next()) {
   // Objects of the right class
   // (Customer or Corporate)
   // are actually returned
   Customer c = (Customer)resultSet.getObject(1);
 }
resultSet.close();
stmt.close();
```

Figure 9: A Java program with a SELECT statement returning Java objects

Programming with Stored Methods

Up to this point, we have discussed how class and inheritance can be integrated into SQL, now let's discuss how to implement active business components that rely on unhampered object programming.

Persistent Stored Module (PSM) is the component of the SQL-99 standard that provides syntactic and semantic constructs for the specification of stored methods. In addition to the obvious advantages of stored procedures such as run-time performance or reusability, you also benefit from the object concepts of polymorphism, encapsulation to achieve full reusability and extensibility of your business components.

Consider now the example of a financial institution, which wants to implement diverse fee management policies per account type as described in the account hierarchy in figure 10.

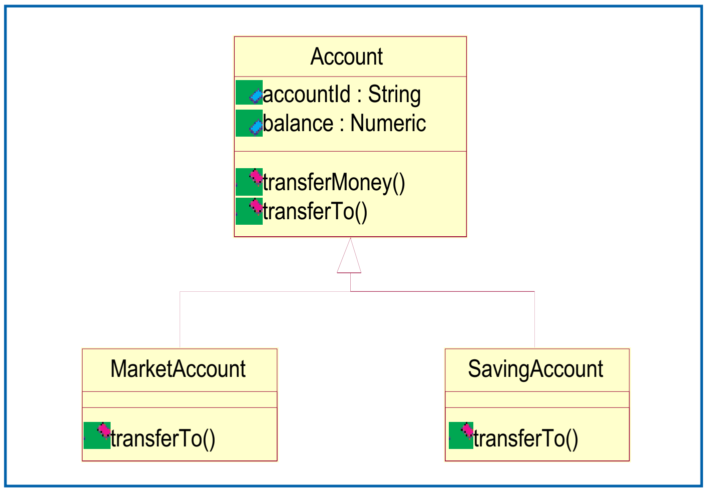

Figure 10: UML diagram for Account classes

The transferMoney() method is a static method, equivalent to a procedure in the relational world, that transfers an amount of money from a source account to a destination one. The transferTo() method is defined on each type of account and it implements the business logic, shown on figure 11, which is associated with each account type.

```
CREATE STATIC METHOD transferMoney (origAccntId STRING,
                                   destAccntId STRING,
                                   amount NUMERIC(19, 2))
RETURNS NUMERIC(19,2)
FOR Account
BEGIN
  DECLARE origAccnt, destAccnt Account; -- Account objects

  SELECT REF(c) INTO origAccnt FROM Account WHERE accountId = origAccntId;
  SELECT REF(c) INTO destAccnt FROM Account WHERE accountId = destAccntId;

  RETURN origAccnt.transferTo (destAccnt, amount);
    -- returns a processing fee
END;
```

```
CREATE METHOD transferTo (destAcct Account, amount NUMERIC(19, 2))
RETURNS NUMERIC(19,2)
FOR Account
BEGIN
  SELF.subtractBalance(amount);
  IF balance < 0 THEN
    ... -- raising exception;
  END IF;
  destAcct.addBalance(amount);
  IF amount > 100.00 THEN
    RETURN 1.20;  -- a processing fee
  ELSE
    RETURN 2.50;  -- a processing fee
  END IF;
END;
```

```
CREATE METHOD transferTo (destAcct Account, amount NUMERIC(19, 2))
RETURNS NUMERIC(19,2)
FOR SavingAccount
BEGIN
  …
  RETURN 1.50;  -- a fixed processing fee
END;
```

```
CREATE METHOD transferTo (destAcct Account, amount NUMERIC(19, 2))
RETURNS NUMERIC(19,2)
FOR MarketAccount
BEGIN
  …
  RETURN amount * 0.0001;  -- a proportional processing fee
END;
```

Figure 11: Stored methods for account management with Matisse

Methods can also be used within regular SQL statements. The `SELECT` statement, in figure 12, has a where-clause which filters customers by area code using the `isInAreaCode()` method.

```
CREATE METHOD isInAreaCode(IN areaCode STRING)
RETURNS BOOLEAN
FOR Customer
BEGIN
  RETURN SUBSTR(telephone, 1, 3) = areaCode ;
END;
```

```
> SELECT * FROM Customer c
          WHERE c.isInAreaCode('415') = TRUE;
```

Figure 12: A SELECT statement using a SQL method and its definition with Matisse

Chances are that you would override the method `isInAreaCode()` for `Corporation`, which searches in `contactPhoneList` as well.

In this case, the object-relational approach shown in figure 13 generates more complex code that is costly to extend and maintain. Extending the class hierarchy, by adding a subclass of Customer, which has a different way of managing contact phone numbers, requires to update all `SELECT` statements that filter by area code.

```
> SELECT name FROM Customer c
  WHERE (SUBSTR(c.telephone, 1, 3) = '415')
  OR (VALUE(c) IS OF (Corporation_objtyp) AND
     (SUBSTR(c.telephone, 1, 3) = '415'
     OR SUBSTR((VALUE(c) AS
      Corporation_objtyp).contactPhone1,1,3)='415'
     OR SUBSTR((VALUE(c) AS
      Corporation_objtyp).contactPhone2,1,3)='415'
     OR SUBSTR((VALUE(c) AS
      Corporation_objtyp).contactPhone3,1,3)='415'
     OR SUBSTR((VALUE(c) AS
      Corporation_objtyp).contactPhone4,1,3)='415'
     OR SUBSTR((VALUE(c) AS
      Corporation_objtyp).contactPhone5,1,3)='415')
```

Figure 13: A relational approach instead of using isInAreaCode() method

Extending the class hierarchy with Matisse consists in defining an overriding method `isInAreaCode()` for the new type, and all the existing logic in place still works without any change.

Joining Classes

Database applications usually require extensive use of multi-table joins. But joins are computationally intensive, and each join is computed at runtime to link information on-the-fly, thereby substancially impacting performance.

To illustrate multi-tables joins, consider the case of a stock management system, where the relevant classes are shown in the UML diagram in the appendix along with its equivalent definition by Matisse Data Definition Language.

```
> SELECT c.name,
         o.orderNo,
         l.lineItemNo
    FROM Customer c, Order o, LineItem l
    WHERE c.orders = o.OID          -- line 5
      AND o.lineItems = l.OID       -- line 6
      AND o.orderNo = 3001;


name          orderNo    lineItemNo
---------- --------- -------------
John Doe        3001          21001
John Doe        3001          21002
```

Figure 14: Joining classes with Matisse

To retrieve the customer and line item data for a specific purchase order, you must join three classes, as illustrated by figure 14. In Matisse SQL syntax, the `OID` property, which represents the object identifier, is implicitly defined for all classes, and plays the role of a primary key. The `orders` property at line 5 is defined in class `Customer` as a reference to `Order` objects, and it acts as the foreign key to the class `Order`.

The query statement shown in figure 15 can be rewritten using a navigational expression through object references as follows:

```
> SELECT o.orderedBy.name,          -- line 1
         o.orderNo,
         o.lineItems.lineItemNo     -- line 3
    FROM Order o
   WHERE o.orderNo = 3001;

name           orderNo     lineItemNo
---------- --------- -------------
John Doe          3001          21001
John Doe          3001          21002
```

Figure 15: Navigation through references with Matisse

Interestingly, you may have noticed that both queries, in figure 14 and 15, return the same results. This demonstrates that join conditions are expressions of relationships between objects. The expression in line 4 in the Join statement (figure 14) is specifying the navigation path through the reference `lineItems` between class `Order` and class `LineItem`, as expressed in line 3 of the query statement figure 15. In fact, Matisse transforms a `SELECT` statement with join conditions into navigational expressions using relationships defined on classes. For both statements, the selection operation (i.e., where clause) filters objects based upon attribute values, and the projection (i.e., select-list) is generated by navigation through inter-objects references.

In summary, the query execution does not need costly join calculations, which are usually computationally intensive operations for relational products.

Matisse eliminates the need for join operations by taking advantage of the explicit references defined between objects. Therefore, for join intensive applications, Matisse easily outperforms relational products by 100x while consuming much less computer resources.

Data Reporting

Since Matisse supports the SQL standard as demonstrated here, as well as ODBC and JDBC, it is easy to connect Matisse to any reporting tool that relies on ODBC or JDBC for easy and quick business data presentation.

A class hierarchy often models a hierarchy of types upon which summary reports can be based.

In figure 16, The `GROUP BY` statement retrieves the average of each customer's total purchase amount by client type. The `SUM(c.orders.totalPurchase)` operation returns each customer's total purchase amount, since `SUM()` aggregates data from all the orders of each customer.

```
> SELECT c.CLASS_NAME AS “Client Type”,
     AVG(SUM(c.orders.totalPurchase)) AS “Avg Odr”
     FROM Customer c
     GROUP BY c.CLASS_NAME;

Client Type         Avg Odr
------------------ -------------
Customer                   124.55
Corporation               2268.23
```

Figure 16: Summary Report using grouping by class

Conclusion

In this brief, we have introduced new ways of managing a class hierarchy with SQL. While in Matisse, SQL queries are relational in their syntax, they also take advantage of the object paradigm by supporting inheritance, polymorphism, and true navigation. Furthermore, query processing takes place on the server to enforce security and achieve best performance. Matisse native object model combined with SQL support demonstrates immediate benefits over relational or object-relational technology:

- Simplicity and extensibility when defining a class hierarchy
- Natural integration with object-oriented languages
- Better performance when joining classes
- True object-oriented programming using SQL methods

These benefits enable you to implement applications and services that require high performance on a level of data complexity that goes beyond the modeling capabilities of legacy relational databases.

Matisse Software Inc.
www.matisse.com
433 Airport Blvd, suite 421
Burlingame, CA 94010
650-548-2581

Download a developer's version of Matisse 6.0 at **www.matisse.com**

Appendix

The following figures represent the UML diagram and class definition for the stock management system, which illustrates multi-tables joins in Matisse.

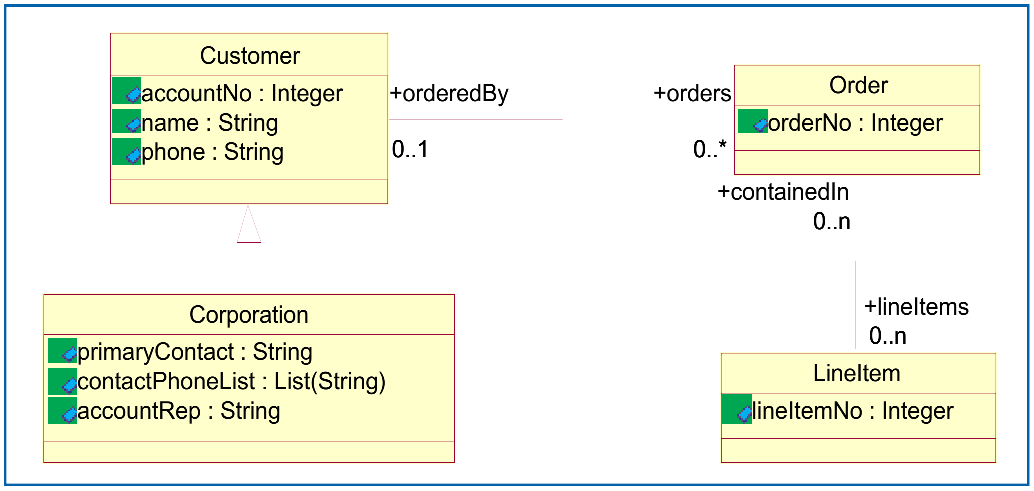

Figure 17: UML diagram for Customer, Order, and LineItem

```
CREATE CLASS Customer (
 AccountNo INTEGER NOT NULL,
 name STRING,
 phone STRING,
 orders REFERENCES (Order)
        INVERSE Order.orderedBy
);

CREATE CLASS Order (
 orderNo INTEGER NOT NULL,
 orderedBy REFERENCES (Customer)
           CARDINALITY (0, 1)
           INVERSE Customer.orders,
 lineItems REFERENCES (LineItem)
           INVERSE LineItem.containedIn
);

CREATE CLASS LineItem (
 lineItemNo INTEGER NOT NULL,
 containedIn REFERENCES (Order)
             INVERSE Order.lineItems
);
```

Figure 18: Class with References in DDL with Matisse