Managing Expressions as Data in Relational Database Systems

Aravind Yalamanchi, Jagannathan Srinivasan, Dieter Gawlick,

Oracle Corporation
{aravind.yalamanchi, jagannathan.srinivasan, dieter.gawlick}@oracle.com

Abstract

A wide-range of applications, including Publish/Subscribe, Workflow, and Web-site Personalization, require maintaining user's interest in expected data as conditional expressions. This paper proposes to manage such expressions as data in Relational Database Systems (RDBMS). This is accomplished 1) by allowing expressions to be stored in a column of a database table and 2) by introducing a SQL EVALUATE operator to evaluate expressions for given data. Expressions when combined with predicates on other forms of data in a database, are just a flexible and powerful way of expressing *interest* in a data item. The ability to evaluate expressions (via EVALUATE operator) in SQL, enables applications to take advantage of the expressive power of SQL to support complex subscription models. The paper describes the key concepts, presents our approach of managing expressions in Oracle RDBMS, discusses a novel indexing scheme that allows efficient filtering of a large set of expressions, and outlines future directions.

1. Introduction

Conditional expressions are a useful way of describing the interest of a user with respect to some expected data. For example in a Content-based subscription system [AS+99], a user may express his interest in an event '`Car4Sale`' (using the syntax defined in [Han92]) as follows:

```
ON    Car4Sale
IF   (Model = 'Taurus' and Price < 20000)
THEN  notify('scott@yahoo.com')
```

Here the expression (`Model = 'Taurus' and Price < 20000`) describes the user's *interest* and the referenced variables, `Model` and `Price`, constitute the relevant *evaluation context* (defined in `Car4Sale`) for this expression.

This paper explores the idea of managing such expressions (along with its evaluation context) as data in a relational database system. We propose to store the expressions defined for a particular evaluation context under a column of a database table. Furthermore, by introducing a special EVALUATE operator that operates on the column storing expressions, one can identify the expressions that evaluate to TRUE for an input data item. The data item constitutes of valid values for all the variables defined in the corresponding evaluation context.

CId	Zipcode	..	Interest
1	32611	..	Model = 'Taurus' and Price < 15000 and Mileage < 25000
2	03060	..	Model = 'Mustang' and Year > 1999 and Price < 20000
..	..	..	..

For example, a `consumer` table can hold expressions as data values in an `Interest` column as shown above. Now the following query can be issued for an input data item in order to identify consumers whose interests are met.

```
SELECT * FROM consumer
WHERE
  EVALUATE (consumer.Interest,
            <data item>[1]) = 1
```

With the ability to store SQL conditional expressions[2] as data in a relational table and query them using the EVALUATE operator, RDBMS becomes an interesting platform for supporting a wide-range of applications including Publish/Subscribe [BA97, AS+99], E-Commerce [MFB00, DHL01], Workflow [CCPP96], Continuous Queries [CDTW00, BW01], Web-site

Proceedings of the 2003 CIDR Conference

[1] The representation of the data items depends on the level of type system support provided by the underlying database system. Section 3.2 provides a solution for Oracle RDBMS and discusses alternate solutions.

[2] Here after, for brevity purposes, 'expressions' are used to mean 'conditional expressions'.

Personalization [CFP99, RSG01], and Resource Management [RLS98].

Unlike existing technologies, which separate characteristics of consumer such as name, telephone number, location, etc. from their interest, our approach allows the interest to be treated as part of consumer characteristics. This allows finer control in identifying relevant consumers. For example, one can identify the consumers based on their interest and `Zipcode`, by issuing a SQL query that uses the EVALUATE operator on the expression column in conjunction with a predicate on the `Zipcode` column as show below:

```
SELECT * FROM consumer
WHERE
  EVALUATE (consumer.Interest,
            <car details>) = 1 AND
  consumer.Zipcode = 03060
```

Thus, expressions, when combined with predicates on other forms of data in a database, are just a flexible and powerful way of expressing *interest* in a data item. The ability to evaluate expressions (via EVALUATE operator) in SQL, enables us to take advantage of the expressive power of SQL to support complex subscription models using constructs such as ORDER BY, GROUP BY and HAVING clauses.

In addition to supporting typical Publish/Subscribe applications, our approach enables applications to do *mutual filtering*. That is, the ability to specify filtering criteria is no longer limited to subscribers. The publisher can as well restrict to whom the data item is delivered, by specifying predicates on the other forms of data associated with the expression (e.g.: `Zipcode` in the above query).

Also, the management of a large number of expressions can only be done by employing database technology. To efficiently process the EVALUATE operator on a large set of expressions, we have developed a novel indexing scheme called the *Expression Filter*. This allows users to optimize filtering of expressions according to the corresponding evaluation context.

Our SQL EVALUATE operator is similar in spirit to the LISP `eval` function [LISP95]. Just like LISP `eval` function, which provides immediate evaluation of the input LISP expression, the SQL EVALUATE operator does immediate evaluation of the input SQL conditional expression for a given input data item.

Our approach of managing expressions as data in RDBMS is somewhat similar to the notion of supporting QUEL commands as a data type in INGRES [SAHR84]. However, QUEL as a data type is primarily motivated to provide a way of representing a set of records through QUEL commands. In contrast, our intent of storing expressions is to capture interest in terms of filtering criteria, which evaluate to TRUE or FALSE for an input data item.

Existing content-based subscription systems and other active systems employ some form of in-memory indexes ([AS+99], [CDTW00], RETE [For82], Ariel [Han92]) for filtering expressions efficiently. In contrast, our indexing scheme creates persistent relational database objects for storage. Index processing involves issuing SQL queries on these objects and this mechanism can potentially scale to large expression sets.

Oracle Rules evaluation engine [ODC01] supports managing ECA rules [WC95] by grouping them as rule sets and evaluating them using procedural interfaces. We are implementing the support for managing expressions as data in Oracle RDBMS. This support complements the Rules evaluation engine functionality by allowing expressions to be stored in database tables that can be evaluated using SQL.

The rest of the paper is organized as follows. Section 2 covers the key concepts related to managing expressions as data. Section 3 discusses our approach of integrating expressions into Oracle RDBMS. Section 4 covers Expression Filter indexing scheme, which enables efficient evaluation of a large collection of expressions. Section 5 discusses future directions. Section 6 concludes with a summary of the paper.

2. Key Concepts

This section covers the key concepts related to managing expressions as data.

2.1 Expression

Expressions are boolean conditions that characterize user's interest in some expected data. The expressions must adhere to SQL-WHERE clause format and can reference variables and built-in or user-defined functions in their predicates. For example, a valid expression can be specified as follows:

```
UPPER(Model) = 'TAURUS' and Price < 20000 and
    HorsePower(Model, Year) > 200
```

Additionally, the expressions can include predicates on data types such as XML, Text and Spatial using special operators. For example, the following expression references the `CONTAINS` operator to identify `Description` that contains the phrase `'Sun roof'`.

```
Model = 'Taurus' and Price < 20000 and
  CONTAINS (Description, 'Sun roof') = 1
```

2.2 Storing Expression as Table Data

Expressions are stored under a column of special data type in a database table and thus for all purposes treated as data. Specifically, expressions can be inserted, updated, and deleted using standard DML statements. For queries projecting the columns holding expressions, the expressions are displayed in string format. The table holding expressions can be replicated like any other table.

Also, by introducing privileges that apply to the column holding expressions one can control the manipulation of expressions via DML operations.

2.3 Expression Metadata

A set of expressions stored in a column of a database table share a common list of variables that can be used in their predicates. Additionally, the expressions can reference any built-in function or approved user-defined functions in their predicates. The list of variable names along with their data types and the list of built-in and approved user-defined functions constitute the metadata for a set of expressions stored in a column. This metadata determines the evaluation context for the corresponding expressions. When a new expression is added or an existing expression is modified (via INSERT or UPDATE), it is validated against this expression set metadata.

2.4 Operator for Evaluating Expressions

The EVALUATE operator is introduced into SQL to evaluate an expression for a given data item. The semantics of the EVALUATE operator can be precisely described by mapping the conditional expression and the associated metadata to an equivalent SQL query. Specifically, the conditional expression maps to the WHERE clause of a SQL query and the corresponding expression metadata determines the structure of the FROM clause of the query. The EVALUATE operator exhibits the semantics of the equivalent query by returning 1 for the expressions that evaluate to true for the data item passed in.

For the example discussed in Section 1, the equivalent query for the `Car4Sale` subscription is as follows:

```
SELECT  1
FROM (SELECT :Model as Model,
             :Year as Year,
             :Mileage as Mileage
      FROM <any_table_with_one_row>)
WHERE
    Model = 'Taurus' and Price < 20000
```

Here, the values for `Model`, `Year`, and `Mileage` variables in the evaluation context are bound to `:Model`, `:Year` and `:Mileage` bind variables respectively. The above query identifies the user only if the conditional expression evaluates to true for the variable values passed in. Additional actions such as notification (`notify('scott@yahoo.com')`) can be performed by including them in the select list items.

2.5 Queries using the EVALUATE Operator

Although a Content-based subscription system is one of the key applications for storing expressions as table data, its usage is not limited to such systems. In the above system, the identification of interested parties and notification can be separated and the `Car4Sale` example can be modelled to use the RDBMS to manage the identification process. Similarly, the `Car4Sale` example can also be used for analyzing the demand for the available inventory.

1) *The ability to evaluate expressions (via EVALUATE operator) in SQL, enables users to take advantage of the expressive power of SQL to support complex applications.*

   For example, when a large number of consumers are interested in an input data item, the *n* most relevant consumers can be identified by using some conflict-resolution [For82] criteria. The query example introduced in Section 1 can be enhanced to use an ORDER BY clause to sort on credit rating and identify the top *n* consumers.

```
SELECT * FROM consumer
WHERE
  EVALUATE(consumer.Interest,
           <car details>) = 1
ORDER BY consumer.Credit_rating DESC
```

2) *Predicates with EVALUATE operator can be combined with predicates on other forms of data to perform filtering based on multiple-domains.*

   For the `Car4Sale` example, the demand for a particular car within a region can be determined by combining the predicate with EVALUATE operator and a spatial predicate using the location of the consumer.

   The ability to filter consumers based on multiple domains can be exploited in a Content-based subscription system to support *mutual filtering*. That is, the ability to specify filtering criteria is no longer limited to subscribers. The publisher can as well restrict to whom the data item is delivered. For the above example, one can limit the notification based on consumer's location by specifying an additional spatial predicate [OSP01]. Furthermore, using a CASE statement in the SELECT list can control the action taken.

```
SELECT
  (CASE WHEN consumer.annual_income
                > 100000
        THEN
             notify_salesperson
                 (consumer.PhoneNumber)
        ELSE
             create_email_msg
                 (consumer.EmailAddress)
   )
FROM consumer
WHERE
  EVALUATE(consumer.Interest,
           <car details>) = 1      AND
  SDO_WITHIN_DISTANCE( consumer.Location,
                       :DealerLoc,
                       'distance=50')
                             = 'TRUE'
```

3) *The EVALUATE operator can be used to perform joins on multiple relations storing expressions and the corresponding data.*

Using the join semantics, a Car dealer in the `Car4Sale` example can sort the available cars based on the demand for them. For this purpose, a batch of data items (Car details) can be stored in a database table and they can be evaluated for a set of expressions by joining the table storing the expressions with this table. These techniques can also be used to perform batch evaluation of data items for a set of subscriptions in a Content-based subscription system.

```
SELECT DISTINCT (inventory.Id),
       count(*) AS Demand
FROM consumer, inventory
WHERE
  EVALUATE (consumer.Interest,
   <car details from inventory table>) = 1
GROUP BY inventory.id
ORDER BY Demand DESC
```

4) *Expressions can be used to maintain complex (N-to-M) relationships between data stored in multiple tables. A join predicate with EVALUATE operator materializes these relationships.*

For example, let there be a table holding information about Insurance policyholders with attributes such as type of insurance, coverage level, and geographical location. Now, a table holding the list of Insurance agents can store expressions defined on policyholder's attributes to maintain an N-to-M relationship between the insurance agents and the corresponding policyholders. By using a join predicate on the column storing (coverage) expressions, the table storing the policyholders can be joined with the insurance agents table to identify all the agents that can attend to each policyholder's needs.

```
SELECT agent.Id,
FROM agent, policyholder
WHERE
  EVALUATE (agent.coverage_expression,
   <policyholder's attributes>) = 1
```

2.6 Indexing Expressions

A special form of index can be defined on a set of expressions stored in a column of a table to process them efficiently with the EVALUATE operator. With this capability, the expressions stored as data in database tables provides a scalable model to manage and evaluate a large collection of expressions.

3. Integrating Expressions into the Oracle RDBMS

This section describes our approach for supporting expressions in Oracle RDBMS.

3.1 Expression Data Type

The data type requirements of a column storing expressions in a database table are complex when compared to those of any built-in or user-defined data type for the following reasons:

1. Since a typical conditional expression is not self-descriptive, the data type should maintain the required metadata for an expression. For example, a predicate `A > '01-AUG-2002'` could produce different results for a particular value of A based on the data type of A. Thus, the metadata for an expression should not only include a list of all the variables that can be used in the expression but also their data types.
2. Since the expressions stored under a column of a database table share common metadata, this columns should be customized for the expression set by associating the corresponding metadata to it. Although a few built-in types like VARCHAR and NUMBER can be customized for length and precision (respectively) at the time of column creation, the different kinds of customizations possible with these types are fixed and limited and hence cannot be extended to handle expressions.

To address these requirements, we adopted the following approach.

Expression Set Metadata, which constitutes of variable names, their data types, and the approved list of user-defined functions acts as the evaluation context for the expression set. The expression set metadata is managed using Oracle RDBMS type system support. Specifically, an object type is defined to capture all the variables that can be referenced in an expression set along with their data types. The expression set metadata with a matching name is created from this object type using a procedural interface. The expression set metadata implicitly includes a list of all the Oracle built-in functions as valid references in the expression set. User-defined functions can be added to this list.

Expression Data Type primarily constitutes of a VARCHAR or CLOB data type to hold the conditional expression. The association of the corresponding Expression Set Metadata is achieved by defining a special *Expression constraint* on the column storing expressions (see Figure 1). This constraint enforces the validity of the expressions stored in the column as well as provides the necessary metadata for expression evaluation.

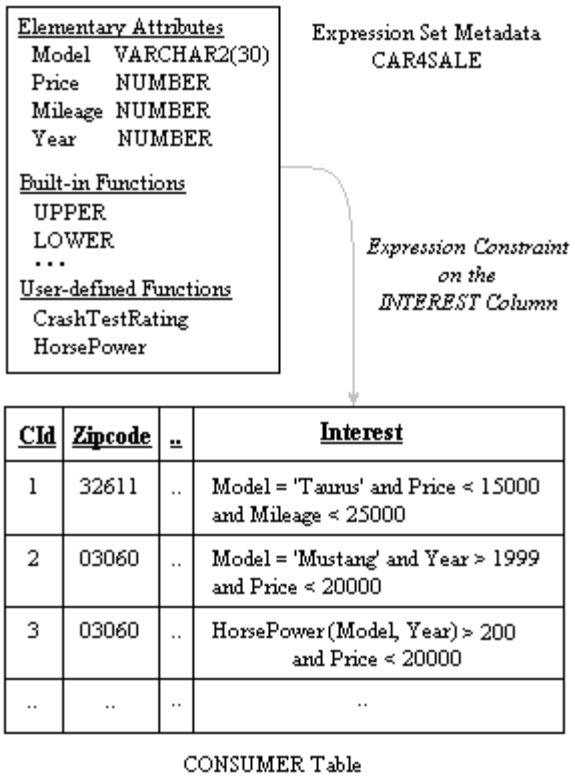

Figure 1: Expression Data Type

Alternately, the Expression data type could be represented as an object (Abstract Data Type) holding the VARCHAR representation of the conditional expression and a reference to the corresponding expression set metadata. When a column of this data type is defined in a table, the Expression constraint can ensure that all the expression instances stored under this column use the same expression set metadata.

3.2 The EVALUATE Operator

The EVALUATE operator, introduced to evaluate expressions, accepts a VARCHAR representation of an expression as the first argument and the data item for which the expression is evaluated as the second argument. This operator returns 1 or 0 based on the outcome of the expression evaluation. While applying the EVALUATE operator on a database column storing expressions, the corresponding expression set metadata is derived from the expression constraint defined on such column. In order to use the EVALUATE operator on a transient expression (not stored in a database column), the corresponding expression set metadata name should be explicitly passed to the operator through an additional VARCHAR argument.

The data item passed to the EVALAUATE operator consists of valid values for all the elementary attributes (or variables) in the corresponding expression set metadata. So, the contents and type of the data item depends on the Expression set metadata associated with the corresponding expression instance. Since the types of the arguments to the EVALUATE operator are predefined, the data item is converted into one of the two canonical forms (string or AnyData[3]) before passing it as the second argument. Hence, the two flavours of EVALUATE operator are:

- *Data item as a string*: For a data item constituting of non-binary data types, it can be represented as a string of name-value pairs.

```
Operator Signature:
  EVALUATE (VARCHAR, VARCHAR)
    returns NUMBER;

Usage:
  SELECT * FROM consumer WHERE
    EVALUATE(consumers.interest,
             'Model=>''Mustang'',
              Price=>22000,
              Mileage=>18000,
              Year=>2000') = 1
```

- *Data item as AnyData*: For a data item constituting of binary data types, the above approach is not possible. For such data items a canonical *AnyData* form of an instance of the corresponding object type should be passed. The above data item can be passed to the EVALUATE operator by converting the instance of `Car4Sale` object type into AnyData as shown in the usage below.

```
Operator Signature:
  EVALUATE (VARCHAR, AnyData)
    returns NUMBER;

Usage:
  SELECT * FROM consumer WHERE
    EVALUATE(consumers.interest,
             AnyData.convertObject(
               Car4Sale('Mustang',
                        22000,
                        18000,
                        2000))
            ) = 1
```

The use of the object type (Abstract Data Type) in the expression set metadata (`Car4Sale` in the above example) is to enable the second flavour of the EVALUATE operator describe above. For database systems with limited type system support, binary types in the data items can be restricted or alternate forms of EVALUATE operator should be considered.

[3] The AnyData data type [OSR01], introduced in Oracle9i, provides interfaces to convert any valid SQL (built-in or user-defined) types into a homogeneous type and convert it back to the original SQL type. The homogeneous type can be stored in database tables or passed as an argument to a function or an operator.

3.3 Expression Evaluation

By default, when an expression and the data item are passed to the EVALUATE operator, a dynamic query is issued to evaluate the expression for the data item. Thus, for a set of expressions (stored in a column), one dynamic query per expression is required. This approach of testing every expression for a data item is a linear time solution and is not scalable for a large set expressions evaluated against a high volume of data items. For this purpose, an optional index can be defined on the column storing the expressions to evaluate them efficiently.

3.4 Indexing a set of Expressions

Using Oracle's Extensibility framework [SM+00], a new mechanism for indexing conditional expressions is introduced. This indexing mechanism is implemented as a new Indextype, Expression Filter, to create and maintain indexes on columns storing expressions. When an Expression Filter index is defined on a column storing expressions, the EVALUATE operator on such column uses the index based on its access cost. For this purpose, the index cost is computed from the expression set statistics like number of expressions in the set, average number of conjunctive predicates per expression, and selectivity of the expressions. For a large set of expressions, the index can quickly eliminate the expressions that are false for a given data item and return only the expressions that evaluate to true.

4. Expression Indexing – Expression Filter

In Oracle RDBMS a new indexing mechanism, Expression Filter, is introduced to efficiently filter a large set of expressions for a data item. As described in Section 3.4, an Expression Filter index can be created on a column storing expressions and the EVALUATE operator on such column may use the index to process the expressions efficiently. This section describes the techniques used to index a set of expressions.

4.1 Approach

Given a large set of expressions, many of them tend to have certain commonalities in their predicates. For a data item, these expressions can be evaluated efficiently if the commonalities are exploited and the processing cost is shared across multiple predicates. For example, given two predicates with a common left-hand side, say `Year = 1998` and `Year = 1999`, in most cases, the falseness or trueness of one predicate can be determined based on the outcome of the other predicate. That is if the predicate `Year = 1998` is true, the other predicate `Year = 1999` cannot be true for the same value of the `Year`. Similar logical relationships can be formed for predicates having common left-hand sides with `range`, `not equal to` and other kind of operators. For example, if the predicate `Year > 1999` is true for a data item, then the predicate `Year > 1998` is conclusively true.

An Expression Filter index defined on a set of expressions exploits the logical relationships between multiple predicates by grouping them based on the commonality of their left-hand sides. These left-hand sides, also called the complex attributes, are arithmetic expressions constituting of one or more elementary attributes and user-defined functions (e.g.: `HORSEPOWER (model, year)`). In the expression set, these left-hand sides (complex attributes) appear in the predicates along with an operator and a constant on the right-hand side (RHS). (Eg : `HORSEPOWER (model, year) >= 150`). Few predicates that do not have constants on the right-hand side can be rewritten to contain a constant on the right-hand side.

4.2 Index Representation

The Expression Filter index internally uses a few persistent database objects to maintain the index information for an expression set. The grouping information for all the predicates in an expression set are captured in a relational table called the *Predicate table*. Typically, the Predicate table contains one row for each expression in the expression set. An expression containing one or more disjunctions is converted into a disjunctive-normal form (Disjunction of Conjunctions) and each disjunction in this normal form is treated as a separate expression with the same identifier as the original expression. The Predicate table contains one row for each such disjunction.

G1 - Predicate Group 1 - with predicates on 'Model'
G2 - Predicate Group 2 - with predicates on 'Price'
G3 - Predicate Group 3 - with predicates on 'HorsePower(Model, Year)'
Op - Predicate operator
RHS - Constant Right-hand side of the predicate
Rid - Identifier of the row storing the corresponding expression in the CONSUMER table
Empty Cells indicate NULL values

	G1		G2		G3		
RId	**Op**	**RHS**	**Op**	**RHS**	**Op**	**RHS**	**Sparse Pred**
r1	=	Taurus	<	15000			Mileage < 25000
r2	=	Mustang	<	20000			Year > 1999
r3			<	20000	>	200	
..	..	..	..	..	..	..	..

Predicate Table for the expressions stored in the INTEREST column of the CONSUMER table

Figure 2: Predicate Table

The most-common left-hand sides of the predicates (complex attributes) in an expression set are identified by

user specification or by statistics collection. For each common left-hand side, a predicate group is formed with all the corresponding predicates in the expression set. For example, if predicates with `Model`, `Price` and `HorsePower(Model, Year)` attributes are very common in the expression set, three predicate groups are formed for these attributes. The Predicate table captures the predicate grouping information as shown in Figure 2.

For each predicate group, the Predicate table has two columns – one to store the operator of the predicate and the other to store the constant on the right-hand side of the predicate. For each predicate in an expression, its operator and the constant on the right-hand side are stored under the corresponding columns of the predicate group. The predicates that do not fall into one of the preconfigured groups are preserved in their original form and stored in a VARCHAR column of the Predicate table as *sparse* predicates (for the above example, the predicates on `Mileage` and `Year` fall in this category). The predicates with IN lists and the predicates involving sub-queries are implicitly treated as *sparse* predicates. Additional indexes are created on the Predicate table as explained in the Section 4.3.

At the time of index creation the Predicate table associated with an expression set is created by the Indextype implementation. The expressions in the set are pre-processed and the predicate table is populated with the predicate grouping information. Additionally, the information stored in the predicate table is maintained to reflect any changes made to the expression set using DML operations on the column storing the expressions

4.3 Index Processing

In order to evaluate a data item for a set of expressions, the left-hand side associated with each predicate group is computed and its value is compared with the corresponding constants stored in the predicate table using appropriate operator. For example, if the `HORSEPOWER('TAURUS',2001)` returns 153, by looking into the predicate table, the predicates satisfying this value are those interested in horsepower being *equal* to 153 or those interested in horsepower being *greater* than a value that is below 153 etc.,. Assuming that the operators and right-hand side constants of the above group are stored in `G3_OP` and `G3_RHS` columns of the predicate table (Figure 2), the following query on the predicate table returns the rows that satisfy this group of predicates.

```
SELECT  exp_id  FROM predicate_table WHERE
   G3_OP = '=' and G3_RHS = :rhs_val   or
   G3_OP = '>' and G3_RHS < :rhs_val   or
   …
```

where `:rhs_val` is the value from the computation of the left-hand side.

Similar techniques are used for `less than` (<), `greater than or equal to` (>=), `less than or equal to` (<=), `not equal to` (!=, <>), `LIKE`, `IS NULL`, and `IS NOT NULL` predicates. Predicates with `BETWEEN` operator are broken into two predicates with `greater than or equal to` and `less than or equal to` operators. Duplicate predicate groups can be configured for a left-hand side if it frequently appears more than once in a single expression (e.g.: `Year >= 1996 and Year <= 2000`).

The WHERE clause shown in the above query is repeated for each predicate group in the predicate table and they are all joined by conjunctions. So, when the complete query is issued on the predicate table, it returns the identifiers for the expressions that evaluate to true with all the predicates in the preconfigured groups. For these resulting expressions, the sparse predicates (if any) stored in the predicate table are evaluated using dynamic queries to determine if an expression is true for the given data item.

```
SELECT exp_id FROM predicate_table
WHERE                  --- predicates in group 1
 (G1_OP is null or      --- no predicate
                        --- involving this LHS
  ((:g1_val is not null AND
    (G1_OP = '=' and G1_RHS = :g1_val or
     G1_OP = '>' and G1_RHS < :g1_val or
     G1_OP = '<' and G1_RHS > :g1_val or
     ...) or
    (:g1_val is null AND G1_OP = 'IS NULL')))

AND                    --- predicates in group 2
 (G2_OP is null or
  ((:g2_val is not null AND
    (G2_OP = '=' and G1_RHS = :g2_val   or
     G2_OP = '>' and G1_RHS < :g2_val   or
     G2_OP = '<' and G1_RHS > :g2_val   or
     ...) or
    (:g2_val is null AND G1_OP = 'IS NULL')))
AND
      ...
```

For efficient execution of the query on the predicate table (shown above), concatenated bitmap indexes [OQ97, ODC01] are created on the {Operator, RHS constant} columns of a few selected groups. These groups are identified either by the user specification or from the statistics about the frequency of predicates (belonging to a group) in the expression set. With the indexes defined on a few preconfigured predicate groups, the predicates from an expression set are classified into three classes:

1. *Predicates with Indexed attributes*: Bitmap indexes are created for the predicates belonging to these groups. In order to evaluate all the predicates in a group, the above query performs a few range scans on the corresponding index and returns the expressions that evaluate to true with just that predicate. Similar scans are performed on the bitmap indexes of other indexed predicates and the results from these index scans are combined using 'BITMAP AND' operations to determine all the expressions that evaluate to true with all the indexed predicates. This enables filtering of multiple predicate groups

simultaneously using one or more bitmap indexes.

2. *Predicates with Stored attributes*: The predicates belonging to this group are captured in the corresponding {Operator, RHS constant} columns of the predicate table, with no bitmap indexes defined on them. For all the expressions that evaluate to true with the indexed predicates, the above query compares the values of the left-hand sides of these predicate groups with those stored in the predicate table. Although bitmap indexes are created for a selected number of groups, the Oracle optimizer may choose not to use the index based on its access cost and such groups are treated as stored predicate groups. The query issued on the predicate table remains unchanged for a different choice of indexes.

3. *Sparse predicates*: For expressions that evaluate to true for all the predicates in its indexed and stored groups, the corresponding sparse predicates (if any), are evaluated in the final stage. If these expressions evaluate to true with the sparse predicates, they are considered true for the given data item.

In order to reduce the number of range scans performed on the bitmap indexes for the evaluation of predicates in the indexed groups, the operators in the predicates are mapped to predetermined integer values. When the < and > operators are mapped to adjacent values (in order), their corresponding range scans can be combined into one. For similar reason, the operators <= and >= are also mapped to adjacent integer values. Optionally, the user can specify the common operators that appear with predicates on a left-hand side and further bring down the number of range scans performed on the corresponding bitmap index. For the above example, `Model` attribute commonly appears in equality predicates and the Expression filter index can be configured to check only for equality predicates while processing the indexed predicate groups. Any remaining predicates on `Model` attribute are processed during sparse predicate evaluation.

4.4 Predicate Table Query

Once the predicate groups for an expression set are determined, the structure of the predicate table is fixed and the query to be issued on the predicate table is fixed. The choice of indexed vs. stored predicate groups does not change the query. So, as part of Expression Filter index creation, the corresponding predicate table query is determined and stored in the dictionary. The same query (with bind variables) is used on the predicate table for any data item passed in for the expression set evaluation. This ensures that the predicate table query is compiled once and reused for the evaluation of any number of data items.

4.5 Cost of Evaluation

The cost of evaluating a predicate in an expression set depends on the group it belongs to. The steps involved in evaluating the predicates in

- an Indexed predicate group is
  - One time computation of the left-hand side of the predicate group.
  - One or more range scans on the bitmap indexes using the computed value.
- a Stored predicate groups is
  - One time computation of the left-hand side of the predicate group.
  - Comparison of the computed value with the operators and the right-hand side constants of all the predicates remaining in the working set (after the filtering of indexed predicates).
- a Sparse predicate group is
  - For all the expressions remaining in the working set after the filtering of indexed and stored predicates, parse of the sub-expression representing the corresponding sparse predicates.
  - Evaluation of the sub-expression through substitution of data values (dynamic query).

4.6 Performance Characterization

Preliminary performance experiments with the Expression Filter indexing scheme showed promising results. For the lack of a standard benchmark for the Expression evaluation, these experiments were conducted using input from Customer Relationship Management (CRM) application. The Expression Filter index performed the best when it is fine-tuned for the given expression set. The tunable characteristics of an index include the list of common predicates, the list of common operators for these predicates and the number of indexed predicates. For a column storing a representative set of expressions, the index can be fine-tuned by collecting expression set statistics and creating the index from these statistics. For expression sets with frequent modifications, self-tuning of the corresponding indexes is possible by collecting the statistics at certain intervals and modifying the index accordingly.

For a set of expressions each having one equality predicate, the best expression evaluation performance can be achieved by creating a simple B^+-Tree index with all the right-hand-side constants in these predicates. For example a large set of expressions with predicates of form `ACCOUNT_ID = :acc_id` can be filtered for a value of `acc_id` by creating a B^+-Tree index with all the values of `acc_id` (RHS constants) in the expression set. However, these indexes should be customized for the given expression set and complex indexing techniques are required for expressions having multiple predicates with possible conjunctions and disjunctions.

For the above expression set, we observed that the performance of the generalized Expression Filter index matched that of the customized index. At the same time, the Expression Filter index can handle complex expressions with multiple predicates with little or no added complexity.

5. Future Directions

5.1 Supporting Expression as a Native Data Type

In the current work, the column of VARCHAR data type with an Expression constraint is treated as a column of Expression data type. So, the semantics of an Expression instance are lost once it is fetched into a programmatic language like JAVA and C. In order to allow operations on a transient (not stored in a table) version of an expression, native support for the expression data type is required.

An Expression data type must be customizable for an expression set by allowing the corresponding evaluation context (or metadata) specification at the time of declaration. For this purpose, the Expression data type should be a special form of parameterized type that accepts the list of valid variables and their data types as parameters. That is, while declaring a column of Expression data type to hold a set of expressions in a table, the variables in the corresponding evaluation context can be passed as arguments to this type in order customize it for the given expression set. Any expression with an invalid variable reference is automatically rejected by the data type check.

Additional operators such as an EQUAL operator to check for logical equivalence of two expressions and an IMPLIES operator to determine if one expression implies another expression can be supported for the Expression data type.

5.2 Supporting EVALAUTE as a Native SQL Operator

Currently, the EVALUATE operator cannot derive the context from the query in which it is used. That is, the data item for which a set of expressions is evaluated should be explicitly passed to the operator as an argument. By supporting EVALUATE operator as a native SQL operator, its functionality can be enhanced to derive the required data items from the current query context. This is beneficial when the data items for which a set of expressions is evaluated are obtained from another table (included in the FROM clause of the same query).

With native SQL support, the EVALUATE operator can also consider alternate execution plans by exploiting any indexes defined on the table storing the data items.

5.3 Indexing Expressions with Text, XML and Spatial predicates

An expression stored in a table may contain domain specific predicates on extensible types like Text, XML and Spatial data types. These domain specific predicates use special operators on the corresponding data. For example, a predicate on XML data can be specified with an `ExistsNode` operator. In order to identify an XML document with a `publication` whose `author` is `Scott` one can use the following predicate.

```
ExistsNode (doc,
   '/publication/author[@name="Scott"]') = 1
```

For the Expression Filter index, the domain specific information for such predicates should be exploited in order to filter them efficiently. The Expression Filter indexing techniques described in Section 4 are being extended to support efficient filtering of XPath predicates on XML Data. For a collection of XPath predicates on a variable of XML data type, these indexes share the processing cost across multiple XPath predicates by grouping them based on the level of XML Elements and the level and the value of XML Attributes appearing in these predicates.

Oracle9i Text engine [OTR01] supports document classification based on the text queries on the expected documents. These text queries are typically the conditions that appear in the CONTAINS operator operating on a Text data type. The document classification uses a specialized index to filter a large collection of text queries for a document.

We plan to integrate the Document Classification index with the Expression Filter index and thus support efficient filtering of expressions involving predicates on Text as well as other data types. In this process, the Expression Filter indexing mechanism will be made extensible to allow easy integration of any new domain-specific classification indexes with the Expression Filter index. In future, this framework can be used to support efficient filtering of predicates on complex data types like Spatial, Image, and Video.

5.4 Characterizing Expressions with respect to data

To identify the most-relevant expression out of the expressions that evaluate to true for a given data item, with the current support, the conflict resolution criteria can be applied on the attributes other than the expression (Section 2.5 Point 1). However, it may be desirable to use certain characteristics of the expressions to choose the most-relevant expression out of a result set. One such characteristic is the *selectivity* of the expressions. For this purpose, each expression can compute a selectivity factor based on the distribution of the expected data items and the most-selective expression in a result set can be chosen as the candidate expression for a data item. Note that the

use of *selectivity* to sort expressions in a result set is similar to the use of *rank* in sorting the results from Text based searches.

The EVALUATE operator can be enhanced to return an ancillary value (selectivity) which can be used to rank the expressions in a result set.

6. Conclusions

The paper explored the idea of managing expressions as data in an RDBMS. Specifically, conditional expressions adhering to SQL-WHERE clause format are allowed to be stored as column values, and an EVALUATE operator is introduced to evaluate expressions as part of SQL queries. With this support RDBMS becomes a platform for supporting wide-range of applications, including Publish/Subscribe, E-Commerce, Workflow, and Resource Management.

A distinguishing aspect of our approach is that the predicates using EVALUATE operator on expressions can be combined with predicates on other relational attributes to perform multi-domain filtering. Furthermore, the expressive power of SQL can be exploited to support complex applications. Since RDBMS manages expressions, the approach implicitly benefits from the database system features, including security, fault-tolerance, and its ability to scale.

We have implemented support for expressions in Oracle RDBMS by leveraging Oracle's Type System, and the Oracle's Extensible Indexing Framework. The indexing scheme introduced to filter a large set of expressions is built into RDBMS and hence can easily scale to large number of expressions.

With the ability to add new SQL data types and Operators into the RDBMS, the expressions stored in tables can easily allow predicates on these new data types. If a classification index exists for the new data type, it can be plugged into the Expression Filter indexing mechanism for efficient expression evaluation. With this support the Expression data type truly becomes extensible.

Acknowledgement

We would like to thank Jayanta Banerjee, Matthieu Devin, Lucy Chernobrod, and Mark Craig for many helpful discussions.

7. References

[AS+99] Aguilera, M., Strom, R., Sturman, D., Astley, M., and Chandra, T. "Matching Events in a Content-based Subscription System". *18th ACM Symposium on Principles of Distributed Computing, 1999: 53-61.*

[BA97] Segall, B., and Arnold, D. "Elvin has Left the Building: A Publish/Subscribe Notification Service with Quenching," *Proc. August 1997.*

[BW01] Babu, S. and Widom, J. "Continuous Queries Over Data Streams", *SIGMOD Record, 30(3), September 2001: 109-120.*

[CCPP96] Casati, F., Ceri, S., Pernici, B., and Pozzi, G. "Deriving Active Rules for Workflow Enactment", *Proc. 7th International Conference on Database and Expert Systems Applications 1996 : 94--110.*

[CDTW00] Chen, J., DeWitt, D. J., Tian F., and Wang, Y. "NiagaraCQ: A Scalable Continuous Query System for Internet Databases", *SIGMOD 2000 : 379-390.*

[CFP99] Ceri, S., Fraternali, P., and Paraboschi, S. "Data-driven one-to-one web site generation for data-intensive applications", *Proc. 25th International Conference on Very Large Databases 1999 : 615-626.*

[DHL01] Dayal, U., Hsu, M., and Ladin, R. "Business Process Coordination: State of the Art, Trends, and Open Issues", *Proc. 27th International Conference on Very Large Databases 2001 : 3-13.*

[For82] Forgy, C. "Rete: A Fast Algorithm for the Many Patterns/Many Objects Match Problem", *Artificial Intelligence 19(1) 1982 : 17-37.*

[Han92] Hanson, E. N. "Rule Condition testing and action execution in Ariel", *SIGMOD Conference 1992 : 49-58.*

[LISP95] Graham, P. "ANSI Common Lisp", *Prentice Hal, 1st edition November 2, 1995.*

[MFB00] Mühl, G., Fiege, L., and Buchmann, A. "Evaluation of Cooperation Models in Electronic Business", *Information Systems for E-Commerce, November 2000.*

[OQ97] O'Neil, P. and Quass, D., "Improved Query Performance with Variant Indexes," *Proceedings of the ACM SIGMOD Inernational. Conference on Management of Data 1997 : 38-49.*

[ODC01] Oracle9i Database Concepts, Release 2, Oracle Corp., http://technet.oracle.com/docs/content.html, *Part# A96524-01 Jun 2001.*

[OSP01] Oracle 9i Spatial Reference, Release 2, Oracle Corp, http://technet.oracle.com/docs/content.html, *Part# A96630-01Jun 2001.*

[OSR01] Oracle9i SQL Reference, Release 2, Oracle Corp., http://technet.oracle.com/docs/content.html, *Part# A96540-01 Jun 2001.*

[OTR01] Oracle9i Text Reference, Release 2, Oracle Corp., http://technet.oracle.com/docs/content.html, *Part# A96518-01 Jun 2001.*

[RLS98] Raman, R., Livny, M., and Solomon, M., "Matchmaking Distributed Resource Management for High Throughput Computing". *Proc. 7th IEEE International Symposium on High Performance Distributed Computing. 1998.*

[RSG01] Rossi, G., Schwabe, D., and Guimaraes, M., "Designing Personalized Web Applications", *Proceedings of the World Wide Web Conference (WWW'10)., May 2001.*

[SAHR84] Stonebraker, M., Anderson, E., Hanson, E. N and Rubenstein, W. B. “Quel as a Data type”, *SIGMOD Conference 1984 : 208-214*.

[SM+00] Srinivasan, J., Murthy, R., Sundara, S., Agarwal, N., and DeFazio, S.: “Extensible Indexing: A Framework for Integrating Domain-Specific Indexing Schemes into Oracle8i”, *ICDE 2000 : 91-100.*

[WC95] Widom, J. and Ceri, S. “Active Database Systems”. *Morgan-Kaufmann, San Mateo, California, 1995.*