Semantic Query Optimization based on Class Partitioning Techniques in an Object Relational Data Warehousing Environment*

Vivekanand Gopalkrishnan
itvivek@cityu.edu.hk

Qing Li
itqli@cityu.edu.hk
City University of Hong Kong
Hong Kong, CHINA

Kamalakar Karlapalem
kamal@iiit.net
International Institute of Information Technology
Hyderabad, INDIA

Abstract
The conventional star schema model of Data Warehouse (DW) has its limitations due to the nature of the relational data model. Firstly, this model cannot represent the semantics and operations of multi-dimensional data adequately. Due to the hidden semantics, it is difficult to efficiently address the problems of view design. Secondly, as we move up to higher levels of summary data (multiple complex aggregations), SQL queries do not portray the intuition needed to facilitate building and supporting efficient execution of complex queries on complex data. In light of these issues, we propose the Object-Relational View (ORV) design for DWs. Using Object-Oriented (O-O) methodology, we can explicitly represent the semantics and reuse view (class) definitions based on the generalization hierarchy (is-a) and the class composition hierarchies (cch), thereby resulting in a more efficient view mechanism. Part of the design involves providing a translation mechanism from the star/snowflake schema to an O-O representation. Associated Horizontal Class Partitioning (AHCP) technique can next be applied upon this O-O schema to further increase the efficiency of query execution by reducing irrelevant disk access. Several indexing methods can be implemented on this partitioned schema to facilitate complex object retrieval and to avoid using a sequence of expensive pointer chasing (or join) operations. Finally, we present the analytical results, based on a cost model we have developed, to demonstrate the effectiveness of our approach vis-a-vis the unpartitioned, pointer-chasing approach.

Keywords: *data warehouse design, object relational views, primary partitioning, associated horizontal partitioning, semantic query optimization.*

1. Introduction

Data warehouse (DW) equips users with more effective decision support tools by integrating enterprise-wide corporate data into a single repository from which business end-users can run reports and perform ad hoc data analysis [CD97]. As DWs contain enormous amount of data, often from different sources, we need highly efficient Indexing structures [OQ97], [Sar97], [GHRU97], [VLK00a], materialized (stored) Views [Rou97], and query processing techniques [VLK99] to efficiently answer on-line analytical processing (OLAP) queries. Materialized Views represent integrated data based on complex aggregate queries, and should be available consistently and

* This work has been supported by City University of Hong Kong under grant no.7001120.

instantaneously. Maintaining the integrity of these Indexes and Views imposes a challenging problem when the source data changes frequently, when the size of the DW keeps growing, and/or when the user queries become more and more complex [GM95], [MK99]. An extensible framework that can accommodate dynamic warehousing [Dayal99] of changing data gracefully, and have adaptive handles for processing OLAP queries efficiently is needed.

1.1 ORV framework

In [VLK98], we examined issues involved in developing the *Object-Relational View (ORV)* mechanism for the data warehouse. Here, *OR* means an *object-oriented* front-end or views to underlying *relational* data sources. So, the architecture and examples we provide follow our interpretation of OR. It must be noted though, that the merits of this proposal can be applied to views in *Object-Relational Databases (ORDBs)* [CMN97] also. The layered architecture of the ORV is as shown in figure 1.

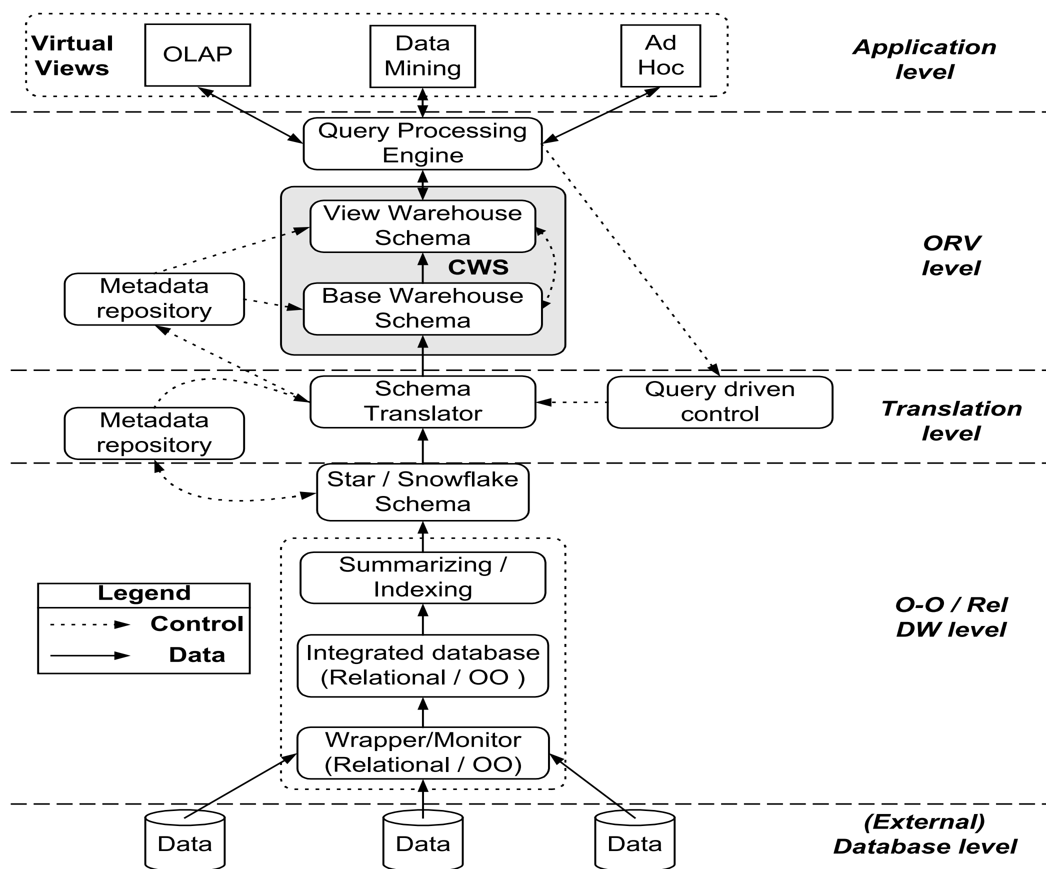

Fig. 1. A Layered Architectural Framework

In this framework, two models are captured; both have a multi layer architecture, consisting of wrapper/monitors, integrator and summarizing units. In the first model, *relational - OO* translation is done after database integration, hence the warehouse data is built on the underlying integrated framework. In the second model, we perform translation into the *O-O* model at the wrapper level, so that the *canonical* model for the integrated schema is the *O-O* model, offering more flexibility in dealing with diverse semantics of the underlying data [NS96].

The Complete Warehouse Schema (CWS) in both models contains *Base classes (BWS)* which include some directly mappable classes and some derived *View* classes *(VWS)* based on the OLAP queries. The CWS views can be implemented as partitions, indexes and aggregate views, as will be described in the next sections. Further more, views (*Virtual classes*) can be inherited from this *CWS*. These views may be partially or completely materialized.

1.2 An Iterative ORDW design methodology

In [VLK98], we showed that besides establishing a semantically richer framework for multi-dimension hierarchies, the *ORV* model provides excellent support for complex object retrieval. Here we illustrate this iterative design methodology for the ORDW.

Our view design methodology depends on the type and pattern of queries that access the DW frequently. By incorporating these access patterns, we can form an efficient framework for retrieving popular queries. Note though that as the queries change, the *O-O schema* may require changes in terms of partitioning and indexing, but the underlying schema is fairly dynamic because of embedded semantics, viz. nested containment relationships, references, *is-a* types, multi-valued objects & object identity. This implicit support of semantics also enables efficient traversal of multiple query paths along the same dimension hierarchy. For example in the *Time* dimension, multiple paths could be along the Week, Month & Season compositions. These are supported by the Class Composition Hierarchy *(CCH)* framework as shown in figure 2.

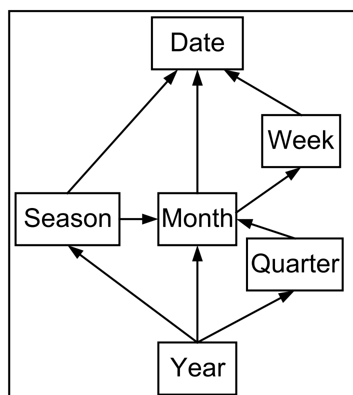

Fig. 2. The Time Hierarchy

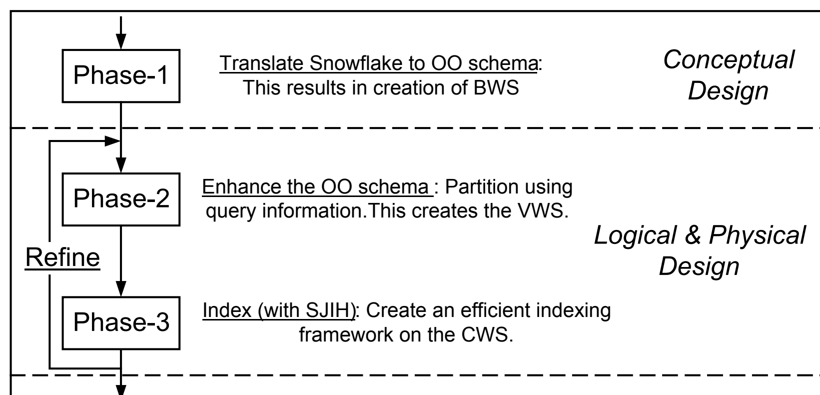

Fig. 3. The ORV Design methodology

As shown in figure 3, we illustrate our methodology in three phases. Phase 1 is the *Conceptual level design*, in which we translate the underlying schema from the snowflake schema model to the *O-O* model, using the Primary Query Set (*PQS*) which contains queries that are fairly stable and most frequently asked. Conceptual schema is generally static and does not reflect internal optimization, hence this phase is outside the refinement loop. The *Logical & Physical design* consisting of Phase-2 (*AHCP* partitioning) & Phase-3 (indexing) are constructed based on the Secondary Query Set (*SQS*) containing the other queries that are less frequently asked but are still significant enough for optimization. These two phases are repeated until the *ORV* schema and indices are optimized. Aggregate Views could be further built on this refined schema and materialized if needed. These phases are explained in detail in the following sections.

1.3 Paper contribution and organization

In this paper, we present the Object Relational Data Warehousing (*ORDW*) methodology as an approach to address many of the issues associated with data warehouse schema design [VLK98].

To put our research in perspective, we review some related work in section 2 and briefly outline our previous work in the contexts of ORDW, Class Partitioning and Indexing on OODBs; we further motivate our study by presenting a sample DW schema and some OLAP query characteristics. In section 3, we create basic sub-query expressions (out of PQS) and utilize a Multiple Query Optimization (*MQO*) approach to trigger our schema design processes. We further explain the *Conceptual level design* of the iterative ORDW design methodology, viz., translation from Star to OO schema and refinements. Section 4 deals with the *Logical & Physical design* aspects, which comprises

of obtaining an optimal AHCP partitioning scheme (4.1), and indexing schemes (4.2). In section 5, we provide a walkthrough of our design algorithms and performance analysis of our iterative methodology. Section 6 illustrates some innovative applications of the ORDW framework, viz. Multi-fact aggregate queries, recursive OLAP, and parameterised queries. Finally we conclude in section 7, and briefly state our future work.

2. Motivations and related work

Here we provide a need for adopting a new methodology viz. Object Relational Data Warehousing. We motivate it by building on the very popular "*Sales*" Data Warehouse schema, and introduce complexities by means of *normalising* and *range* queries. Further, we state work done in the realm of logical and physical design, with encouraging results that justify this approach.

2.1 Motivations

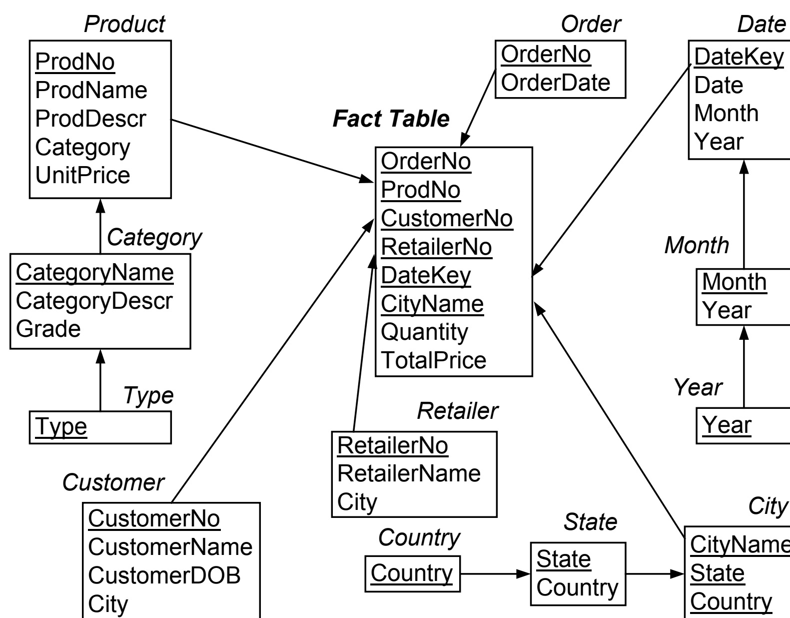

Fig. 4. A Sample Snowflake Schema

Let us consider the sample *snowflake schema* for a *Sales* DW (taken from [CD97]), with one *fact table* and *dimension* tables representing *Time, Product, Customer,* and *Address* hierarchies, as shown in figure 4. As shown in table 4, OLAP queries could be posed on various predicates along a single hierarchy, as well as on predicates along multiple hierarchies. Summary tables could be defined along a predicate or set of predicates by separate *fact* tables and corresponding *dimension* table(s). These summary tables could be materialized depending on various materialization selection algorithms to improve querying cost. As seen in the figure, the dimension tables in the *snowflake schema* (along with schema for summary tables) are in a *composition hierarchy*.

For the purpose of this paper, queries involving Nested Facts can be considered as sub-sets of inter-Fact queries. They are distinguished by the presence of a semantic disjoint-ness between the Facts involved. It must be noted though that this disjoint-ness does not preclude the Facts from sharing the same component objects. A query processing scheme that is built on separate Facts will inadvertently need costly joins. This inefficiency is amplified for queries with low selectivity and high frequency. This calls for a need for a partitioning scheme that transcends Facts and is not restricted by the hierarchies mentioned. It must be noted that such a partitioning scheme may well be overlapping and hence will suffer due to storage space restrictions.

For the example in the figure, some OLAP queries could be on the entire range of Sales and would need to access multiple dimensions for the commonly used *Group By* clauses. However, other queries

could also have a predicate range in place (such as "Categ=Elec" or "Country=US"). In such cases, the search space on the Fact "Sales" is reduced by a factor equal to the selectivity of the predicate. However, this does not help during query processing (normal unpartitioned case), as the entire Fact table is processed while searching for relevant tuples. Even in cases where indexes are built [VLK00a], the benefit could be reduced, as index creation takes up more time due to the enormity of the Fact. Further, as the OLAP queries involve multiple paths (multiple selections and group bys), the size of the Forward and Reverse Joins is dependent on the size of the Root (Fact). This calls for the need to ***partition*** the Fact according to the query characteristics [VLK00b].

On the other hand, indexing is definitely a complementary means (to class partitioning) for efficient query processing. As noted above, the dimension tables in the *snowflake schema* (along with schema for summary tables) are in a *composition hierarchy*, hence they can be naturally represented as an Object-Oriented schema. Therefore, querying costs (*join*) on complex predicates along this *snowflake schema* should be analogous to querying costs by *pointer chasing* mechanism in an O-O framework. From [FKL98], we see that the *Structural Join Index Hierarchy (SJIH)* mechanism is far superior to *pointer chasing* operations for Complex Object retrieval, especially in queries involving predicates from multiple paths. Experimental results [Won98], [VLK00a] conform to the analytical results of this cost model. It therefore makes sense to incorporate the semantic-rich SJIH into our ORDW framework [VLK00a][1], as an additional step to embed query semantics for efficient query processing (cf., Fig. 3).

2.2 Related work

Partitioning has been vastly researched in Relational and OO database systems. Excellent work has been done in Vertical Partitioning (VP) and Horizontal Partitioning (HP) in both systems, but the unique features of OO systems have made it possible to experiment with different variations such as Derived Horizontal Class Partitioning (DHCP) [BK98], Associated Horizontal Class Partitioning (AHCP), Path Partitioning (PP) and Method Induced Partitioning (MIP). [KL00] presents a comprehensive framework for devising partitioning schemes based on different types of methods and their classification. The issue of fragmentation transparency is addressed by considering appropriate method transformation techniques. While those methods were extremely successful in the transactional environment of an OODB, to the best of our knowledge, no work has been done in partitioning of an Object Relational DB. Our research in partitioning an Object Relational Data Warehouse (ORDW) [VLK00b] is the first work in this direction.

Recently, we have conducted some preliminary studies on developing the ORDW framework. In [VLK98], we showed that the ORV (Object Relational View) model offers inherent features that are conducive to managing a data warehouse. We listed the various issues that arise during the design of an OR-DWMS (Object Relational Data Warehouse Management Systems). Here, OR means an object-oriented front-end or views to underlying relational data sources. Based on the issues discussed in [VLK98], we articulated a three-phased design approach in [VLK99], which also provided a query-driven translation mechanism from the star/snowflake schema to an object oriented (O-O) representation. Some query processing strategies utilizing Structural Join Index Hierarchy (SJIH) techniques for complex queries on composite objects were addressed in [VLK00a]. In this paper, we focus on the efficacy of class partitioning techniques in the context of our ORDW framework, for the purpose of semantic query optimization.

3. Conceptual Design

As seen in figure 3, the *Conceptual level design* of the iterative ORDW design methodology consists of Phase-1, i.e. the translation from *Star* schema to *OO* schema. Conceptual schema is generally static and does not reflect internal optimization, hence this phase is outside the refinement loop, which operates only on phases 2 and 3. The schema at the conceptual level should be immune to changes in query patterns and frequency. As mentioned in section 1.2, we use the Primary Query Set (*PQS*) as

[1] In this paper we omit further coverage and evaluation of the SJIH indexing scheme on the ORDW, due to space limitations. The reader may refer to [VLK00] for further details.

input for the construction of the ORV schema (the translation phase). In order to implement this, we create atomic sub query expressions by utilizing a Multiple Query Optimization (*MQO*) approach. Assigning weights to the intermediate nodes based on frequency and degree of sharing of the query sub-expressions, we derive ORV schema from the *snowflake* schema.

It must be noted here that as the queries change, the *ORV* schema may require changes in terms of partitioning and indexing, but the underlying schema is fairly *static* because of embedded semantics. This implicit support of semantics also enables efficient retrieval of multiple query paths along the same dimension hierarchy.

3.1 MQO hierarchy (MVPP)

To illustrate the derivation of our ORV schema, we assume the Primary Query Set (PQS) giving three sub-queries that have the highest frequency and degree of sharing that induce *ORV* schema range derivation:

Q1 : Sales to Customers whose age is no more than 19 years (i.e. *Teenager*).
Q2 : Sales to Customers whose age is at least 20 years old (i.e. Adult).
Q3 : Sales to Customers whose age is more than 50 years, group by Product.

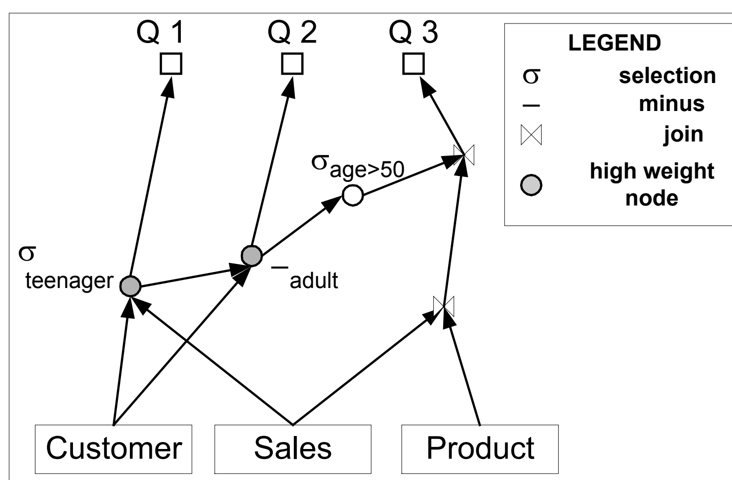

Fig. 5. MQO on the Primary Query Set

From this simple sub-query set, we create a sample MQO as shown in figure 5. The MQO is a DAG (Directed Acyclic Graph) from the ORDW classes to the sub-queries in PQS. Each node represents an operation (e.g., select, project, join), and is given weights according to the frequency of the queries accessing it and the degree by which it is shared. In the above queries, we see that pushing down the select operation on the age of Customer creates nodes for "adult" and "teenager" that are most frequently accessed and are thereby assigned higher weights.

3.2 ORV schema derivation

The fundamental star schema model consists of a single Fact Table (FT) and multiple Dimension Tables (DTs). This can be further sub-classed as *snowflake* (normalizing along DTs) and *multi-star* (normalizing along FTs) and combinations of *multi-star* & *snowflake schema* models. We illustrate our translation mechanism here on the single *star / snowflake schema* model. Note that a generic extension to include *multi-star schema* models can be easily derived due to advantages of the O-O model as stated in section 2.

Star / Snowflake Schema

A snowflake schema consists of a single Fact Table (FT) and multiple Dimension Tables (DT). Each tuple of the FT consists of a (foreign) key pointing to each of the DTs that provide its multidimensional coordinates. It also stores numerical values (non-dimensional attributes, and results

of statistical functions) for those coordinates. The DTs consist of columns that correspond to attributes of the dimension. DTs in a *star* schema are denormalized, while those in *snowflake* schema are normalized giving a Dimension Hierarchy. A generalized view of the snowflake schema is presented in figure 6.

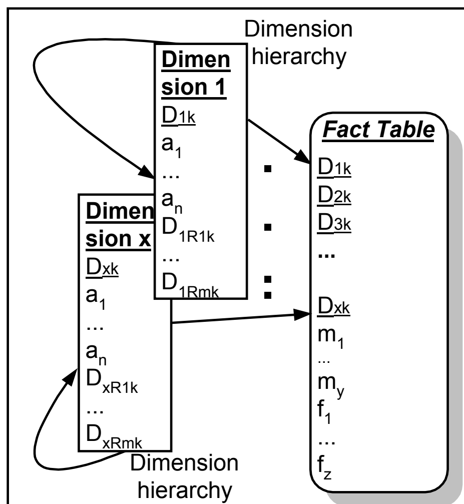

Fig. 6. Generalized view of snowflake schema

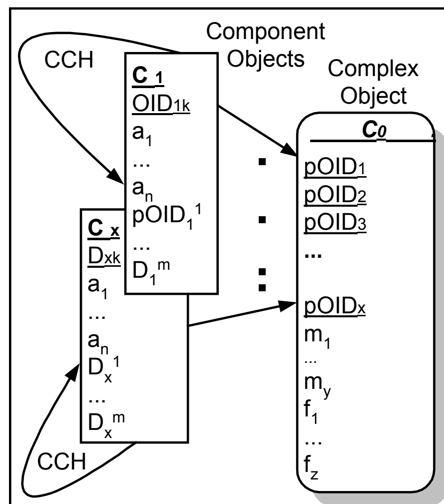

Fig. 7. Corresponding O-O schema

Preliminaries

Every tuple in the FT consists of the *fact* or *subject* of interest, and the dimensions that provide that *fact.* So each tuple in the FT corresponds to one and only one tuple in each DT, whereas one tuple in a DT may correspond to more than one tuple in the FT. So we have a 1:N relationship between FT : DTs.

Let the *snowflake schema* be denoted as *SS.*

No. of FT = 1; No. of DT = x.

We denote the relations between the FT and DTs as:

$Rel\ (FT, DT_i) = R_i$

$1 \leq i \leq x$; where x is the no. of DTs

Let the Relations between DTs in a dimension hierarchy be denoted as:

$Rel\ (DT_i^r, DT_i^{r+1}) = R_i^r$

$0 \leq r \leq m$;

where m is the no. of relations in the hierarchy under DT_i.

and $DT_i^0 = DT_i$

Table 1. Elements of the Fact Table (FT)

$\{D_{ik}\}$	set of Dimension keys, each corresponding to a Dimension Table (DT). $1 \leq i \leq x$; where x is the no. of DTs
$\{m_j\}$	set of member attributes. $0 \leq j \leq y$; where y is the no. of attributes
$\{f_s\}$	set of results of statistical functions. $1 \leq s \leq z$; where z is the no. of

Table 2. Elements of the Dimension Table (DT_i)

D_{ik}	Index of the DT
$\{a_j\}$	Set of member attributes. $0 \leq j \leq n$; where n is the no. of attributes.
$\{R_{irk}\}$	set of keys of relations that form its Dimension Hierarchy.

	function results.

	$0 \leq r \leq m$; where m is the no. of relations in the hierarchy under DT_i

3.2.1 Translation Algorithm

Our methodology intends to capture the *hidden* semantics behind a DW schema design, by incorporating the *star / snowflake schema* information with the *query type* and *pattern* information. Frequent Data warehousing queries can be thought of being decomposed and categorized into the following form:

$Q \rightarrow \{ Q_n \cup Q_r \}$

where Q_n is the set of sub-queries that leads to *normalising* the schema, and Q_r is the set of sub-queries (from the MQO hierarchy) that act on a *range* of the schema, with the highest weights (cf. figure 5).

Based on this classification, we can refine the resultant schema in two complementary ways: Refinement-1, involving *normalising* sub-queries Q_n, and Refinement-2, involving *range* sub-queries Q_r.

Refinement 1 - *normalising*

In the ORDW environment, *normalising* can be regarded as a technique for refining the ORDW schema through utilizing the query semantics to generate a finer class composition hierarchy of any class. The refinement can be accomplished in a step-by-step manner, as shown below.
We note that in terms of predicates accessed in the DTs, queries of type Q_n can be defined as

$Q_n \rightarrow (DT_i^r . \{a_j\})$ where $\{a_j\}$ is a set of attributes of DT_i^r.

Step N1. For the Fact Table FT in the snowflake schema, create a class C_0 in the O-O schema.

$$\text{Create } C_o$$

Step N2. For each Dimension Table DT_i in the snowflake schema, create a class C_i in the O-O schema.

$$\forall\ DT_i \text{ Create } C_i$$

Step N3. For each relation R_i in the snowflake schema, create a pointer to OID, $pOID_i$ in class C_0 in the O-O schema.

$$\forall\ R_i \text{ Create } C_0 . pOID_i = OID(C_i)$$

Step N4. For each member attribute m_j in FT in the snowflake schema, create an attribute m_j in class C_0 in the O-O schema.

$$\forall\ m_j \text{ in FT Create } C_0 . m_j$$

Step N5. For each result-value attribute fs in FT in the snowflake schema, create an attribute fs in class C0 in the O-O schema.

$$\forall\ f_s \text{ in FT Create } C_0 . f_s$$

Step N6. For each relation R_i^r in the snowflake schema, create a class C_i^r in the O-O schema.

$$\forall\ R_i^r \text{ Create } C_i^r$$

Step N7. For each member attribute a_j in DT_i^r in the snowflake schema, create an attribute a_j in class C_i in the O-O schema.

$$\forall\ i(\forall r\ DT_i^r . a_j \text{ Create } C_i . a_j)$$

Step N8. For each relation R_i^r in the snowflake schema, create a pointer to OID, $pOID_i^r$ in class C_i in the O-O schema.

This is a recursive step, as it navigates through the dimension hierarchy. The relations between the various nodes of the DT are explicitly captured, so steps 6-7 can be repeated in the hierarchy loop.

$$\forall\ R_{irk} \text{ Create } C_{ir} . pOID_i^r = OID(C_i^r)$$

Step N9. For each Query Q_i in Q_n, which accesses a set of $\{a_j\}$ belonging to a DT in D_i, vertically partition the corresponding class C_i in the O-O schema.

$\forall\ Q_i\ (\forall d\ DT_n.\ \{a_j\}\ \text{Create}\ C_{nj} \leftarrow C_n)$

Refinement 2 - range derivation

In terms of values of predicates accessed in the DTs, queries of type $\mathbf{Q_r}$, can be defined as

$Q_r \rightarrow (\ DT_{ir}.\ a_j\ .\ \{v_k\}\ )$ where v_k is a set of values of attribute a_j of DT_i^r.

Step R1. For each Sub-Query Q_i in $\mathbf{Q_r}$, which accesses a record containing a range of values $\{v_k\}$ for attribute a_j belonging to a DT in D_i, derive the range of corresponding class C_i in the O-O schema.

$\forall\ Q_i\ (\forall d\ DT_d.\ a_j\ .\ \{v_k\}\ \ \text{Create}\ C_{djk} :: C_d)$

This forms the primary *is-a hierarchy* of the *O-O* schema. Here, the classes mapping to the DTs are divided into range groups according to the queries acting on them. This *range derivation* ensures that specific *subsets* of *classes* are available while maintaining a high degree of *reusability*. As seen in figure 7, the generalized view of the O-O schema is similar to that of the snowflake schema. The class corresponding to FT is C_0 .

3.2.2 Resultant schema

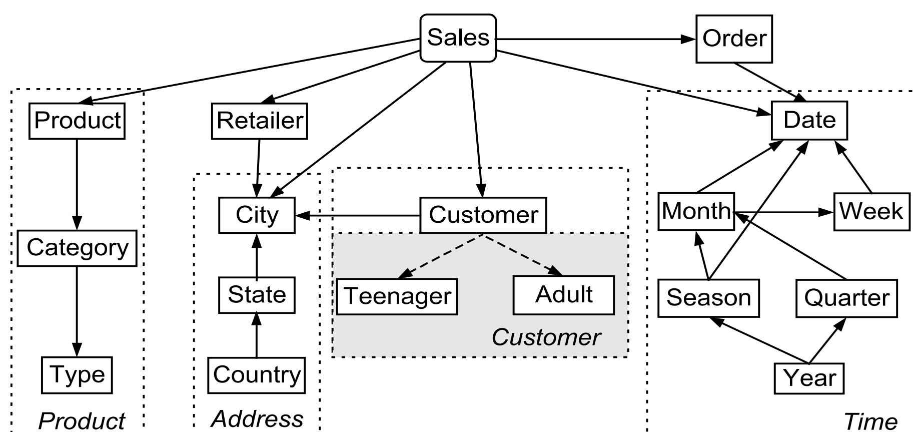

Fig. 8. The OO Schema.

The figure shows the class composition hierarchy for the *Time* dimension after refinement 1, and the is-a hierarchy (shaded area) for the *Customer* dimension after refinement 2.

Figure 8 (without the shaded area) shows the translated *O-O schema* for the *Sales* example taken in previous sections, which is generated by tracing the steps of the above algorithm step-by-step: Note that this hierarchy is not a mere mapping of FTs and DTs from the *snowflake schema*. The classes mapping to the DTs are further vertically partitioned according to the queries acting on them. For an example of multiple paths within a single dimension hierarchy, let us consider the *Time (Date)* hierarchy. If the queries access *Date* by multiple paths like *Day_of_Week*, or *Day_of_Month* or *Week_of_Quarter*, they must be supported within the same path, instead of having to access disjoint *entities (classes)*.

4. Logical & Physical Design

As seen in figure 3, the *Logical & Physical level design* of the iterative ORDW design methodology consists of Phase-2, i.e. enhancing the *ORV* schema (*VWS*) with query-driven Associated Partitioning and Phase-3, i.e. creating indexing mechanisms on the Complete Warehouse Schema (CWS). These two phases are repeated until the *ORV* schema and indices are optimized, as shown in the refinement iteration loop (cf. figure 3). Aggregate Views could be further built on this refined schema and materialized if needed. This design level is influenced by the Secondary Query Set (*SQS*) containing

the queries that are less frequently asked (as compared with PQS), but are still significant enough for optimization.

Based on classifications by DW operations & by OO concepts, we consider the following queries listed in Table 3 as our sample SQS for subsequent discussions.

Table 4. Sample OLAP queries - SQS

No.	*Query*	*Query type*
Q1	Sales by Prod by State in US	Only along cch (pivot)
Q2	Sales by Prod by State by Year in US	-> Drill-down
Q3	Sales by Prod for Categ=Elec	-> Roll-up
Q4	Sales by Prod by City for Categ=Elec	Only along cch, Drill-down
Q5	Sales by Prod by Country for Categ=Elec	Only along cch, Roll-up
Q6	Sales by Prod to Teenagers by State for Categ=Elec & in US	Only along cch, Slice_and_dice
Q7	Sales of Prod 1 compared with Sales of Prod 2 to Teenagers for Categ=Elec	Only along cch, Drill-down, Slice_and_dice
Q8	% increase in Sales to Teenagers over Sales to Adults, of Prod 1 / 2 for Categ=Elec & in US	Combination of is-a & cch, Drill-down, Slice_and_dice

4.1 Phase - 2 : Associated Horizontal Class Partitioning (AHCP)

The Associated Horizontal Class Partitioning (AHCP) methodology creates semantic-rich hybrid class partitions for efficient query processing. It is a technique by which several classes can be partitioned according to the semantics of another class in its aggregation hierarchy. We employ the AHCP on our ORDW schema, and propose to extend its applicability from class composition hierarchies to also include is-a hierarchies and links quantified by partial participation, thereby encompassing the Complete Warehouse Schema (CWS) in the ORDW.

4.1.1 AHCP preliminaries

The total cost of the AHCP framework can be broadly categorized as partition storage cost, partition retrieval cost and partition maintenance cost. In this paper, we also incorporate query-centric information including selectivity and frequency to determine the selection of minimal complete set of partition fragments for optimal storage, maintenance and retrieval costs.

Primary Horizontal Partitions (PHP) :

Classes in the ORDW schema can be denoted as C_i^p, indicating the *i* th class in the *p* th path. The root class (FC) is denoted as C_0. Primary Horizontal Partitions on these classes can be denoted as sub-classes and placed in the is-a hierarchy under the original partitioned class. Note that the (sub) is-a hierarchy in our examples is denoted by the subscript *i.j* , denoting the *j* th sub-class of the *i* th class (in the *p* th path). The Primary Horizontal Partitioning (PHP) operation can be denoted as:

$$PHP(C_i^p)_{p1} \rightarrow \{ C_{i.1}^p , C_{i.2}^p , \ldots, C_{i.n}^p \}$$

where (C_i^p) is the Class that is Primary Horizontally Partitioned according to a predicate (p1), resulting in *n* fragments which are treated as classes $\{C_{i.n}^p\}$. Note however, that since FC is the only root in the realm of our OLAP query sets, any primary partition of the root need not display the path suffix; i.e. ($C_{0.1}^0 = C_{0.1}$).

The example in figure 9 shows classes C_0, C_2^1 and C_1^2 in the class composition hierarchy (CCH). Some of the PHPs are { $C_{21.1}^2$, $C_{21.2}^2$ and $C_{21.3}^2$ }, connected by dashed lines (is-a) to the super-class C_1^2 which was partitioned.

As the PHPs can be considered as subclasses of the class on which the PHP were performed, they are placed in the is-a hierarchy of the schema. We can have any no. of PHPs on a single class based on a number of predicates. For a single simple predicate, the PHPs are disjoint, i.e. they do not share any

objects. PHP schemes based on multiple or complex predicates on the same class, may induce overlapping fragments, however we do not consider such schemes in this work to avoid complexity.

Associated Horizontal Class Partitions (AHCP) :

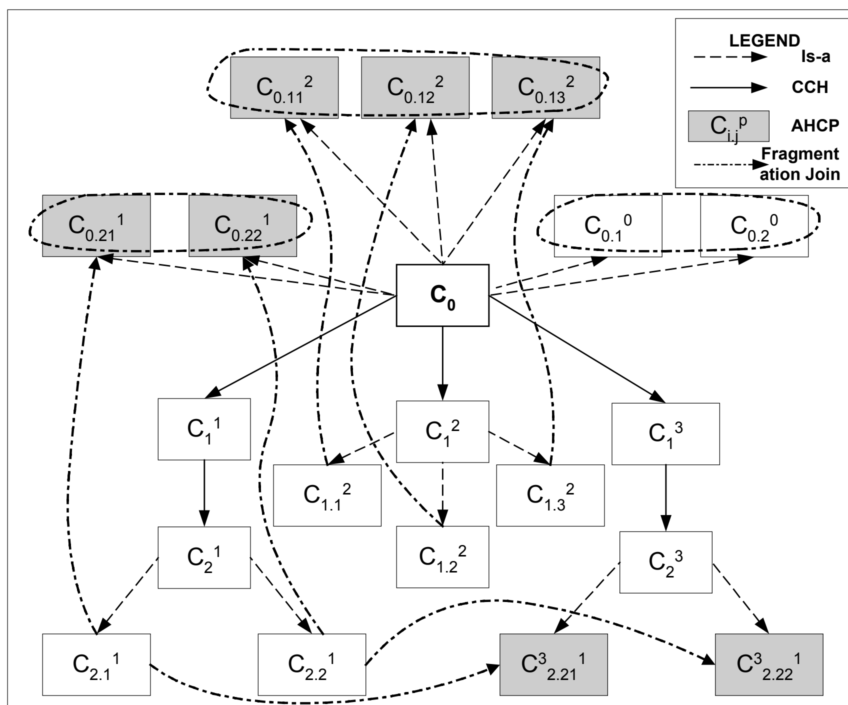

Fig. 9. An AHCP example on the Fact and Dimension Classes

After the PHPs are created, the AHCP operation may be performed on some other classes in the schema. As noted above, most queries in the SQS access the root (FC) for its value based attributes, and hence this paper deals primarily with AHCP of the root class. The AHCP operation can be denoted as follows:

$\text{AHCP}\ (C_j^q,\ \text{PHP}(C_i^p)_{p1}) \rightarrow \{ C^q_{j.i1}{}^p,\ C^q_{j.i2}{}^p,\ ...,\ C^q_{j.im}{}^p \}$ and

where (C_j^q) is the Class that is Associate Horizontally Class Partitioned (AHCP) according to the PHP on class C_i^p, resulting in *m* fragments which are treated as classes $\{ C^q_{j.im}{}^p \}$. Here again since FC is the only root in the realm of our OLAP SQSs, any AHCP of the root need not display the path suffix; i.e. ($C^0_{0.11}{}^2 = C_{0.11}{}^2$).

As seen in the figure, the examples indicate that two sets of AHCPs are created from the root C_0. They can be created by:

$\text{AHCP}\ (C_0,\ \text{PHP}(C_1^2)_{p1}) \rightarrow \{ C_{0.11}{}^2,\ C_{0.12}{}^2,\ C_{0.13}{}^2 \}$ and

$\text{AHCP}\ (C_0,\ \text{PHP}(C_2^1)_{p1}) \rightarrow \{ C_{0.21}{}^1,\ C_{0.22}{}^1 \}$

These partition fragments are denoted as subclasses in the figure by means of the shaded boxes to indicate Associate Partitioning.

The AHCP operation can also be preformed on classes other than the root, i.e. the Dimension Classes. For example, as seen in the figure, C_2^3can be AHCPed based on the PHPs of C_2^1.

$\text{AHCP}\ (C_2^3,\ \text{PHP}(C_2^1)_{p1}) \rightarrow \{ C_{3.21}{}^1,\ C_{3.22}{}^1 \}$

The result is also shown in shaded boxes under C_2^3 in the figure.

An important point to be noted here is that while the Fragments obtained by any single AHCP operation on the root are always disjoint, the same cannot be said about Fragments obtained by AHCP on any other (Dimension) class. This indicates the storage overhead to be incurred while performing AHCPs on the Dimension classes, and must be taken into account by the cost model.

4.1.2 AHCP cost model

In an ORDW, partitioning can be implemented by means of Method Induced Partitioning techniques [KL98]. Moreover due to the structural and cardinal differences inherent between Dimension Classes (DC) and the Fact Class (FC), we can assume that the DCs need not be physically partitioned as they may be wholly or partially stored in memory (under both medium and large memory hypothesis). Hence the cost of the traditional *join* between the PHP fragments and the AHCP fragments can be ignored. This *join* can be achieved by employing the methods of the FC.

Storage Cost :

The Storage cost (SC) has two components: Primary Horizontal Partition (PHP), and the Associated Horizontal Class Partition (AHCP). It can be stated as:

$$SC = SC_{PHP} + SC_{AHCP}$$

They are given as follows:

1. SC_{PHP} (C_1) :

We assume that in most cases, and especially in this paper, we consider only one PHP per class. This ensures that the partitions are disjoint for simple predicates. In such cases, there is negligible overhead for storage cost as $SC_{PHP}(C_1) = | C_1 |$ (no. of pages occupied by the class C_1 + catalog entries for the no. of PHPs of C_1). These catalog entries give details of the partitioned Class structure, extent and qualifying rules. Hence they are very small and can easily be accommodated in memory (in both the medium and large memory hypothesis).

In case of multiple complex predicates on a Dimension (C_1), resulting in overlapping fragments, we propose not to replicate the entire class extent, but rather only replicate the Class OIDs (and some frequently accessed attributes) in the separate Partitions.

In this case the storage overhead can be estimated as:

$$SC_{PHP}(C_1) = || C_1 || \times No_{Attr} \times (sizeof(Attr)) \times No_{PHP}$$

where No_{Attr} = No. of Attributes replicated.

where No_{PHP} = No. of Partition schemes.

Given a maximum of 2 replicated attributes or 20% of the class structure, and a uniform size of attributes, we can accommodate upto 5 different Partitioning schemes for an increase of 100% in SC_{PHP} (C_1).

2. SC_{AHCP} (C_0) :

This is by far the biggest increment for storage cost in the AHCP ORDW. As noted above, the root (C_0) would be the widely used as the candidate for performing AHCP. Since any predicate on a single dimension can only induce disjoint partitions in the root, the partitioning overhead is negligible for multiple partitioning schemes in a single DC.

$$SC_{AHCP}(C_0) = | C_0 | + No_{PHP} \times Size_{Cat}(PHP_i).$$

where $Size_{Cat}$ = Catalog entry size (structure, extent, qualifying rules).

But as we incorporate multiple predicates on different dimension classes, SC_{AHCP} (C_0) grows linearly as the no. of dimensions (assuming only single complex predicates on each dimension). This can be a large overhead, as C_0 as the FC, is very large (~order of Gigabytes).

Hence we intend to reduce this overhead by means of a Multiple Partition Processing Plan (MP^3), based on MVPP [YKL97]. This would entail a compromise between duplication and efficiency of the

partitions, as sub-fragments will have to be created to support the AHCPs. The Join needed to produce the final result from these sub-fragments constitutes the increase in retrieval cost.

Maintenance cost :

As noted above, since inter-fragment *join* is avoided between the PHP and AHCP fragments, maintenance cost is considerably simplified due to the AHCP operation.

As the ORDW is a read_mostly and append_only environment, we can safely estimate the maintenance cost even though the schema is vastly enhanced (and complicated) by semantics. For example, once the Warehouse has achieved full functionality, in each update cycle of the ORDW, we can expect upto 0.5% addition of the FC (this is a very conservative estimate based on our same DW, maintaining 10 years worth of "Sales" data and updated daily). The updates to DCs can be ignored mainly because their percentage will be even smaller and also because most of the DCs will be in memory anyway. Only these 0.5% FC objects have to be processed in order to maintain the Partitioning scheme.

The Maintenance cost for the AHCP partitioning scheme (MC) can be defined as the extra cost of maintaining the AHCPs and the PHPs catalogs.

$$MC = MC_{Cat}(AHCP_i) + MC_{Cat}(PHP_i)$$

Since $MC_{Cat}(PHP_i)$ is negligible as the PHPs are in memory, the main cost is on the AHCP maintenance, which is comprised of maintaining catalog entries of the AHCP, Generally this meta-information is small enough to be stored completely in memory.

Retrieval cost :

To determine retrieval cost, we break up the complex queries into smaller atomic sub-query expressions. We denote this by means of a MQO (Multiple Query Optimization) graph in the MP^3, which is further explained in section 3.3.

The Retrieval Cost (RC) is the cost of parsing the catalog, accessing the relevant AHCPs (as union) and the cost of the *join* with corresponding PHPs.

$$RC = RC_{Cat} + RC_{AHCP} + RC_{PHP} + RC_{join}$$

However, as we store the PHPs and the *join* in memory, and the Catalog is relatively small, RC is mainly composed of AHCP loading cost. Since this is smaller than the complete FC by a factor of min (sel_{pi}), where sel_{pi} indicates the selectivity of the predicates on query Q_i , we achieve a considerable savings in retrieval cost.

This saving is also obtained when indexing schemes like the SJIH [VLK00a] are built on top of the AHCPs, and also when aggregate views have to be developed.

4.1.3 AHCP selection procedure

We approach the problem of performing AHCP in the ORDW in a different manner from the case of DHCP in a normal OODB [BKS98]. IN [BKS98], various techniques (candidates) were considered to decide the best PHP candidate for performing DHCP. Here we consider all the PHP candidates, and our AHCP algorithm generates an optimal combination of complete and minimal set of AHCPs.

AHCP Algorithm (also called MP^3 algorithm)

The algorithm can be broken into three parts:

1. Generating an exhaustive set of AHCPs based on query characteristics (selectivity, fan-out) obtained from the entire query space.
   1.1 For each query Q_i in the *SQS*, generate logical associated fragments $\{C_{0,j}^p\}$ from $\{C_j^p\}$, that satisfy sub-expressions of Q_i completely.

1.2 Perform an intersection of the $C_{0.j}^{p}$ fragments for all Q_i . This creates the complete disjoint AHCP set, on which the queries will be based.

2. Assigning query weights depending on priority and importance (frequency).

2.1 For each query Q_i , evaluate the minimal set of query processing fragments : $QPF_i = \{ C_{0.j1}^{p1}, C_{0.j2}^{p2}, \ldots, C_{0.jm}^{pm} \}$

2.2 Create query plans for each Q_i having nodes involving unions of fragments that exist in multiple QPF_i .

2.3 Assign cumulative weights to the nodes depending on their utility to consecutive Q_i (based on frequency and cardinality).

3. Selecting a minimal complete set of AHCPs based on the query weights, subject to storage and maintenance cost. This part is similar to the Algorithm for selecting views to be materialized given in MVPP [YKL97].

3.1 For each Q_i , perform top-down evaluation of nodes in its query plan.

3.2 Select lower nodes (breakup) if the retrieval cost is lesser.

Figure 10 shows examples of AHCPs (AHCP-1, AHCP-2) and PHPs (PHP-1) on the Fact Class (C_0). These fragments are then merged by intersecting them and obtaining a complete disjoint set of Partitions. It must be noted that this is obtained from the query characteristics, and are very exhaustive. Due to this reason, it may not be feasible to materialize them all, and hence the MP^3 is used to determine which fragments should be materialized and which should be kept virtual [VLK98]. The cost model is based on the MVPP [YKL97], and incorporates SC and MC besides RC. As shown in the figure, the shaded classes are materialized.

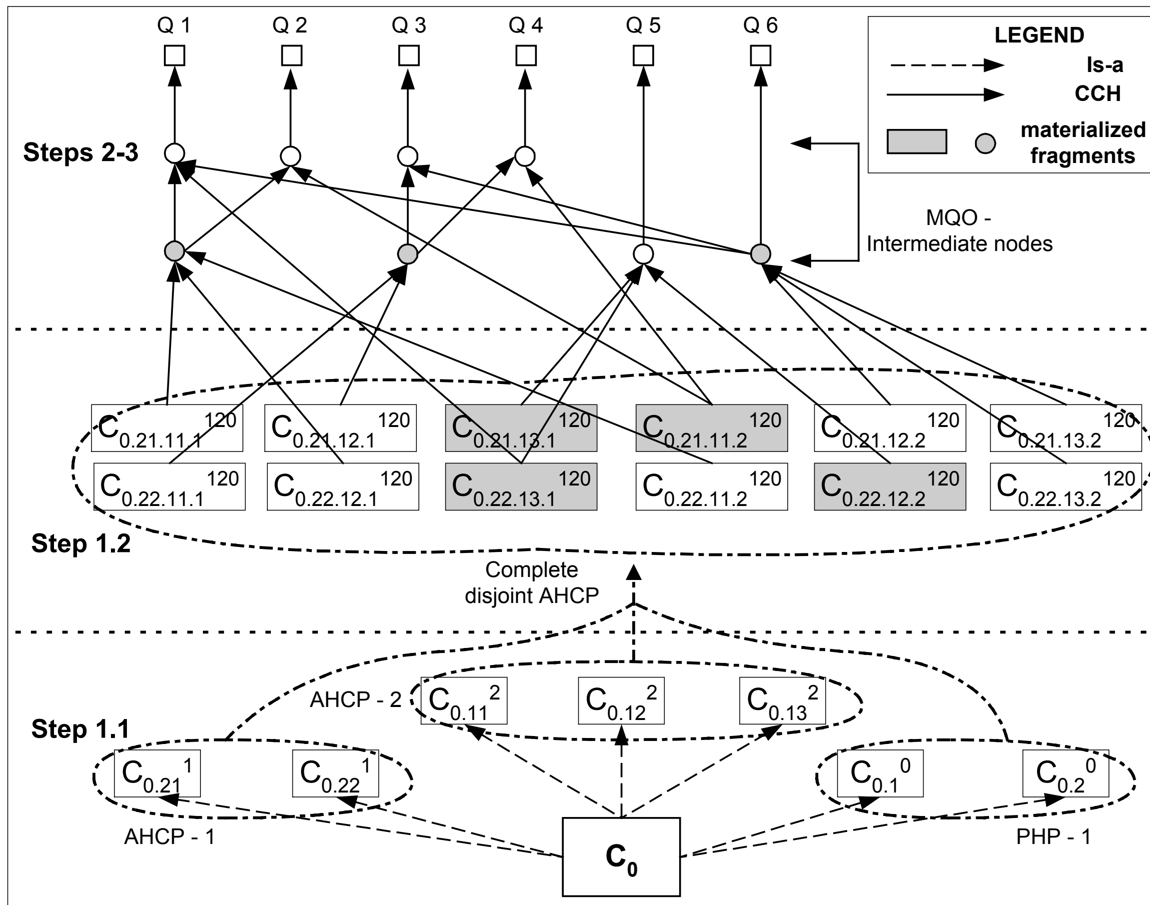

Fig. 10. Multiple Partition Processing Plan (MP3).

4.2 Phase - 3: Indexing

Though not a direct topic of this paper, we have incorporated the indexing scheme as part of our ORDW design methodology due to its complementary nature to partitioning. In particular, several indexing methods as illustrated in [OQ97], [Sar97], can be implemented on the partitioned ORV schema to facilitate complex object retrieval and to avoid using a sequence of expensive pointer chasing (join) operations. As shown in [VLK00a], a query-driven indexing approach based on the structural join index hierarchy (SJIH) mechanism [FKL98] can be very effectively devised, which can demonstrate a tremendous efficiency over plain pointer chasing approach. We omit further coverage and evaluation of the SJIH indexing scheme in this paper due to space limitation; further details can be found from [VLK00a].

5. Sample evaluation and analysis

In this section, we analyze the fragment retrieval cost for processing queries in SQS using AHCP. A comparison of the results with that of plain query processing approach using pointer chasing is then conducted.

5.1 Fragment retrieval cost

In order to evaluate the AHCP methodology, we use the sample ORDW schema and queries as detailed in section 4. Here we note that there are 8 eight queries in the SQS, and we assume them all to be of equal importance.

Running our example through the algorithm given in section 4.1.3 :

Step 1.1 : For each query Q_i in the SQS, generate logical associated fragments $\{C_{0,j}^p\}$ from $\{C_j^p\}$ that satisfy sub-expressions of Q_i completely.

We see that there are 4 main predicates, by which the Dimensions are partitioned,

viz. p1 : "Country = US", p2a : "Customer = Teen " , p2b : "Customer = Adult", p3a : "Product = P1 ", p3b : "Product = P2", and p4 : "Categ = Elec". Performing the AHCP function wrt. these PHPs as shown in section 3.1, we arrive at an exhaustive set of AHCPs of the Sales Class (FC).

Step 1.2 : Perform an intersection of the $C_{0,j}^p$ fragments for all Q_i .

Consequently, by intersection, we see that a complete set of 16 different AHCPs of the FC (Sales) can be created based on these 4 predicates, encompassing all possible and non-empty fragments:

F1	p1 ^ p2a ^ p3a ^ p4	F9	!p1 ^ p2a ^ p3a ^ p4
F2	p1 ^ p2a ^ p3b ^ p4	F10	!p1 ^ p2a ^ p3b ^ p4
F3	p1 ^ p2a ^ !p3a ^ !p3a ^ p4	F11	!p1 ^ p2a ^ !p3a ^ !p3a ^ p4
F4	p1 ^ p3a ^ !p4	F12	!p1 ^ p3a ^ !p4
F5	p1 ^ p2a ^ p3a ^ p4	F13	!p1 ^ p2a ^ p3a ^ p4
F6	p1 ^ p2a ^ p3b ^ p4	F14	!p1 ^ p2a ^ p3b ^ p4
F7	p1 ^ p2a ^ !p3a ^ !p3a ^ p4	F15	!p1 ^ p2a ^ !p3a ^ !p3a ^ p4
F8	p1 ^ p3a ^ !p4	F16	!p1 ^ p3a ^ !p4

Step 2.1: For each query Q_i of the SQS, evaluate the minimal set of query processing fragments.

The query processing fragments (QPF) are shown in the following table:

QPF_1 , QPF_2	F1, F2, F3, F4, F5, F6, F7, F8
QPF_3 , QPF_4, QPF_5	F1, F2, F3, F4, F5, F6, F7, F9, F10, F11, F13, F14, F15
QPF_6	F1, F2, F3
QPF_7	F1, F2, F9, F10
QPF_8	F1, F2, F5, F6

Step 2.2 : Create query plans for each Q_i having nodes involving unions of fragments which exist in multiple QPF_i .

The intermediate nodes are created by a combination of fragments noting their affinity in the QPFs. For the sake of completeness, we also create un-accessed nodes, for example, N12 (F12 U F16), though these fragments are not accessed by any query in the SQS.

Node	Definition	Node	Definition
N1	F1 U F2	N7	N2 U N6 U N9
N2	N1 U F3	N8	F9 U F10
N3	F5 U F6	N9	N1 U N8
N4	N2 U N3	N10	F11 U F13 U F14 U F15
N5	N3 U F7	N11	N5 U N8 U N10
N6	N5 U F4 U F8	N12	F12 U F16

Step 2.3 : Assign cumulative weights to the nodes depending on their utility to consecutive Q_i (based on frequency and cardinality).

For each of the queries Q_i, we know the optimal query processing plan op_i , which is an ordered list of nodes and fragments. We also know the frequency (f_{qi}) of each query, and the

selectivity (sel_{pj} of the clause that its (sub-query) is based on. Depending on those parameters, we give weights to the nodes in the op_i of each query.

For example, in processing for Q1, we have:

<op_1> = <N7, N6, N2, N9, N3, N5, N8, N1, F1, F2, F4, F5, F6, F7, F8, F9, F10>

$\therefore$ the weights for all these nodes (and fragments) is $f_1 * sel_1$.

Processing for Q6, we have:

<op_6> = <N2, N1, F1, F2, F3>

$\therefore$ the weights for all these nodes (and fragments) is $f_6 * sel_6$.

.. and so on.

For simplification, we consider equal frequencies and 100% selectivity in the fragments; hence at the end of this step, we have weights:

Frag	Weight	Frag	Weight	Node	Weight	Node	Weight
F1	4	F9	3	N1	4	N7	1
F2	4	F10	3	N2	2	N8	3
F3	2	F11	1	N3	3	N9	2
F4	1	F12	0	N4	1	N10	1
F5	3	F13	1	N5	2	N11	1
F6	3	F14	1	N6	1	N12	0
		F7	1	F15	1		
		F8	1	F16	0		

Step 3.1 : For each Qi , perform top-down evaluation of nodes in its query plan.

As the <opi> are ordered (tree structured), for each Qi, we can traverse the list in a top-down manner. Initially all top -level nodes can be considered marked for materialization.

Step 3.2 : Select lower nodes (breakup) if the retrieval cost is lesser.

This is a recursive step, in which the node is unmarked (for materialization) if any node under it has a weight higher than itself. In that case the lower nodes are considered marked for materialization, and the process is repeated with them.

For example, processing for Q8, we mark N4 as it is the first node:
but the weights are : N4 : 1, N2 : 2, N3 : 3.
hence N4 is discarded for N2 and N3.

Now N2 : 2, N1 : 4, F3 : 2.

So N2 is discarded for N1 and F3.
.. and so on.

Repeating this process for all the queries, the following nodes are materialized:
F3, F4, F7, F8, N1, N3, N8, N10.

This is our optimal minimal AHCP set.

Comparing HCF retrieval cost with pointer traversal cost

In this section, we evaluate our AHCP scheme for its performance gain over the un-partitioned case during query retrieval. As noted in the previous section, we have derived an optimal complete minimal AHCP set of the Sales FC.

The DCs and associated *joins* are in memory and evaluating a query branch dealing with them would involve CPU cost. This is ignored here, as the disk i/o cost is the major component of response time in most query retrieval costs.

The following study shows disk i/o cost ratios for varying relative frequencies of queries in the *SQS*.

$$\text{cost ratio (CR)} = \frac{\text{cost of disk i/o for unpartitioned case}}{\text{cost of disk i/o after AHCP}}$$

The query frequencies are varied from 10% to 90%. As these are relative frequencies, it must be noted that the frequencies of the other queries in SQS are modified equally in each case. The parameters for the study are stated in the Appendix.

Observations

Relative frequency	10%	30%	50%	70%	90%
Q1	0.05	0.15	0.25	0.35	0.45
Q2	0.05	0.15	0.25	0.35	0.45
Q3	0.03	0.09	0.15	0.21	0.27
Q4	0.03	0.09	0.15	0.21	0.27
Q5	0.03	0.09	0.15	0.21	0.27
Q6	0.02	0.06	0.1	0.14	0.18
Q7	0.01	0.03	0.05	0.07	0.09
Q8	0.01	0.03	0.05	0.07	0.09

As can be seen from the above table, there is always a minimum gain obtained when the ORDW is partitioned; the range of the gain varies from 1% to 50% in this case study.

Note that the above results appear to exhibit a linear relation between the selectivity of the query and the cost gain obtained from the AHCP operation. However this should be interpreted only as the best-case scenario, because in real-world cases some level of data replication is expected which can cause redundant data access. This may lead to higher cost for the partitioned case than what this example indicates, although the difference will not be too significant.

5.2 Resultant schema

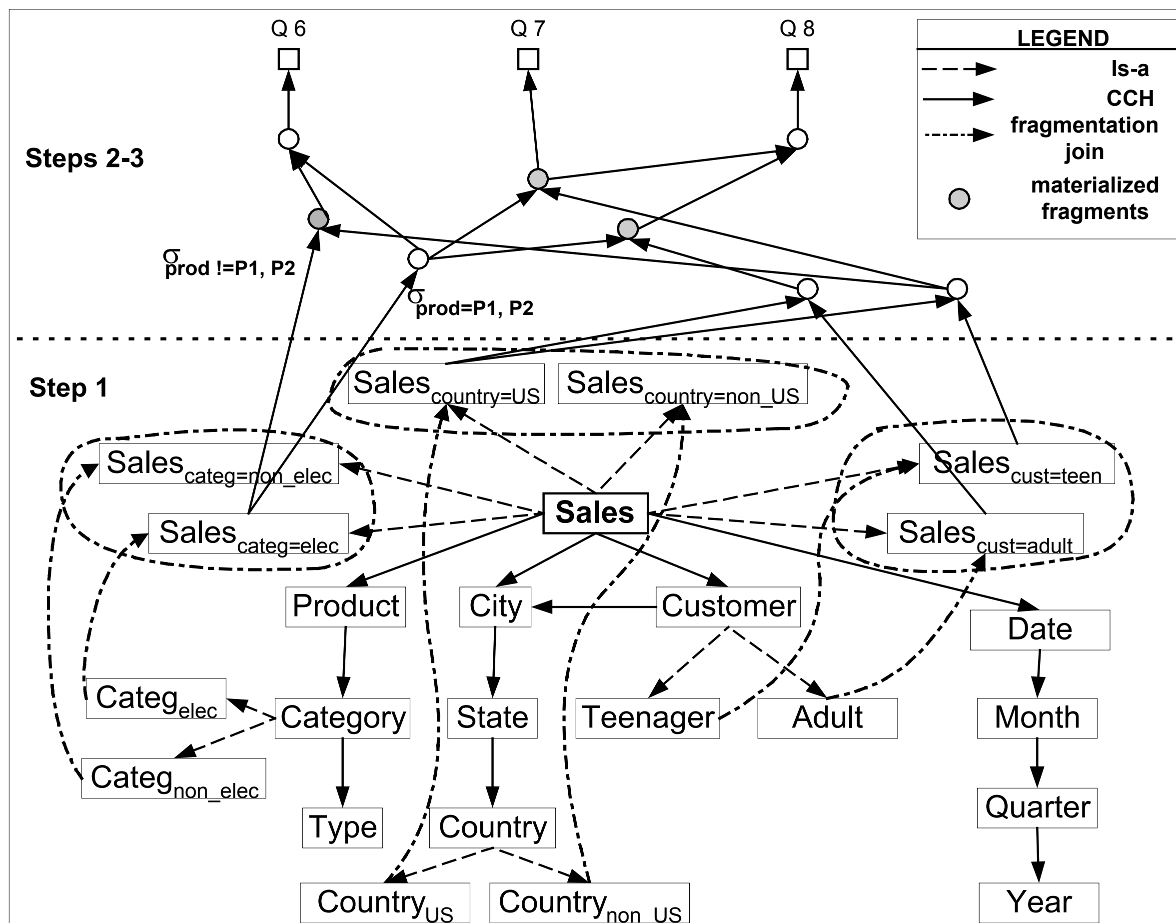

Fig. 11. The OO Schema.
The figure shows the MP3 algorithm's partial results (for queries Q6, Q7, Q8 only).

Figure 11 shows the *O-O schema* for the *Sales* example, which is generated by tracing the steps of the MP^3 algorithm step-by-step, for queries Q6, Q7 and Q8. Due to the exhaustive nature of the algorithm, it is not possible to depict all internal nodes and fragments obtained in the intermediate steps, hence only these three queries are selected without loss in generality. This schema shows PHPs on the DCs (Category, Customer and Country), while *Sales* (FC) is Associate Horizontally Class Partitioned. Step 1, shows first level AHCPs of *Sales*, viz. $Sales_{categ=elec}$, $Sales_{country=US}$, etc. The fragmentation links between PHPs and the corresponding AHCPs are also shown. Now in the next two steps of the algorithm, the intermediate nodes are created, and weighted according to selectivity and degree of sharing. As indicated in the above figure, three intermediate nodes are materialized for queries Q6, Q7 and Q8.

6. Applications of ORDW framework

Current research work on data warehouses has only focused on Single Fact schemas. Moreover, most work is concentrated in SPJ queries, some on queries with Aggregates, and very little on aggregates in the presence of hierarchies. To the best of our knowledge, no work has been done in recursive queries, or in true Multi-Fact queries, i.e. involving multiple Measures. In this paper we have presented a methodology towards efficient query processing in an object-relational data warehousing (ORDW) environment, through devising and incorporating Associated Horizontal Class Partitioning (AHCP) techniques over the ORDW schema. Our methodology starts with a given set of data warehouse queries, comes up with a near-optimal AHCP scheme for the queries, and selects AHCP fragments as materialized views to facilitate efficient evaluation of these queries. Through an initial analytical

study, we are already able to demonstrate the gains of our approach vis-a-vis the unpartitioned approach in terms of disk i/o in the ORDW environment.

The query space on DWs can be shown as in Figure 12:

```
Recursive Aggregate Views   Parameterized Aggregate Views
          |      |          |     |     |
          |      Aggregate Views  |     |
          |      |          |     |     |
          |      |          SPJ Views   |
          |      |          |     |     |
          |      |          |     |     |
   Multi – Fact      :    Star /  Snowflake schema
```

Figure 12. ORDW inter-cube/view relationships

1.1 Benefited applications

Based on the query characteristics, we present some new and innovative applications that benefit from our ORDW methodology. Specifically, these applications arise out of queries that are complex, hierarchical, or ad-hoc.

Multi-Fact queries

Consider the following multi-Fact OLAP queries that are based on two distinct *facts* FT_1 and FT_2

FT_1 (Product, Supplier, Date, Sales)

FT_2 (Category, Region, Excise_Duty)

DT_1 (Product, Category, Type)

Product (Name, Size, Colour, Weight, …)

Q_1 : (Product_by_Size, Month, ΣSales)

Q_2 : (Category, State, %of Duty)

A multi-fact OLAP query is one that has to be resolved by one or more Views, whose schema definition includes Dimensions from more than one Fact. This also implies that the Measure of interest may involve attributes from the multiple FTi 's.

Recursive OLAP queries

Consider a Census Data Warehouse whose typical applications include birth control, health/ hygiene advertising, forecasting or target marketing would involve aggregate queries on entire family trees. For example, the DW stores Personal details such as Parents, Siblings, Address, Salary, Tax paid, etc. Now, a hierarchical query, which needs to recursively navigate the Parent attribute, could be of great interest in determining a Measure (say, Salary earned) over generations of different ethnic groups determined by family ties.

Such a nested recursive query is extremely difficulty (and virtually impossible) to achieve with standard SQL, and would involve redundant temporary storage structures, and call for "external" routines (e.g., stored procedures and/or embedded approach). Whereas in an OODW, we can easily formulate a recursive query based on the expressive power of, e.g., OQL [Cat94].

Ad-hoc (parameterised) queries

Furthermore, there is also Ad-hoc Query, which need to be evaluated efficiently:

Q_3 : Get ΣSales of Large Products whose %of Duty > *p*% in Eastern States

In this case the Ad-hoc query also needs to access both Facts. In the ORDW, this query can be answered by Oids assigned to the cells of materialised views V_1 & V_2, and with a navigation of the Category→Product hierarchy. The Ad-hoc query is based on (selectivities of) distinct Measures

present in two (or more) Facts, and correspondingly two (or more) Cubes. Hence determination / computation of the ad-hoc query cannot be done by a single Cube. It is also impossible with a join of the two Cubes (in Relational sense using Views), because each cell in Q_1 is based on a range of Products that overlaps with the range of Products (Category) along cells in Q_2. Since this query is parameterised, i.e. the user (analyst) can specify different values for "*p*", materialisation has to be optimised, and view-sharing strategies have to be decided at run-time.

7. Future work & conclusion

In this paper we have presented a methodology towards efficient query processing in an object-relational data warehousing (ORDW) environment, through devising and incorporating Associated Horizontal Class Partitioning (AHCP) techniques over the ORDW schema. Our methodology starts with a given set of data warehouse queries, comes up with a near-optimal AHCP scheme for the queries, and selects AHCP fragments as materialized views to facilitate efficient evaluation of these queries. Through an initial analytical study, we are already able to demonstrate the gains of our approach vis-a-vis the unpartitioned approach in terms of disk I/O in the ORDW environment.

Note that the work we have described in this paper (hence the result obtained) should be only regarded as an intermediate stage towards efficient ORDW query processing; further advanced techniques and mechanisms should and can be naturally added. In particular, an adaptive and extensible indexing framework is currently being developed, so as to better accommodate the requirements of dynamic data warehousing [Dayal99] which demands the incorporation of more semantics into the data warehouse schemata. As shown in [VLK00a], a query-driven indexing mechanism built on the SJIH (structural join index hierarchy) [FKL98] seems to be very effective, and is supplementary to the AHCP work on materialised views [VLK00b]. We are extending the ORDW framework to include the notion of an Object *Cube*, where the Cube or aggregated view in the presence of dimensions, is treated as a Class. Currently we are in the process of combining these complementary approaches into the same framework, and are building an experimental ORDW prototype system, which will be validated by empirical studies based on, example, TPC-H benchmark queries.

References

[BK98] L. Bellatreche, K. Karlapalem and A. Simonet, "Algorithms and Support for Horizontal Class Partitioning in Object-Oriented Databases", *Distributed and Parallel Databases*, Kluwer Academic Publishers, accepted in 1998.

[Cat94] Cattell, R. et al., The Object Database Standard: ODMG-93, release 1.1, Morgan Kaufman, 1994.

[CD97] Surajit Chaudhuri and Umeshwar Dayal, "An Overview of Data Warehousing and OLAP Technology", *ACM SIGMOD Record*, 26(1), March 1997, pp. 65-74.

[CMN97] Michael J. Carey, Nelson Mendonça Mattos, Anil Nori, "Object-Relational Database Systems: Principles, Products, and Challenges (Tutorial)", *SIGMOD Conference* 1997.

[Dayal99] Umeshwar Dayal, "Dynamic Data Warehousing", *Proc. First International Conference on Data Warehousing and Knowledge Discovery (DaWaK),* Florence, Italy, 1999.

[FKL98] Chi-wai Fung, Kamalakar Karlapalem, Qing Li, "Structural Join Index Hierarchy: A Mechanism for Efficient Complex Object Retrieval", *Proc. FODO Conference 1998*, pp. 127-136.

[KL00] K. Karlapalem and Q. Li, "A Framework for Class Partitioning in Object-Oriented Databases", *Distributed and Parallel Databases*, 8(3): 317-350, Kluwer Academic Publishers, 2000.

[GHRU97] H. Gupta, V. Harinarayanan, A. Rajaraman, and J.D. Ullman, "Index Selection for OLAP", *Proc. ICDE 1997*, pp. 208-219.

[GM95] A. Gupta, and I. S. Mumick, "Maintenance of Materialized Views: Problems, Techniques, and Applications", *IEEE Data Engineering Bulletin*, June 1995.

[MK99] Mukesh Mohania and Y. Kambayashi, "Making Aggregate Views Self-Maintainable", *Data and Knowledge Engineering*, 32(1), 2000, pp: 87-109.

[NS96] Shamakant Navathe, Ashoka Savasere, "A Schema Integration facility Using Object-Oriented Data Model", in Omran A. Bukhres, Ahmed K. Elmagarmid (eds.), *Object-Oriented Multidatabase Systems: A solution for Advanced Applications*, Prentice Hall, 1996, pp. 105-128.

[OQ97] P. O'Neil, D. Quass, "Improved query performance with variant indexes", *Proc. ACM SIGMOD '97*, pp. 38-49.

[Rou97] Nick Roussopoulos, "Materialized Views and Data Warehouses", *Proc. KRDB 1997*, pp. 12.1-12.6.

[Sar97] Sunita Sarawagi, "Indexing OLAP Data", *IEEE Data Engineering Bulletin*, 1997, 20:36-43.

[VLK98] Vivekanand Gopalkrishnan, Qing Li, Kamalakar Karlapalem, "Issues of Object-Relational View Design in Data Warehousing Environment", *Proc. IEEE SMC Conference 1998*, pp. 2732-2737.

[VLK99] Vivekanand Gopalkrishnan, Qing Li, Kamalakar Karlapalem, "Star/Snow-flake Schema Driven Object-Relational Data Warehouse Design and Query Processing Strategies", *Proc. First International Conference on Data Warehousing and Knowledge Discovery (DaWaK)*, LNCS 1676, Florence, Italy, 1999, pp. 11-22.

[VLK00a] Vivekanand Gopalkrishnan, Qing Li, Kamalakar Karlapalem, "Efficient Query Processing with Structural Join Indexing in an Object Relational Data Warehousing Environment", *Proc. 11th Information Resources Management Association International Conference (IRMA'00),* Anchorage, Alaska, May 21-24, 2000 (to appear).

[VLK00b] Vivekanand Gopalkrishnan, Qing Li, Kamalakar Karlapalem, "Efficient Query Processing with Associated Horizontal Class Partitioning in an Object Relational Data Warehousing Environment", DMDW 2000, pp. 4-1 – 4-9.

[Won98] Wong Hing Kee, "Empirical Evaluation of Vertical Class Partitioning & Complex Object Retrieval in Object Oriented Databases", *M.Phil. thesis*, HKUST, 1998.

[YKL97] Jian Yang, Kamalakar Karlapalem, Qing Li, "Algorithms for Materialized View Design in Data Warehousing Environment", Proc. VLDB 1997, pp. 136-145.

Appendix

Table A. Query Parameters
f_o = fan-out
R - reference (reverse links)
$||C_i||$ - cardinality

Reference (i→j)	f_o	**R**	$\|\|C_i\|\|$	$\|\|C_j\|\|$
Sales→Product	1	100	50M	.5M
Sales→Customer	1	50	50M	1M
Sales→Teenager	1	250	50M	.2M
Sales→Date	1	500	50M	36.5K
Prod→Category	1	10	.5M	1K
Product→Retailer	50	100	.5M	50K
Category→Type	100	5	1000	10
Retailer→City	1	4	50,K	12.5K
Customer→City	1	80	1M	12.5K
Year→Mon	12	1	10	120
Mon→Date	30	1	120	3.6K
Year→Date	365	1	10	3.6K
Country→State	25	1	10	250
State→City	5	1	250	1.2K
Country→City	125	1	10	1.2K

Table B. Selectivity (%).

Country = 'US'	50
Category = 'Elec'	30
Product = 'P1'	5
Product = 'P2'	5
Customer = 'Teen'	20