Evaluating Functional Joins Along Nested Reference Sets in Object-Relational and Object-Oriented Databases*

Reinhard Braumandl Jens Claussen Alfons Kemper

Lehrstuhl für Informatik, Universität Passau, 94030 Passau, Germany
⟨lastname⟩@*db.fmi.uni-passau.de* — *http://www.db.fmi.uni-passau.de/*

Abstract

Previous work on functional joins was constrained in two ways: (1) all approaches we know assume references being implemented as physical object identifiers (OIDs) and (2) most approaches are, in addition, limited to single-valued reference attributes. Both are severe limitations since most object-relational and all object-oriented database systems do support nested reference sets and many object systems do implement references as location-independent (logical) OIDs. In this work, we develop a new functional join algorithm that can be used for any realization form for OIDs (physical or logical) and is particularly geared towards supporting functional joins along nested reference sets. The algorithm can be applied to evaluate joins along arbitrarily long path expressions which may include one or more reference sets. The new algorithm generalizes previously proposed partition-based pointer joins by repeatedly applying partitioning with interleaved re-merging before evaluating the next functional join. Consequently, the algorithm is termed $P(PM)^*M$ where P stands for partitioning and M denotes merging. Our prototype implementation as well as an analytical assessment based on a cost model prove that this new algorithm performs superior in almost all database configurations.

*This work was supported in part by the German National Research Foundation DFG under contracts Ke 401/6-2 and Ke 401/7-1.

Proceedings of the 24th VLDB Conference
New York, USA, 1998

1 Introduction

Inter-object references are one of the key concepts of object-relational and object-oriented database systems. These references allow to directly traverse from one object to its associated (referenced) object(s). This is very efficient for navigating within a limited context—so-called "pointer chasing"-applications. However, in query processing a huge number of these inter-object references has to be traversed to evaluate so-called functional joins. Therefore, naive pointer chasing techniques cannot yield competitive performance. Consequently, several researchers have investigated more advanced pointer join techniques to optimize the functional join. First of all, there are approaches to materialize (i.e., precompute) the functional joins in the form of generalized join indices [Val87]. [BK89] was the first proposal for indexing path expressions, and the access support relations [KM90] are another systematic approach for materializing the functional join along arbitrarily long path expressions; it was later augmented to join index hierarchies [XH94]. Many more proposals exist by now.

[SC90] were the first who systematically evaluated three pointer join techniques (naive, sorting, and hash partitioning) in comparison to a value-based join. Their work was augmented by [DLM93] to parallel database systems. [DLM93] also allowed nested sets of references, but they did not report on how to re-establish the grouping of the nested sets after performing the functional join. [CSL+90] incorporated some of the key ideas of pointer joins in the Starburst extensible database system. The emphasis in this work was on supporting hierarchical structures, i.e., one-to-many relationships, with hidden pointers. [GGT96] concentrate on finding the optimal evaluation order for chains of functional joins. Thus it complements our work: We devise an algorithm to efficiently evaluate a functional join within the query engine while they are concerned with finding the best evaluation order.

Unfortunately, the previous work on functional joins was constrained in two ways: (1) all approaches we know assume references being implemented as physical object identifiers (OIDs) and (2) most approaches are, in addition, limited to single-valued reference attributes. Both are severe limitations since most object-relational and all object-oriented database systems do support nested reference sets for modelling many-to-many and one-to-many object associations and many object systems do implement references as location-independent (logical) OIDs. In this work, we develop a new functional join algorithm that can be used for any realization form for OIDs (physical or logical) and is particularly geared towards supporting functional joins along reference sets. The algorithm can be applied to evaluate joins along arbitrarily long path expressions which may include one or more reference sets. The new algorithm generalizes previously proposed partition-based pointer joins by repeatedly applying partitioning with interleaved re-merging before evaluating the next functional join. Consequently, the algorithm is termed $P(PM)^*M$ where P stands for partitioning and M denotes a merging.

Before going into the technicalities let us motivate our work by way of a simplified order-inventory example application. In an object-oriented or object-relational schema the *LineItems* that model the many-to-many relationship between *Orders* and the ordered *Products* would most naturally be modelled as a nested set within the *Order* objects:[1]

```
create type Order as (                                create type Product as (
  OrderNumber number,                                   ProductID number,
  LineItems set(tuple( Quantity number,                 Cost number,
                 ProductRef ref(Product)))              ...);
  ...);
```

Here, *Order* refers to the ordered *Product*s via a nested set of references in attribute *LineItems*. Let *Orders* be a relation (or type extension) storing *Order* objects. Then, in an example query we could retrieve the *Orders*' total values:

```
select o.OrderNumber, (select sum(l.Quantity * l.ProductRef.Cost)
                       from o.LineItems l))
from Orders o;
```

Logically, the query starts at each *Order o* and traverses via the nested set of references to all the ordered *Products* to retrieve the *Cost* from which the total cost of the *Order* is computed. Note that, unlike in a pure (flat) relational schema, the nested set of *LineItems* constitutes an explicit grouping of the *LineItems* belonging to one *Order* that is, in most systems, also maintained at the physical level. Our new algorithm exploits this physically maintained grouping. However, we do, of course, avoid the danger of "thrashing" that is inherent in a naive nested loops pointer chasing approach. Our prototype implementation as well as a comprehensive analytical assessment based on a cost model prove that this new algorithm performs superior in almost all configurations. In particular, our $P(PM)^*M$-algorithm performs very well even for small memory sizes.

[1] Throughout the paper we will use some (pseudo) SQL syntax that is close to the commercial ORDBMS product that we used for comparison purposes. Unfortunately, the commercial ORDBMS products do not entirely obey the SQL3 standard. Length limitations prevent us from additionally showing the standardized ODMG object types and OQL queries [CBB+97]—but they are very similar.

The remainder of this paper is organized as follows: In Section 2 we give a short overview of physical and logical OID realization techniques. Section 3 overviews the known algorithms and introduces our new $P(PM)^*M$-algorithm. Section 4 explains the integration of our algorithm into our iterator-based query engine and gives an initial performance comparison. In Section 5 we present a more comprehensive performance analysis based on a cost model by comparing the algorithms for different database configurations. Section 6 concludes the paper.

2 Realization of Object Identifiers

Object identity is a fundamental concept to enable object referencing in object-oriented and object-relational database systems. Each object has a unique object identifier (OID) that remains unchanged throughout the object's life time. There are two basic implementation concepts for OIDs: physical OIDs and logical OIDs [KC86].

2.1 Physical Object Identifiers

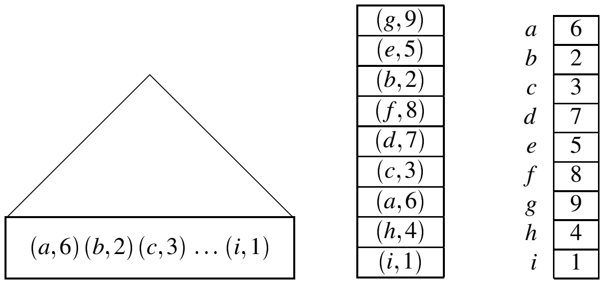
Physical OIDs contain parts of the initial permanent address of an object, e.g., the page identifier. Based on this information, an object can be directly accessed on a data page. This direct access facility is advantageous as long as the object is in fact stored at that address. Updates to the database may require, however, that objects are moved to other pages. In this case, a place holder (forward reference) is stored at the original location of the object that holds the new address of the object. When a moved object is referenced, two pages are accessed: the original page containing the forward and the page actually carrying the object. With increasing number of forwards, the performance of the DBMS gradually degrades, at some point making reorganization inevitable. O_2 [O2T94], ObjectStore [LLOW91], and (presumably) Illustra [Sto96, p. 57] are examples of commercial systems using physical OIDs.

2.2 Logical Object Identifiers

Logical OIDs do not contain the object address and are thus location independent. To find an object by OID, however, an additional mapping structure is required to map the logical OID to the physical address of the object. If an object is moved to a different address, only the entry in the mapping structure is updated. In the following, we describe three data structures for the mapping. [EGK95] give details and a performance comparison.

2.2.1 Mapping with a B^+-Tree

The logical OID serves as key to access the tree entry containing the actual object address (cf. Figure 1 (a)). In this graph, a letter represents a logical OID and a number denotes the physical address of the corresponding object (e.g., the object identified by a is stored at address 6). Here, we use simplified addresses; in a real system the address is



(a) B^+-tree (b) Hash-Table (c) Direct Mapping

Figure 1: Mapping Techniques

composed of page identifier and location within that page. For each lookup, the tree is traversed from the root. Alternatively, if a large set of sorted logical OIDs needs to be mapped, a sequential scan of the leaves is possible. Shore [CDF+94] and (presumably) Oracle8 [LMB97] are systems employing B-trees for OID mapping.

2.2.2 Mapping with a Hash Table

The logical OID is used as key for a hash table lookup to find the map entry carrying the actual object address (cf. Figure 1 (b)). For example, Itasca [Ita93] and Versant [Ver97] implement OID mapping via hash tables.

2.2.3 Direct Mapping

The logical OID constitutes the address of the map entry that in turn carries the object's address. In this respect, the logical OID can be seen as an index into a vector containing the mapping information. Direct mapping is immune to hash collisions and always requires only a single page access (cf. Figure 1 (c)). Furthermore, since the logical OIDs are not stored explicitly in the map, a higher storage density is achieved. Direct mapping was used in CODASYL database systems and is currently used in BeSS [BP95].

3 Functional Join Algorithms

The subsequent discussion of the algorithms is based on the following very simple abstract schema:

```
create type R_t as (             create type S_t as (
    R_Data char(200),                S_Attr number,
    SrefSet set(ref(S_t)),           S_Data char(200),
    ...);                            ...);
create table R of R_t;           create table S of S_t;
```

The example queries we wish to discuss are the following—one with an aggregation, the other without:[2]

```
select r.R_Data,                 select r.R_Data,
    (select sum(s.S_Attr)            (select s.S_Attr
     from r.SrefSet s)                from r.SrefSet s)
from R r;                        from R r;
```

[2]Note that the query on the right-hand side is not standard SQL because the nested query returns a set of tuples. However, some ORDBMS products do already support this—and in OQL this query is also possible (in a slightly different syntax, though).

3.1 Known Algorithms

3.1.1 The Naive Pointer-Chasing Algorithm

The naive, pointer chasing algorithm scans *R* and traverses every reference stored in the nested set *SrefSet* individually. For logical OIDs, first the *Map* is looked up to obtain the address of the referenced *S* object which is then accessed. If the combined size of the *Map* and *S* exceeds the memory capacity this algorithm performs very poorly.

In a system employing physical OIDs the naive algorithm does not need to perform the lookup in the *Map*. However, the access to the page the physical OID is referring to may reveal that the object has moved to a different page. In this case, the *forward* pointer has to be traversed in order to retrieve the object. Again, the algorithm performs very poorly if the size of *S* exceeds the memory capacity.

3.1.2 The Flatten-Algorithms

These algorithms flatten (unnest) the *SrefSet* attribute and partition or sort the flat tuples to achieve locality. For logical OIDs the evaluation plan looks as follows:

$$\nu_{S\_Attr:SattrSet}((\mu_{SrefSet:Sref}(R) \bowtie_{\leadsto} Map) \bowtie_{\leadsto} S)$$

Here, $\mu_{SrefSet:Sref}$ denotes the unnest (flatten) operator which replicates the *R* objects and replaces the set-valued attribute *SrefSet* with the single-valued attribute *Sref*. The nest operator $\nu_{S\_Attr:SattrSet}$ forms a set-valued attribute *SattrSet* from *S_Attr* [SS86]. The functional join is denoted by $\bowtie_{\leadsto}$ to indicate that for every (left) argument the corresponding join partner of the right argument is "looked up." To perform the two functional joins with the *Map* and with *S*, respectively, two techniques can be applied to achieve locality: partitioning and sorting.

If partitioning is applied, the flattened *R* tuples are partitioned such that each partition refers to a memory-sized partition of the *Map*. Upon replacing the logical OID in *Sref* by the address obtained from the *Map* the tuples are once again partitioned for the next functional join with *S*. Instead of partitioning, one could also sort the flattened *R* tuples. For the *Map* lookup the tuples are sorted on the *Sref* attribute and for the second functional join they are sorted on the addresses of *S* objects.

The final ν (nesting) operation is evaluated by grouping the flattened *R* tuples based on the OIDs of the original *R* objects. Grouping can be done by a sort-based or by a hash-based algorithm.

For physical OIDs the evaluation plan omits the first functional join with the *Map*.

3.1.3 Value-Based Join

The value-based join plan is as follows:

$$\nu_{S\_Attr:SattrSet}(\mu_{SrefSet:Sref}(R) \bowtie_{R.Sref=S.OID} S)$$

We assume that every object "knows" its OID—here *S.OID*. Note that this plan is equally applicable for logical and physical OID realizations because the object references are not traversed but only compared.

3.2 The Partition/Merge-Algorithm $P(PM)^*M$

The partition/merge-algorithm is an adaptation of the above flatten/partition-algorithm in the way that it retains the grouping of the flattened R tuples across an arbitrary number of functional joins. This is achieved by interleaving partitioning and merging in order to retain (very cheaply) the grouping after every intermediate partitioning step. This is captured in the notation $P(PM)^*M$. We will first describe the basic $P(PM)^1M$-algorithm which is applied when evaluating a single functional join under logical OIDs. More intermediate PM-steps are needed when longer functional join chains are evaluated (cf. Section 3.4).

In the $P(PM)^1M$-algorithm two joins are performed: (1) R is joined with the *Map* to replace the logical OIDs by their physical counterparts and (2) the result is joined with S. For evaluating the joins we will adapt the hash join algorithm. The *probe input* is R for the first join and R with the logical OIDs replaced by their physical counterparts—then called *RM*—in the second phase. Unlike the original hash join algorithm, only the probe input is explicitly partitioned.[3] The *build input*, i.e., the *Map* and S, are either faulted into the buffer or—if range partitioning is applied—loaded explicitly (i.e., prefetched) into the buffer. In both cases, however, a partitioning step for the *Map* and S involving additional disk i/o is not required.

The successive steps of the partition/merge-algorithm can be visualized as follows:

$$\begin{array}{ccccc} \text{flatten and partition} & \to & \text{join with } \mathit{Map} \text{ to obtain } \mathit{RM} & \to \cdots \\ R\ N\text{-way} & & \text{and partition } \mathit{RM} & \\ & & (N*K)\text{-way} & \end{array}$$

$$\cdots \to \begin{array}{c} \text{re-merge } \mathit{RM} \text{ to } K \text{ partitions} \\ \text{and join with } S \end{array} \to \text{merge}$$

That is, the partition/merge-algorithm first flattens the R objects and partitions them, then applies the mapping from logical to physical OIDs, partitions the resulting *RM*, then re-merges the initial partitioning and performs the join with S, and finally merges the partitions to restore the over-all grouping of the flat R tuples belonging to the same R object.

We need two partitioning functions h_M and h_S:

- h_M partitions the *Map* into N memory-sized chunks by mapping logical OIDs of S to the partition numbers 1 to N and
- h_S partitions S into K memory-sized chunks by mapping addresses of S objects to the partition numbers 1 to K.

That is, *Map* is partitioned into partitions $M_1, \ldots, M_N$ and S into $S_1, \ldots, S_K$. Actually, these partitioning functions are not applied on *Map* and S but on the logical OIDs stored in the nested sets of R and on their physical counterparts—in *RM* after applying the mapping.

In more detail, the algorithm performs the following four steps:

1. Flatten the nested *SrefSet*s and partition the flat R objects/replicas into N partitions, denoted $R_1,\ldots,R_N$. That is, for every object $[r,\{Sref_1,\ldots,Sref_l\}] \in R$ generate the l flat tuples $[r,Sref_1]$, …, $[r,Sref_l]$ and insert these tuples into their corresponding partitions $h_M(Sref_1), \ldots, h_M(Sref_l)$, respectively. Of course, the R attributes (*R_Data* in our example query) need not be replicated. It is sufficient to include them in one of the flat tuples or, often even better, to leave them out and re-merge them at the end (cf. Section 3.5). The partitions are written to disk.
2. For all $1 \le i \le N$ do:
   - For (every) partition R_i the K initially empty partitions denoted $RM_{i1}, \ldots, RM_{iK}$ are generated.
   - Scan R_i and for every element $[r,Sref] \in R_i$ do:
     - Replace the logical OID *Sref* by its physical counterpart *Saddr* obtained (probed) from the i-th partition M_i of the *Map*.
     - Insert the tuple $[r,Saddr]$ into the partition RM_{ij} where $j = h_S(Saddr)$.

   Note that all OID mapping performed in this step concerns only partition M_i of the *Map*, which is either prefetched or faulted into the buffer.

   Having completed step 2., all the $N*K$ partitions $RM_{11},\ldots,RM_{1K}, RM_{21},\ldots, RM_{NK}$ are on disk.
3. For all $1 \le j \le K$ do:
   - Scan the N partitions $RM_{1j},\ldots,RM_{Nj}$ simultaneously and merge them into a single tuple stream. The merging is done to restore the grouping of the flat R tuples according to R OIDs; that is, the merging generates the tuple stream $[r_1,\ldots], \ldots, [r_1,\ldots], [r_2,\ldots], \ldots$.
   - For every tuple $[r,Saddr]$ the functional join with S is performed by looking up the S object at location *Saddr* and the relevant information, here *S_Attr*, is retrieved. Insert the tuple $[r,S\_Attr]$ into partition RMS_j.

     All S objects referenced in this step belong to the j-th partition S_j of S which is prefetched or faulted into the buffer—again, the partitioning ensures that the entire S_j fits into memory.

   After completion of step 3., the K partitions $RMS_1,\ldots,RMS_K$ are on disk.
4. Scan all partitions $RMS_1,\ldots,RMS_K$ simultaneously and re-assemble the flat tuples into the nested representation, i.e., group the tuples according to R-OIDs.

[3] For simplifying the presentation, we assume that the partitioning can be done in one recursion level—however, this is not required for the algorithm to work.

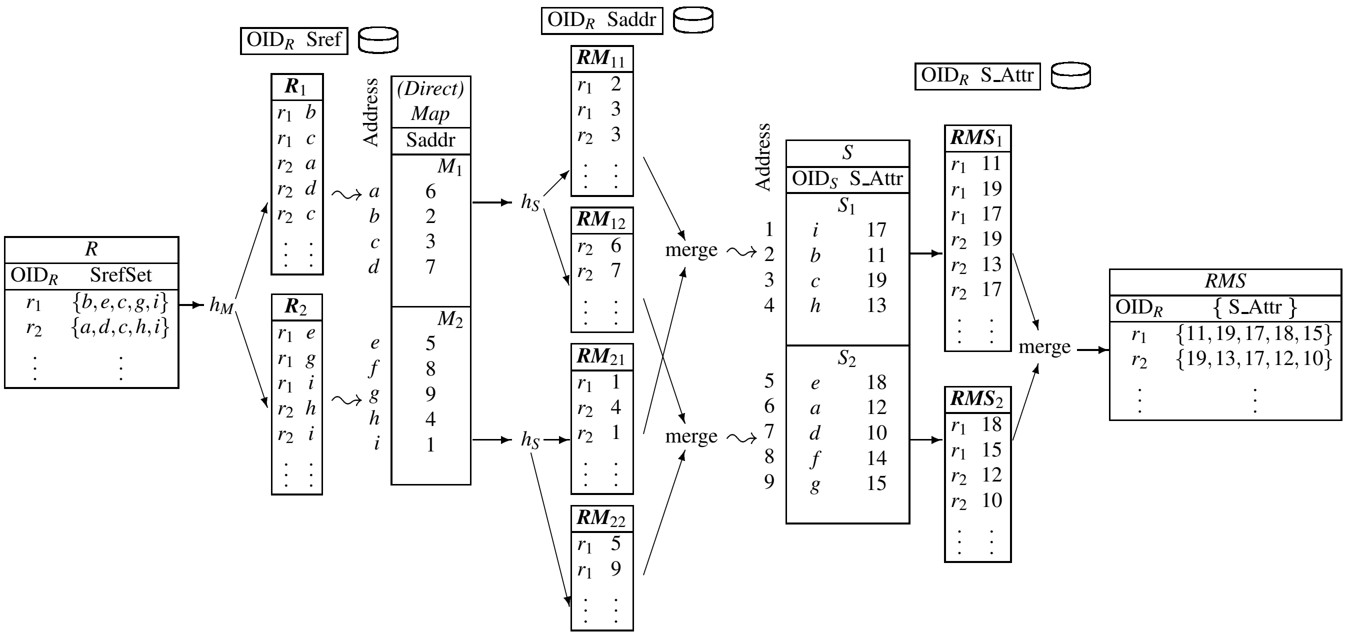


Figure 2: An Example Application of the Partition/Merge-Algorithm $P(PM)^*M$ ($\leadsto$ denotes the lookup of the functional join; for simplicity, the handling of the additional R attribute *R_Data* is not shown)

We note that the partition/merge-algorithm writes the (augmented) R to disk three times: (1) to generate the N partitions of the probe input for the application of the *Map*, (2) to generate the $N*K$ partitions after applying the *Map*, and (3) the K partitions obtained after joining with S. The intermediate $N*K$-way partitioning and subsequent N-fold merging of the $N*K$ partitions into K partitions is the key idea of this algorithm. This way the grouping of the flattened R tuples is preserved across the two partitioning steps with different partitioning functions h_M and h_S. Please observe that immediately distributing the objects into the K partitions after applying the *Map* would have destroyed the grouping on R that we want to retain in every partition.

In this respect the partition/merge-algorithm induces the same i/o-overhead as the basic flatten-algorithms of Section 3.1.2. However, the CPU cost of the partition/merge-algorithm is far lower than for the basic flatten-algorithms because there is no in-memory re-grouping involved. The flat tuples of the same R object are always in sequential order in all the partitions—it may only happen that some partitions do not contain any tuple. Furthermore, the $P(PM)^*M$-algorithm gives room for optimizations based on the retained grouping that are not applicable to other algorithms (cf. Section 3.5).

3.3 An Example of the $P(PM)^*M$-Algorithm

Figure 2 shows a concrete example application of the $P(PM)^*M$-algorithm with two partitioning steps. The tables R_i, RM_{ij} and RMS_j are labelled by a disk symbol to indicate that these temporary partitions are stored on disk.

We start with table R containing two objects with logical OIDs r_1 and r_2—for simplicity, any additional R attributes are omitted. The set-valued attribute *SrefSet* contains sets of references (logical OIDs) to S. The first processing step flattens these sets and partitions the stream of flat tuples. In our example, the partitioning function h_M maps $\{a,\ldots,d\}$ to partition R_1 and $\{e,\ldots,i\}$ to partition R_2. The next processing step starts with reading R_1 from disk, maps the logical OIDs in attribute *Sref* to object addresses using the portion M_1 of the *Map* (note that the *Map* is not explicitly partitioned) and in the same step partitions the tuple streams again with partitioning function h_S (h_S maps $\{1,\ldots,4\}$ to partition 1 and $\{5,\ldots,9\}$ to partition 2. The resulting partitions RM_{1j} (here $1 \le j \le 2$) are written to disk. Processing then continues with partition R_2 whose tuples are partitioned into RM_{2j} ($1 \le j \le 2$). Once again, let us emphasize that the fine-grained partitioning into the $N*K$ (here $2*2$) partitions is essential to preserve the order of the flat R tuples belonging to the same R object. The subsequent merge scans N (here 2) of these partitions in parallel in order to re-merge the fine-grained partitioning into the K partitions needed for the next functional join step. Skipping the fine-grained partitioning into $N*K$ partitions and, instead, partitioning *RM* into the K partitions right away would not preserve the ordering of the R tuples. In detail, the third phase starts with merging RM_{11} and RM_{21} and simultaneously dereferences the S objects referred to in the tuples. In the example, [r_1,2] is fetched from RM_{11} and the S object at address 2 is dereferenced. The requested attribute value (*S_Attr*) of the S object—here 11—is then written to partition RMS_1 as tuple [r_1,11]. After processing [r_1,3] from partition RM_{11}, [r_1,1] is retrieved from RM_{21} and the object address 1 is dereferenced, yielding a tuple [r_1, 17] in partition RMS_1. Now that all flattened tuples belonging to r_1 from RM_{11} and RM_{21} are processed, the merge continues with r_2. After the partitions RM_{11} and RM_{21} are processed, RM_{12} and RM_{22} are merged in the same way to yield a single partition RMS_2. As a final step, the partitions RMS_1 and RMS_2 are merged to form the result *RMS*. During this step, the flat tuples [*r*,*S_Attr*] are nested (grouped) to form set-valued attributes [*r*,{*S_Attr*}]. If aggregation of the nested *S_Attr* values had been requested in the query, it would be carried out in this final merge.

3.4 $P(PM)^*M$, Physical OIDs, Path Expressions

At first glance, repeatedly partitioning and re-merging appears unnecessary for systems employing physical OIDs where the intermediate join with the *Map* is not needed. In fact, in the simple case of a one-step functional join the variant $P(PM)^0M$ is applied. However, the full-fledged $P(PM)^*M$-algorithm is necessary if the query traverses a longer path expression. Consider, for example, the following query where we want to group to each *Customer* the set of *Manufacturers* from which he or she has ever ordered goods:

select *c*.Name, (**select** *l*.ProductRef.ManufRef.Name
from *c*.OrderRefSet *o*, *o*.LineItems *l*)
from Customers *c*

Here we assume additional types *Customer* and *Manufacturer* with the attribute *Name*. *Customers* refer via a nested reference set *OrderRefSet* to the given *Orders*. *Manufacturers* are referenced from *Products* via the reference attribute *ManufRef*.

Another query would be to determine the *Customers*' aggregated *Order* volumes:

select *c*.Name, (**select sum** (*l*.Quantity * *l*.ProductRef.Cost)
from *c*.Orders *o*, *o*.LineItems *l*)
from Customers *c*

Let us, however, concentrate on the Manufacturer query. The $P(PM)^*M$ evaluation plans for physical OIDs and logical OIDs are outlined in Figure 3 (a) and (b), respectively. Both plans contain two unnesting operations to flatten the *OrderRefSet* and the *LineItems* sets, respectively. When comparing the two plans, they differ mainly in the higher number of functional joins needed for mapping logical OIDs. We assume different *Maps* Map_O, Map_P, and Map_M for *Orders*, *Products*, and *Manufacturers*, respectively. The plan based on physical OIDs draws profit from the interleaved partition/merge (*PM*) steps in the same way as the one based on logical OIDs, i.e., the grouping of *Customers*' *Orders* and their *LineItems* is retained across the successive functional joins. Therefore, the final grouping operation is realized as a (very cheap) merge, in both plans.

3.5 Fine Points of the $P(PM)^*M$-Algorithm

There are still some fine points in the design of the $P(PM)^*M$-algorithm that we have to address.

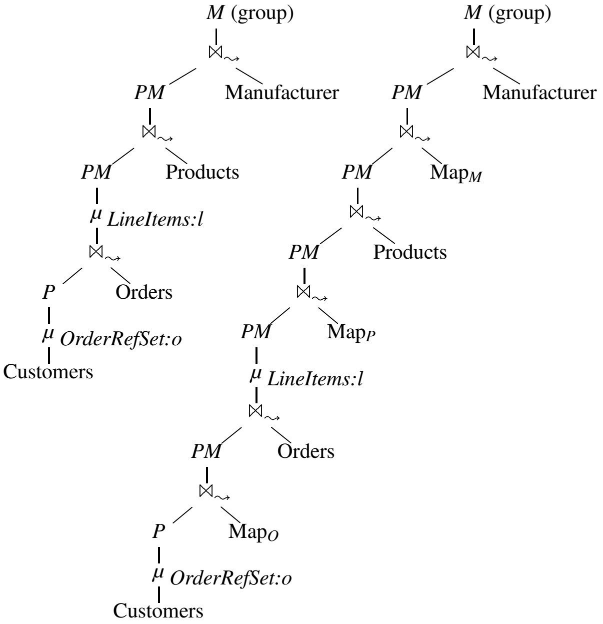
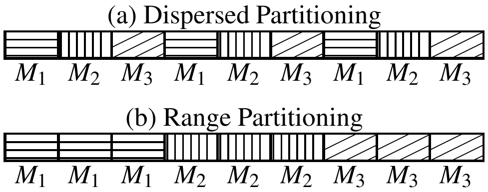


Figure 3: Manufacturer Query Using (a) Physical and (b) Logical OIDs

Obtaining an Order on *R* The algorithm requires an order on the *R* tuples for the merge iterators. When comparing tuples from different partitions—e.g., [r_2,6] from RM_{12} and [r_1,5] from RM_{22} in Figure 2—it has to be determined in what order r_1 and r_2 were contained in the original *R*. If there is no such order given by the key on *R*, an additional sequence number is inserted during the first "flatten and partition" step and used for the succeeding merge steps. Note that all flattened tuples of one *R* object are assigned the same sequence number.

***Map* Access** For the first partitioning phase of the $P(PM)^*M$-algorithm the particular mapping technique has to be taken into account. In general, a partitioning function h_M that achieves range partitioning is favorable. For direct mapping and hash table mapping the difference between range partitioning and "dispersed" partitioning is highlighted as follows:



Range partitioning has two advantages: (1) Even if the pages of one partition are individually faulted into the buffer, the disk accesses are all in the same vicinity. (2) Instead of demand paging, range partitioning allows to prefetch the entire partition from disk, thereby transforming random i/o into chained i/o. However, prefetching is only reasonable if (almost) all pages of the *Map* are actually accessed in performing the OID mapping. In a system where the same *Map* is shared by many object types prefetching would not be reasonable.

Achieving range partitioning for direct mapping is quite simple given the range of pages in the *Map* because every logical OID is composed of page identifier and slot within that page.

If a hash table mapping scheme is used, the same hash function has to be applied to the logical OIDs and then the hashed values (containing the page identifier) can be range partitioned.

For a B^+-tree mapping scheme prefetching a partition is only feasible if the pages of one partition are physically adjacent which is typically not the case because of the dynamic growth of the B^+-tree. However, for OIDs that are sequentially generated the B^+-tree may be built (or reorganized) such that adjacent leaf pages are actually physically adjacent

Access to S Similarly to the partitioning step for the *Map* access, range partitioning is beneficial in any following partitioning step if the accessed data structure is clustered and the query engine has enough knowledge about the physical organization. If the partitioning function for the access to S achieves range partitioning on the pages S is stored on, the accesses to S objects belonging to one partition are more localized. Furthermore, instead of faulting in each individual page of S, the whole range of S that is used for a particular partition can be prefetched in large chunks of sequential i/o.

Full Unnesting vs. Retaining Sets The $P(PM)^*M$-algorithm as described above flattens the *SrefSet* attribute to a single-valued attribute *Sref*. Instead of fully unnesting *SrefSet*, it is also feasible to keep the set intact as much as possible, i.e., the partitioning operators split the set into subsets each of which containing those elements that belong to the same partition. That is, for every object $r \in R$ there is at most one tuple in every partition. This tuple contains a nested set containing those references belonging to the particular partition. For example, the first partitioning operator gets the complete set *SrefSet* as input and partitions it into at most N tuples and at most one tuple per partition, each of which again contains a set-valued attribute. Referring to the example in Figure 2, the first R object $[r_1, \{b,e,c,g,i\}]$ would be split into $[r_1, \{b,c\}]$ (written to partition R_1) and $[r_1, \{e,g,i\}]$ (written to R_2).

This approach avoids multiple flat tuples for the same R object in the same partition; thus it is most beneficial for larger sets *SrefSet*, for small partitioning fan-outs and for non-uniformly distributed references. Of course, keeping the sets requires higher implementation effort. The query engine has to offer "set-aware" variants of some iterators: The partitioning iterator must be capable of splitting nested sets, and the join iterators must iterate through all elements of the nested sets. Our query engine—described briefly in Section 4—is capable of processing nested sets.

Projecting R **Attributes** If "bulky" attributes of R are requested in the result, they may severely inflate the amount of data that is written three times to partition files. To reduce this effect, several measures can be taken: First, the replication of attributes during flattening is unnecessary. Instead, for every $r_i \in R$ the attributes are written only once. Second, since the algorithm retains the order of R, the attributes could be projected out and merged in later for the final result. In contrast to the value-based join and the standard flatten-algorithm, the re-insertion of R attributes is in fact very cheap, since both R and the result have the same order and the R attributes are simply handled as an additional—$(K+1)$-st—input stream of the last merge operator. If the second scan on R would be expensive (e.g., because of high selectivity on R), the bulky attributes of the qualifying R objects might be saved in a temporary segment during the initial scan for reuse in the final merge.

Early Aggregation If aggregation is requested on the result sets in addition to grouping, the aggregation can be folded such that it is already applied to the subgroups belonging to the same R object before they are written to RMS_j. This may result in storage savings for RMS_j. During the final merge, the intermediate aggregation results are then combined. This is easily achieved for the aggregations *sum*, *min*, *max*, *count* which constitute commutative monoids [GKG$^+$97]—i.e., operations that satisfy associativity and have an identity. For, e.g., *avg* more information has to be maintained to enable early aggregation.

Buffer Allocation The algorithm consists of several consecutive phases, each of which stores its intermediate results entirely on disk. This simplifies database buffer allocation, since the memory available to the query can be allocated exclusively to the current phase. The four phases may be easily derived from the example in Figure 2: They are delimited by the three sets of partitions R_i, RM_{ij}, and RMS_j that are stored on disk. Consequently, the four phases are: (1) initial processing of R ending with the first set of partitions R_i, (2) *Map* lookup, (3) dereferencing S, and (4) final merge. For phases (2) and (3), the major amount of memory is allocated to cache the *Map* and S, respectively, and only a small amount is allocated to input and output buffers for the partitions. Summarizing, the $P(PM)^*M$ algorithm is very modest in memory requirements; that is, because of its phased "stop and go"-approach and since it does not require a costly grouping, it tolerates small main memory sizes very well whereas other algorithms easily degrade if main memory is scarce in comparison to the database size.

4 Proof of Concept

To compare the evaluation algorithms, we have implemented them in our iterator-based query engine. In this section we will first outline the implementation of the $P(PM)^*M$-algorithm and then describe a few performance measurements we have taken with our query engine. A more comprehensive assessment based on a cost model is given in Section 5.

4.1 Partition/Merge-Implementation

Our query engine is based on the iterator model [Gra93] and is implemented in C++. Figure 4 gives an outline of the implementation of our $P(PM)^*M$-algorithm. The dashed boxes indicate the new special-purpose *PM*-operators that are composed of two "off-the-shelf" iterators. For the basic functional join from R to S along *SrefSet*, processing starts with a scan of R, applying an optional selection predicate and projecting out unwanted attributes, but keeping at least the set-valued attribute *SrefSet* of R and a key for R. *SrefSet* is then flattened by the unnest operator μ yielding tuples with single-valued attribute *Sref*. The first partitioning iterator P_N partitions the input into N partitions based on a partitioning function h_M—as introduced before. The second half of the iterator scans the partitions one at a time and passes the tuples to the functional join with the *Map*, probing every logical OID in *Sref* against the *Map* and replacing it by the address *Saddr*. The partition size and the partitioning function h_M are chosen depending on the OID mapping technique such that the *Map* lookup can be evaluated in memory (see Section 3.5). The join output is directly fed into the next partitioning operator PM_{NK}. This time the physical OIDs, now stored in *Saddr*, serve as partitioning key, and each of the N input partitions obtained from the prior join is split into K output partitions yielding totally $N*K$ partitions stored on disk. The dotted arrow between P_N and PM_{NK} symbolizes a communication channel that tells PM_{NK} to start a new set of partitions each time an input partition has been completed. Instead of processing all $N*K$ partitions consecutively, we first re-merge those N partitions referring to the same partition of S objects. That is, we merge one matching subpartition from each of the initial N partitions, yielding K partitions as output of the merge operator. Each of the K partitions is then in turn joined with S and the join result is again written to a partition file by an operator called "cache." In a final step, the K partitions are merged to form a single output stream. If we projected out some "bulky" R attributes that are needed in the final result, we would very cheaply re-merge this information in this final merge by simply adding R (or the temporary segment containing the projected data of R) as the $(K+1)$-st merge stream to the merge operator. If required, an aggregation is performed on the result, as indicated by "Aggr." Note that since the order of R is retained, the result is already grouped by the key of R such that only the aggregation function itself (like *sum*) has to be computed.

4.2 Benchmark Setup

The benchmarks were performed on a Sun SparcStation 20 running under Solaris 2.6. The database was held on the

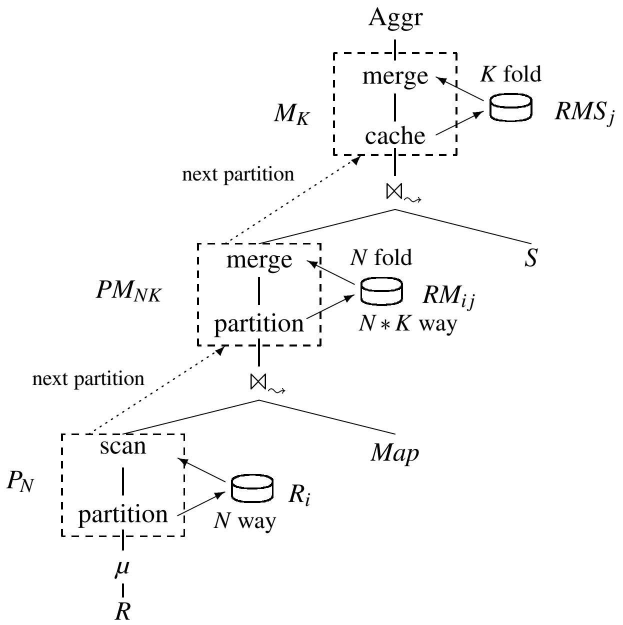


Figure 4: $P(PM)^*M$-Implementation

operating system disk together with the database system, and another disk was used for temporary files. In order to avoid side effects because of file system caching, the direct i/o option was turned on for all file accesses. Thus, we ensured that *each* file access of the database system definitely caused disk i/o and could not be satisfied from the OS file system cache. This is especially important for our first experiments on the small database. Writing the result tuples to display/file was suppressed for all queries—also the relational ones (see below).

The database buffer cache was segmented and configured according to the optimizer (cost model) estimation individually for each query plan. For the run time experiments, the total amount of cache available to a query did not exceed 2 MB at any time. Direct mapping was employed to resolve logical OIDs.

We restrict ourselves to the query with aggregation given in Section 3. For the query without aggregation, the final aggregation operator would be replaced by a grouping operator. Since for each R object the complete group of *S_Attr* had to be conserved instead of a single aggregation result, all algorithms requiring an explicit grouping, i.e., the flatten-algorithms and the value-based join, would become even more expensive, whereas the algorithms that retain the initial grouping would not suffer much from the larger result.

We have tried to evaluate the query on a commercial relational DBMS with object-relational extensions (O/RDBMS).[4] For this purpose, we have created types and tables as described in Section 3. The references were scoped, i.e., they were constrained to point only to a single table (S) using the SQL3-like *scope* clause. However, the query crashed after a few hours with DBMS errors for our

[4] Our license prohibits to identify the particular product.

database	$\|R\|$	$\|S\|$	$\|SrefSet\|$	R pages	S pages
small	10000	10000	10	994	667
larger	100000	100000	10	9933	6667

database	*Map* pages	*R* object size	*S* object size
small	61	348	228
larger	591	348	228

Table 1: Database Cardinalities

initial (larger) database, such that we had to fall back to a smaller database in order to get any results for comparison. The cardinalities of both the small and the larger database are given in Table 1. The references in *SrefSet* were distributed uniformly over the full extent of *S*.

The O/RDBMS was installed on a faster machine (SS20 with CPUs clocked 50% higher and faster disks) and configured with 2 MB buffer—as in our other setup. For comparison, we have also used the same commercial O/RDBMS for a pure (flat) relational representation without using references and nested sets. In this schema, the reference set *SrefSet* has been omitted and an additional table *RS* has been created to implement the association between *R* and *S*. Consequently, the *RS* table contained $|R| * 10$ rows. To compensate for the lacking OIDs, the tables *R* and *S* were extended to contain additional integer (key) attributes *R_key* and *S_key*, respectively. Tuples of *R* and *S* had a size of 204 byte each—the smaller *R* tuple size in comparison to the *R* object size of 348 bytes is due to the missing nested reference set in the pure relational schema. The table *RS* contained only the two attributes *RS_Rkey* and *RS_Skey* that served as foreign keys for *R* and *S*, respectively. The following query, which (apart from *R_key*) yields the same result as the object-relational one, was measured:

select *r*.R_key, *r*.R_Data, **sum**(*s*.S_Attr)
from R *r*, S *s*, RS *rs*
where *rs*.RS_Skey = *s*.S_key **and** *rs*.RS_Rkey = *r*.R_key
group by *r*.R_Key, *r*.R_Data

4.3 Comparison of Measured Run Times

We have created both databases described in Table 1 on our prototypical OODBMS and implemented all algorithms described in the previous section. For the plans either a cheap "stream" aggregation algorithm (only calculating the sum) was employed if the grouping on *R* was retained ($P(PM)^*M$ and naive) or a hash aggregation was used if the initial grouping has been destroyed (sort, partition and value-based plans). The value-based plan was implemented using a hybrid hash join with *S* as build input. For the $P(PM)^*M$-algorithm, some of the optimizations described in Section 3.5 were implemented: The sets were retained (no full unnesting), range partitioning was applied for the access to *S*, and the bulky *R_Data* attribute was projected out and re-merged in the final step.

Table 2 gives an overview of the observed run times for all algorithms. For comparison, the predictions of our cost model (cf. Section 5) are also shown. Furthermore, the run times of the queries on the O/RDBMS are given for two variants: (1) based on the object-relational schema of Section 3 with the nested reference set *SrefSet* and (2) on the pure flat relational schema with the additional table *RS*.

The value-based join performs quite well on the small database since the build input (*S*) is projected to contain only two attributes, the OID and *S_Attr*. It is, however, not cheaper than the $P(PM)^*M$-algorithm since the final hash aggregation causes additional cost that does not occur in the $P(PM)^*M$ plan. On the larger database, it can no longer keep its complete build input in memory and, as a consequence, has to perform an expensive hash aggregation. When comparing the $P(PM)^*M$ run time to the naive algorithm, there is a performance gap of more than an order of magnitude: On the larger database, the absolute run time of the naive algorithm amounts to more than five *hours*, while our $P(PM)^*M$-algorithm requires only less than five *minutes*. The $P(PM)^*M$-algorithm also outperforms all the flatten-algorithms, though not as drastically as the naive pointer chasing algorithm. The sort-based flatten plan suffers from high CPU cost for sorting and small run files due to the restricted amount of memory.

For the object-relational schema, the commercial O/RDBMS shows an even worse performance than the naive algorithm. On the other hand, the query on the flat relational schema takes reasonable run time, although for the larger database still more than twice as much as $P(PM)^*M$ (in spite of the faster host for the O/RDBMS).

5 Analytical Evaluation

We developed a cost model as vehicle for a broader analysis. I/o costs are modelled according to [HCLS97] and the CPU operation assumptions are mostly based on [PCV94] and [HR96]. Our cost model contains extensions to deal with set-valued attributes and our new $P(PM)^*M$-algorithm. Due to space limitations, we cannot discuss individual formulas. The cost formulas model disk i/o quite precisely by means of differentiating between seek, latency, and transfer time. As a consequence, we are able to grasp the difference between sequential and random i/o and the influence of the transfer block size. In modelling the CPU costs, we have included those operations that have major influence on CPU time, e.g., sorting, hashing, buffer management (page hit/page fault) and iterator calls.

Unless stated otherwise, the analyses are based on the larger database as described in Table 1. Similarly, the default configuration was chosen as before, i.e., 2 MB of memory was available and logical OIDs were resolved using direct mapping. The labels of the plots are constructed from two parts, the first one describing the access method to the *Map*, the second part describing the access to the *S* extent. The access methods are: no partitioning (N), i.e., directly chasing each individual pointer, partitioning (P) and

database	small				larger			
method	our prototype	cost model	commercial O/RDBMS with ref. sets	commercial O/RDBMS flat rels w/o refs	our prototype	cost model	commercial O/RDBMS with ref. sets	commercial O/RDBMS flat rels w/o refs
naive	356	461			14893	18219		
flatten/partition	125	136			1868	2029		
flatten/sort	140	168	1110		4874	5432	–	
value-based	40	56		51	1811	1389		721
$P(PM)^*M$	29	34			289	295		

Table 2: Run Times in Seconds (2 MB Memory, avg. $|SrefSet|$=10, Direct Mapping)

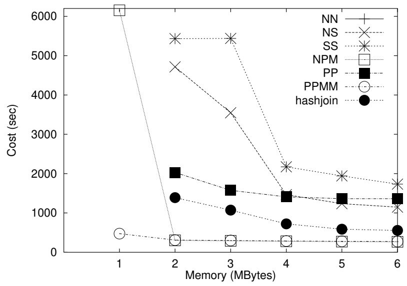


Figure 5: Cost Model Results for Larger Database (Direct Mapping, $|SrefSet|$=10)

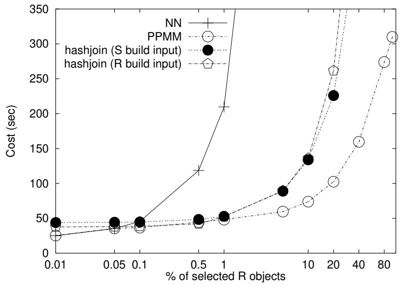


Figure 6: Selection on R (Direct Mapping, $|SrefSet|$=10, 2 MB Memory)

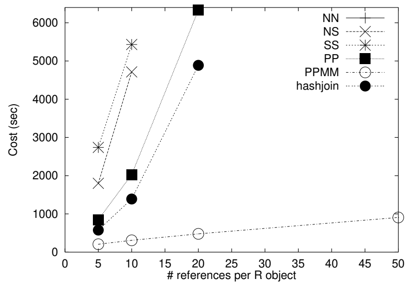


Figure 7: Varying the Cardinality of *SrefSet* (2 MB Memory, Direct Mapping)

merging (M), and sorting (S). The value-based hash join does not fit into this classification and is simply labelled "hashjoin."

5.1 Varying the Memory Size

The run times of the various algorithms under varying memory sizes are reported in Figure 5. The NN plan using (naive) pointer chasing both for *Map* lookup and dereferencing S does not even show up in the plot due to its run time of 6'20 hours for 1 MB to 4'10 hours for a 6 MB buffer. The NS query still uses naive *Map* lookup, but sorts the physical OIDs before accessing S. When comparing NS with SS, sorting the flattened R tuples for the *Map* lookup does not pay off because the *Map* is smaller than 2 MB (For 1 MB the sort-based plans are out of the range of the curve because for such small memory configurations they need several merge phases.) Both sort variants suffer from high CPU costs for sorting. The partition plan PP yields already significantly better performance than sort-based plans for small memory sizes. The performance advantage of partitioning over sorting for small memory sizes is due to the large number of run files generated for sorting. The value-based hash join performs even better than PP, but is still quite costly compared to the winners PPMM (=$P(PM)^1M$) and NPM (=$P(PM)^0M$). The latter one omits the first partitioning step and shows poor performance for very small memory sizes. For 2 MB and larger, the two plans have the same run time since PPMM uses only one partition for the *Map* access anyway and, therefore, coincides with NPM. The most impressive result of this curve is that the $P(PM)^*M$-algorithm tolerates very small memory sizes under which all other algorithms degrade.

5.2 Varying the Selectivity on *R*

In Figure 6 the percentage of R objects taking part in the functional joins is varied on the (logarithmically scaled) x-axis. For a small number of R objects, most pages of the *Map* are hit at most once and some pages of S are not referenced at all, such that one might expect a break-even point between $P(PM)^*M$ and the naive algorithm. However, for a high selectivity (e.g., 0.01% corresponding to 10 R objects) they have nearly the same run time. That is, even if there are only very few references to be resolved, there is no significant overhead induced by our $P(PM)^*M$-algorithm. On the other hand, the naive algorithm very quickly degrades if the number of references to be mapped increases. Furthermore, we have plotted the value-based hash join with two configurations, using either R or S as build input. Both variants are, however, worse than $P(PM)^*M$ over the full selectivity range, and for a small number of R objects they are—due to the fix cost for the hash join and hash aggregation—even worse than the naive plan.

5.3 Varying the Set Cardinality

In the previous experiments, the number of elements in *SrefSet* was constantly 10. Figure 7 shows run times of the algorithms with different set sizes. While the $P(PM)^*M$-algorithm scales linearly, the run times for all others explode. The flatten variants behave poorly. The naive plan suffers from an enormous amount of random i/o (up to $50 * 100,000$ references, calculated run time of roughly 25 hours and is therefore not shown) and the flatten plans suffer from large temporary files.

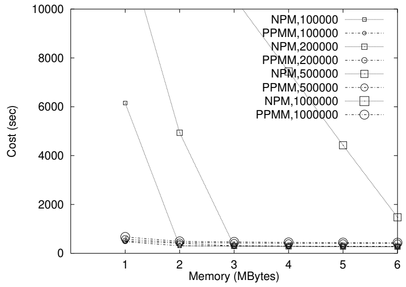


Figure 8: Inflating the OID *Map* under Varying Memory Sizes (Direct Mapping)

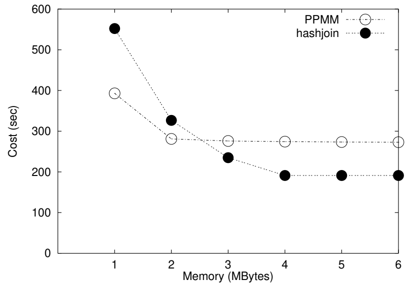


Figure 9: Value-Based vs. $P(PM)^*M$ Pointer Join: $|SrefSet|$=3, Direct Mapping, 1,000,000 *Map* Entries

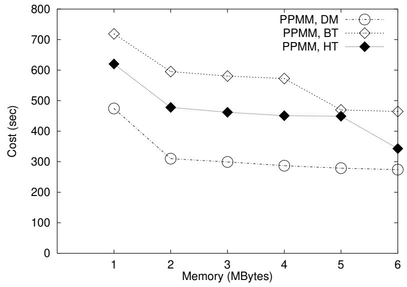


Figure 10: Comparison of Different OID Mapping Techniques: DM, BT, and HT, all with $P(PM)^*M$-Algorithm

5.4 Inflating the OID *Map*

So far we assumed a distinct *Map* for the *S* objects which, as a consequence, is perfectly clustered. In the following experiment, we analyze the behavior of $P(PM)^*M$-algorithms for not-so-well clustered OID *Maps*, as they may occur if there is one global OID *Map* or if only a small fraction of *S* is referenced, e.g., because of a selection on *R*. The OID *Map* for *S*—previously containing 100,000 entries—has been inflated by inserting unused entries—uniformly distributed over all pages of the *Map*—to contain up to one million entries. The NPM and PPMM queries have been run on the standard database (100,000 objects of *R* and *S* each, 10 elements in *SrefSet*) with different amounts of memory available. The legend of Figure 8 indicates the size of the *Map* (100000, ..., 1000000). The smallest symbols denote the configuration that was used in Figure 5, i.e., the *Map* was optimally clustered. For larger *Maps*, the PPMM plan shows only a slight run time increase, caused by the inevitably higher number of i/o accesses to the larger *Map*. However, each *Map* page is fetched from disk only once, since the number of partitions in the first partitioning step is adapted such that one partition of the *Map* can be cached in memory. On the other hand, NPM cannot cope with larger *Maps* since it induces an enormous number of page faults as long as the *Map* does not entirely fit into memory.

Figure 9 compares the $P(PM)^*M$-algorithm with the value-based hash join in an extreme scenario: The set-valued attribute *SrefSet* contains only three references on average and the *Map* is inflated to contain one million entries—of which 900,000 are obsolete. The number of *R* and *S* objects remains at 100,000, respectively. This set-up favors the value-based hash join extremely, since it does not use the *Map* anyway. Furthermore, the hash join draws profit from larger amounts of memory in a larger scale than $P(PM)^*M$ because of the projection on *S*: The (projected) *S* that serves as build input for the hash join can be kept in memory for large memory configurations (beyond 4 MB) such that the join is an in-memory operation. On the other hand, the $P(PM)^*M$-algorithm loads and keeps the *S* pages in their entirety in memory. Since the whole *S* extent of ca. 26 MB still does not fit in memory, the additional memory does not avoid a partitioning step of $P(PM)^*M$ and the flattened *R* must still be written to disk partitions.

5.5 Comparing Different OID Mapping Techniques

Figure 10 compares the three OID mapping techniques that we have discussed in Section 2.2 for our application, i.e., in combination with the $P(PM)^*M$-algorithm. Both B^+-tree (BT) and hash table mapping (HT) show two performance steps. The first step occurs when increasing memory from 1 MB to 2 MB. Here, the scan and merge operators reach their optimal amount of memory. The second step occurs when the $P(PM)^*M$-algorithm omits the first partitioning phase since the OID mapping structure can be completely cached in memory. Since the total size of the B^+-tree is smaller than that of the hash table,[5] this point is reached with a smaller memory size for the BT curve. In addition, BT is generally more expensive due to higher CPU cost for the tree lookup. The direct mapping (DM) approach is the cheapest: The first partitioning step can already be omitted at a memory size of 2 MB due to the compact representation of the *Map*. Furthermore, the compact storage of the (direct) *Map* reduces the total number of i/o calls. In addition, the CPU overhead for a single *Map* lookup is cheaper for DM than for the other two mapping techniques.

5.6 Logical OIDs in Comparison to Physical OIDs

So far, we have assessed our algorithms for different scenarios using logical OIDs. Next, we turn to physical OIDs. This simplifies all algorithms since the extra *Map* lookup operation is omitted. Thus, the algorithms are no partitioning (N), sorting (S), partitioning (P), and $P(PM)^0M$ (labelled PM). The value-based hash join is independent of the underlying OID realization. For comparison, Figure 11 additionally includes the NPM and PPMM plans for logical OIDs realized with direct mapping. The naive plan does not show up in the plot since it ranges between four and five hours. The run time for the partition plan P is similar to the value-based hash join while the sort-based query performs still significantly worse. Not surprisingly, the PM plan performs slightly better than the $P(PM)^*M$ plan for logical

[5] Due to prefix compression and a specialized splitting procedure [EGK95] the B^+-tree contains more entries per page than the hash table.

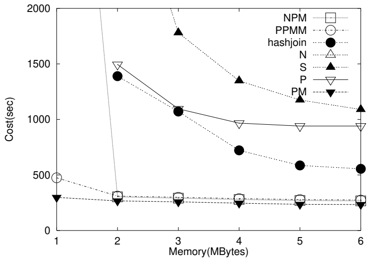


Figure 11: Physical OIDs vs. Logical OIDs with Direct Mapping

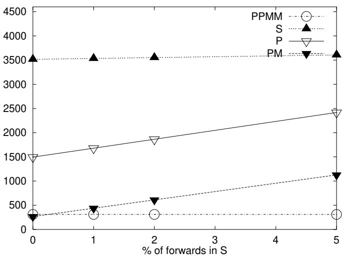


Figure 12: Effect of Forwards

OIDs. However, the additional cost of the *Map* lookup is kept at a low level. For example, for 3 MB of memory the PM plan was only 14% cheaper than $P(PM)^*M$.

While physical OIDs are definitely advantageous on a "clean" database, they incur a severe performance penalty in the presence of forwards. We have created a varying percentage of forward references (0% to 5%) in the S extent. Figure 12 shows that the sort-based plans are fairly robust against forwards—although at a high cost level—because they "hit" the same forwarded object consecutively whereas the multiple hits of the forwarded object are non-consecutive for partition-based plans. Therefore, sort-based plans need to allocate only one additional page for loading the currently "active" forwarded object whereas partition-based plans need to allocate more buffer for a partition containing forwards. P and PM behave similarly (the lines are parallel), such that partition/merge retains its advantage. For comparison, the PPMM plan under logical OIDs is also shown. Evidently, even for very low levels of forward references (e.g., 1%) logical OIDs are superior to physical OIDs.

6 Conclusion and Future Work

In object-relational and object-oriented database systems, one-to-many and many-to-many relationships are typically represented as nested sets of references—instead of a separate relation as in the pure relational model. Very often, queries along these nested reference sets require to retain the implicit grouping given by the set of references. In this paper we have developed a new algorithm that is based on successively partitioning and merging. This algorithm retains the grouping within the partitions and restores the overall grouping by (efficient) merge operations. Our prototype implementation and the quantitative assessment based on a cost model have proven that the algorithm is superior to other methods.

The key idea of the $P(PM)^*M$-algorithm consists of retaining an order (or grouping) given by nested reference sets across multiple functional joins. Of course, this idea is also applicable to a pure relational database schema if one wants to preserve the order (or grouping) across regular hash-based joins. We will investigate this in the future.

References

[BK89] E. Bertino and W. Kim. Indexing techniques for queries on nested objects. *IEEE Trans. Knowledge and Data Engineering*, 1(2):196–214, 1989.

[BP95] A. Biliris and E. Panagos. A high performance configurable storage manager. In *Proc. IEEE Conf. on Data Engineering*, pages 35–43, Taipeh, Taiwan, 1995.

[CBB+97] R. Cattell, editor. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers, San Mateo, CA, USA, 1997.

[CDF+94] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up persistent applications. In *Proc. of the ACM SIGMOD Intl. Conf.*, pages 383–394, Minneapolis, MI, USA, May 1994.

[CSL+90] M. J. Carey, E. Shekita, G. Lapis, B. Lindsay, and J. McPherson. An incremental join attachment for Starburst. In *Proc. of the VLDB Conf.*, pages 662–673, Brisbane, Australia, August 1990.

[DLM93] D. DeWitt, D. Lieuwen, and M. Mehta. Pointer-based join techniques for object-oriented databases. In *Proc. of the Intl. IEEE Conf. on Parallel and Distributed Information Systems*, San Diego, CA, USA, January 1993.

[EGK95] A. Eickler, C. A. Gerlhof, and D. Kossmann. A performance evaluation of OID mapping techniques. In *Proc. of the VLDB Conf.*, pages 18–29, Zürich, Switzerland, September 1995.

[GGT96] G. Gardarin, J.-R. Gruser, and Z.-H. Tang. Cost-based selection of path expression processing algorithms in object-oriented databases. In *Proc. of the VLDB Conf.*, pages 390–401, Bombay, India, September 1996.

[GKG+97] T. Grust, J. Kröger, D. Gluche, A. Heuer, and M. H. Scholl. Query evaluation in CROQUE – calculus and algebra coincide. In *Proc. Brit. Natl. Conf. on Databases (BNCOD)*, London, UK, Jul 1997.

[Gra93] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[HCLS97] L. M. Haas, M. J. Carey, M. Livny, and A. Shukla. Seeking the truth about *ad hoc* join costs. *The VLDB Journal*, 6(3):241–256, 1997.

[HR96] E. P. Harris and K. Ramamohanarao. Join algorithm costs revisited. *The VLDB Journal*, 5(1):64–84, 1996.

[Ita93] Itasca Systems Inc. Technical summary for release 2.2, 1993. Itasca Systems, Inc., 7850 Metro Drive, Mineapolis, MN 55425, USA.

[KC86] S. N. Khoshafian and G. P. Copeland. Object identity. In *Proc. of the ACM Conf. on Object-Oriented Programming Systems and Languages (OOPSLA)*, pages 408–416, November 1986.

[KM90] A. Kemper and G. Moerkotte. Access support in object bases. In *Proc. of the ACM SIGMOD Intl. Conf.*, pages 364–374, Atlantic City, USA, April 1990.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Comm. ACM*, 34(10):50–63, 1991.

[LMB97] L. Leverenz, R. Mateosian, and S. Bobrowski. *Oracle8 Server – Concepts Manual*. Oracle Corporation, USA, 1997.

[O2T94] O_2 Technology, Versailles Cedex, France. *A Technical Overview of the O_2 System*, July 1994.

[PCV94] J. M. Patel, M. J. Carey, and M. K. Vernon. Accurate modeling of the hybrid hash join algorithm. In *Proc. of the ACM SIGMETRICS*, pages 56–66, Santa Clara, CA, May 1994.

[SC90] E. Shekita and M. J. Carey. A performance evaluation of pointer-based joins. In *Proc. of the ACM SIGMOD Intl. Conf.*, pages 300–311, Atlantic City, NJ, May 1990.

[SS86] H.-J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, 1986.

[Sto96] M. Stonebraker. *Object-Relational DBMSs: The Next Great Wave*. Morgan Kaufmann Publishers, San Mateo, CA, USA, 1996.

[Val87] P. Valduriez. Join indices. *ACM Trans. on Database Systems*, 12(2):218–246, June 1987.

[Ver97] Versant Object Technology. Versant release 5, October 1997. `http://www.versant.com/`.

[XH94] Z. Xie and J. Han. Join index hierarchies for supporting efficient navigations in object-oriented databases. In *Proc. of the VLDB Conf.*, pages 522–533, Santiago, Chile, September 1994.