PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F 17/30 | A1

(11) International Publication Number: **WO 99/09494**

(43) International Publication Date: 25 February 1999 (25.02.99)

(21) International Application Number: PCT/NZ98/00128

(22) International Filing Date: 14 August 1998 (14.08.98)

(30) Priority Data:
328552 14 August 1997 (14.08.97) NZ

(71) Applicant *(for all designated States except US)*: AORAKI CORPORATION LIMITED [NZ/NZ]; 17 Sir William Pickering Drive, Bishopdale, Christchurch (NZ).

(72) Inventor; and

(75) Inventor/Applicant *(for US only)*: McCOLL, Hugh [NZ/NZ]; 17 Sir William Pickering Drive, Bishopdale, Christchurch (NZ).

(74) Agents: CALHOUN, Douglas et al.; A.J. Park & Son, Huddart Parker Building, 6th floor, Post Office Square, P.O. Box 949, Wellington 6015 (NZ).

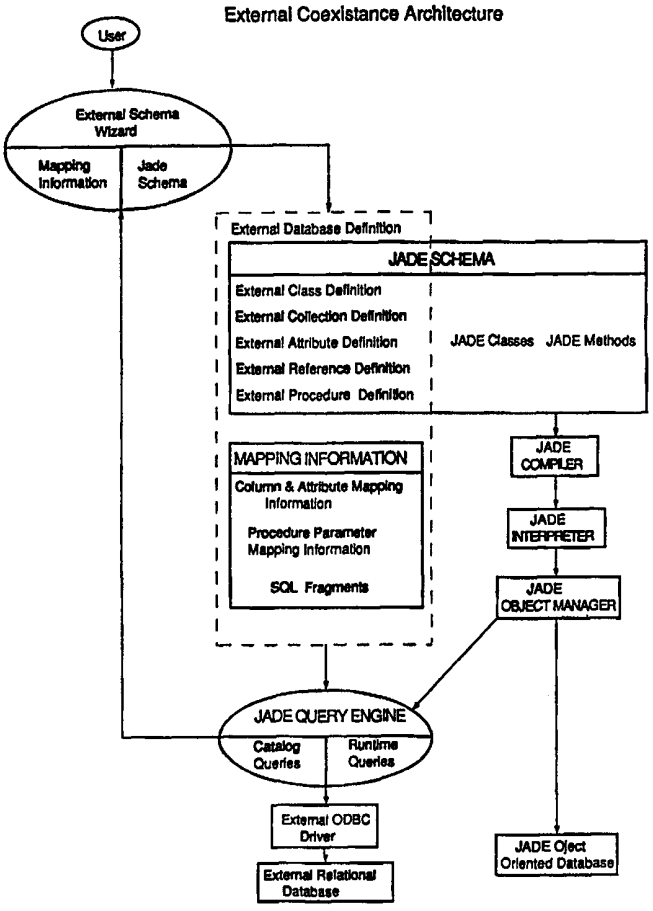
(81) Designated States: AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, DE, DK, EE, ES, FI, GB, GE, GH, GM, HR, HU, ID, IL, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, UA, UG, US, UZ, VN, YU, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published
With international search report.

(54) Title: RELATIONAL DATABASE COEXISTENCE IN OBJECT ORIENTED ENVIRONMENTS

(57) Abstract

Software methodology to enable "legacy" relational database co–existence with an object database in an object oriented system. This comprises a declarative relational to object schema mapping process which is implemented during development and a query engine which makes use of the mapping at run–time to allow access to data in the relational database by object oriented access methods. The methodology allows access to data in the relational database in the same manner as access to the co–existing object database, that is without embedded SQL code, to provide data location transparency from the application developer's viewpoint. Application of the methodology to the JADE™ object oriented development and run–time environment is disclosed.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	**ES**	Spain	**LS**	Lesotho	**SI**	Slovenia
AM	Armenia	**FI**	Finland	**LT**	Lithuania	**SK**	Slovakia
AT	Austria	**FR**	France	**LU**	Luxembourg	**SN**	Senegal
AU	Australia	**GA**	Gabon	**LV**	Latvia	**SZ**	Swaziland
AZ	Azerbaijan	**GB**	United Kingdom	**MC**	Monaco	**TD**	Chad
BA	Bosnia and Herzegovina	**GE**	Georgia	**MD**	Republic of Moldova	**TG**	Togo
BB	Barbados	**GH**	Ghana	**MG**	Madagascar	**TJ**	Tajikistan
BE	Belgium	**GN**	Guinea	**MK**	The former Yugoslav Republic of Macedonia	**TM**	Turkmenistan
BF	Burkina Faso	**GR**	Greece			**TR**	Turkey
BG	Bulgaria	**HU**	Hungary	**ML**	Mali	**TT**	Trinidad and Tobago
BJ	Benin	**IE**	Ireland	**MN**	Mongolia	**UA**	Ukraine
BR	Brazil	**IL**	Israel	**MR**	Mauritania	**UG**	Uganda
BY	Belarus	**IS**	Iceland	**MW**	Malawi	**US**	United States of America
CA	Canada	**IT**	Italy	**MX**	Mexico	**UZ**	Uzbekistan
CF	Central African Republic	**JP**	Japan	**NE**	Niger	**VN**	Viet Nam
CG	Congo	**KE**	Kenya	**NL**	Netherlands	**YU**	Yugoslavia
CH	Switzerland	**KG**	Kyrgyzstan	**NO**	Norway	**ZW**	Zimbabwe
CI	Côte d'Ivoire	**KP**	Democratic People's Republic of Korea	**NZ**	New Zealand		
CM	Cameroon			**PL**	Poland		
CN	China	**KR**	Republic of Korea	**PT**	Portugal		
CU	Cuba	**KZ**	Kazakstan	**RO**	Romania		
CZ	Czech Republic	**LC**	Saint Lucia	**RU**	Russian Federation		
DE	Germany	**LI**	Liechtenstein	**SD**	Sudan		
DK	Denmark	**LK**	Sri Lanka	**SE**	Sweden		
EE	Estonia	**LR**	Liberia	**SG**	Singapore		

"RELATIONAL DATABASE COEXISTENCE IN OBJECT ORIENTED ENVIRONMENTS"

TECHNICAL FIELD

This invention relates to methods and techniques for integrating relational databases effectively with object oriented applications while ensuring co-existence with an object database.

BACKGROUND ART

Relational databases employed in existing systems represent a significant investment which impedes any more to an object oriented model. It has been difficult to map a relational database to an object oriented model, particularly where it is to co-exist with an object database. This has made it difficult to programmatically access the data of a relational database for an object oriented model.

The problem is difficult to that presented by the interface required when an object oriented language sits on a relational database. There the relational database can be defined as part of the development of the object oriented application and the application code can make use of embedded SQL commands. Further in a dedicated object oriented application - relational database combination the database can be accessed directly rather than through the Open Database Connectivity (ODBC) layer provided with most relational databases.

US Patent 5,765,161 discloses a system where data from non-relational, non-object-oriented data stores, such as IBM IMS™, are encalculated as persistent objects for use in an object oriented application. Such a system does not allow for co-existence with a populated object database.

US Patent 5,542,078 discloses a system for accessing and integrating non-object oriented data stores with object applications. An application using an object database management system is provided with an interface to access foreign data

stores which include external data that is mapped and converted into objects for use by the application. This system does not allow the mapping of complex relationships between data to manipulate external data with the same degree of effectiveness as data held in the integrated object database.

DISCLOSURE OF INVENTION

It is an object of the present invention to provide within an integrated object oriented system true co-existence with non-object-oriented databases and in particular relational databases in a manner which allows maximum use of object oriented techniques and obviates the need for knowledge of structural query language (SQL) code on the part of application developers.

Accordingly the invention in one aspect consists in a method of mapping a relational database schema to an object oriented schema to allow co-existence of said relational database with an object database comprising the steps of:

(a) extracting the schema data from said relational database, including database tables, columns, special columns, primary keys, foreign keys, indexes and stored procedures and parameters,

(b) enhancing said relational database scheme by

  (i) mapping tables to classes and columns to attributes,

  (ii) mapping columns to attributes,

  (iii) defining classes of set, array and dictionary collections based on primary key and index data,

  (iv) establishing relationships from said primary keys, foreign keys and columns having the same names,

  (v) defining relationship implementation based on ehuristics by using the cardinality of said tables to define the cardinality of the relationship between the corresponding classes,

(c) generating structural query language (SQL) code fragments to support collection operations from the information derived in step (b) and storing same for run-time access to said relational database, and

(d) storing a description of the classes and collections established in step (b) for use in run-time by an object oriented application whenever access to data in said relational database is required.

In a further aspect the invention consists in a database query engine for an object oriented application which enables access to data in one or more external non-object oriented databases in the same manner as an internal integrated object oriented database to provide data location transparency, said engine comprising:

an external class corresponding to each non-object oriented database which determines how a proxy object is populated by instances from the corresponding external database,

sub classes which re-implement methods to populate external class instances from said external databases while maintaining an identical interface to internal object classes,

a mechanism which produces a representation of the external database schema, and an object identifier to unique key mapping between an external proxy object instance and a row in the external database,

said engine carrying out the following processes:

assembling SQL statements as determined by the definition of the appropriate external class,

appending a JOIN query predicate when accessing a property which is an end point of a bi-directional relationship,

binding parameter values obtained from the attribute values of an external proxy instance to parameter markers in said SQL statements, binding parameter values obtained from the operation request which represent dictionary keys

binding parameter values obtained from the attribute values of the external proxy instance to parameter markers specified in the JOIN predicate of the query, and

executes the resultant SQL query against the respective non-object oriented database.

The said external query engine comprises the parts:

a) External classes, which have two components -namely, an interface and a query specification. The interfaces defines the properties (attributes and relationships) and methods if any and their domains; the query specification (comprising query fragments generated by the Schema Mapping) defines how to populate a proxy with instances (tuples from the external relational database).

b) Subclasses of existing classes (in a single-inheritance hierarchy), which re-implement methods for purposes of populating external class instances from an external data source while maintaining an identical interface and (interface contract) to the existing internal object classes.

c) A mechanism to construct an in-memory "agent" representation of the enriched schema for purposes of caching schema information and implementing meta-schema operations

d) An OID (object identifier) to (unique keys) mapping between an external (proxy) object instance and a row in the external database.

e) A re-implementation of object manager operations particularly get and set property operations for read or write access to simple attribute values

f) A further reimplementation of get property operations for external object properties, which represent an end-point in a bi-directional relationship.

g) A further reimplementation of get and set property operations, which trigger access to 'long binary data' (or blobs - binary large objects).

h) A mechanism to manage a 'result-set' of rows fetched from an external RDMS, including browsing, locking, direct and relative access to rows.

i) An External Collection class, using h) provides the protocol to access individual or multiple instances from a subset of a class extent. The membership of the collection is constrained by attributes of the collection description as defined in the enriched schema.

j) Three types of External Collection:

- Dictionary (providing single or multiple key access to members)
- Sets (an unordered collection of instances
- Array (an ordered collection of instances).

k) An external Iterator class using h) and/or in collaboration with the External Collection class provides the protocol to iterate 'a collection' of external instances.

The said external query engine comprises the steps:

l) Assembles various SQL statement "fragments" as determined by the definition of the external class (determines column and table selection lists) and the context of the class and operation to produce an SQL statement

m) May append filtering and sort predicate fragments as determined by an external collection description.

n) By way of part f) above, when accessing a property which is an end-point of a relationship, appends the JOIN query predicate (generated by schema enrichment)

o) Binds parameter values obtained from the attribute values of an external proxy instance to parameter markers specified in the SQL statement produced by steps l), m), n)

p) Further: for external dictionary operations binds parameter values obtained from the operation request (method invocation) which represent dictionary keys

q) Further: when accessing a relationship end-point property, by means of a get property implementation, binds parameter values obtained from the attribute values of the external proxy instance to parameter markers specified in the JOIN predicate of the query

r) Executes the resultant SQL query against the external RDMS.

s) Automatically maintains a single-valued property (object reference) which is the inverse property of a multi-valued property in a one-many relationship.

The said external query engine allows:

a) Location transparency of data to a Global Schema user,

b) By way of object identifier to external key mapping:

- direct access to objects by reference in local database cache (avoiding external query)
- Ability to synchronise a cached replicas (or proxy instance) with the equivalent representation in the external database.
- Test for membership of a cached object retrieved from one external collection in any other external collection

c) Read or write access to external relational data (along with access to internal

database objects) by way of an Integrated Object-Oriented language using the Class protocols as implemented by the external Collection and Iterator subclasses.

d) Queries over the extent (or all instances) of an external class

e) Queries over a subset of instances of an external class as constrained by the schema description for an external collection

f) Queries over a subset of instances of an external class as further constrained by the schema description for a property that participates in a bidirectional relationship

g) Using an external Dictionary, direct access to a single 'external object' via specified keys

h) Using an external Dictionary, relative key access to a single 'external object' via specified keys (gtr keys, lss, leq keys, first last etc.)

i) Dictionary iterations to be started using a relative key access in either a forward or reverse direction.

j) Using the automatic inverse step s) allows a direct in cache access to the replicated proxy instance using an object reference as opposed to issuing a further query to the external database.

k) The access to long binary data can to be deferred until actually referenced by way of explicit reference to the attribute in an external proxy instance.

l) Ability to dynamically override filtering and sort criteria as statically defined in schema enrichment process.

The invention therefore provides two complementary areas of functionality: a relational to object schema mapping process, and a query engine that implements in the relational database object oriented access methods.

It is possible to map one or more external relational databases to the object model, by providing a higher level integrated view of external relational databases as part of the existing object schema, hiding structural differences between the existing object model and the relational model giving the appearance of assessing a single unified database. The relationship between the tables in the relational database are mapped to bi-directional relationships using inverse references.

Access to "relational objects" in an external database, may then be achieved using an object oriented language, rather than constructing a structured query language (SQL) statement. Mapping is specified declaratively using a "wizard" style mapping tool. The external schema wizard (ESW) enables mapping of:

Tuples (which may be the result of a project/join or a view) to objects.
SQL types to object language primitive types.
Joins to bi-directional relationships based on foreign and primary keys or arbitrary column pairs.

In the present invention existing relational database schemas are enhanced to map the relational database into an object model. Object concepts and language constructs can then access the data in the external relational database in the same way as data is accessed in the co-existing object database. This then allows a query engine that sits between the object environment and the external databases to implement at runtime the object database access methods.

To accommodate more than one external database a Global Conceptual Schema (GCS) is the principal architecture adopted to map a unified schema within the object environment. A partially automated mapping utility is provided to map

relational schema into semantically enriched object model schemas. The GCS provides a high-level integrated view of external relational databases as part of an existing object database schema, hiding structural differences between the object model and the relational model. From the application developer's view there appears to be a single unified database. The key to the mapping is to discover the relationships between tables in the relational schema and to map these to bi-directional relationships implemented using inverse references.

Developers will construct applications which will access relational objects in a procedural manner using an object oriented language rather than by the use of embedded SQL statements in the application code. The mapping will be able to map tuples to objects, SQL types to object primitive types and joins to bi-directional relationships as previously outlined. The mapping is declarative, using a 'wizard-style' mapping tool, rather than programmatic. This reduces coding requirements and increases flexibility.

Data connectivity is provided by the relational database Open Database Connectivity (ODBC) interface, which will be used during the schema transformation phase to access catalogue information and later at 'runtime' to populate virtual object instances via SQL queries.

ODBC is a widely accepted application-programme interface for database access. It is based on the Call-Level Interface (CLI) specifications from X/Open and ISO/IEC for database APIs and uses Structured Query Logic (SQL) as its database access language. ODBC is designed for maximum interoperability - that is the ability of single application to access different database management systems (DBMSs) with the same source. The query engine will call functions in the ODBC interface which are implemented in database specific modules called drivers.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 shows diagrammatically the architecture of an object oriented system

supporting non-object database co-existence,

Figure 2 shows functions performed by the external schema mapping "wizzard" according to the present invention,

Figure 3 shows the architecture of the external query engine of the present invention, and

Figure 4 shows a typical run-time configuration of an object oriented system with external database co-existence.

BEST MODES FOR CARRYING OUT THE INVENTION/OVERVIEW

The following description will, be way of example, refer to the JADE object oriented language and integrated object database produced by Aoraki Corporation Limited and described in the JADE Technical Overview (4.0) published August 1997. JADE provides both development and run-time environments. The invention however is not limited in application to JADE and has application to object oriented systems generally.

OVERVIEW

Global Conceptual Schema

In loosely coupled architectures (such as a gateway architecture), the user is responsible for understanding the structural differences and access languages of the individual database systems. The present invention supports the concept of a Global Conceptual Schema (GCS) which will provide a high-level integrated view of an OO schema plus one or more external relational databases. The intent is to hide the structural differences between different database schema, giving the user the impression that a single database is being accessed.

The proposed GCS architecture is architectured to ensure that it can be readily adapted to support what are commonly referred to as multi-database or federated database systems including non-relational databases such as hierarchical, network or even other OO databases. In addition the actual connectivity may employ mechanisms other than ODBC, such as a native call level interface or other high

level abstractions such as OLEDB.

Semantic Schema Enhancement

A crude approach to mapping a relational schema to a JADE model would be to produce a one-to-one mapping of tables to classes and columns to attributes in JADE. With this approach the domain (or type) of a property could never be another class, i.e. modelling of references or aggregation via collections would not be supportable. In addition, access to the relational database would be reduced to value-based query semantics instead of making use of the inherent navigational power provided by the JADE object model. In order to make effective use of the JADE object model with a uniform conceptual schema approach a process of semantic schema enhancement is employed. This process cannot be a simple deterministic mapping from a pure relational schema to a JADE object model. In general, a relational database schema will not provide the additional higher level semantic information required for deriving useful relationships between classes. This means that the transformation of a relational schema into a JADE object model will be a partially automated process requiring certain additional input from a user with some knowledge of the semantics of the relational database being mapped.

In most cases, JADE will derive certain useful information directly from the relational catalog, such as primary and foreign key specifications. Often, foreign key relationships will also be deduced based on naming conventions. For example, if "dept-id" is the primary key of the department table then it may be deduced that "dept-id" in the employee table is probably a foreign key. Other information, such as relationship implementation, can be deduced heuristically.

Object Identifier Mapping

An 'Object Identifier' to primary key (or special columns) mapping allows the direct mapping of a proxy instance to a row in the relational system. The Object identifier property of the proxy denotes the primary key of the relational entity that the proxy

is modelling. As a simple example, the attribute "dept_num" provides a unique handle for each tuple (row) in a Department table. The attribute dept_num would play the role of a foreign key in an associated Employee table. The OID to primary key mapping is of significance for updating in certain collection operations. For updating a proxy with a searched update it is necessary to uniquely identify the row in the external database which the proxy represents.

External proxy classes

One of the outputs of the Schema Transformation process will be the external proxy classes. The purpose of the proxy classes is to act as a mediator between the OO world and the relational world. These classes will be derived from a common abstract class: ExternalObject a subclass of the Object class. The external proxy classes are instances class type: ExternalClass.

The proxy classes that comprise the transformation schema have two components: (a) a interface, and (b) a mapping or query specification. The interface defines the properties (and methods, if any) and the mapping specification, comprising of 'SLQ-like' queries, defines how to populate a proxy with instances from an external database. The tuples or instances retrieved by this query are 'virtual-instances' of the proxy. The user must deal with instances of the external classes defined in the schema.

An external class definition can be mapped from any one of the following relational 'entities': a base table, a view defined in the relational schema, a query e.g. a join, or the result set of a stored procedure

External class properties

The properties of an external class may be either: (a) attributes, or (b) external references. Attributes are simple types whose domain is one of the supported JADE primitive types. The type of an external reference can be another external class type or an external collection. External references are used to represent the endpoints of a

relationship. External references are always defined as explicit inverses. These references are implemented using mapping methods, which employ either an external object or an external collection method to retrieve an instance or instances.

External collection types

ExternalArray

An ordered collection - requires a sort specification (ORDERED BY clause), optional filtering expression (WHERE clause)

External Dictionary

Requires keys, sort (ORDERED BY) and optionally a filtering specification (WHERE clause)

ExternalSet

An unordered collection, may be Class extend optional filtering expression (WHERE clause)

SQL to JADE data type mapping

The following table lists the ANSI SQL data types and shows the JADE primitive to which they are mapped by default.

ANSI SQL data type	**JADE primitive type**
BINARY, VARBINARY, LONGVARBINARY	Binary
BIT	Boolean
CHAR[1], TINYINT	Character
CHAR, VARCHAR, LONGVARCHAR	String
DECIMAL(p, s), NUMERIC(p, s)	Decimal[p, s]
REAL, FLOAT, DOUBLE	Real
INTEGER, SMALLINT	Integer
BIGINT (64-bit integer)	Decimal[20, 0]
DATE	Date
TIME	Time

TIMESTAMP	Timestamp

[1]p=precision, s=scalefactor

Inheritance

External proxy classes can be organised into an inheritance hierarchy adding further semantics above the relational model. In the mapping process, an external proxy class could be made a subclass of another provided it met the following requirements:

The properties mapped in the superclass are a subset of the properties of the subclass

The 'set of instances' of the subclass are a subset of the 'set of instances' of the superclass

Summary

Mapping technology is used to enhance relational schemas with object model semantics.

Tables (base, derived or predefined views) are mapped to subclasses of External object.

Columns are mapped to attributes.

Foreign key definitions are used to deduce inverse relationships.

Tuples (or rows) map to instances of external classes.

Each external database is represented by a subclass of ExternalDatabase and a singleton instance.

Object Identifier to 'primary key' mapping allows unique identification of a tuple and map this to a 'proxy' object.

Inverse relationships will be defined and implemented using references to class types.

Multi-valued properties will be implemented with Dictionary, Set or Array types.

Schema Transformation Support

Overview

In order to support the process of transforming a relational schema into an object model a schema transformation "Wizard" is provided within the development environment. The Wizard accesses catalog information from selected external data bases and generates classes under the direction of the developer. Some data sources or ODBC drivers may not support all the catalog functions used and some catalogs may not contain all the information required for deriving or suggesting potential relationships.

The Wizard:

(1) allows selection of data sources configured on the development machine,
(2) imports information from relational catalog, including tables, columns, primary and foreign keys, indexes, stored procedures,
(3) supports table selection,
(4) maps base tables or views as well as derived tables join to classes,
(5) provides column selection and column to attribute mapping,
(6) suggests default names for all entities but user can override defaults,
(7) automatically maps SQL types to JADE types,
(8) allows default OO types and length mappings to be overridden,
(9) supports definition of external collection types - will suggest potential dictionaries using primary key and index information obtained from the catalog,
(10)supports the definition of relationships between derived classes - will suggest potential relationships using foreign key information with user specifying cardinality and kind,
(11)stored procedures with parameters and a result set will be mapped to methods on the external database subclass of an ExternalDatabase, class
(12)supports iterative and round-trip definition, and
(13)provides a mechanism to update a mapped object model when the relational schema changes (without having to start from scratch).

Accessing a relational database from JADE

Overview

The user will access the relational database using the JADE language and by navigating relationships via inverse references, e.g. it is possible to code the following style of iterative access:

```
for each emp in dept.employees do
        emp.display( );
endforeach;
```

This is made possible using the collaboration of a new 'external iterator' and an 'external collection'. The 'external collection' will issue the required SQL query to create a 'virtual collection' of instances. That is, in the employee and department tables example, the virtual collection of instances will be the rows of the employee table where employee.dept_id = department.dept_id. The dept_id foreign key is obtained from the virtual department instance, which is the parent of the virtual employees dictionary. The external iterator class will implement the standard iterator protocol i.e. next() and back() methods etc. The external iterator/collection collaboration, will allow constructs such as for each to function in a transparent fashion.

Using external collections

External collections implement most of the non-updating collection methods e.g. first, last, size etc.

External dictionaries provide direct key access to an 'external object instances' i.e. random access to a row or tuple in the relational database.

Example:

```
department := E_Company.departments [name];
```

Relative key access is provided using getAtKeyGtr, Geq, Leq and Lss style methods.

ADHOC query extensions

'ADHOC queries' are supported to some extent by allowing filtering (WHERE

clause) and sort specifications (ORDER BY) to be explicitly set on external collections at runtime, overriding the 'canned query' associated with the collection.

Sequential access

External iterators implement the standard iterator protocol and may be used directly. Standard JADE for each syntax is fully supported including reversed and where clause.

Class::instances to retrieve 'class extent'.

Exception handling

An exception class is provided to provide additional information specific to using the ODBC interface. Special consideration is given to detecting and handling changes to the relational schema which may become inconsistent with the schema mapping.

Where clause optimisation

In the JADE 'for each' instruction, JADE currently optimise special cases when the predicate following the where consists of a conditional key expression. A similar optimisation will be useful when iterating with a for each over an external proxy collection. A special case occurs when the predicate following the where clause contains operands which are all attributes of the collection member type. In this case, the where predicate from the for each can either become the where predicate in the SQL statement or can be conjoined with an existing where predicate in the query specification associated with the external collection.

Updating external databases

An External Database class will provide transaction support and direct SQL execution with beginExternalTransaction, commitExternalTransaction, abortExternalTransaction.

Searched updates will be possible using the ExternalDatabase::executeSQL method.

Positioned or cursor-based updates will be supported by an external collection method to create instances and external object methods to update or delete instances. Attributes of a proxy can be updated in the same fashion as normal JADE objects, by using the assignment operator.

MORE DETAILED SPECIFICATIONS WITH REFERENCE JADE OBJECT CONCEPTS AND LANGUAGE CONSTRUCTS

(1) THE DECLARATIVE MAPPING TOOL

Definition Requirements

Class Definition

A class can be defined from one or more base tables. If multiple tables are specified it is equivalent to a JOIN. If a class is defined as a JOIN of two or more tables its unique or primary key columns will be the combination of all the tables unique or primary key columns. By selecting only those attributes which are required for a class a table projection can be effected, ie the class can access a subset only of the columns for the table. This has resource and performance benefits for large tables and joins.

Each external class defined must include attributes (possibly hidden) that correspond to the unique or primary key columns of the tables it uses. This is required to give each object uniqueness. This is necessary for random updating and the collection method 'includes'.

Attribute Mapping methods are created for all attributes that have a base type of String or Binary and whose length is > 540 or have an unbounded length. This hence includes all SLOBS and BLOBS for which no space would be allocated in the object buffer. This allows the reading of large strings and binaries only when required.

Reference Mapping methods are created for all for single references. This method

calls getProxy with feature number of the reference property to allow the returning of the appropriate proxy.

Rules for collection candidate creation

Create a dictionary for each index of the class if all the columns of the index have corresponding user defined attributes ie not hidden. Create a dictionary based on the primary key of the class if all the columns of the primary key have corresponding user defined attributes ie not hidden. Create an array based on the primary key of the class if all the columns of the primary key have corresponding user defined attributes ie not hidden. Create a set for each class.

Rules for creating candidate relationships

Create a candidate for each primary key/foreign key pair between the tables on each side. Create a candidate for each matching pair of column names that have the same base type. Exclude any that have either end already involved in an actual or candidate relationship.

Relational to Object Oriented Heuristics

Cardinality

Use the cardinality of the respective tables to indicate the cardinality of the relationship between the corresponding classes.

Given two tables with a primary key/foreign key relationship defined between them :

cardinality of primary key table < cardinality of foreign key table => one to many relationship.

cardinality of primary key table >= cardinality of foreign key table => one to one relationship.

Relationship Type

Using the 'delete rule' of a foreign key we can infer parent-child relationships.

Cascade => parent-child relationship

Delete SetNull or SetDefault => optional => one to one relationship.

Restrict, Set To Null, Default => peer -peer relationship.

Is it possible to build a relationship graph from the primary key/foreign key definitions and table cardinality. As we define relationships this can be modified to better guess the cardinality, implementation and parent /child relationship of subsequent relationships.

If cardinality information is not available then a relationship is based on a primary key/foreign key pair is likely to be a one to many relationship. If no delete rule information is available assume a peer-peer relationships.

Other factors that could be used to infer relationship characteristics are connectivity, common substrings in names and by analysing the frequency counts of data in an actual database.

Example Transformation

Relational Schema:

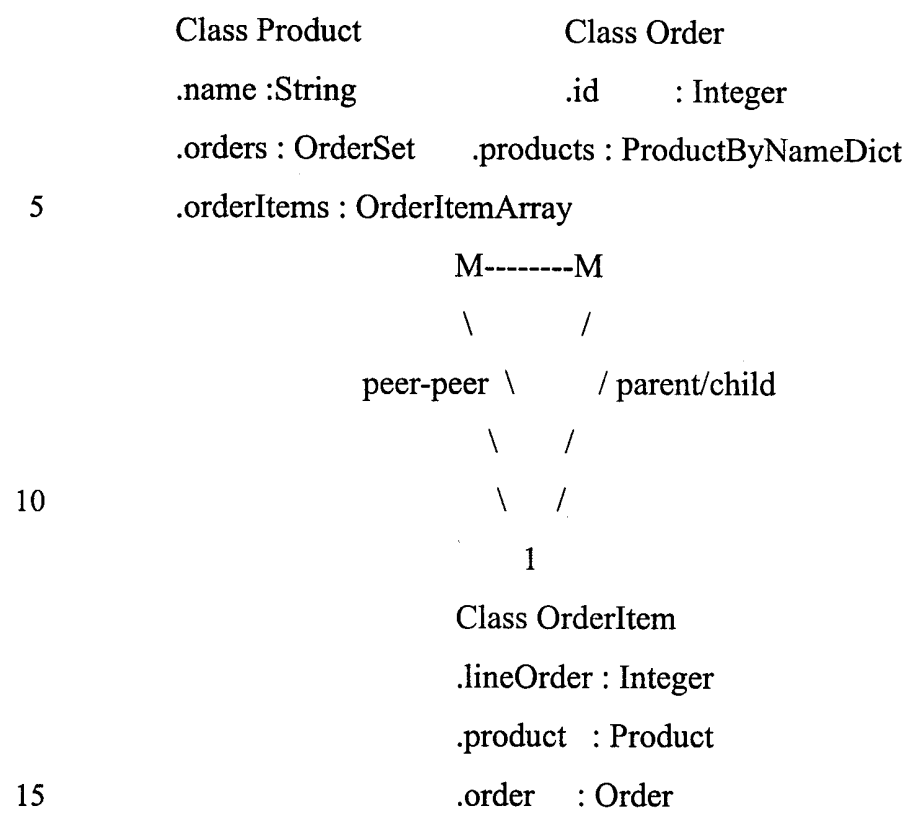
Table Product 1---- restrict ----M Table OrderItem M------cascade------1 Table Order

.name : String(pkey) .lineOrdinal : Integer .id : Integer (pkey)

.orderId (fkey for Order.id, cascade)

.productName (fkey for Product.name)

Jade Schema:



OrderSet = Set of Order

OrderItemArray = Array of OrderItem ordered by .lineOrder

ProductByNameDict = Dictionary of Product ordered by .name

M:M Relationships and Intermediate Tables

Intermediate tables are tables that are used to allow an M:M relationship to be implemented between two other tables. They can have additional columns beyond those participating in the relationship or they can consist solely of those columns.

These tables can be identified for a particular pair of tables by the following heuristics:

1. They have foreign keys to both pairs of tables.
2. They have existing 1:M or 1:1 relationships to both tables

M:M relationships are usually implemented as Sets.

Reference FromList SQL fragments are required to be appended

SQL Fragment Generation

SELECT List

Class Based

List of column identifiers corresponding to all the attributes of the class. This includes any hidden attributes.

FROM List

Class Based

List of table identifiers corresponding to all the tables included in the class

Reference Based

List of table identifiers corresponding to all the tables included in a reference query that are not class based ie. all intermediate tables in an M:M relationship.

WHERE Predicates

User specified Class Extent and Table Join

This fragment is optionally specified by the user.

Reference Join

The ExternalReference.joinPredicate and the ExternalReference.joinPredicateInfo properties. are used for reference querying using the late-binding technique.

Given a relationship defined as follows :

lhsClass	**rhsClass**
lhsColumn1 -	- rhsColumn1
lhsColumn2 -|----------|-	rhsColumn2
lhsColumn3 -	- rhsColumn3
... etc	

The joinPredicate consists of an equality term for each rhs column of the relationship and a '?' placeholder for bound parameters at runtime.
ie joinPredicate =(rhsColumn1 = ?) AND (rhsColumn2 = ?) AND... etc

The joinPredicateInfo consists of an integer for each '?' in the joinPredicate. Each integer is the column/attribute ordinal of the parameter to be bound at runtime. ie,. joinPredicateInfo = <column/attribute ordinal for lhsColumn1>, <column/attribute ordinal for lhsColumn1> ...etc

Collection getAtKey

Determines the keyEqPredicate and keyEqPredicateInfo required for a keyEq operation on an ExternalDictionary. If either of these properties is null then the keyEq operation is undefined.

The keyEqPredicate consists of an equality term for each key column of the collection. Each term consists of the columns identifier, an '=' sign and a '?' placeholder for bound parameters. All these terms are ANDed together and enclosed in braces. The keyEqPredicateInfo consists of an integer for each '?' in the keyEqPredicate. Each integer is the ordinal of the key to be bound as a parameter at runtime. This method must handle both single and multiple key collections.

Eg.
Given a dictionary with key1, key2 etc corresponding to column1, column2 etc
keyEqPredicate = (column1=?) AND (column2=?) AND ... etc
keyEqPredicateInfo = ordinal of key1, ordinal of key2, ... etc.

Collection geqKey,gtKey,leqKey,ltKey

Determines the key<CmpOp>Predicate and key<CmpOp>PredicateInfo required for a key<CmpOp> operation on an ExternalDictionary, where <CmpOp> can

be Geq, Gtr, Leq, Lss.

If either of these properties is null then the key<CmpOp> operation is undefined.

The key<CmpOp>Predicate consists of a comparison term for each key of the collection.

As keys have an implicit precedence each key consists of two factors ORed together, except for the last key. The first factor handles the major domain range for the key ie the '>' part of a '>=' comparison. The second factor ie the '=' part of a '>=' is ANDed with the first factor of the next key. If the next key is the last key then its term is used in its entirety. Each factor consists of a column identifier, a comparison operator and a '?' placeholder for bound parameters.

The key<CmpOp>PredicateInfo consists of an integer for each '?' in the key<CmpOp>Predicate. Each integer is the ordinal of the key to be bound as a parameter at runtime. This method must handle both single and multiple key collections.

Eg.

Given a dictionary with key1, key2 etc corresponding to column1, column2 the terms for geqKey are :

keyGeqPredicate = ((column1>?) OR ((column1 = ?) AND ((column2>?) OR(column2 = ?) AND ... etc.

keyGeqPredicateInfo = ordinal of key1, ordinal of key1, ordinal of key2, ordinal of key2 ... etc.

Collection includes

Determines and then sets the includesPredicate and includesPredicateInfo properties for this collection. If either of these properties is null then the includes method is undefined. The includesPredicate consists of an equality term for each unique column of the tables that are used by the collections member class. Each term consists of the columns identifier, an '=' sign and a '?' placeholder for bound parameters. All these terms are ANDed together and enclosed in braces. The

includesPredicateInfo consists of an integer for each '?' in the includesPredicate. Each integer is the column/attribute ordinal of the parameter to be bound at runtime. This method must handle both single and multiple table classes, and both single and multiple unique column tables.

Eg.

Given a collection with a member class whose table(s) have unique columns column1, column2 the terms for includes are :

includesPredicate = (column1=?) AND (column2=?) AND ... etc.

includesPredicateInfo = <column/attribute ordinal for column1>, <column/attribute ordinal for column2> ...etc

Collection first, last

Not currently used.

ORDER BY List

Collection Key

List of column identifiers and ASC/DESC specifiers corresponding to all the keys of a dictionary or the order attributes of an array.

User specified Class

This fragment is optionally specified by the user.

Column and Table Identifiers

Table identifies are generated from the table name, and any schema and catalog names specified for the table. These are combined according to the driver information for their usage, prefix/suffix usage and the driver quote character. Instead of using table identifiers directly, correlation names (also known as aliases) are used, unless the driver does not support this. Table name aliases are defined in the FROM clause and are used instead of table identifiers in SELECT,

and WHERE clauses. Table name aliases are generated that are unique across the entire External Database definition and not just for a particular query.

Column identifiers are a combination of a column's table identifier (or alias) and its name separated by a '.' character.

2. EXTERNAL DATABASE QUERY ENGINE

The relational Query engine has two functions as indicated in figures 1 and 3. The first is to implement the behaviour of external proxy classes and collections that provide access to external databases (see also figure 4). The second is to provide catalog Query functions, which import catalog information from an external database populating meta information objects used by the External Schema ESW.

Catalog import methods

The query engine will provide methods on the external schema entities used by the external schema Wizard to import catalog information from a relational schema. These methods will employ the relevant ODBC catalog functions. Some drivers/data sources may not support all the required functions, in which case the external schema Wizard have less information to use for producing a default mapping. The JADE query engine will always query driver capability to determine whether a required function is available before using it.

The following entities will be imported from relational catalogs and stored as persistent meta information in the users schema: Tables, Columns, Special Columns, Primary Keys, Foreign Keys, Indexes, Stored Procedures and parameters. The catalog import functions will either create new entities in the JADE schema or update existing definitions if they already exist. This is required to allow the External database wizard to provide schema upgrade capabilities.

Outputs from the Schema Transformation Process

The schema transformation process creates an object model, which can be employed by developers as an abstract O-O view of a relational database. The query engine will implement the required access methods to allow this abstract object model to work. In addition to the external class, database and collection definitions, the query engine will also depend on a number of stored SQL queries and mapping methods generated by the external database wizard.

Each external database is mapped to a subclass of the a class called ExternalDatabase, which has a singleton persistent instance. The external database class provides properties and methods relevant to the external data source and connection. At run-time a transient instance is created; this instance can be accessed in user logic to invoke database specific operations.

A meta schema class, ExternalClass, implements behaviour specific to external proxy classes. The class is used by the query engine when creating virtual proxy instances. In particular, each external class will contain the SQL query required to populate a 'class extent' or to do a join query for a single valued reference.

Another meta schema class, ExternalCollClass, implements behaviour specific to external collection classes. The class is used by the query engine to populate an external virtual collection.

Usage

The user will access the relational database using the OO language and by navigating relationships via inverse references, e.g. in JADE it will be possible to code the following style of iterative access:

foreach emp in dept.employees do
emp.display();
endforeach;

This is made possible using the collaboration of an 'external iterator' and an 'external collection'. The 'external collection' will issue the required SQL query to create a 'virtual collection' of instances, that is the rows of the employee table where employee.dept_id = department.dept_id. The dept_id foreign key will be obtained from the virtual department instance, which is the parent of the virtual employees dictionary. The external iterator class will implement the standard iterator protocol i.e. next() and back() methods etc. The external iterator/collection collaboration, will allow constructs such as foreach to function in a transparent fashion.

External collections

External collections will implement non-updating collection methods e.g. first, last, size etc. External dictionaries will provide direct key access to an 'external object instances' i.e. random access to a row or tuple in the relational database.

Example JADE:

department := E_Company.departments[name];

Relative key access will be provided using getAtKeyGtr, Geq, Leq and Lss style methods.

The external database wizard will generate stored SQL queries for the standard key search operations implemented at the external dictionary level. These will be in the form of parameterised SQL statements.

DHOC QUERY EXTENSIONS

'ADHOC queries' will be supported to some extent by allowing filtering (WHERE clause) and sort specifications (ORDER BY) to be explicitly set on external collections at runtime, overriding the 'stored query' associated with the collection. The filtering and sort specifications can be set using the external collection properties:

ExternalCollection::filterExpression
ExternalCollection::sortExpression

These properties may be set before a collection is used in a foreach statement or in conjunction with an iterator. The filterExpression allows the SQL WHERE clause associated with a collection to be specified or overridden and the sortExpression allows the SQL 'ORDER BY' clause to be specified or overridden. These expressions will be combined with the column list and table selection expressions to build a SQL query. If the resultant query is invalid, an exception will be raised.

One way of using this facility would be to create a shared external collection instance, set the filter and sort expressions and then use an iterator or foreach to fetch the instances. For example:

```
findRichCustomers();
        accounts : CustomerAccountSet;
        account : Account;
begin
        create accounts;
        accounts.filterExpression := "account.balance > 100000";
        accounts.sortExpression := "account.balance";
```

```
foreach account in accounts do
display(account.number, account.name);endforeach;
end;
```

It is more appropriate to use the filter and sort expressions with external sets, than with dictionaries and arrays.

Sequential access

External iterators implement a standard iterator protocol and may be used for iterating over any external collection. Standard JADE foreach syntax is fully supported including **reversed** and **where** clause.

The ExternalClass::instances property is implemented to provide access to the class extent in a similar fashion to the existing JADE Class::instances 'virtual collection' property. If the external class has a defined ORDER BY, then instances will be retrieved in this order, otherwise the order will be data-source dependent. A filter ('WHERE' clause) may be defined in the External database wizard to restrict the rows included class extent.

Schema mismatch exceptions

Special consideration will be given to detecting and handling changes to the relational schema that may become inconsistent with an existing schema mapping. In some cases, changes are handled transparently. For example: when a column is added or a column type changed and a valid type mapping still exists, then the query engine will create an external proxy, mapping the actual columns to the attributes as defined in JADE. Certain possible changes to the relational schema will render mappings and/or queries invalid, for example when a table that is mapped to an external class is deleted. When a mapping or query that is no longer valid is referenced, then an appropriate ODBC exception will be raised. When the schema mapping gets too far out of date to be useable then developers must return to the schema External database wizard in order to update the mapping

information.

Updating external databases

Two styles of updating are supported, which in relational terminology are referred to as:

- Non-cursor updates
- Cursor-based updates

Non-cursor Updates

The ExternalDatabase class will provide transaction support using beginExternalTransaction, commitExternalTransaction and abortExternalTransaction methods. NonCursor or searched updates will be possible using the ExternalDatabases::executeSQL method.

Cursor-based updates

Positioned or cursor-based updates will be supported by an external collection method to create a new instance (insert a row) and external object methods to delete or apply updates to an external object.

ExternalCollection::createObject
ExternalObject::deleteSelf
ExternalObject::update

Attributes of a proxy can be updated in the same fashion as normal JADE objects, by using the assignment operator.

Interoperability

To allow a single application to operate with many different DBMSs difference ODBC drivers are use (see figure 4) This is because the use of the ODBC standard does not itself enure interoperability. The ODBC standard is very broad and one of its goals is to allow a large number of DBMSs to expose as wide a

feature set as possible. Drivers are provided to work with a single DBMS and, by definition, are not interoperable. They play a role in interoperability by correctly implementing and exposing ODBC over a single DBMS.

The query engine will aim to be capability driven as opposed to employing driver or data source specific code as far as possible. The query engine will interrogate connected ODBC drivers and make use of certain features when available. Examples include catalog functions, block mode and scrollable cursors. In other cases the way we use the driver has to be tailored based on the requirements of the driver and the way the data source behaves.

SUPPORTING CLASSES AND METHODS

getExternalDatabase method

Application::getExternalDatabase

getExternalDatabase(dbName : String) : ExternalDatabase;

The getExternalDatabase method returns a reference to the shared transient instance of the external database identified by dbName.

ExternalDatabase Class

The ExternalDatabase class represents a connection to an external database and provides methods that operate on the data source.

ClassExternalDatabase public, abstract

Inherits From: Object

Properties

Property	Description
connectionString	Contains parameters required to connect to a data source
serverName	The name of the server as defined for the data source
userName	Contains a user-id to be used to establish a connection
password	Contains password is required by the data source

ExternalDatabase::connectionString

connectionString : String

This string should contain any parameters required for connecting to a data source. The parameters are generally driver/data source specific. A default connection string can be obtained automatically when connecting to a data-source using the External database wizard browse facility. However, the default string can be overridden at runtime on a per user or connection basis by setting this attribute. The UID and PWD parameters should not be included in this string.

xternalDatabase::serverName

serverName : String

This attribute contains the name of the database server if this is defined for the data source.

ExternalDatabase::userName

userName : String

This attribute should contain the name of a valid user id, used for authentication at the data source. A default user-id is established at design

time using the External database wizard. However, this of course may be changed at run-time on a per user basis, prior to opening a database connection.

ExternalDatabase::password

password : String

The password attribute is used in conjunction with userName for authentication, if required, at the data source. A default password may be stored on the database object, if the schema translator has allowed this. In any case, this may be changed at run-time on a per user basis, prior to opening a database connection.

Methods

Method	Description
open	Open a connection to an external database
close	Close the connection to an external database
beginExternalTransaction	Start a database transaction
commitExternalTransaction	Commit a transaction
abortExternalTransaction	Rollback the changes made during the current transaction
executeSQL	Execute an SQL statement directly
canTransact	Returns true if database supports transactions
isUpdateable	Returns true if database is updateable
isSQLValid	Check the syntax of an SQL statement

ExternalDatabase::open

open();

This method establishes a connection to the external database. The connectionString, userName and password(if this is required by the data source) must be set before opening a connection. Default values for these are

established by the External database wizard.

ExternalDatabase::close

close();

This method closes any open connection to the external database.

ExternalDatabase::canTransact

canTransact() : Boolean;

Call this method to determine whether the connected database supports transactions.

ExternalDatabase::isUpdateable

isUpdateable() : Boolean;

Call this method to determine whether the connected database allows updates. Not all drivers support updating, an example would be the JADE ODBC driver.

ExternalDatabase::beginExternalTransaction

beginExternalTransaction();

If the external database supports transactions, this method should be called at the start of a series of updating operations: creates, deletes and updates that must be applied atomically to the target database for consistency and recoverability reasons. By default, updates are committed immediately; calling beginExternalTransaction delays the commitment of updates until commitExternalTransaction is called. If the external database doesn't support transactions(use the canTransact method to determine this), then calling this method will have no effect.

ExternalDatabase::commitExternalTransaction

commitExternalTransaction();

If the external database supports transactions, this method should be called at the end of a series of updating operations: creates, deletes and updates to commit or apply the changes to the external database. If the external database doesn't support transactions(use the canTransact method to determine this), then calling this method will have no effect.

ExternalDatabase::abortExternalTransaction

abortExternalTransaction();

If the external database supports transactions, this method can be used to undo the effects of a transaction. All updating operations: creates, deletes and updates made since the last beginExternalTransaction call are reversed to the state that existed at the time of that call. If the external database doesn't support transactions(use the canTransact method to determine this), then calling this method will have no effect.

ExternalDatabase::executeSQL

executeSQL(sql : String);

Call this method to execute an SQL command directly. The SQL statement is passed via the sql string parameter. This method does not return any data so it is not suitable for data retrieval operations. It is most suited to performing searched updates or calling stored procedures, which don't return a result set. If the SQL statement is invalid, an ODBC exception containing driver and/or data source diagnostics will be raised.

ExternalDatabase::checkSQL

isSQLValid(sql : String) : Boolean;

Call this method to check that the syntax of an SQL statement is both valid and supported by the driver and data source. The method returns true if the syntax is valid and supported, otherwise it returns false. No exception is raised if the syntax is not acceptable.

ExternalClass Class

Method	Description
IsUpdatable	Returns true if instances are updateable (i.e. can create, delete or update instances)

ExternalClass::isUpdatable

isUpdateable() : Boolean;

Call this method to determine whether instances of the external class can be updated This method will return false if either the data-source is read-only or the class is read-only. An external class will be read-only if it is based on a relational view or join query defined by the External database wizard.

ExternalObject Class

Method	Description
DeleteSelf	Deletes the external object
Update	Completes a create or update operation, saving changes to the external database

External Collection Classes

The external collection classes represent the 'result set' of a selection from an external data source. The external collections are virtual in the sense that member instances don't actually exist until they are first referenced. When a member is first referenced by a direct key access or via iteration an external proxy instance is created which represents the corresponding row in the result set. External collections provide the operations to support direct and relative key access and may be used in collaboration with external iterators to access

rows in a result set sequentially.

There is a corresponding external collection type for each of the existing JADE collection types as shown by the following table:

Collection type	Semantics
ExternalCollection	Abstract superclass of all external collections
ExternalArray	An ordered collection - requires an 'ORDER BY' clause, optional filtering expression (WHERE clause)
ExternalDictionary	Requires keys, requires an 'ORDERED BY' on keys and optionally a filtering specification (WHERE clause)
ExternalSet	An unordered collection, no 'ORDER BY', optional filtering expression (WHERE clause)

ExternalCollection Class

The ExternalCollection class inherits the interface of the Collection class and most non-updating methods will be implemented and supported. External collections are in themselves read-only. Operations such as add, remove, clear and purge are not supported. The compiler will prevent the use of updating methods for external collections in the same way that they are prevented for automatic collections participating in an inverse relationship.

Class ExternalCollection public, abstract

Inherits From: Collection

Properties

Property	Description
filterExpression	Used as a filter to select specific rows
sortExpression	Used to control how instances are ordered in the collection

ExternalCollection::filterExpression

filterExpression : String;

This attribute may be specified to override the default filtering or WHERE predicate defined for an external collection. This may be useful for selecting a subset of records at runtime. The filtering expression should not contain the WHERE keyword. The filtering expression must be defined in terms of external column names, not the attribute names to which they are mapped. If the resultant SQL statement is not valid then an ODBC exception will be raised.

ExternalCollection::sortExpression

sortExpression : String;

This attribute may be specified to override the default sort or SQL 'ODER BY' specification defined for an external collection. The sort expression should not contain the 'ORDER BY' SQL keywords. The filtering expression must be defined in terms of external column names, not the attribute names to which they are mapped. If the resultant SQL statement is not valid then an ODBC exception will be raised.

Methods

The methods listed in the following table are defined for the ExternalCollection class

Method	Description
createIterator	Creates an external iterator for an external collection
first	Returns the first entry in the collection
getOwner	Returns the parent or owner of the collection
includes	Returns **true** if the external collection contains a specified object
inspect	Provides an inspector for external instances
last	Returns the last entry in the collection
canCreate	Returns true if member type instances may be created

createObject	Create a new external proxy instance
size	Returns the current number of entries in the collection

ExternalCollection::createIterator

createIterator(): Iterator;

The **createIterator** method is used to create an external iterator for use with external collections.

ExternalCollection::first

first(): MemberType;

The **first** method returns a proxy reference representing the first entry in the virtual collection. For ordered collections such as dictionaries and arrays the proxy will represent the first row selected as determined by the 'ORDER BY' clause.

ExternalCollection::getOwner

getOwner(): Object;

The **getOwner** method returns the owner or parent of the external collection

ExternalCollection::includes

includes(value: MemberType): Boolean;

The **includes** method returns **true** if the virtual collection or result set contains the specified object. This method will result in a key equal query based on the attributes which comprise the primary keys of the proxy.

ExternalCollection::last

last(): MemberType;

The **last** method returns a proxy reference representing the last entry in the virtual collection. For ordered collections such as dictionaries and arrays the proxy will represent the last row selected as determined by the 'ORDER BY' clause. The SQL query used to retrieve the last record will be constructed to give the optimiser the opportunity to perform a singleton select. In cases where an index exists on attributes in the ORDER BY a sort should also be avoided.

ExternalCollection::size

size(): Integer;

The **size** method returns the number of entries in the virtual collection. This method will result in an SQL query that will actually count the rows in the selected tables mapped to the proxy class. This method should therefore be used with caution if there are expected to large number of rows in the result-set.

ExternalCollection::canCreate

canCreate() : Boolean;

Call this method to determine whether instances of the member-type of the collection can be created. This method will return false if either the data-source is read-only or the class of the collection members is read-only. An external class will be read-only if it is based on a relational view or join query defined by the External database wizard.

ExternalArray Class

An external array is an ordered virtual collection which represents the rows in a result set generated from an SQL query containing a sort specification i.e. an 'ORDER BY' clause. The member instances occur in the order determined by the 'ORDER BY'

Subscripting External Arrays

The bracket [ ] subscript operators may be used to access rows at a specified position in the result-set, for example to access the nth transaction in an external array called transactions, you may do the following:

transaction := transactions[n];

The [] notation is a syntax shortcut for the ExternalArray::at method

Class ExternalArray public, transient

Inherits From: ExternalCollection

The methods provided by the **ExternalArray** class are listed in the following table.

Method	Description
at	Returns the entry at a specified index in the array

ExternalArray::at

at(index: Integer): MemberType;

The **at** method returns the entry in the virtual array at the position specified by the **index** parameter. This corresponds to accessing the nth row in the result-set.

If there is no row at the specified index in the result set, an exception will be raised.

ExternalSet Class

An external set is an unordered virtual collection which represents the rows in a result set generated from an SQL query which has no sort specification (ORDER BY). The order in which instances are fetched is data-source dependent.

Class ExternalSet public, transient

Inherits From: ExternalCollection

The methods provided by the **ExternalSet** class are listed in the following table.

Method	Description
includes	Returns **true** if the specified object occurs in the set

Methods

ExternalSet::includes

includes(key: MemberType): Boolean;

The **includes** method returns **true** if the virtual collection or result set contains the specified object. This method will result in a key equal query based on the attributes which comprise the primary keys of the proxy.

ExternalDictionary Class

An external dictionary is an ordered virtual collection which represents the rows in a result set generated from an SQL query with a sort specification (ORDER BY). The 'ORDER BY' specification is generated by the External database wizard and represents the order specifications defined for the member-key attribute values.

Associative key access using subscript notation

The bracket [ ] subscript notation can be used as a shortcut for the getAtKey method which supports random access with a key equal search condition. For example:

customer := customers[name];

Class ExternalDictionary public, transient

Inherits From: ExternalCollection

The methods provided by the **ExternalDictionary** class are listed in the following table.

Method	Description
getAtKey	Returns the object at the specified key
getAtKeyGeq	Returns the object with a key greater or equal to the specified key
getAtKeyGtr	Returns the object with a key greater than the specified key
getAtKeyLeq	Returns an object with a key less than or equal to the specified key
getAtKeyLss	Returns an object with key less than the specified key
includesKey	Returns **true** if the receiver contains an entry at the specified key
startKeyGeq	Sets a start position within a collection for an **iterator** object
startKeyGtr	Sets a start position within a collection for an **iterator** object at the next object after the specified key
startKeyLeq	Sets a start position within a collection for an **iterator** object at the object equal to or before the specified key
startKeyLss	Sets a start position within a collection for an **iterator** object at the object before the specified key

Methods

ExternalDictionary::getAtKey

getAtKey(keys: KeyType): MemberType;

The getAtKey method issues a singleton SQL select which searches for an exact match between the member-key attribute values of virtual instances

(rows) and the corresponding key parameters. If a row is selected, then a proxy reference representing the row is returned; otherwise, the method returns null.

ExternalDictionary::getAtKeyGeq

getAtKeyGeq(keys: KeyType): MemberType;

The getAtKeyGeq method issues a singleton SQL select that searches for a 'greater than or equal' key match between the member-key attribute values of virtual instances (rows) and the corresponding key parameters. If a row is selected, then a proxy reference representing the row is returned; otherwise, the method returns null.

ExternalDictionary::getAtKeyGtr

getAtKeyGtr(keys: KeyType): MemberType;

The getAtKeyGtr method issues a singleton SQL select that searches for a greater than key match between the member-key attribute values of virtual instances (rows) and the corresponding key parameters. If a row is selected then a proxy reference representing the row is returned; otherwise, the method returns null.

ExternalDictionary::getAtKeyLeq

getAtKeyLeq(keys: KeyType): MemberType;

The getAtKeyLeq method issues a singleton SQL select that searches for a 'less than or equal' key match between the member-key attribute values of virtual instances (rows) and the corresponding key parameters. If a row is selected then a proxy reference representing the row is returned; otherwise, the method returns null.

ExternalDictionary::getAtKeyLss

getAtKeyLss(keys: KeyType): MemberType;

The getAtKeyLss method issues a singleton SQL select that searches for a 'less than' key match between the member-key attribute values of virtual instances (rows) and the corresponding key parameters. If a row is selected then a proxy reference representing the row is returned; otherwise, the method returns null.

ExternalDictionary::includesKey

includesKey(key: KeyType): Boolean;

The includesKey method issues a singleton SQL select which searches for an exact match between the member-key attribute values of virtual instances (rows) and the corresponding key parameters. The method return true if a row is selected, otherwise, it returns false.

ExternalDictionary::startKeyGeq

startKeyGeq(keys: KeyType; iter: ExternalIterator);

The **startKeyGeq** method sets a start position within a collection for an **iterator** object.

ExternalDictionary::startKeyGtr

startKeyGtr(keys: KeyType; iter: ExternalIterator);

The **startKeyGtr** method sets a start position within a collection for an **iterator** at the next object after the key specified in the **Keys** parameter

ExternalDictionary::startKeyLeq

startKeyLeq(keys: KeyType; iter: ExternalIterator);

The **startKeyLeq** method sets a start position within a collection for an **iterator** object at the object equal to or before the key specified in the **keys** parameter.

ExternalDictionary::startKeyLss

startKeyLss(keys: KeyType; iter: ExternalIterator);

The **startKeyLss** method sets a start position within a collection for an **iterator** object at the object before the key specified in the **keys** parameter.

ExternalIterator Class

The **ExternalIterator** class is used in collaboration with an external collection. An external iterator instance is used to access the virtual instances of the collection sequentially, either forwards or in a reverse direction. In reality, external iterators will provide the operations to scroll an SQL cursor associated with the result set of the query, which was used to populate the external collection.

Class ExternalIterator public, real, transient

Inherits From: Iterator

The methods provided by the **ExternalIterator** class are listed in the following table.

Method	Description
back	Accesses entries in reverse order in the collection to which the iteration is attached
getCollection	Returns the collection associated with the receiver
next	Accesses successive entries in the collection to which the iteration is attached
reset	Initializes an iterator
startAtIndex	Sets the starting position of the iterator to a relative index in the result-set of the external collection query

Methods

ExternalIterator::back

back(value: ExternalObject output) : Boolean updating;

The **back** method is used to scroll backwards through an SQL result-set and return a proxy-reference representing each row as the cursor is scrolled.

ExternalIterator::getCollection

getCollection() : ExternalCollection;

The **getCollection** method returns the external collection currently associated with the receiver. A **null** value is returned if no collection is associated with the receiver.

ExternalIterator::next

next(value: ExternalObject output): Boolean updating;

The **back** method is used to scroll forwards through an SQL result-set and return a proxy-reference representing each row as the cursor is scrolled.

ExternalIterator::reset

reset() updating;

The **reset** method resets the state of an iterator. After an iterator has been reset, the first next or back operation will cause the default SQL query associated with the attached external collection to be re-issued.

ExternalIterator::startAtIndex

startAtIndex(index: Integer) updating;

The **startAtIndex** method is used to position a cursor at a specified row in the result-set.

The required position is specified in the **index** parameter.

ODBCException Class

Class ODBCException public, real, transient

Inherits From: NormalException

Properties

Property	Description
NativeError	Native data-source specific error code
State	Five-character ODBC defined state variable

ODBCException::nativeError

nativeError : Integer;

This attribute will contain a native error code, specific to the data source. A description of the meaning of this should be available in the documentation for the data source.

ODBCException::state

state : String[5];

This attribute will contain the five-character ODBC state code returned by the ODBC driver. The first two characters indicate the class of the error; the next three indicate the subclass.

CLAIMS

1. A method of mapping a relational database schema to an object oriented schema to allow co-existence of said relational database with an object database comprising the steps of:

(a) extracting the schema data from said relational database, including database tables, columns, special columns, primary keys, foreign keys, indexes and stored procedures and parameters,

(b) enhancing said relational database schema by

(i) mapping tables to classes and columns to attributes,

(ii) mapping columns to attributes,

(iii) defining classes of set, array and dictionary collections based on primary key and index data,

(iv) establishing relationships from said primary keys, foreign keys and columns having the same names,

(v) defining relationship implementation based on heuristics by using the cardinality of said tables to define the cardinality of the relationship between the corresponding classes,

(c) generating structural query language (SQL) code fragments to support collection operations from the information derived in step (b) and storing same for run-time access to said relational database, and

(d) storing a description of the classes and collections established in step (b) for use in run-time by an object oriented application whenever access to data in said relational database is required.

2. A method according to claim 1 wherein statistical information is extracted from the relational database and is used in step (b)(v) as a determinant in defining relationship implementation.

3. A database query engine for an object oriented application which enables access to data in one or more external non-object oriented databases in the same manner as an internal integrated object oriented database to provide data location transparency, said engine comprising:

an external class which determines how a proxy object is populated by instances from the corresponding external database,

sub classes which re-implement methods to populate external class instances from said external databases while maintaining an identical interface to internal object classes,

a mechanism which produces a representation of the external database schema, and an object identifier to unique key mapping between an external proxy object instance and a row in the external database,

said engine carrying out the following processes:

assembling SQL statements as determined by the definition of the appropriate external class,

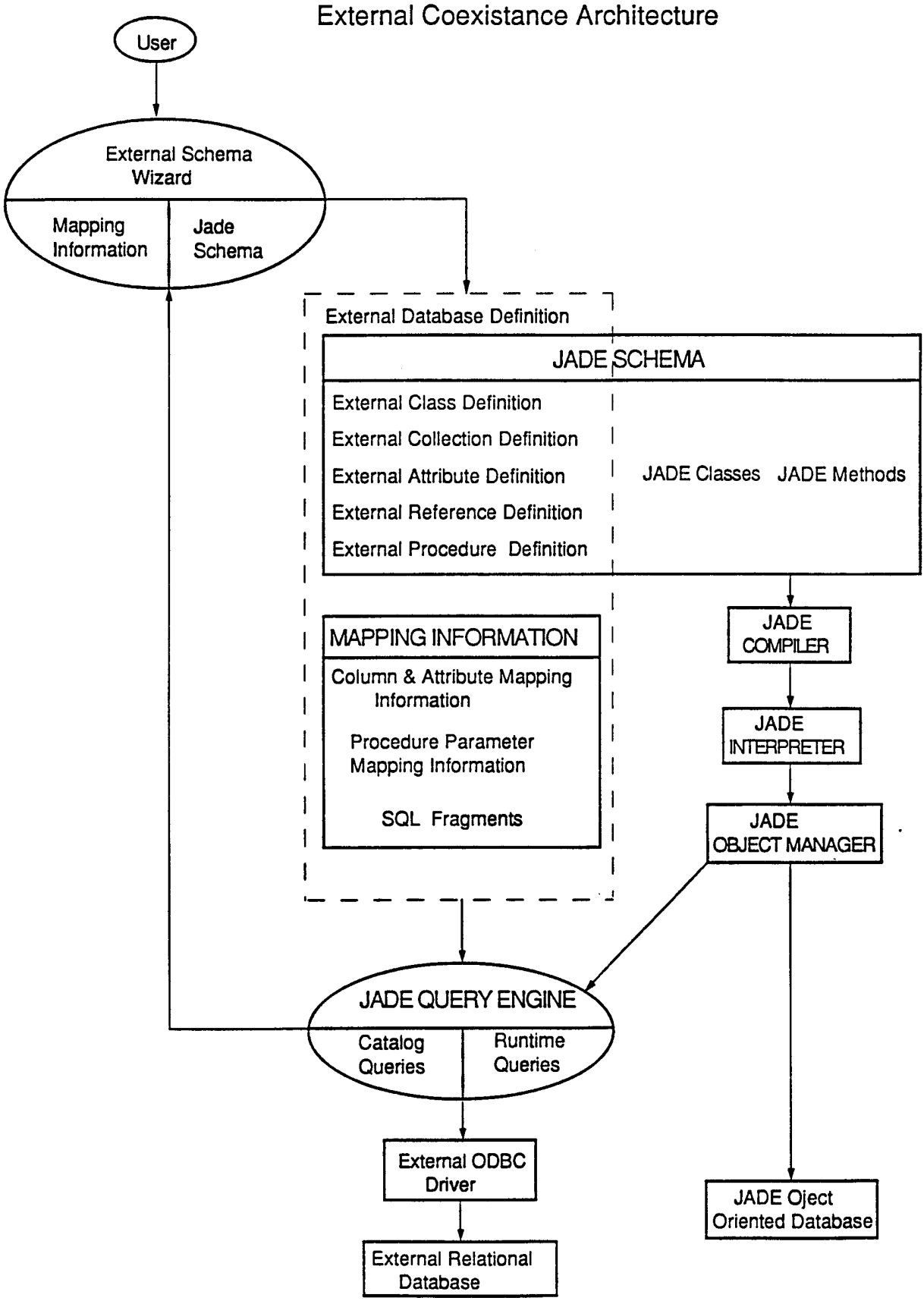
appending a JOIN query predicate when accessing a property which is an end point of a bi-directional relationship,

binding parameter values obtained from the attribute values of an external proxy instance to parameter markers in said SQL statements, binding parameter values obtained from the operation request which represent dictionary keys,

binding parameter values obtained from the attribute values of the external proxy instance to parameter markers specified in the JOIN predicate of the query, and

executes the resultant SQL query against the respective non-object oriented database.

FIG. 1


External Coexistance Architecture
User
External Schema Wizard
Mapping Information
Jade Schema
External Database Definition
JADE SCHEMA
External Class Definition
External Collection Definition
External Attribute Definition
External Reference Definition
External Procedure Definition
JADE Classes
JADE Methods
MAPPING INFORMATION
Column & Attribute Mapping Information
Procedure Parameter Mapping Information
SQL Fragments
JADE COMPILER
JADE INTERPRETER
JADE OBJECT MANAGER
JADE QUERY ENGINE
Catalog Queries
Runtime Queries
External ODBC Driver
External Relational Database
JADE Oject Oriented Database

External Schema Wizard
Mapping Technology

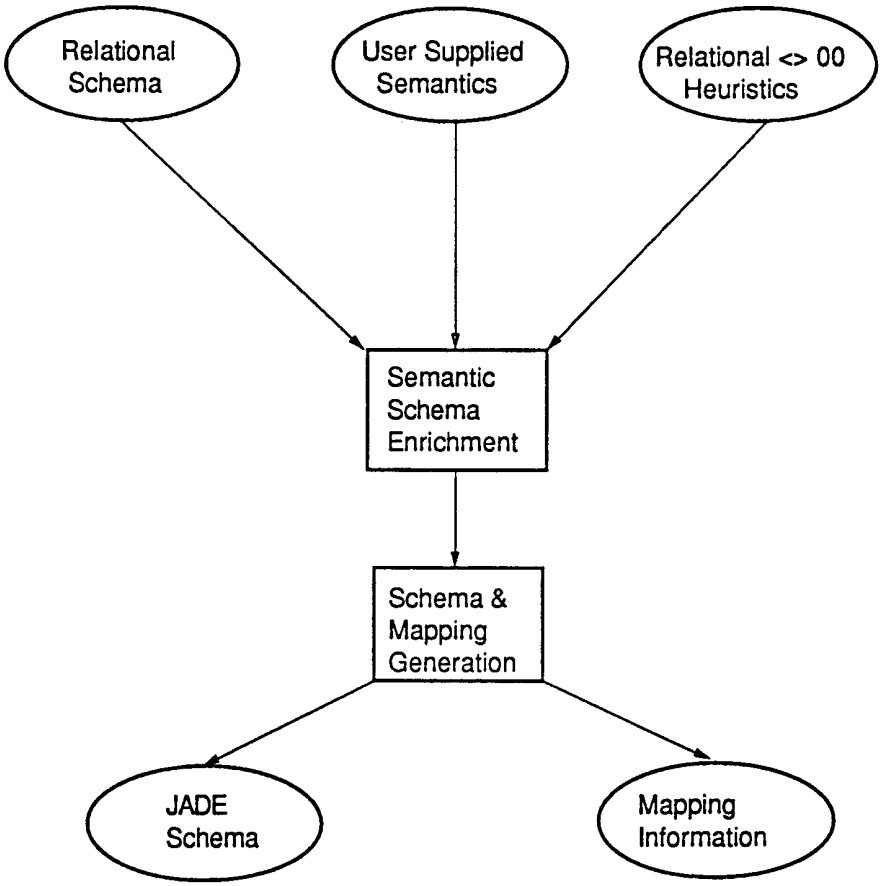


FIG. 2

External Query Engine Architecture

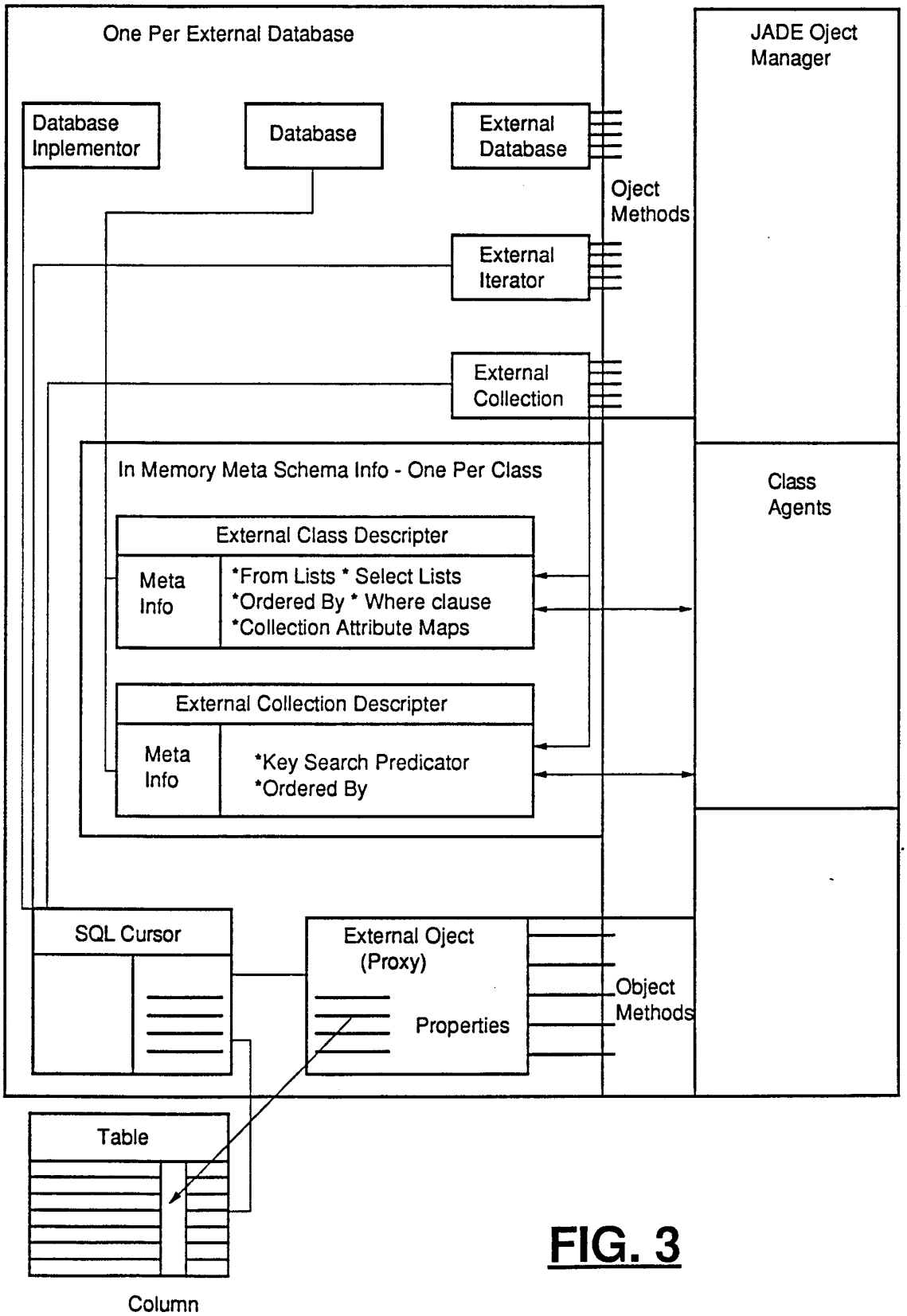


FIG. 3

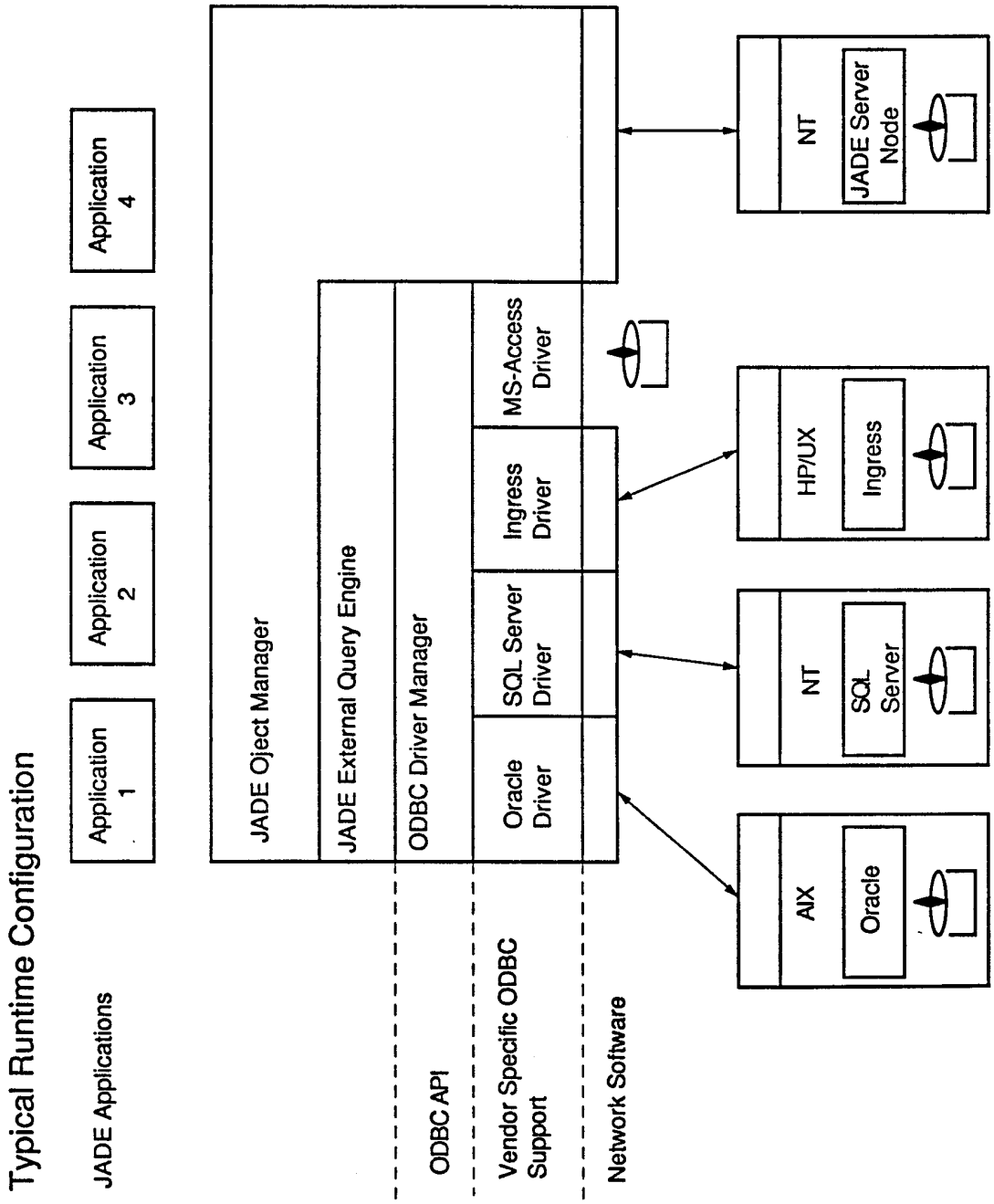

Typical Runtime Configuration
JADE Applications
Application 1
Application 2
Application 3
Application 4
JADE Oject Manager
JADE External Query Engine
ODBC Driver Manager
Oracle Driver
SQL Server Driver
Ingress Driver
MS-Access Driver
ODBC API
Vendor Specific ODBC Support
Network Software
AIX
Oracle
NT
SQL Server
HP/UX
Ingress
NT
JADE Server Node

FIG. 4

INTERNATIONAL SEARCH REPORT

International application No.
PCT/NZ 98/00128

A. CLASSIFICATION OF SUBJECT MATTER

Int Cl[6]: G06F 17/30

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC[6] - G06F 17/30 , IPC[5] - G06F 15/40

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
WPAT - OBJECT ( ) ORIENT: AND RELAT:

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document. with indication. where appropriate, of the relevant passages	Relevant to claim No.
X	WO 96/10232 (ONTOS) 4 April 1996 See whole document	1-3
X	EP 0504085 (INTERNATIONAL BUSINESS MACHINES CORPORATION) 16 September 1992 See whole document	1-3
X	WO 95/03586 (PERSISTANCE SOFTWARE) 2 February 1995 See whole document	1-3

[X] Further documents are listed in the continuation of Box C

[X] See patent family annex

* Special categories of cited documents:

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier application or patent but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
30 October 1998	10 NOV 1998
Name and mailing address of the ISA/AU AUSTRALIAN PATENT OFFICE PO BOX 200 WODEN ACT 2606 AUSTRALIA Facsimile No.: (02) 6285 3929	Authorized officer **J.W. THOMSON** Telephone No.: (02) 6283

INTERNATIONAL SEARCH REPORT

International application No.
PCT/NZ 98/00128

C (Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	US 5504885 (ALASHQUR) 2 April 1996 See whole document	1-3
P, X	US 5765162 (BLACKMAN et al.) 9 June 1998 See whole document	3

INTERNATIONAL SEARCH REPORT
Information on patent family members

International application No.
PCT/NZ 98/00128

This Annex lists the known "A" publication level patent family members relating to the patent documents cited in the above-mentioned international search report. The Australian Patent Office is in no way liable for these particulars which are merely given for the purpose of information.

Patent Document Cited in Search Report	Patent Family Member	
EP 504085	US 5212787	
WO 9503586	US 5499371	
WO 9610232	EP 783738	US 5542078

END OF ANNEX