Book Title
Book Editors
IOS Press, 2006

Combining Associative and Navigational Access in Persistent Object Stores

Markus Kirchberg, Weena Nusdin and Alexei Tretiakov
Information Science Research Centre and Department of Information Systems, Massey University, Private Bag 11 222, Palmerston North 5301, New Zealand
[M.Kirchberg | A.Tretiakov]@massey.ac.nz

Abstract. We introduce an approach allowing to support queries involving both navigation along references (i.e. navigational access) and imposition of conditions on object data (i.e. associative access) in Persistent Object Stores. Our approach does not directly rely on the knowledge of the database schema and of the queries to be supported, and thus can be used to support ad-hoc querying of semistructured data. The basic access structures it uses (i.e. selector meta-objects representing sets of values connected via references to database objects and references) are stable with respect to changes in database data in a sense that changes in the value of one object do not necessitate changes of the elements of the support structure facilitating access to other objects. In addition, we propose an extension of the matrix index approach to support navigational access such that main memory space usage is decreased. Also, we demonstrate how matrix index coding can be enhanced further to support the combined navigational and associative access by providing guarantees for encoded object sequences by referencing selector meta-objects.

Keywords. Persistent Object Store, Querying of Semistructured Data, Associative and Navigational Access

1. Introduction

The area of providing access to stored data is one of the important research topics of database systems. It relates to several functions of a database management system (DBMS) such as query languages (QLs) and secondary storage management.

[1] defines that a QL is '*a notation for expressing queries, coupled with a mechanism for assigning meaning to the expression*'. In other words, a QL enables users to pose questions about data in a database (DB) . For object-oriented data models (OODMs) there have been a number of QLs proposed, including algebraic, calculus, logic programming oriented and SQL-like approaches [1].

In relation to the secondary storage management of an object-oriented database system (OODBS), which affects the efficiency of retrieval of requested data from a DB, there are several issues concerned. Such issues include the need to build auxiliary data structures (i.e. indices) that speed up the search for requested data. Efficient data access approaches reduce the number of I/O operations needed and the time spent during the search for requested data in secondary (or tertiary) storage devices. The criteria used to evaluate the performance of indices are for example the cost to search for requested data

using indices and the storage size required to store indices [3,8,10,22]. The cost performance of indices can be estimated in terms of the number of I/O operations required to perform index-based data lookup. In particular, if the size of indices is small enough for them to fit into the main memory, the lookup time is dramatically improved. Therefore, minimising the index size is of prime importance.

As commonly agreed, queries in an OODBS are supported by three types of data access including direct, navigational and associative access [8,10,22,9]:

- Direct access is retrieval of a set of objects via their explicitly given object identifiers (OIDs).
- Navigational access is retrieval of a set of OIDs based on navigation along references between objects beginning from any given object.
- Associative access is retrieval of a set of OIDs that meet conditions specified on object values.

Associative access in an OODBS often includes navigational access to referenced or referencing objects of the object being accessed [8]. It was found that the existing approaches proposed for associative access are tied to specific paths in the object-oriented schema, and do not perform well in case of ad-hoc queries of semi-structured or close to semi-structured data [15].

Our research addresses these problems. In this paper, we study how associative access can be supported in a persistent object store (POS) regardless of paths or inheritance hierarchies involved with a query. We propose extensions to a model of a POS such that approaches that are proposed for navigational access can be used to support associative access in the extended POS. The extensions include:

1. Clustering storage objects in a POS on their classes or values of attributes;
2. Clustering references in a POS on classes or values of attributes of objects they connect; and
3. Providing guarantees for groups of references, and for objects they connect.

The rest of the article is organised as follows. Section 2 discusses the related work on indexing OODBMS and XML data. Sections 3 and 4 introduce a generic POS-level data model and the QL used in this paper. Sections 5 and 6 describe an approach to minimise access to permanent storage via clustering objects and references by using selector meta-objects. Section 7 extends the matrix-index coding approach to POS navigation, and demonstrates how the extended approach can be used to provide guarantees for groups of objects and references that can be used in query optimisation. Section 8 discusses further optimisations achieved by replacing references to meta-objects by flags. Section 9 discusses a proof of concept implementation. Finally, Section 10 concludes our article.

2. Related Work

There have been a number of data access approaches proposed for OODBSs. For direct access - that is the simplest type of data access - [22] suggests that it may be implemented by associative access approaches used in RDBSs such as B-tree or hashing indices. This is because the direct access can be compared with associative access to data records in a RDBS via search key values.

Examples of the approaches that have been proposed to support navigational access in an OODBS are navigation index [22], ring and spider data structures [18], object skeletons [5], join index hierarchies [4,21], triple-node hierarchies [12], and Matrix-Index Coding (MIC) [10]. The join index hierarchies and triple-node hierarchies are modifications of join indices used in a RDBS. Object skeletons maintain networks of object IDs as structures to support navigation among objects. In the ring and spider data structures, an object stores reference pointers that indicate referenced or referencing objects of the object. The navigation index and MIC employ in-memory calculation techniques by appending codes with IDs of objects. The code of an object can be expanded to a set of object IDs of referenced or referencing objects of the object.

Among the approaches proposed to support navigational access, MIC employs relatively simple and efficient concepts compared with others. The MIC works like join indices but allows to retrieve a sequence of object IDs, rather than a single object ID in single index access by applying an encoding technique based on simple continuous fractions [10].

[9] suggests that the original concept of the navigation index and MIC can be generalised by making the navigation index and MIC independent from a coding technique used, i.e. any appropriate coding technique can be selected to guarantee optimal performance in different circumstances. Especially, data compression techniques should be considered as alternative coding techniques because they can reduce the storage size of the navigation index and MIC.

The approaches that have been proposed to support associative access in an OODBS are for example Single-Class (SC) and Class-Hierarchy (CH) indices [8], H-tree [11], hierarchy class Chain (hcC) tree [17], Class-Division (CD) index [16], nested, path and multi indices [3], access support relations [7], and Nested-Inherited index (NIX) [2]. These approaches are modifications of approaches used in a RDBS such as B+ tree and join indices. In these approaches, a class is regarded as a relation and a logical object, which is an object defined in a data model, is regarded as a tuple over a relation. A UID, which is a system-defined unique identifier for a tuple, is regarded as an *OID* of a logical object. Unfortunately, these approaches cannot perform well when associative access includes navigational access involved with multiple paths over attribute/inheritance hierarchies [5].

Recently, the related problem of indexing XML documents received a lot of attention (see e.g. [20,23,14] for some recent publications, and [13] for a recent review). Most of the existing approaches to XML indexing take advantage of the tree structure of XML documents, and support queries relying on that structure. On the other hand, if references are considered, XML can also be viewed as a general graph, resulting thus in a situation similar to the one in OODBSs. [15] is the first publication on XML indexing that takes that view; it offers an approach based on indexing all possible paths (to support ad-hoc queries of semi-structured data). The consequence is that the resulting index is extremely large. [19] and [6] improved the approach applied by [15] by restricting the paths to be indexed, resulting in smaller indices, but made little progress into another problem associated with indexing paths: the resulting index structures are highly unstable with respect to updates of the underlying data: an update at a single location in the data graph may affect how other locations reachable from it via indexed paths are represented in the access structure, leading to the necessity to rebuild a large part of it.

Defined Attributes:

dom(class) = OODBSchemaClass
dom(number) = Integer
dom(name) = String

$value_1$ *: {class: paper, number: 157367}*
$value_2$ *: {class: student, name: Brown}*
$value_3$ *: {class: student, name: Smith}*

Figure 1. An instance of the POS-level data model.

In this article, we offer an approach based on clustering object IDs, references, and join index entries in extended matrix-index coding (which we introduce). The access structures we propose do not directly rely on the knowledge of the underlying schema or queries they are to support, and thus support ad-hoc querying of semi-structured data while still offering a possibility of tuning for specific data and access patterns, and are stable with respect to changes in the data, namely, small data updates never necessitate dramatic restructuring of indices.

3. Data Model

The POS-level data model is defined as follows. Objects are pairs $(OID, value)$, where OID is a unique identifier (which can appear in no more than one such pair), and $value$ is the object's data, which from the point of view of POS has no structure. References are ordered pairs of object ID's (OID_1, OID_2). Object IDs used in references should be valid objects (and thus, presented as $(OID, value)$ pairs). Thus, from the point of view of POS, the DB is structured as a labelled directed graph with objects acting as vertices, values - as labels, and references - as edges. For simplification, we do not label references, and assume that any references with attached data are reified to objects.

Attributes are strings with associated domains. We require that all of the employed domains have partial or total orders defined on them (if no natural partial order exists for a domain, we will use a partial order in which each element can only be compared, and is equal, to itself.) For each $value$, there is a mapping defined assigning to each attribute from a finite set of attributes constant values from their respective domains, resulting in a tuple representing the object's state. For a given $value$, we will say that the attributes forming the domain of such a mapping are “defined" for the $value$. For a given $value$, attributes that are not defined are referred to as “undefined". For an OID, defined and undefined attributes are the attributes defined or (respectively) undefined for its $value$. Attributes representing an object do not necessarily assume a unique set of values: different objects may result in the same representing tuple. An example is shown in Figure 1.

4. Query Language

Approaches to object clustering introduced in this article are general enough to support a variety of QLs. To verify its viability, we demonstrate its application in case of a simple QL based on path expressions. We assume that the service provided by POS is limited to returning subgraphs of the data graph, so we consider a language that does not create new objects and references. (Certainly, POS would also offer an interface allowing to modify data. However, we assume that views of the data graph offered by queries are limited to its subgraphs.) This implies that joins can be realised via navigation, namely, objects that may need to be joined in queries are already connected via references.

In the QL, a select query is comprised of two sequences: a navigation sequence and a filter sequence. The navigation sequence is a sequence in the form $(O_1^n, R_{1,2}^n, O_2^n, R_{2,3}^n, O_3^n, ..., R_{m-1,m}^n, O_m^n)$, where O_n^i are object navigation selectors given as expressions in the form $attr(O_i^n) \in sset(O_i^n)$, where $attr(O_i^n)$ is a set of attributes and $sset(O_i^n)$ is a subset of the Cartesian product of these attributes' domains. $R_{i-1,i}^n$ are reference navigation selectors given as expressions in the form $attr_1(R_{i-1,i}^n) \bigcup^{disjoint} attr_2(R_{i-1,i}^n) \in sset(R_{i-1,i}^n)$, where $attr_1(R_{i-1,i}^n)$ and $attr_2(R_{i-1,i}^n)$ are sets of attributes, while $sset(R_{i-1,i}^n)$ is a subset of the Cartesian product of these attributes' domains.

An object navigation selector O_i^n evaluates to $true$ on a given OID if all attributes in $attr(O_i^n)$ are defined for the OID, and the tuple they form once they are assigned values (on the $value$ for this OID) is an element of $sset(O_i^n)$.

A reference navigation selector R_i^n evaluates to $true$ on a reference (OID_1, OID_2) if attributes in $attr_1(R_{i-1,i}^n)$ and $attr_2(R_{i-1,i}^n)$ are defined on OID_1 and OID_2, respectively, and the tuple formed by concatenating their values is a member of $sset(R_{i-1,i}^n)$.

A filter sequence should be of the same length as the navigation sequence. The syntax for filter sequences is defined similarly to navigation sequences.

In the following, we often use the term "selector set" to refer to a set of tuples on some attribute set. A selector set may be infinite, and it does not need to be defined by explicitly listing all tuples it contains.

Select queries are evaluated as follows. The navigation sequence is matched to all possible paths in the graph. It matches if all selectors for object IDs and for references connecting them match. For each matched path, the filter sequence is applied, and objects and references for which selectors in the filter sequence match are included in the result of the query. However, for a given match of a path via the navigation sequence, references for which some of the objects they connect are not included after applying the filter sequence are also not included. We observe that selectors on references may be regarded as realising join conditions.

Let us consider some examples. Let objects representing papers be connected by references to objects representing students enrolled. Then a query returning students enrolled in paper number 157367 can be formulated as follows. The navigation sequence consists of three selectors. The first selector would select objects belonging to the class $paper$, with number 157367 (attribute set is $\{class, number\}$, and selector set is $\{\{class : paper, number : 157367\}\}$). The second navigation selector is a trivial selector evaluating to $true$ on all references. The third navigation selector would evaluate to $true$ on all objects of class $student$. Now, the filter sequence would include two selectors that always evaluate to $false$ to get rid of the $paper$ and the references, and the

last selector that always evaluates to $true$ to include all students matched by the navigation sequence. As a variation, one could choose not to filter anything out in the filter sequence, with the query returning the paper, the students, and the references connecting the paper to the students (which, perhaps, at the layer using the POS services could be assembled into a higher level object representing the paper and encapsulating the references to students as part of the object, and into a number of separate higher level objects representing the students.) Finally, if we were to request a number of papers, with students enrolled, rather than a single paper, the query would effectively realize a kind of a join, and the result would be a directed graph with some of the students possibly shared by more than one paper.

As a second example, consider a DB in which objects represent cities, and references represent flights. Consider the following query: return all cities in Russia that can be visited if one flies from London to Tokyo with two stopovers, such that the time difference between any of the subsequent stops is less than 6 hours. To formalise this query, one needs selectors for navigation sequence that would evaluate to $true$ on references if objects they connect have attribute values for attribute *diffgmt* (difference from GMT) close enough (condition that is easy to represent as set membership). In addition, selectors for the filter sequence should ensure that only cities in Russia are returned, with objects corresponding to other cities and references representing the flight connections left out.

It is easy to extend the select queries to allow infinite or bound number of repetitions, as in XML XPath and in regular expressions. However, as far as object and reference clustering is concerned, such extensions do not introduce new issues, so we limit ourselves to the relatively restricted definition of select queries given above. For the same reason we omit queries involving set operations on results of select queries, and queries involving semi-joins for vertices not connected via references.

5. Selector Sets and Selector Meta-Objects

Let us require that sets used in selectors are such that membership and set inclusion tests are relatively inexpensive. This is the case if selector sets are hypercubes formed as Cartesian products of intervals $[low, high]$, with low and $high$ constants taken from domains of attributes used in the selector (one interval for each attribute). We require that $low \leq high$, which implies that in case of partially ordered domains (such as class hierarchy) low and $high$ are comparable. We observe, that a conjunction of equalities can be represented as a selector with a hypercube selector set for which on all attributes $high = low$.

In addition, for a set of attributes with domains forming a Euclidean space, we may form selector sets by delimiting areas with hyper-planes. Finally, we allow selector sets formed by explicitly enumerating a small number of points, selector sets obtained as unions of a small number of selector sets (formed for the same attribute set), and selector sets obtained as Cartesian products of selector sets (formed for disjoint attribute sets).

While object IDs are relatively small, we expect that the $value$ part of a POS object may be rather large. Therefore, we assume that POS object values are stored on permanent storage. Generally, to retrieve attribute values for an object $value$ (to evaluate query selectors), one needs to retrieve the $value$ from permanent storage, which is computationally very expensive. Therefore, there is a need to maintain in the main memory some

auxiliary structures that would allow one to minimise the number of objects for which the $value$ has to be retrieved. Such auxiliary structures should determine the smallest possible superset of objects that need to be retrieved from permanent storage (with the ideal auxiliary structure allowing to retrieve only the objects that need to be returned by the query). In fact, as it should have been clear from the introduction, the focus of this article is on defining such structures.

To define auxiliary structures, we rely on meta-objects representing selector sets, which we call selector objects. The $value$ of such a POS object would encode the definition of a selector set. We do not dwell on how such encoding could be achieved, as it is quite easy to define (e.g. for a selector which is a hypercube, one could store a flag indicating that this is a hypercube, and for each attribute store attribute name and domain together with $high$ and low values). Selector objects can be retrieved by the query compiler, which can then take advantage of various guarantees provided via references to and from a selector object OID. We observe here, that while POS is unaware of the structure of values for objects encapsulating DB data, it has to be aware of the structure of selector meta-objects, hence, for POS, selector meta-objects should be distinguishable from objects encapsulating DB data.

6. Clustering Objects and References

To facilitate the execution of queries involving evaluation of selectors, we add references between objects and selector meta-objects. We will refer to references to a selector meta-object as upstream references, and to references from a selector meta-object as downstream references. In this section, we assume that references are maintained as join indices: sequences of object IDs pairs (OID_1, OID_2) sorted by OID_1, with two join indices provided: one for references between objects encapsulating DB data, and another one for references to and from meta-objects. In addition, we maintain a join index allowing to associate selector meta-objects with references as a sequence of object triplets $(OID_1, OID_2, MOID)$, where (OID_1, OID_2) is the reference, and $MOID$ is the ID of the associated meta-object. In the following section we will extend our consideration to a more general access structure based on MIC.

Let us consider how selector meta-objects can be applied to support select queries (see Figure 2). To match the first selector in navigation sequence, the query compiler would find a selector meta-object for which the selector set contains the first selector set in the navigation sequence of the query, while being as close to it as possible. For attribute sets forming Euclidean spaces, closeness can be judged by comparing enclosing volumes. For some attribute domains, closeness can be difficult or impossible to define, then, a random choice can be made. For the chosen meta-object, downstream references are followed, resulting in a superset of object IDs matching the first selector in navigation sequence. For all of these objects, upstream references are followed to determine if there is a meta-object for the current object with a selector set contained by the selector set from the navigation sequence of the query, which provides a guarantee that the object ID matches the query selector and should be retained for further processing. For the rest of the objects in the superset, values are retrieved from permanent storage, attribute values are matched against the selector set from the query, and only object IDs actually matching the selector are retained for further processing of the query. We note that if

Figure 2. Matching a path involving two objects and a reference by using selector meta-objects.

selector meta-object suitable for assembling the initial superset were not available, the query engine would have to retrieve values of all DB objects, which from performance point of view is clearly not acceptable.

The rest of the query is executed by accepting or rejecting objects and references on paths of the graph according to navigation and filter sequences by following upstream references and verifying if there is a meta-object containing the selector set (for a reference or for an object) with the selector set contained by the selector set from the query. If such a meta-object is available, it provides a guarantee that the reference (or the object) matches the selection criteria. Otherwise, object values are retrieved and attribute values are matched against the selectors directly. Object values already retrieved may be cached. If an object with a cached value is available, upstream references are not followed, and determination is made based on matching attribute values against the selectors directly. This approach is applied for the rest of the navigation sequence (apart from the first match), and for the whole of the filter sequence (which is applied only if all of the navigation sequence has matched).

The use of selector meta-objects allows minimising the necessity to retrieve values from permanent storage. It is the job of the query optimiser to generate suitable meta-objects for objects and references. Providing an algorithm for achieving it is beyond the scope of this article and remains a topic for further research.

The cost of employing selector meta-objects in terms of main memory space can be estimated as

$$(2N_{DOREF} + 2N_{UOREF} + 3N_{URREF}) * sizeof(OID) \tag{1}$$

where N_{DOREF} is the number of downstream references to objects, N_{UOREF} is the number of upstream references from objects, and N_{URREF} is the number of upstream references to meta-objects from references between DB objects.

The cost of matching a navigation sequence in terms of execution time can be estimated as

$$t_{om} + \frac{(l_{path} + n_{uom}) * t_{om} + (l_{path} - 1 + n_{urm}) * t_{rm}}{2} \tag{2}$$

where t_{om} is the time required to access the join index that matches DB objects to selector meta-objects, t_{rm} is the time required to access the table matching references between DB objects to selector meta-objects, l_{path} is the length of the sequence, n_{uom} is

the number of upstream references from objects in the sequence, and n_{urm} is the number of upstream references from references in the sequence.

If an object is added, removed or updated, upstream and downstream references from and to this object are added or removed as needed, but (unlike in approaches involving indexing paths) it does not directly affect how other objects are reflected in the access structure. Also, addition or removal of a reference between DB objects affects only upstream references to meta-objects from that reference. Hence, the access structure based on using selector meta-objects is relatively stable with respect to changes in DB data. On the other hand, to maintain the integrity of the access structure, it is essential that selector meta-objects with downstream references are referencing all of the objects they match. Therefore, in case of value updates or object addition, updates to the access structure can not be deferred.

7. Extended Matrix-Index Coding Approach and Navigation Guarantees

The MIC approach [10,9] allows to speed up navigation by replacing the second object ID in a join index (OID_1, OID_2) by a compressed sequence of object IDs: $(OID_1, encode(OID_2, OID_3, ..., OID_n))$. The sequence is formed as follows: for an object with two or more outgoing references (the so-called branching object) the object IDs of reference ends are encoded in arbitrary order. For an object OID_1 with a single outgoing reference, followed (along this reference) by a number of similar objects, the sequence of object IDs is encoded in the order they are encountered when moving from OID_1 to the next branching object whose ID is placed as the last ID in the encoded sequence. To distinguish entries corresponding to branching objects from entries corresponding to objects with a single outgoing reference, a flag is attached to each join index entry. MIC allows to retrieve several references via a single access to the join index (references are reconstructed from the decoded sequence).

Here, to benefit from guarantees provided via selector meta-objects, we generalise the MIC technique to make it more flexible. While in its original definition, the matrix-index approach suggests that encoding is applied in all cases when it is applicable (so that an OID appears in the left column of the resulting join index only once), we relax this requirement. For example (refer Figure 3), in case of OID_1 connected directly to OID_2, OID_3, OID_4 and OID_5 (case of a branching object), we would allow the situation where OID_1 enters the join index twice: once as $(OID_1, encode(OID_2, OID_3))$, and another time - as $(OID_1, encode(OID_4, OID_5))$. The advantage of such an approach is that the query optimiser can apply the encoding as needed for fastest operation. For example, for a long sequence of non-branching object IDs that is often navigated through starting from the first object ID, the whole sequence can be encoded only once and stored with the object used to retrieve it. There is no need to encode partial sequences many times, with the entire object IDs in the sequence, thus saving significant amount of memory space. The cost for such flexibility is that even if an object ID enters the join index only once, to retrieve all references starting from that object ID two accesses to the join index are necessary (the second one to confirm that there are no further entries). However, as the join index is sorted by the object ID by which it is accessed such additional costs are likely to be very minimal.

Another significant benefit of the more flexible approach to MIC is the ability to provide navigation guarantees via selector objects. With each entry of the join index we

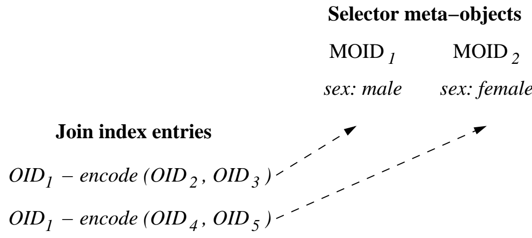

Figure 3. Extended MIC: Using selector meta-objects to provide guarantees for encoded objects.

associate a number of object IDs of selector meta-objects, so that each join index entry now appears as

$$(OID_1, encode(OID_2, OID_3, ..., OID_n), \{MOID_1, MOID_2, ..., MOID_m\})$$

(we expect that n would normally be much greater than m). We require that all object IDs under $encode$ should match all of the selector meta-objects in the same join index entry. (This approach can be easily extended to allow providing guarantees for all references in a join index entry.)

As an example, consider again a DB in which objects representing papers are connected via references to objects representing students, with some of the students not enrolled in any papers. Consider a query that would retrieve all female students enrolled in papers (it is easy to see that such a query can be expressed by using the QL introduced above). To support queries of this kind, the query optimiser would create two join index entries for each paper, one for male students, and another one for female students, and add object IDs of the appropriate meta-objects to the join index entries. Then, when executing the query the system would not navigate along the entries for which a guarantee is provided that all of the objects in them are in a wrong category (in our case, male).

8. Further Optimisation by Using Flags Instead of References

To check if an object has an upstream reference to a certain selector meta-object, access to the join index connecting objects to meta-objects is required. Further optimisation can be achieved by adding one more level of indirection: selector meta-object IDs are matched to flags, which are stored with object IDs. We require that sizes of such flags would be smaller or equal to the size of an object ID. (Ideally, flags should be much smaller than object IDs. It is trivial to introduce flags equal in size to object IDs, as IDs of meta-objects themselves can be used as flags.) This way, the availability of an upstream reference to a meta-object can be verified every time the OID is available without any further accesses to auxiliary structures. The drawback of such an approach is that for a single object participating in many references the OID can enter the join index many times, so that compared to using upstream references the approach using flags may turn out to be more expensive in terms of main memory space. However, both approaches could be used simultaneously. The query optimiser could then choose the currently most efficient approach based on the number of references, usage patterns etc.

We note that flags can also be used in conjunction with the join index to provide guarantees for references and for join index entries (with flags attached to "target" object

IDs in join index entries, and to join index entries themselves). In case of references, redundancy is possible if matrix-index encoding is applied (as then, the same reference can be reflected in the index many times). On the other hand, for simple join indices each reference enters the index only once, so that the overhead associated with using flags is minimal. For join index entries, using flags never leads to redundancy, and since in that case flags are replacing meta-object IDs that are larger than the flags, using flags is likely to be advantageous in most cases.

9. Proof of Concept Implementation

A limited version of the approach outlined above (involving the use of selector meta-objects for object IDs only without the selection of references) was implemented as a C++ program, and proved to be effective in executing a variety of queries involving both navigation and conditions on object attributes without accessing object data.

10. Conclusions

We introduced an approach allowing to support queries involving both navigation along references (navigational access) and imposing conditions on object data (associative access) at the level of Persistent Object Stores (POS). Our approach does not directly rely on the knowledge of the DB schema and of the queries to be supported, and thus can be used to support ad-hoc querying of semistructured data. The basic access structures it uses (selector meta-objects representing sets of values connected via references to DB objects and references) are stable with respect to changes in DB data in a sense that changes in the value of one object do not necessitate changes of the elements of the support structure facilitating access to other objects.

In addition, we proposed an extension of the MIC approach to navigational access allowing to save main memory space, and demonstrated how MIC can be enhanced to support the combined navigational and associative access by providing guarantees for encoded object sequences by referencing selector meta-objects.

As a topic for further research we suggest combining the approach based on referencing selector meta-objects with the existing approaches based on directly indexing navigation paths to complement the support for ad-hoc queries and semistructured data with the ability to configurably target specific queries and data patterns.

References

[1] ABITEBOUL, S., AND BEERI, C. The power of languages for the manipulation of complex values. *The VLDB Journal 4*, 4 (1995), 727–794.

[2] BERTINO, E. An indexing technique for object-oriented databases. In *Proceedings of the 7th Conference on Data Engineering* (1991), IEEE Computer Society, pp. 160–170.

[3] BERTINO, E., AND KIM, W. Indexing techniques for queries on nested objects. *IEEE Transactions on Knowledge and Data Engineering 1*, 2 (1989), 196–214.

[4] HAN, J., XIE, Z. A., AND FU, Y. Join index hierarchy: An indexing structure for efficient navigation in object-oriented databases. *IEEE Transactions on Knowledge and Data Engineering 11*, 2 (1999), 321–337.

[5] HUA, K. A., AND TRIPATHY, C. Object skeletons: An efficient navigation structure for object-oriented database systems. In *Proceedings of the 10th Conference on Data Engineering* (1994), IEEE Computer Society, pp. 508–517.

[6] KAUSHIK, R., SHENOY, P., BOHANNON, P., AND GUDES, E. Exploiting local similarity for indexing paths in graph-structured data. In *Proceedings of the 18th Conference on Data Engineering* (2002), IEEE Computer Society, pp. 129–140.

[7] KEMPER, A., AND MOERKOTTE, G. Access support in object bases. In *Proceedings of the SIGMOD Conference on Management of Data* (1990), H. Garcia-Molina and H. V. Jagadish, Eds., ACM Press, pp. 364–374.

[8] KIM, W., KIM, K.-C., AND DALE, A. Indexing techniques for object-oriented databases. 371–394.

[9] KIRCHBERG, M., KUCKELBERG, A., SCHEWE, K.-D., AND TRETIAKOV, A. On coding navigation paths for in-memory navigation in persistent object stores. In *Proceedings of the 19th Brazilian Symposium on Databases* (2004), S. Lifschitz, Ed., UnB, pp. 259–268.

[10] KUCKELBERG, A. The matrix-index coding approach to efficient navigation in persistent object stores. In *Workshop on Foundations of Models and Languages for Data and Objects* (1998), pp. 99–112.

[11] LOW, C. C., OOI, B. C., AND LU, H. H-trees: a dynamic associative search index for oodb. In *Proceedings of the SIGMOD conference on management of data* (1992), ACM Press, pp. 134–143.

[12] LUK, F. H.-W., AND FU, A. W. Triple-node hierarchies for object-oriented database indexing. In *Proceedings of the 7th conference on information and knowledge management* (1998), ACM Press, pp. 386–397.

[13] LUK, R. W., LEONG, H. V., DILLON, T. S., CHAN, A. T., CROFT, W. B., AND ALLAN, J. A survey in indexing and searching XML documents. *Journal of the American Society for Information Science and Technology 53*, 6 (2002), 415–437.

[14] MENG, X., JIANG, Y., CHEN, Y., AND WANG, H. Xseq: an indexing infrastructure for tree pattern queries. In *Proceedings of the SIGMOD conference on management of data* (2004), ACM Press, pp. 941–942.

[15] MILO, T., AND SUCIU, D. Index structures for path expressions. In *Proceeding of the 7th International Conference on Database Theory* (1999), Springer-Verlag, pp. 277–295.

[16] RAMASWAMY, S., AND KANELLAKIS, P. C. OODB indexing by class-division. In *Proceedings of the SIGMOD conference on management of data* (1995), ACM Press, pp. 139–150.

[17] SREENATH, B., AND SESHADRI, S. The hcC-tree: An efficient index structure for object oriented databases. In *Proceedings of the 20th Conference on Very Large Data Bases* (1994), Morgan Kaufmann Publishers Inc., pp. 203–213.

[18] SUBIETA, K. LOQIS: The object-oriented database programming system. *Lecture Notes in Computer Science 504* (1991), 403–421.

[19] VAGENA, Z., MORO, M. M., AND TSOTRAS, V. J. Twig query processing over graph-structured xml data. In *Proceedings of the 7th Workshop on the Web and Databases* (2004), ACM Press, pp. 43–48.

[20] WANG, H., AND MENG, X. On the sequencing of tree structures for XML indexing. In *Proceedings of the 21st Conference on Data Engineering* (2005), IEEE Computer Society, pp. 372–383.

[21] XIE, Z., AND HAN, J. Join index hierarchies for supporting efficient navigations in object-oriented databases. In *Proceedings of the 20th Conference on Very Large Data Bases* (1994), Morgan Kaufmann Publishers Inc., pp. 522–533.

[22] ZEZULA, P., AND RABITTI, F. Object store with navigation acceleration. *Information Systems 18*, 7 (1993), 429–459.

[23] ZOU, Q., LIU, S., AND CHU, W. W. Ctree: a compact tree for indexing XML data. In *Proceedings of the 6th workshop on Web information and data management* (2004), ACM Press, pp. 39–46.