Ersch. in: Kurzfassungen 8. Workshop "Grundlagen von Datenbanken", Friedrichsbrunn (Harz) (28.05.-31.05.96) / [Hrsg.: Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik]. Hrsg. Stefan Conrad ... - Magdeburg, 1996. - S. 21-25. - (Preprint / Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik ; 1996,3)

Physical Design in OODBMS

Dieter Gluche and Marc H. Scholl

University of Konstanz
Department of Mathematics and Computer Science
P.O.Box 5560/D188, D-78434 Konstanz, Germany
E-mail: {Dieter.Gluche, Marc.Scholl}@Uni-Konstanz.de
Tel.: +49 7531 88-4436 Fax: +49 7531 88-3577

1 Introduction and Motivation

Physical database design is the process of identifying the optimal internal layout of data to be stored on disk to represent the logical information contained in the logical database schema. This terminology originates from the ANSI-3-Schema-Architecture, where the idea of the separation was that the logical schema represents the global view of the whole database semantics, whereas the physical schema reflects performance considerations.

This paper describes the approach to physical database design taken by the CROQUE project. CROQUE stands for Cost and Rule-based Optimization of object QUEries, a DFG-funded joint project of the Universities of Konstanz and Rostock. The overall objective of CROQUE is to implement an ODMG-style object database engine with a very flexible storage manager and a corresponding query and update transformer and optimizer. The idea is that, for a given (logical) database schema, which is described in terms of the ODMG object definition language ODL, we optimize the *physical database schema*, that is, the stored data structures w.r.t. a given transaction load. As a consequence, the internal database layout may look rather different from its logical counterpart. Particular emphasis is put on (partial) replication of data, so as to provide different clustering strategies of the same set of objects, if that is beneficial for the overall system performance. The obvious trade-off between the benefits of replicated data for retrieval operations and the extra maintenance overhead for updates needs to be quantified (in a cost model) that goes into the optimization of the physical database design. Partially replicated storage structures are very similar to materialized views, the difference being that the former need not correlate (but often can) with user views, but may just result from analyzing the access statistics given in the transaction load.

In CROQUE, the transformations between the logical and the physical database schemas are defined in a (formal) query language, a mixture of an object algebra and object calculus (see [GS96b] for details of that language and the translation of ODMG OQL into it). The advantages of such a query language-based mapping approach are that (i) there is no need for a specific storage structure description language, and more importantly, that (ii) user queries against the logical database schema are easily translated into queries on the physically stored schema by a simple textual substitution akin to the classical view substitution mechanism. The translation of updates requires essentially the same mechanisms that are necessary for the maintenance of materialized views in relational databases, with the proper extensions for the object-oriented model. Unlike other approaches, we do not restrict the queries defining the physical schema, allowing to use an orthogonal query language. Since the mapping is described on the schema level, this information can be used for the static query optimization.

We proceed by shortly reviewing related previous work and then describing the choices that we envision for the physical database design. After that, we concentrate on the update propagation problem. Throughout this paper, we will use the term materialized view in the sense of a storage structure defined by a query over the logical database schema, that is, the physical schema consists of a bunch of materialized views.

2 Related Work

In earlier work, we already considered mapping-based physical database designs using nested relations as the physical storage description model [SPS87, Sch92]. We presented how to describe various physical design techniques, such as horizontal and vertical partitioning, materialized views, and hierarchical clustering, by means of a (nested relational) query algebra and also showed that an algebraic query rewrite optimizer can produce efficient execution plans. The GMAP tool is a recent attempt in the same direction [TSI94], they also provide a descriptive way to define physical schemas within a query language and provide query transformations and optimization. They have

also started to look into update transformation, but in general did not consider heavily replicated physical designs. Furthermore, GMAPs cover only Select-Project-Join mappings.

[GMS93] shows how to maintain views incrementally on base relations. Only flat relations are materialized. The basic idea is to detect changes to the base relations and to compute the changes to the materialized view incrementally.

3 Logical Data Model

The logical data model is described in an ODL [Cat93] like syntax. It allows to define objects, structs, complex attributes like sets, bags and lists and relationships between objects. Relationships may be defined explicitly inverse.

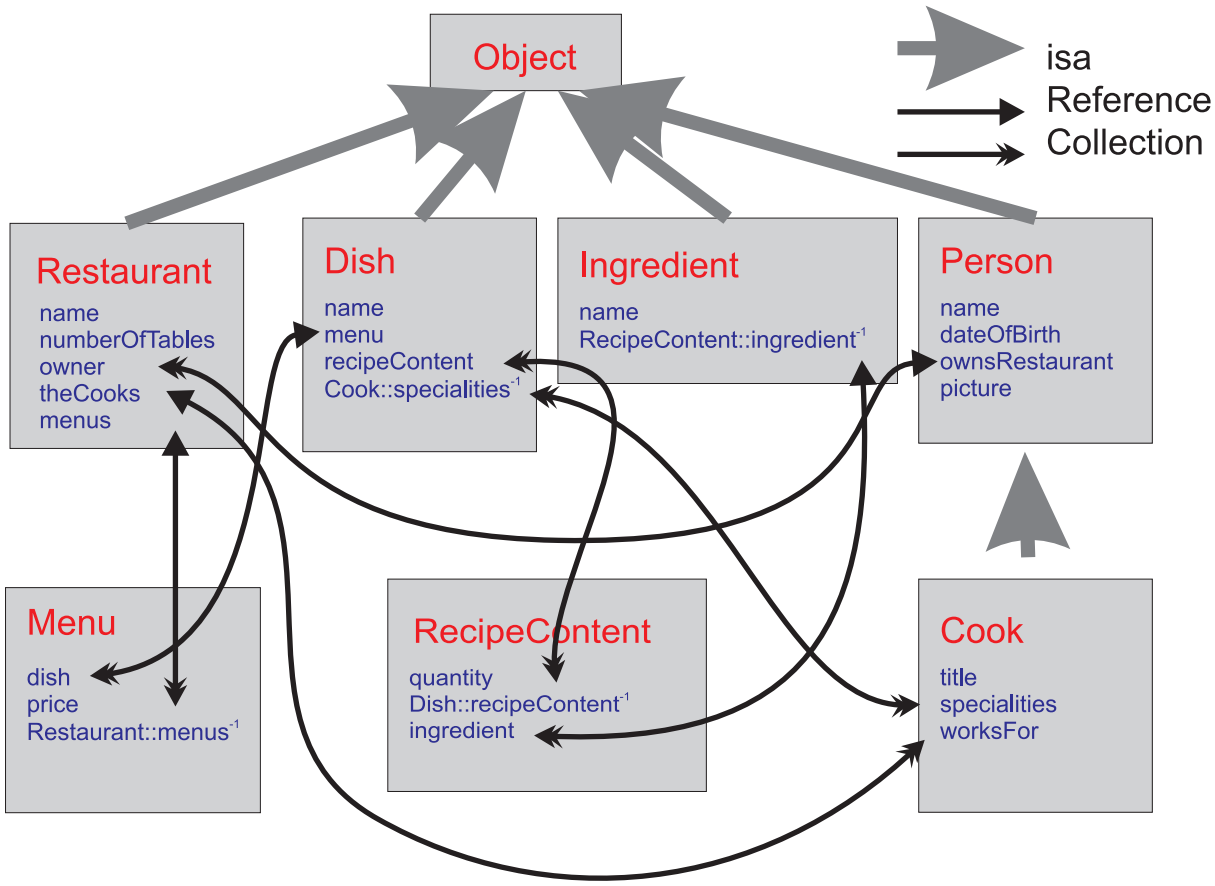

Figure 1: Example Logical Schema

Figure 1 illustrates an example of a logical database schema. The logical model can be represented completely by unary and binary relations called base relations. In the example, the logical structure type **Restaurant** will be decomposed into a unary relation *Restaurant* containing the OIDs of the extent of the type **Restaurant**, and five binary relations containing tuples of the OID and the value of a property. E.g. *Restaurant::name* contains tuples of OIDs and Strings, *Restaurant::theCooks* contains tuples of OIDs of Restaurants and OIDs of Cooks and so on. The inverse relationship *Cook::worksFor* contains the same as information *Restaurant::theCooks* with swapped columns. If there is no inverse relationship defined in the logical schema, this attribute is added by the system, e.g. *Cook::specialities*

4 Physical Schema and Materialized Views

The physical schema is described as the tuple *PDB* (physical DataBase), containing all physical materializations of the database. Each materialization is a non-recursive view based on logical base relations and on previously defined physical views.

4.1 Mappings between Logical and Physical Schema

Base relations need not be stored as unary or binary relation, defined in the logical schema. To guarantee that the mapping form the logical to the physical schema is lossless, there must exist queries against the physical schema that compute each logical base relation. We denote these queries as *inverted queries*. Since it is allowed to define several views with a high degree of replication, there might exist choices to build the inverted queries.

When a logical query is fired against the logical schema, each logical base relation is substituted by the choices of its corresponding inverted query. This is the starting point for query optimization.

4.2 Splitting Collections

Splitting objects can be defined by an OQL selection. The next example defines three materializations splitting the extent of *Restaurant* in two overlapping physical collections.

```
allRestaurants :    select <r.#>
                    from   Restaurant as r
bigRestaurants :    select <r.#>
                    from   Restaurant as r
                    where  r.numberOfTables > 10
```

In contrast to ODL we use $<$ ... $>$ to define a tuple construction, while # denotes the logical OID value. We distinguish between the logical OID and the physical address of a stored object, unlike many implementations of OODBMSs, since the unique logical OID value lets us identify replicas.

4.3 Indexing and Clustering

Each materialized collection is stored as clustered index. Indexes specify how to store a collection, while the query specifies what to store. Different storage structures are offered, e.g. hash tables, B-trees, lists. Each index clusters its elements in its corresponding order. Furthermore the default storage structure is the physical bag. A bag clusters its elements without any order.

```
allRestaurants :    btree[name]
                    select <r.#, r.name>
                    from   Restaurant as r
```

In the example *allRestaurants* denotes a clustered B-tree (organized by the attribute *name*) containing the OIDs and the names of all restaurants of the extent. Each materialized attribute can be used for indexing, so updates to elements can be recognized easily by the local index instance and reorganization can be performed. Since we allow orthogonal combination of materialization operators, indexes may be nested.

4.4 Splitting Objects

Splitting an object can be defined by a simple projection on the logical structure. This projection may be included in the select clause. The physical projections may overlap. To merge a decomposed object, the logical OID can be stored inside of the physical object.

```
personDates    :    select <p.#, p.name, p.dateOfBirth >
                    from   Person as p
personPictures :    select <p.#, p.name, p.picture >
                    from   Person as p
```

4.5 Physical Addresses

To speed up merges of replicas, physical addresses pointing to physical objects can be stored. This allows to link replicas of the same logical object:

```
personPictures :    select <p.#, p.name, p.picture >
                    from   Person as p
personDates    :    select <p1.#,
                            @1:  element(select p2.@
                                         from personPictures as p2
                                         where p2.# = p1.#
                                        ), p1.name, p1.dateOfBirth>
                    from   Person as p1
```

The second view definition uses a subquery against the physical view *personPictures*. The internal attribute @ is used to select the physical address of a materialized tuple. The next query merges the object with the OID *#100* using the stored address. * denotes the pointer dereferencing of @1.

```
element(select <p.#, p.name, p.dateOfBirth, p.*@1.picture>
        from   PersonDates as p
        where  p.# = #100) ;
```

Though all materializations of collections are implemented as clustered indexes, it is easy to emulate secondary indexes, through the use of addresses.

4.6 Combining Related Objects

In many cases it makes sense to cluster related objects. Therefore the nested objects have to be included in the projection. In this example the restaurant objects are stored with its owner and all its cooks inside.

```
restaurants : select <r.#, theOwner: <r.owner.#>
                      theCooks: select <c.#> from r.theCooks as c>
              from Restaurant as r
```

4.7 Materialized Joins

Flat joins can be materialized based on top level collections, e.g. the extents, or on local collections. This technique allows to build path indexes and access support relations [KM94].

```
restaurants : btree[cook.#]
              select <restaurant: <r.#>, cook: <c.#>>
              from Restaurant as r, r.theCooks as c
```

The example shows a materialized natural join between restaurants and their corresponding cooks. The collections is organized as a B-tree. It can be considered as a access support relation.

In general, path indexes are based on local collections. To build a path index inside an object, a materialized join can be nested within the select clause.

4.8 Orthogonality

Each of the presented operations can be combined orthogonally. This leads to nesting in the select, from, and where clause of the defining OQL queries.

5 Performing Updates

Representing the logical schema as unary and binary relations is also amenable to decompose complex logical update operations into elementary update operations against the unary and binary relations.

Furthermore, many logical elementary updates can be collected before an (incremental) update operation is executed physically on a materialized view. With the collections of elementary update operations we can determine if a part of a view is involved in this update.

For update propagation, it has been shown in [GS96a] how incremental view maintainance can be applied to our physical storage structures. Some nested queries can be computed incrementally using the materialized information in the nesting. Finally, some open problems were shown where incremental computation fails. In these cases, some replicated parts of the physically stored objects need to be recomputed.

6 Conclusions

If object database systems are to take data independence serious, the physical database design needs to provide a high degree of freedom to prevent that physical design decisions (i.e., performance issues) have to be moved to the logical schema (i.e. the semantic level). We have sketched the CROQUE physical data model which allows to describe clustering, nesting, replication, indexing and partitioning, and orthogonal combinations thereof, each with or without redundancy. With the definition of pointers we are able to build fast access paths and connect physical materialized objects via pointers between views. Since we use stable addresses, no pointer reorganization is required on object level.

The transformations between the logical and the physical database schema are defined in a descriptive way, using a formal query language. Hence, query transformation is easily established by substitution, similar to classical view queries.

References

[Cat93] R.G.G. Cattell. *The Object Database Standard: ODMG - 93, Release 1.1*. Morgan Kaufman Publishers, 1993.

[FVM92] J.-C. Freytag, G. Vossen, and D.E. Maier, editors. *Query Processing for Advanced Database Applications*. Morgan Kaufman Publishers, 1992.

[GMS93] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proc. ACM SIGMOD Conf.*, page 157, Washington, DC, May 1993.

[GS96a] Dieter Gluche and Marc H. Scholl. Physical OODB Design Exploiting Replication. Technical Report in preparation, Department of Mathematics and Computer Science, Database Research Group, University of Constance, 1996.

[GS96b] Torsten Grust and Marc H. Scholl. Translating OQL into Monoid Comprehensions — Stuck with Nested Loops? Technical Report 3/1996, Department of Mathematics and Computer Science, Database Research Group, University of Konstanz, March 1996.

[KM94] Alfons Kemper and Guido Moerkotte. *Object-Oriented Database Management: Applications in Engineering and Computer Science*. Prentice–Hall, 1994.

[Sch92] M.H. Scholl. *Physical Database Design for an Object-Oriented Database System*, chapter 14, pages 419–447. [FVM92], 1992.

[SPS87] Marc H. Scholl, H.-Bernhard Paul, and Hans-Jörg Schek. Supporting flat relations by a nested relational. In *Proceedings of The 13th International Conference on Very Large Data Bases*, pages 137–146, 87.

[TSI94] Odysseas G. Tsatalos, Marvin H. Solomon, and Yannis E. Ioannidis. The GMAP: A Versatile Tool for Physical Data Independence. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 367–378, Santiago, Chile, 1994.