

Singapore Management University
Institutional Knowledge at Singapore Management University

Research Collection School Of Information Systems

School of Information Systems

12-1996


Export Database Derivation Approach for supporting Object-Oriented wrapper queries

Ee Peng LIM

Singapore Management University, eplim@smu.edu.sg

Hon-Kuan LEE

Follow this and additional works at: https://ink.library.smu.edu.sg/sis_research

 Part of the [Databases and Information Systems Commons](#), and the [Numerical Analysis and Scientific Computing Commons](#)

Citation

LIM, Ee Peng and LEE, Hon-Kuan. Export Database Derivation Approach for supporting Object-Oriented wrapper queries. (1996). *Cooperative databases and applications: Proceedings of the International Symposium on Cooperative Database Systems for Advanced Applications, Kyoto, Japan, December 5-7, 1996*. 337-346. Research Collection School Of Information Systems.

Available at: https://ink.library.smu.edu.sg/sis_research/1024

This Conference Proceeding Article is brought to you for free and open access by the School of Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email libIR@smu.edu.sg.

Export Database Derivation and Query Processing for Object-Oriented Wrappers

EE-PENG LIM HON-KUAN LEE*

School of Applied Science
Nanyang Technological University
Nanyang Avenue, Singapore 639798, SINGAPORE

Abstract

*Wrappers export the schema and data of existing heterogeneous databases and support queries on them. In the context of cooperative information systems, we present a flexible approach to specify the derivation of object-oriented export databases from local relational databases. Our export database derivation consists of a set of **extent derivation structures** which defines the extent and deep extent of export classes. Having well-defined semantics, the extent derivation structures can be readily used in transforming wrapper queries to local queries. Based on the extent derivation structures, we developed a wrapper query evaluation strategy which handles object-oriented queries on the export databases. The strategy further considers the limited query processing capabilities of local database systems and the language constraints on the local query languages.*

1 Introduction

Traditionally, a wrapper has been defined to be a software component that converts data and queries from one model to another. However, in order to deploy wrappers in a cooperative information systems[6, 1] that provide integrated access to multiple existing heterogeneous databases (also known as local databases), it's functionalities have to be further extended. In the following, we enumerate the functionalities that are required by a wrapper:

- **Exporting and homogenizing the schemas and data of existing databases (DBs):**

Wrappers export schemas and data of existing DBs to the global users or applications of a cooperative information system. The set of schema and data exported from a local DB is called an **export DB**. Unlike the traditional view definition, the export DBs may be designed with an intention to ease the resolution of inter-database schema- or instance-level conflicts. An export DB may contain or incorporate extra semantics that are not found in the local DB[4].

- **Encapsulating the data model and query language differences among local database**

systems:

Apart from exporting local DBs, wrappers have to process queries on the export DBs. Queries on the export DBs are also known as **wrapper queries**. By adopting a common data model and query language to represent and query export DBs respectively, wrappers hide the differences of local database systems from the cooperative information systems and facilitate the development of global applications which query the export DBs.

- **Controlling the subset of local DBs accessible by the global users:**

Another important reason for having wrappers is to allow local DBAs to control the subsets of local DBs (both in terms of schema elements and instances) accessible by the cooperative information system users through the definition of export DBs. This also prevents some classified local information from being read by unauthorized people.

In our work, we have designed a flexible DB mapping approach which allows local DBAs to derive an OO export DB from a local relational DB declaratively and graphically using a computational structure known as **extent derivation structure(EDS)**. In this approach, basic OO semantics, such as *class inheritance*, *aggregation relationship*, and *object identifiers*, can be readily supported. We have also developed the wrapper query evaluation strategy which includes some translation algorithms to transform wrapper queries into local DB queries. We note that it is not always possible to translate a wrapper query into a single local DB query. This is due to some local query language constraints, such as: (a) some operations required by wrapper query are not supported by the local DBMS; (b) the wrapper query involves input data not found in the local DB; or (c) the local query language grammar imposes some constraints on the translation. For example, a SQL statement may not support selection or projection over a union of tables.

Related Work

In [4], the concept of **metaclass** has been introduced to integrate relational DBs into a federated DB system based on an OO data model known as VO-DAK. This approach requires methods to be defined

*The author is now with the Port of Singapore Authority.

to extract the attributes of relations into the VODAK data model as properties of export classes. Our work, on the other hand, has further studied the derivation of class inheritance from relational databases and the evaluation of wrapper queries.

The Penguin project at Stanford University[9] addresses the problem of storing data as relations but retrieving them using an OO query interface. There is a subtle difference between this problem and designing OO query interface to *existing* relational DBs. While the former designs relations to support OO views, the latter assumes that relational DBs and their applications have existed for some time and the OO views must be supported without any modification to them. Hence, Penguin adopts the *top-down* approach in designing the OO views instead of the *bottom-up* approach which is more appropriate in our context.

2 Export DB Definition versus Derivation

To satisfy a wide variety of interoperability requirements, every local to export DB mapping strategy must allow different OO export DBs to be defined for the same local relational DB. The reason is that the same local DB may have to participate in different cooperative information systems that have different export DB requirement. Moreover, we need to distinguish between the **definition** of export DBs from their **derivation**. The former describes the schema of export DBs. The latter describes the mapping between local DBs and export DBs. In this paper, we shall only focus on export DB derivation.

3 Mapping between Local Relational DB and OO Export DB

3.1 Example of Local DB and Export DBs

To demonstrate the export DB derivation process, the following company database is used as a local relational DB example.

Example: (Company DB Example)¹

```
Employee(eno, name, etype, salary, dob, sales, dno)
LocalDept(dno, dname, mgr)
OverseasDept(dno, dname, mgr)
OfficeAssign(rno, dno, floor)
Project(pno, ptitle)
ProjBudget(pno, budget)
EmpProj(eno, pno)
```

In the **Employee** table, each employee record is identified by the employee number (**eno**), and it contains attributes such as name (**name**), type of employment (**etype**), i.e. part-time or full-time, salary, date of birth (**dob**), sales amount (**sales**) and the department number (**dno**) in which the employee works. While part-time employees are paid daily, full-time employees are paid monthly. Hence, the **salary** attribute has been overloaded by two different meanings. For administrative purposes, department information has been stored in two tables, **LocalDept**

¹The key of each table has been underlined.

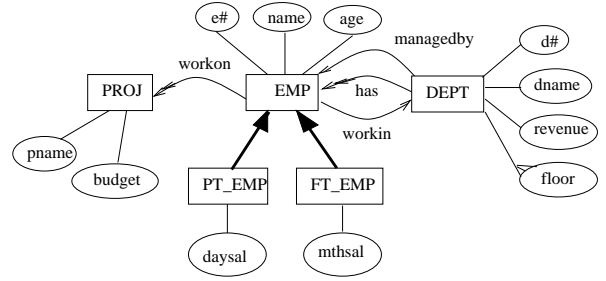


Figure 1: Example Export Schema

and **OverseasDept**. For the local department offices, **OfficeAssign** contains the office room records which contain the room numbers (**rno**), departments occupying the office, and the floors on which the offices are located. The project title and budget informations have been stored in separate relations. **Project** table contains project number (**pno**) and project title (**ptitle**) whereas **ProjBudget** table contains the budget for each project. **EmpProj** contains the employee-to-project assignment.

We assume that the local DBA, after having used some DB re-engineering tool and negotiated with the global users, has decided to export the local database using the OO schema given in Figure 1. In the export schema, the **PT_EMP** and **FT_EMP** classes have been defined to differentiate between part-time and full-time employee records, respectively. The **age** attribute does not exist in the local database but can be computed from the date of birth **dob**. The **revenue** of a **DEPT** object is defined to be the sum of sales made by employees in the department. Since each department may be allocated several offices located on different floors, **floor** is a set attribute. The **PROJECT** class contains **pname** and **budget** as attributes. Notice that the project number (**pno**), the key of **Project** local table, has been excluded from **PROJECT**. Though not shown in the export schema, an object id is implicit in every object class, and can be queried.

3.2 Extent Derivation Structures and their Algebraic Semantics

Extent derivation structure (EDS) defines how the extent and deep extent of an export class can be derived separately from a set of local relations. We define the **extent** of a class to be the set of objects that directly belong to the class, and **deep extent** to be set of objects that directly or indirectly belong to the class. We denote the extent and deep extent of a class C by E_C and E_C^* , respectively. For example, the extent of **EMP** class (denoted by E_{EMP}) includes all objects that belong to the **EMP** class only, but not **PT_EMP** or **FT_EMP**. On the other hand, the deep extent of the **EMP** class (denoted by E_{EMP}^*) includes all the objects that belong to **EMP**, **PT_EMP** and **FT_EMP**. In our OO query model, we allow queries to be directed at both the extents and deep extents of classes.

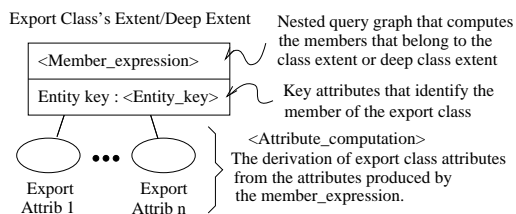


Figure 2: Extent Derivation Structure

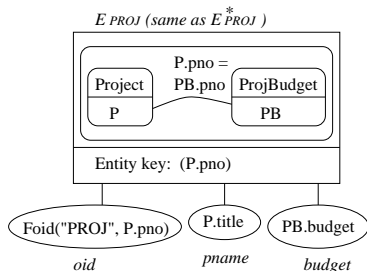


Figure 3: Extent Derivation Structure for E_{PROJ} and E^*_{PROJ}

EDSs are declarative in nature. Unlike other previously proposed derivation languages[7, 4], the class derivation structures are expressible in a graphical form. It is therefore easy to adopt extent derivation structures in a graphical tool for specifying export database derivation.

Definition: (Extent Derivation Structure)

An extent derivation structure (EDS) is defined as a 3-tuple, $\langle member_expression, entity_key, attribute_computation \rangle$, and is represented graphically in Figure 2.

An EDS example for deriving both E_{PROJ} and E^*_{PROJ} is shown in Figure 3 (to be further explained in Section 4). The *member_expression* is itself a **nested query graph** that computes, from the local relations, the relation containing the necessary information about object members of a class extent or deep class extent. At present, the operations involved in the *member_expression* operate on relations only. An export object may therefore correspond to a set of tuples computed by the *member_expression*. **Entity key** refers to the set of attributes used to identify these tuples that represent an object in the *deep extent* of an export class. Usually, entity keys are also primary keys of some local relations but are not always so. Note that this piece of information is required because *entity_key* is not always retained as attributes in the export class. For example, **pno** from the **Project** relation is not kept in the **PROJ** class. Attributes of the export class extent (or deep class extent) are extracted or computed from *member_expression* using *attribute_computation*. In the case of a relationship attribute from a source export class to a destination export class, the *attribute_computation* of the source export class must include the derivation of the entity key of the destination export class.

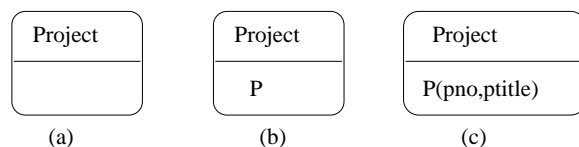


Figure 4: Primitive Nested Query Graph

The nested query graph representing *member_expression* extends the well-known query graph model by accommodating a large set of algebraic operations, i.e. selection (σ), projection (π), join (\bowtie), full-outerjoin ($\overline{\bowtie}$), one-way-outerjoin ($\overline{\bowtie}$), groupby, generalization attribute derivation (*GAD* - defined later in this section), union (\cup), intersection (\cap), subtraction ($-$) and aggregation. Clearly, the query graph can be further extended with new operations if the derivation of export DB requires. Due to its nested nature, the evaluation of a nested query graph should begin with the innermost component(s), unless wrappers perform some algebraic transformations that alter the implicit ordering of operations. Where necessary, the nested query graph also allows the relation represented by any of its components to be assigned an alias, and the relation's attributes to be renamed.

Definition: (Nested Query Graph)

A nested query graph is formed by two components, namely (a) a graph component, and (b) an optional relation alias with an optional list of attribute aliases. A nested query graph can be recursively defined as follows: (Due to space constraint, we do not show the nested query graphs constructed by $\overline{\bowtie}$, \cap , $-$ and aggregation.)

- *Primitive nested query graph*: In this case, the graph component contains just the local relation. If necessary, a relation alias may be assigned or a list of attribute aliases can be specified to replace attribute names. This is illustrated by Figure 4. Figure 4(a) shows a **Project** relation. Figure 4(b) shows that the **Project** relation is assigned a new relation alias; Figure 4(c) shows that the **Project** relation is assigned both relation and attribute aliases.
- *Selection and projection on nested query graph*: Selection and projection can be specified on a nested query graph as shown in Figure 5. Relation and attribute aliases are not required since selection and projection do not create any new attribute or tuple.
- *Join of nested query graphs*: Two or more nested query graphs can be joined together to form another nested query graph as shown in Figure 6(a). For any two nested query graphs involved in a join, we connect them by an undirected edge labeled with the join predicate. Relation and attribute aliases are optional.
- *One-way-outerjoin of nested query graphs*: Two

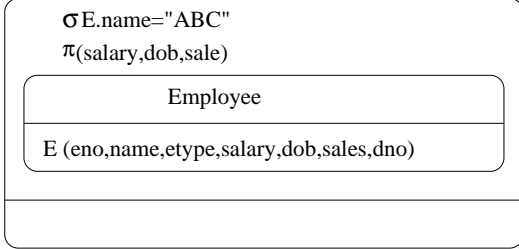


Figure 5: Selection and Projection on Nested Query Graph

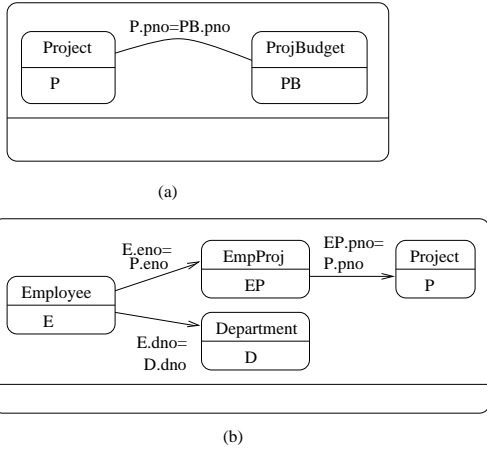


Figure 6: (a) Join of Nested Query Graphs, (b) One-Way-Outerjoin of Nested Query Graphs

or more nested query graphs can participate in a series of non-cyclic one-way-outerjoins as shown in Figure 6(b). We further restrict the outerjoin connectivity to be originated from a single class. Relation and attribute aliases are optional.

- *Generalized attribute derivation on query graph:* In order to perform computations on attributes, we introduce an operation known as **generalized attribute derivation (GAD)**. *GAD* is a unary operation that computes an output relation which contains attributes derived by applying system- or user-defined functions on the input relation. It is formally defined as:

Definition: (Generalized Attribute Derivation - GAD)

Let R be a relation with attributes A , and F_i 's be attribute functions.

$$GAD(R, F_1(X_1), F_2(X_2), \dots, F_m(X_m)) = \{ \langle F_1(X_1(r)), F_2(X_2(r)), \dots, F_m(X_m(r)) \rangle \mid r \in R \}$$

where $X_i \subseteq A$

Like the functions used in *attribute_computation*, the commonly used functions such as average function, identity function, etc. can be built-into the wrapper. Other functions can be user-defined and can be registered with the wrapper when required. A nested query graph example for *GAD* is shown in Figure 7(a). In this example, the employee's salary in Marks and age are computed by $F_{USStoMARK}$ and F_{age} respectively. Since new tuples are generated, new relation and attribute aliases are assigned.

- *Groupby nested query graph:* Groupby divides a relation horizontally into several partitions of records and summarizes selected attributes for each partition. This can be represented in a nested query graph as shown in Figure 7(b). Since the groupby operation creates new summary attributes, it is mandatory to assign relation and attribute aliases to the groupby nested query graph.
- *Union of nested query graphs:* Nested query graphs can be unioned together as shown in Figure 7(c). The resultant nested query graph must be given new relation and attribute aliases due to the merging of attributes.

4 Examples of Using Extent Derivation Structures

To illustrate the use of EDSs, we describe how the OO export DB example in Figure 1 can be derived from the set of local relations given in Section 3.1. Recall that an EDS must be defined for every export class extent and deep class extent. Nevertheless, it is clear that when an export class does not have any subclass, it's extent and deep extent are equivalent.

Deriving E_{PROJ} and E_{PROJ}^*

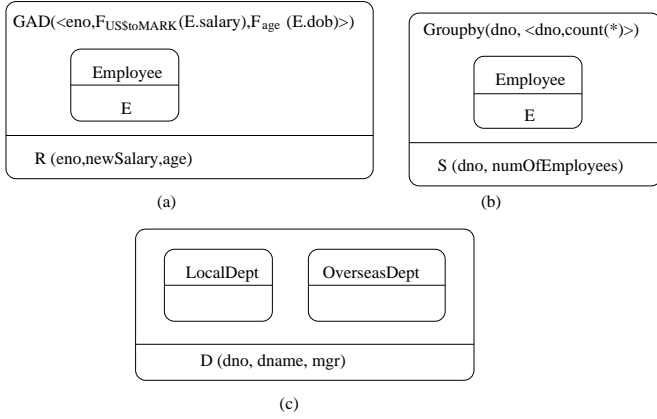


Figure 7: (a) GAD Nested Query Graph, (b) $Groupby$ Nested Query Graph, (c) Union of Nested Query Graphs

The EDS in Figure 3 is defined for both E_{PROJ} and E_{PROJ}^* . In the figure, the member expression indicates that the E_{PROJ} (or E_{PROJ}^*) members can be derived from a join between **Project** and **ProjBudget** assuming that every project must have a budget. The attribute, **P.pno**, has been designated to be the entity key. This implies that all export objects in E_{PROJ} can be uniquely identified by **P.pno** and therefore, any reference from other classes to E_{PROJ} has to use **P.pno** to obtain the corresponding E_{PROJ} (or E_{PROJ}^*) member(s). **P.pno** together with the export class name $PROJ$ are used to generate the export oids.

Deriving E_{DEPT} and E_{DEPT}^*

The EDSs of the $DEPT$ class extent and deep class extent are identical since $DEPT$ does not have any subclass. The information of E_{DEPT} (or E_{DEPT}^*) objects can be derived from several relations as shown in Figure 8.

The EDS in Figure 8 indicates that **LocalDept** and **OverseasDept** tables have to be unioned together to obtain the department numbers, department names and managers' employee numbers. The union'ed relation is aliased **LO**, and its attributes are also assigned new aliases. A department's revenue can be determined by the total sales made by its employees. To obtain the revenue information, the tuples in **Employee** are grouped by department numbers and the sum of sales for each group is computed. The groupby result is given a new relation alias(**GE**) and new set of attribute aliases. One-way outerjoins from **LO** to **E**, **GE** and **OA** relations collect all information needed to compute attributes of the department objects.

The entity key of E_{DEPT} (or E_{DEPT}^*) objects is **LO.dno** and it, together with "DEPT", are used to generate the oids of $DEPT$ objects. The other $DEPT$ attributes, e.g. **dname**, **d#**, **floor** and **revenue**, are derived or computed from the relation generated by the nested query graph. For relationship attributes **has** and **managedby**, the entity keys of the domain classes, in this case $E.eno$ and $LO.mgr$ respectively,

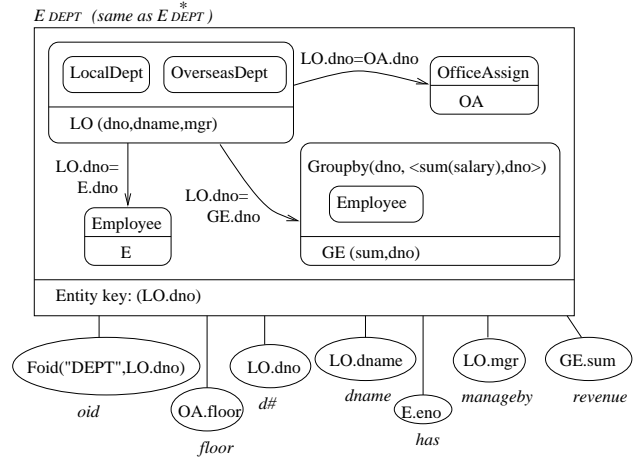


Figure 8: Extent Derivation Structure for E_{DEPT} and E_{DEPT}^*

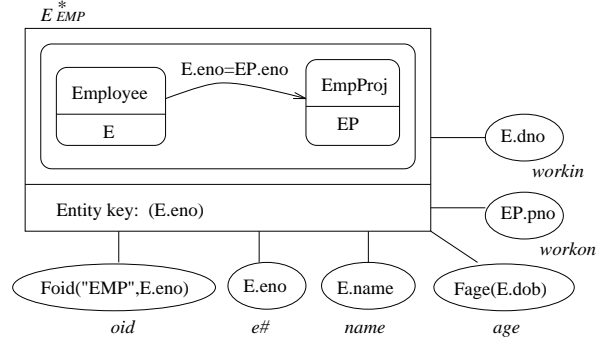


Figure 9: Extent Derivation Structure for E_{EMP}^*

are computed.

Deriving E_{EMP} and E_{EMP}^*

The EDS of E_{EMP} is not shown here since it is empty. Figure 9 depicts the EDS of E_{EMP}^* , the deep extent of EMP . To derive the relationship attribute, **workon**, we need the **Employee** to be extended with the project assignment information. Therefore, a one-way outerjoin from **Employee** to **EmpProj** is specified in the nested query graph. The EMP oids and attributes are generated or derived in a way similar to those of $PROJ$ and $DEPT$. Note that EMP , being selected as the id of the class poset involving EMP , PT_{EMP} and FT_{EMP} , has been used to compute the EMP oid. Note that the **age** attribute is computed by applying a function F_{age} on **E.dob**. Since no F_{age} exists in the local database, the function has to be included as part of the wrapper's data dictionary and is used during query evaluation.

Deriving E_{FT_EMP} and $E_{FT_EMP}^*$

Figure 10 shows the common EDS shared by E_{FT_EMP} and $E_{FT_EMP}^*$.

Deriving E_{PT_EMP} and $E_{PT_EMP}^*$

This is similar to that of E_{FT_EMP} and $E_{FT_EMP}^*$,

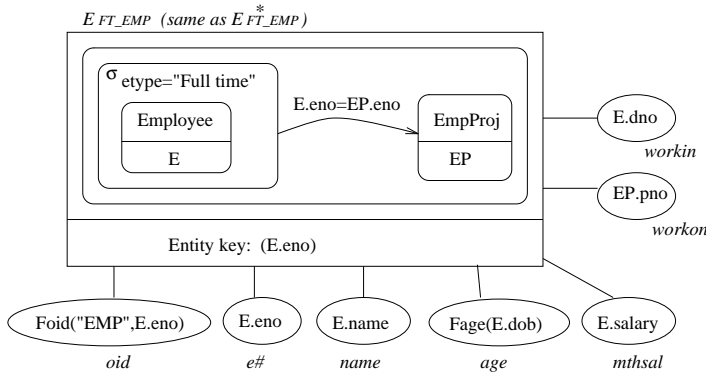


Figure 10: Extent Derivation Structure for E_{FT_EMP} and $E_{FT_EMP}^*$

and we do not show it here.

5 Wrapper Query Processing

The transformations of OO wrapper queries into local relational queries require both the export DB definition and derivation information. The former is needed to ensure the given wrapper queries are correctly formulated while the latter is used to replace the export classes by their corresponding derivation expressions in order to evaluate the queries. Our processing strategy decomposes a wrapper query into one or more local relational queries, generates intermediate results, and stores them in the local database during query processing. To handle operations not supported by the local DBMS, we incorporate query processing capabilities into the wrapper. The wrapper further performs query simplification to reduce the processing overhead.

5.1 Wrapper Query Processing Steps

The wrapper query processing steps are as follows:

- Step 1: Set up the initial query graph**
 An initial query graph consists of nodes representing export class extents, and edges representing the relationships between the export class extents referenced by the query. Let the class extent appearing in the **FROM** clause be called the **anchor**. We construct directed edges from the anchor to the other class extents referenced by path expressions found in the **SELECT** and **WHERE** clauses.
- Step 2: Replace the export class extents in the query by simplified member expressions**
 The export class extents in the initial query graph are replaced by the member expressions of their corresponding EDSs. In the process, the member expressions are simplified by removing those subexpressions which do not contribute to the query result. We call the resulting query graph the **augmented query graph**.
- Step 3: Generate the query tree**
 From the augmented query graph, a query tree

which indicates the order of evaluating the query operations is generated. The leaf nodes and internal nodes of the query tree denote the local relations and query operations, respectively. Since multiple query trees can be generated from a single augmented query graph, the query tree must be carefully chosen to reflect the query optimization strategy adopted by the wrapper. As part of query optimization, the query tree may be simplified by a set of heuristic rules to reduce its processing cost.

- Step 4: Determine the wrapper and local DBMS query fragments**

Since not all operations in a query tree may be evaluated by the local DBMS, a wrapper query processor has to distinguish between the operations to be executed by the local DBMS and by itself. By clustering the operations to be performed at the wrapper and the local DBMS, we obtain the **wrapper query fragments** and **local DBMS query fragments**, respectively.

5.2 An Example Wrapper Query and Its Processing

In this section, we demonstrate wrapper query processing using the following query example ($Q1$). The query retrieves, for the full-time employees who are younger than 20 years old and who work on some project of budget greater than \$1,000, their names, department names, and the project names they work on.

```
Q1: SELECT F.name, F.workin.dname,
      F.workon.pname FROM FT_EMP F
      WHERE F.age < 20 and F.workon.budget > 1000
```

Step 1: Set up the initial query graph

Figure 11 depicts the initial query graph constructed for $Q1$. The export class extents referenced are assigned unique class aliases and the directed edges between export class extents are marked with the corresponding relationship attribute names. The class aliases $C1$, $C2$ and $C3$ have been assigned to E_{FT_EMP} , E_{DEPT}^* and E_{PROJ}^* , respectively. The simple attributes referenced by the query, i.e. *name*, *age*, *dname*, *pname* and *budget*, are attached to the export class extents they belong to. The target attributes, *name*, *dname* and *pname*, are marked by *. The **WHERE** predicates are also indicated next to their attributes.

Step 2: Replace the export class extents by simplified member expressions

Figure 12 shows the augmented query graph obtained by replacing the export class extents by their simplified member expressions. To avoid the same relation alias to be used by different member expressions, we prefix the relation aliases in the member expressions by the unique class aliases of their corresponding export class extents. By examining attributes referenced by the query, some member expressions can be simplified. For example, the E_{DEPT}^*

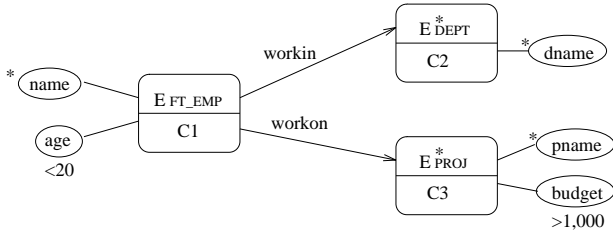


Figure 11: The Initial Query Graph for Q_1

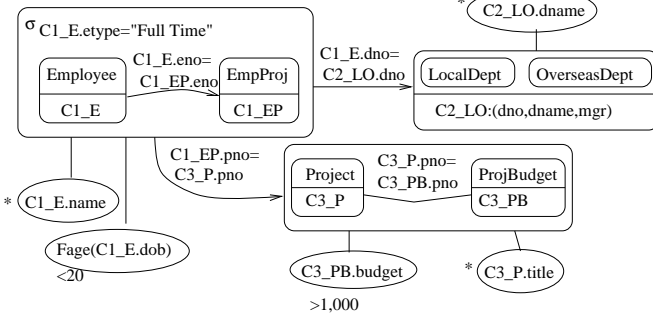


Figure 12: The Augmented Query Graph for Q_1

member expression used in this example has been simplified since the employee, floor and revenue information of departments are not referenced by the query. The simple attributes are replaced by their relational correspondences while the relationship attributes are replaced by the appropriate outerjoin predicates. For example, the *workin* relationship attribute of E_{FT_EMP} has been replaced by the join predicate involving $C1_E.dno$, computed by the EDS of E_{FT_EMP} , and $C2_LO.dno$ which is the entity key of the $DEPT$ class.

Step 3: Generate the query tree

The generation of the query tree from an augmented query graphs is a query optimization problem. In this paper, we do not intend to delve much into the wrapper query optimization issue since it is beyond the scope of this paper. We will, however, describe some heuristic optimization that can be performed on the query tree. Details of the algebraic transformation rules that make the heuristic optimization possible can be found in [2, 5, 3]. The generation of query trees can be divided into two sub-steps.

- *Sub-step 1: (Decide the ordering of joins and outerjoins)*

By deciding the ordering of joins and outerjoins, a preliminary query tree can be constructed. A selection operation attached with the **WHERE** predicates, and a projection operation that keeps only the target attributes are added as the last two operations in the query tree as shown in Figure 13. The $[< relation_alias > : < attribute_alias_list >]$ notations attached to some nodes indicate places where relations or attribute names are named/renamed.

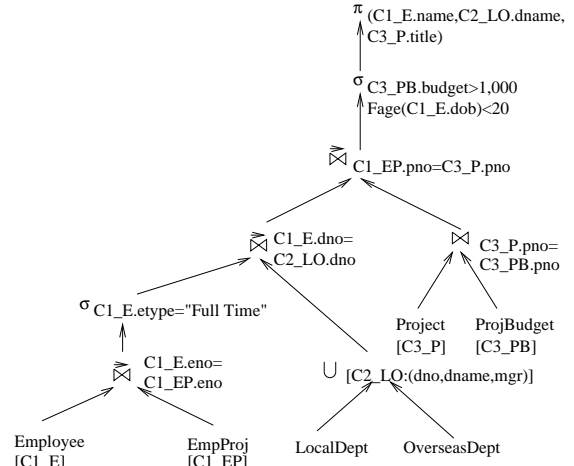


Figure 13: Query Tree Before Heuristic Optimization

- *Sub-step 2: (Perform heuristic optimization on the query tree)*

Without any local cost model information, one can only perform heuristic optimization on the preliminary query tree using some algebraic transformation rules. To reduce the amount of processing, we push the selection operations and projection operations down the tree nearer to the leaf nodes so that they are evaluated as early as possible. In the process, we also convert some outerjoins into joins without affecting the final result. Figure 14 shows the query tree after heuristic optimization. Note that the selection predicate $Fage(C1_E.dob) < 20$ has been moved to right above the **Employee** leaf node. The predicate $C3_PB.budget > 1,000$ is also moved to right above the **ProjBudget** node. The outerjoin operation along this move is transformed into a join operation since records with $C3_PB.budget = NULL$ have to be discarded. Interested readers can refer to [5] for information about the transformation rules.

Step 4: Determine the wrapper and local DBMS query fragments

Assuming that the local DBMS is SQL-based and does not handle $Fage()$ and outerjoins, the optimized query tree can be decomposed into wrapper and local DBMS query fragments as shown in Figure 14. Each local DBMS query fragment can be translated into a SQL query and be submitted to the local query processor. In the figure, each query fragment is enclosed by a dotted region annotated by a label (GQF_i for wrapper query fragment and LQF_j for local DBMS query fragment, for some i and j). While the wrapper query fragments are evaluated by the wrapper query processor, the intermediate results produced may have to be created as local database tables in order for the local DBMS to execute the next local DBMS query fragments. Note that LQF_3 and LQF_4 , though being next to each other, cannot be evaluated as one

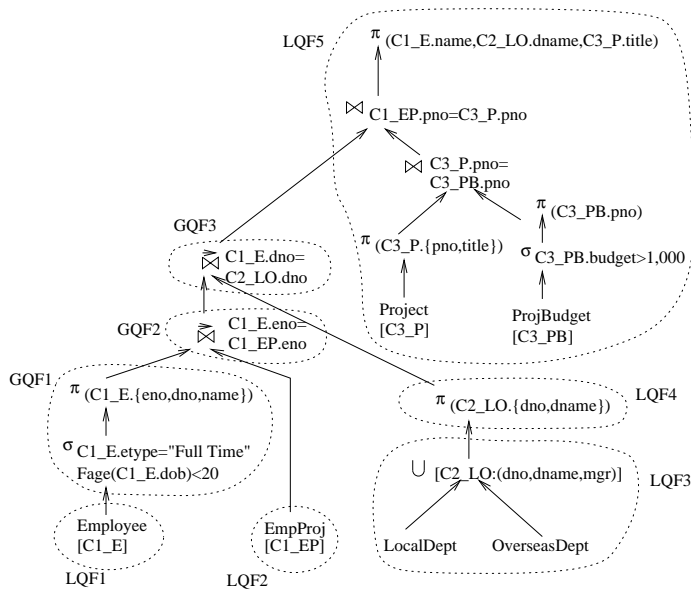


Figure 14: Query Tree After Heuristic Optimization and Query Fragment Generation

local DBMS query fragment. This is due to the SQL language constraint which disallows a selection to be performed on a union expression within a single query statement.

6 Conclusions

In this paper we have described in detail the design of a wrapper to support object-oriented queries on an export DB view built on an existing relational DB. We design the wrapper such that it can fulfil the different DB integration needs. We propose the concept of **extent derivation structure (EDS)**, which allows us to derive the export class extents and deep class extents. The EDS representation also supports relationships between export classes. Due to its inherent algebraic semantics, the expressive power of EDS can be defined mathematically as the set of algebraic operations it can support. A wrapper query can be translated into more than one local queries depending on the complexity of the export DB derivation. To support query operations not found in the relational DBMS, we allow extra query processing capabilities to be incorporated into the wrapper query processor.

As part of our cooperative information system project, we have prototyped the core components of the wrapper query processor. The wrapper prototype is able to support OO queries on export DBs constructed on relational databases implemented in Postgres[8]. The future research directions to be pursued include:

- **Cost-based**

Optimization of Wrapper Queries: So far, we have determined a set of algebraic transformations for heuristics optimization. This can be further improved if the cost model of the exist-

ing DBMS is made available or can be calibrated. With a cost model, we can perform better query optimization on the wrapper queries.

- **Modeling of Legacy Applications:** In this paper, the focus is on the reuse of existing relational databases. A large amount of information can, however, be found embedded in the legacy applications. At present, the modeling of legacy applications has not been studied much in the literature. We plan to extend our approach to model such application semantics in the future.

References

- [1] M.W. Bright, A.R. Hurson, and S.H. Pakzad. A taxonomy and current issues in multidatabase systems. *IEEE Computer*, March 1992.
- [2] A.L.P. Chen. Outerjoin optimization in multidatabase. In *Proceedings of Databases in Parallel and Distributed Systems*, pages 211–217, 1990.
- [3] C. Galindo-Legaria and A. Rosenthal. How to extend a conventional optimizer to handle one- and two-sided outerjoin. In *Proceedings of the 8th Int'l Conf. on Data Engineering*, pages 402–409, 1992.
- [4] Wolfgang Klas, G. Fischer, and K. Aberer. Integrating relational and object-oriented database systems using a metaclass concept. *Journal of Systems Integration*, 4(4), 1994.
- [5] E.-P. Lim, J. Srivastava, and S.-Y. Hwang. An algebraic transformation framework for multidatabase queries. *Distributed and Parallel Database Journal*, 3(3), 1995.
- [6] W. Litwin and A. Abdellatif. Multidatabase interoperability. *IEEE Computer*, December 1986.
- [7] C.C. Liu and A.L.P. Chen. Object view derivation and object query transformation. In *Proc. IEEE COMPSAC*, 1994.
- [8] M. Stonebraker and G. Kemnitz. The postgres next-generation database management system. *Communications of the ACM*, 34(10), Oct. 1991.
- [9] T. Takahashi and A.M. Keller. Implementation of object view query on relational database. In *Int'l Conf. on Data and Knowledge Systems for Manufacturing and Engineering (DKSME)*, Hong Kong, May 1994.