The VLDB Journal
© Springer-Verlag 2000

Functional-join processing[*,**]

R. Braumandl, J. Claussen, A. Kemper, D. Kossmann

Universität Passau, Lehrstuhl für Informatik, 94030 Passau, Germany; e-mail: {braumandl,claussen,kemper,kossmann}@db.fmi.uni-passau.de

Edited by O. Shmueli. Received February 28, 1999 / Accepted September 27, 1999

Abstract. Inter-object references are one of the key concepts of object-relational and object-oriented database systems. In this work, we investigate alternative techniques to implement inter-object references and make the best use of them in query processing, i.e., in evaluating functional joins. We will give a comprehensive overview and performance evaluation of all known techniques for simple (single-valued) as well as multi-valued functional joins. Furthermore, we will describe special *order-preserving* functional-join techniques that are particularly attractive for decision support queries that require ordered results. While most of the presentation of this paper is focused on object-relational and object-oriented database systems, some of the results can also be applied to plain relational databases because *index nested-loop joins* along key/foreign-key relationships, as they are frequently found in relational databases, are just one particular way to execute a functional join.

Key words: Object identifier – Logical OID – Physical OID – Query processing – Pointer join – Functional join – Order-preserving join

1 Introduction

1.1 Background and motivation

Inter-object references are one of the key concepts of object-relational and object-oriented database systems. In an object model, it is, for instance, natural to represent an *Order* object as an object with a reference to a *Customer* object and a set of references to *Lineitem* objects. In such a database, it is just as natural that users initiate queries that involve functional joins (also called pointer-based joins). A user could, for example, request the names of all *Customers* that ordered goods in the last two weeks, or a user could be interested in the prices of the *Lineitems* of this month's *Orders*.

Inter-object references are implemented by storing the referenced object's object identifier in the referencing object. Object identifiers can be implemented in different ways and they come in different flavors: references can, for example, be generated externally (e.g., within a legacy order-processing system), or they can be generated internally by the database system. As a consequence, different techniques that depend on the kind of references used are applicable in order to implement functional joins. Furthermore, our example from above shows that inter-object references can occur as part of single-valued and multi-valued attributes, and again special functional-join techniques are required to deal with both cases. Finally, the choice of the right functional-join method can be impacted by other operations that must be carried out in a query (e.g., sorting, other joins, or aggregation).

We point out that the presented techniques are applicable to object-relational systems in the same way as they are to object-oriented systems.

1.2 Related work

The purpose of this paper is to give a comprehensive overview and thorough performance analysis of all known functional-join techniques. The first (classic) work on functional joins was carried out by Shekita and Carey [SC90]. This work was partially incorporated into Starburst [CSL+90], and it was later extended by [DLM93] to deal with parallel database systems. In our own previous work, we studied alternative ways to implement inter-object references [EGK95], developed special techniques for functional joins in the presence of multi-valued attributes [BCK98], and special techniques that can be used for decision support queries [CKK98]. All these papers, however, are each focused on one particular aspect of functional-join processing, and in this paper, we will give a complete overview and fill in all the open holes which were not addressed in previous work.

Throughout this paper, we will concentrate on *ad hoc* functional joins. That is, we will assume that there are no specialized join or path indices available. Join indices have been proposed in [Här78] and [Val87], and work on path

[*] Some excerpts of this work appeared in the conference publications [EGK95] and [BCK98].

[**] This work was partially supported by the German Research Council (DFG) under contract Ke 401/7-1.

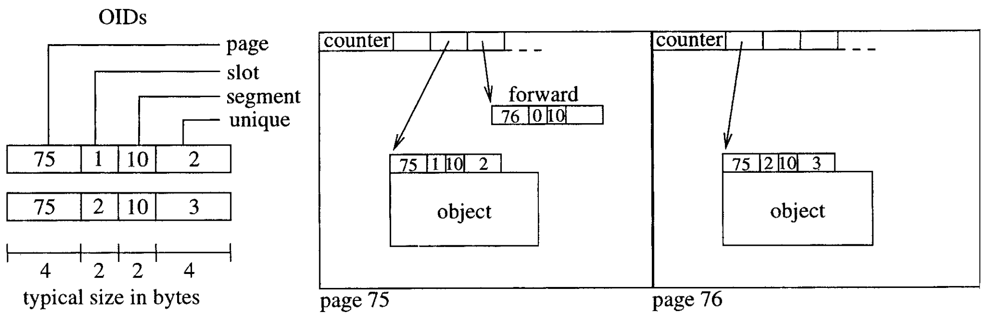


Fig. 1. Physical OIDs

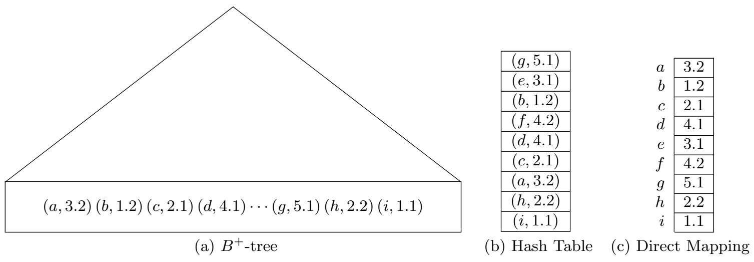


Fig. 2. Mapping techniques

indices is reported in [BK89, KM90, XH94]. A recent paper describes two new join algorithms that are based on join indices [LR99]. Both algorithms store the join result in two files on disk, which need to be merged to obtain the actual join result. Therefore, their algorithms achieve their performance gains mostly in situations where the join needs to be materialized and less in situations where the join result is further processed (then the tuples have to be re-read from disk because their algorithm does not pipeline tuples to the next operator).

Another line of work which we will not cover in detail is so-called *object assembly* [KGM91, MGS+94]. Object assembly influences the order in which objects are read from disk or retrieved from remote servers in a distributed system in order to reduce (disk or network) I/O cost. Object assembly is specifically designed to *assemble* complex objects that are hierarchically composed of sub-objects, and object assembly does not work well for general-purpose functional joins which have been the focus of our work. Also, we will concentrate on the execution of functional joins and ignore query optimization issues. Query optimization techniques which aim at finding the best evaluation order for chains of functional joins are presented in, e.g., [GGT96] and [CD92].

1.3 Overview of this paper

The remainder of this paper is organized as follows. In Sect. 2, we investigate alternative ways to implement object identifiers. Section 3 describes different algorithms for performing functional joins along single-valued reference attributes. Section 4 describes analogously different algorithms for performing functional joins along multi-valued reference attributes. Section 5 presents a new class of functional-join techniques which are particularly attractive for decision support queries that require ordered results. Section 6 presents the results of a comprehensive performance analysis comparing all alternative techniques with the help of a detailed cost model. Section 7 concludes this paper.

2 Implementation of object identifiers

Before embarking on the details of alternative functional-join techniques, we would like to present different ways to represent and implement object identifiers (OIDs for short) in a database system. Two different kinds of OIDs can be found in databases today: (1) physical OIDs and (2) logical OIDs.

Physical OIDs encode the storage location, whereas logical OIDs are storage location independent [KC86]. Logical OIDs can, furthermore, be implemented in three different ways. In this section, we will describe physical OIDs and logical OIDs and the three different ways to implement logical OIDs, and we will discuss the trade-offs of all the alternative approaches.

2.1 Physical OIDs

A physical OID contains the (disk) storage location of an object at the time the object was created. In a centralized database system, the storage location is typically defined as a *segment number*, which identifies a file on a disk, a *page number*, which identifies a block within a segment, and a *slot number*, which is the position used to find the object within a page [GR93]. In a distributed system, a physical OID also contains the (IP) address of the server at which the object was created [EKK97]. In addition to the storage location, a physical OID also contains a *unique field*, so that the database system works correctly if objects are deleted. Suppose, for example, that object A references object B using a physical OID. Now object B is deleted and a new

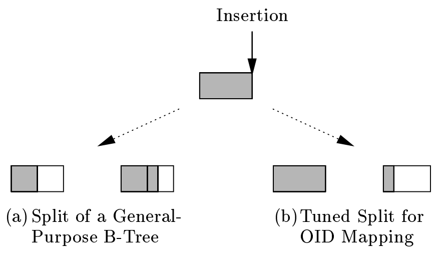

Insertion
(a) Split of a General-Purpose B-Tree
(b) Tuned Split for OID Mapping

Fig. 3. Splitting of a leaf page in a B-tree

object C is created at the storage location at which object B was originally stored. To be able to trap that the object referenced by A no longer exists, the physical OIDs of objects B and C must differ in the value of their *unique fields*. In Exodus [CDRS86], for example, this is achieved by maintaining a counter for every data page, which is increased whenever a new object is created on that page. The current counter value becomes part of the OID.

Working with physical OIDs is very simple: to dereference a physical OID (e.g., traverse the inter-object reference from A to B), the database system simply decodes the storage location of the referenced object which is part of the physical OID. Special precautions must only be taken if objects migrate to different pages. Migrations are, for instance, necessary if objects grow as a result of update operations. If an object migrates, a *forward* which contains the new storage location of the object is established at the place at which the object was originally stored. If an object migrates several times, this forward is updated so that it always contains the right storage location and an object can be read with at most two "hops."

Two example physical OIDs are shown in Fig. 1. The figure also shows that the object referenced by the first OID is still stored at its original location, whereas the object referenced by the second OID was migrated to another page, so that a *forward* for that object had to be established. Examples of commercial systems that make use of physical OIDs are O_2 [O2T94], ObjectStore [LLOW91], Objectivity [Obj96], and (presumably) Illustra [Sto96, p. 57].

2.2 Logical OIDs

Logical OIDs do not contain the storage locations (or addresses) of objects; i.e., logical OIDs are *location independent*. To find an object using its logical OID, a mapping structure is required, which *maps* the logical OID to the object's address. If an object is migrated, the object's entry in the mapping structure is updated in a similar way as *forwards* are updated when objects migrate and physical OIDs are used. Three different kinds of mapping structures are used in practice: (1) B-trees, (2) hash tables, and (3) direct mapping tables. These three mapping structures are shown in Fig. 2 (letters denote logical OIDs and number pairs denote addresses of objects composed of the object's *page number*, and *slot number*. The segment number is ignored in the illustrations of this paper). We will describe these three mapping structures in the following. Furthermore, we will describe how these mapping structures can be partitioned.

2.2.1 Mapping logical OIDs with a B-tree

B-trees or B^+-trees [BM72, Com79] can naturally be used to map logical OIDs to object addresses. B^+-trees are implemented in probably every commercial database system, so that no significant additional implementation effort is required to effect logical OIDs with B^+-trees. For the purpose of OID mapping, however, it is advisable to use a specifically tuned implementation of a B^+-tree, because logical OIDs are usually generated and inserted into the B^+-tree in ascending order. This insertion pattern is the worst case for standard textbook B^+-trees, because many (unnecessary) splits occur and 50% of the storage space is wasted, as shown in Fig. 3a. For this reason, splits should be implemented as demonstrated in Fig. 3b: rather than moving half of the entries of an over-full node into the new node, only the last entry is moved. Similar optimizations for insertions in ascending order were incorporated in the AP-tree [SG89] in the context of temporal databases. Examples of systems that support logical OIDs and use B-trees are Gemstone [MS87], SHORE [CDF+94], and Oracle8 [LMB97].

2.2.2 Mapping logical OIDs with hash tables

As an alternative to a B^+-tree, a hash table can be used to map logical OIDs. A variety of different hashing techniques that can be used for this purpose have been described in the literature; see, e.g., [ED88] for an overview. An important tuning factor for any kind of hash table is the *hash function*. To map OIDs, a good hash function can easily be found because OIDs are usually generated in ascending order, so that simple *mod* hash functions work well. Examples of database systems that use hash tables to map logical OIDs are Versant [Ver97] and Itasca [Ita93].

2.2.3 Direct mapping

B^+-trees and hash tables find the addresses of objects by *comparing* OIDs. The third approach we describe is called *direct mapping* and it works by *encoding* an address of a so-called *handle* into an OID. *Handles* look and work like *forwards* used if physical OIDs are employed: a *handle* of an object contains the address of the object, and if an object is migrated, the *handle* is updated. The difference between systems that employ logical OIDs with direct mapping and systems that rely on physical OIDs is that systems that use direct mapping allocate a *handle* for every object at the time the object is created, whereas systems that use physical OIDs allocate a *forward* only when an object is migrated.

Going into more detail, *handles* are organized in extensible disk-resident arrays, so-called *handle segments*. A *handle segment* contains any number of *handle pages*, and every *handle page* contains a fixed amount of *slots* with *handles*. A *handle* contains the address of an object and a *unique field* which is used to detect dangling references that refer to deleted objects in the same way as in systems that employ physical OIDs. A logical OID is composed of the address of a *handle* (i.e., *handle segment number, handle page number*, and *slot number*) and a *unique field*.

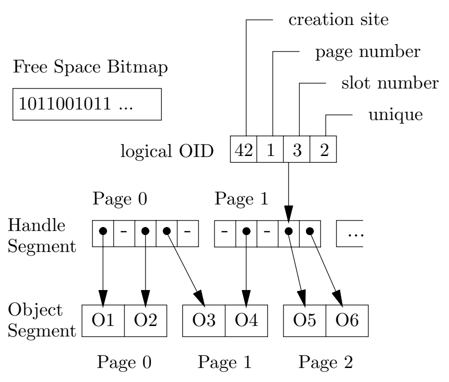


Fig. 4. OIDs, handles, handle pages, and handle segments

Figure 4 shows an example logical OID and how it references a *handle* which, in turn, references an object. The figure also shows a *free space bitmap* which is maintained for every *handle segment*. This bitmap is used to find empty slots in the handle segment when a new object is created. In Fig. 4, this bitmap has a bit set for every *slot* used in the handle segment. The bitmap could, however, also be organized in the granularity of *handle pages*, and, of course, it is also possible to compress the *free-space bitmap* because most of its bits will be set if objects are only rarely deleted.

Currently, we know of no commercial system that supports logical OIDs using direct mapping. Direct mapping has, however, been used in a couple of research prototypes; e.g., [HZ87, BR90, WW90, BP95]. Also, variants of direct mapping have been used in CODASYL database systems.

2.2.4 Partitioning of mapping structures

In very large databases with many objects, it is usually not a good idea to keep the whole mapping information in a single mapping structure. For direct mapping, we already showed in the previous subsection how the mapping information can be partitioned into multiple *handle segments*. Fortunately, partitioning is also possible if B$^+$-trees or hash tables are used. One popular approach is to establish a separate B$^+$-tree (or hash table or *handle segment*) for every class of objects in an object-oriented database system. (In an object-relational system, one mapping structure would be established for every type/table of objects – same concept, just different terminology.) This approach is, for example, used by Itasca [Ita93]. In Itasca, OID mapping is, thus, carried out in two steps. First, the class name which must be encoded in the OID, is hashed to get the right per-class hash table. Then, that per-class hash table is probed to get the object's address. Since the number of classes is typically relatively small, the whole class-name hash table can be kept in memory, while the whole per-class hash tables can usually not be kept in main memory. Note that this kind of partitioning of the mapping information is carried out implicitly by relational database systems: in relational databases, primary indices are naturally constructed for every table individually and there is no *global index* that keeps the keys of all the tuples of the whole database.

Table 1. Number of disk I/O requests to read an object

Physical OIDs	B$^+$-trees	Linear Hashing	Direct Mapping
1 or 2	$1\ldots(1+\log n)$	2	1 or 2

Table 2. Size of mapping structures

# of objects	B$^+$-trees	Linear Hashing	Direct Mapping
200,000	9 MB	13 MB	5 MB
5,000,000	204 MB	252 MB	118 MB

2.3 Logical OIDs with probable position pointers

Probable position pointers (PPP) have been proposed in the late 1970s [GR93]. The idea is to generate logical OIDs which contain a PPP, which is an address at which the object can most likely be found. When such a PPP-enhanced OID is dereferenced, the PPP is traversed first, and only if the object is not found at that address the *logical* part of the OID is used to find the object (using, e.g., direct mapping) and the PPP is updated. Because this approach is a *hybrid* of logical and physical OIDs and, thus, inherits most of the advantages and disadvantages of physical and logical OIDs, we will not discuss this approach further but, rather, concentrate on the two underlying mechanisms, i.e., physical and logical OIDs. One particular disadvantage of PPPs is that PPPs require the use of very large OIDs and, thus, databases tend to become very large if PPPs are used.

2.4 Discussion

In this section, we would like to briefly summarize the tradeoffs of physical OIDs and the three approaches to effect logical OIDs.

Retrieval performance. Using physical OIDs, an object can be read from disk with a single I/O request if the object was not migrated and with at most two requests if the object was migrated. Using logical OIDs and direct mapping, an object can be read from disk with at most two I/O requests. As shown in [EGK95], it is possible to read an object with a single request for many applications because the relevant *handle pages* can effectively be cached in main memory. Using a B$^+$-tree, $1 + \log n$ pages, the height of the B$^+$-tree plus one, must be read from disk in the worst case. As shown again in [EGK95], however, one or two requests are usually sufficient if the cache is large enough to keep the relevant parts of the B$^+$-tree main-memory resident. Using hash tables and linear hashing [Lit80], our experiments showed that objects can be retrieved from disk with two requests (one request for the hash table lookup and one for accessing the object), almost independent of the size of the cache. Table 1 summarizes these results.

Size of the mapping structure. Table 2 shows the size of the mapping structure if logical OIDs are used. We can see that independent of the number of objects in the database, the mapping structure is the smallest if direct mapping is used and the largest if hash tables are used (linear hashing as of [Lit80]); B^+-trees (with prefix compression) lie somewhere in between.

Obviously, it is not always possible to exactly determine the space overhead of systems that employ physical OIDs. If no objects are migrated, the space overhead is 0, and if objects are migrated, the overhead corresponds to the space occupied by *forwards*. Note, however, that *forwards* fragment data pages and cannot be stored without off-cuts, whereas *handles* can nicely be packed into *handle pages* if direct mapping is used.

Size of OIDs. Since OIDs are used to represent inter-object references in the whole database, the size of an OID strongly impacts the size of the entire database. Physical OIDs are usually 12 bytes long: 4 bytes each for the *page number* and the *unique field*, and 2 bytes each for the *segment number* and the *slot number*. Likewise, logical OIDs are 12 bytes long if direct mapping is used. If B^+-trees or hash tables are used, 8 bytes are usually enough to implement logical OIDs. (With 8 bytes, the database system can generate up to 2^{64} objects, which is more than enough.) Most systems that use logical OIDs with B^+-trees or hash tables, nevertheless, have OIDs which are 12 bytes long; e.g., in order to encode the class name of an object into an OID.

Other considerations. As noted in [EGK95], concurrency control and recovery of the mapping structure are easier to implement and faster if direct mapping rather than B^+-trees or hash tables are used. Another observation made in [EGK95] is that hash tables require tuning: hashing works best if the size of the hash table is known in advance, and hashing shows very poor (insertion) performance if the size of the hash table exceeds the anticipated size. Even if the size is known in advance, it is more expensive to bulkload a hash table than to bulkload a B^+-tree or a *handle segment*.

From all this discussion, we might conclude that either physical OIDs or logical OIDs with direct mapping are the way to go. However, throughout this section, we implicitly made the assumption that all OIDs are generated by the database system. But some applications require externally defined OIDs. Externally defined OIDs are, for example, necessary to integrate legacy systems. Neither physical OIDs nor logical OIDs with direct mapping can be used to support externally defined OIDs. In this case, B^+-trees and hash tables are the only viable options, so that we will continue to consider these two options to implement OIDs when we describe techniques to implement functional joins in the following sections.

3 Functional joins along single-valued references

We now turn to a description of alternative ways to implement functional joins (also called *pointer-based joins*) along single-valued reference attributes. We first present an example that involves a query with such a functional join and then present alternative techniques, some of which depend on the way OIDs are implemented.

3.1 Example schema and query

Throughout this section, we will describe algorithms using the following example schema. The schema consists of two tables R and S, and the objects stored in table R refer to objects stored in table S.

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

The example query we wish to discuss traverses all *Sref* references of the objects stored in table R in order to retrieve the *S_Attr* attributes of the matching objects stored in table S. That is, this query involves a functional join between R and S and it can be defined as follows:

```
select r.*, r.Sref.S_Attr
from R r;
```

In the following, we will simply ignore the handling of the *R_Data* (and *S_Data*) attributes—they are just included in the schema because our performance analysis includes these bulky attributes to model realistically sized objects.

3.2 The naïve approach

The naïve approach to execute our example query is to scan through table R and follow every *Sref* reference individually. This approach corresponds (conceptually) to a traditional nested-loop join, and this approach can be applied independent of the kind of OID (physical or logical) and mapping technique (B^+-tree, hashing, or direct mapping) used. Figure 5 illustrates this naïve approach for a system that uses logical OIDs and direct mapping.

3.3 Value-based functional joins

The second way to execute our example query is to ignore the fact that the *Sref* fields contain OIDs and carry out the query using regular join methods such as nested-loops, sort/merge joins, or hash joins. This approach can be effected by re-writing the query as follows (of course, a good query optimizer performs the re-writing).

```
select r.*, s.S_Attr
from R r, S s
where r.Sref = s.OID;
```

Obviously, this approach works again for any kind of OIDs and independent of the mapping technique used. One restriction, however, is that all objects stored in table S must be *self-identifiable*, i.e., the *OID* can be considered as a (possibly not fully materialized) field of an object. The second restriction is that the *Sref* references must be *scoped*; i.e., all the objects referenced by R objects must be stored in table S. The approach is shown in Fig. 6 using a (traditional) hash join method to execute the join.

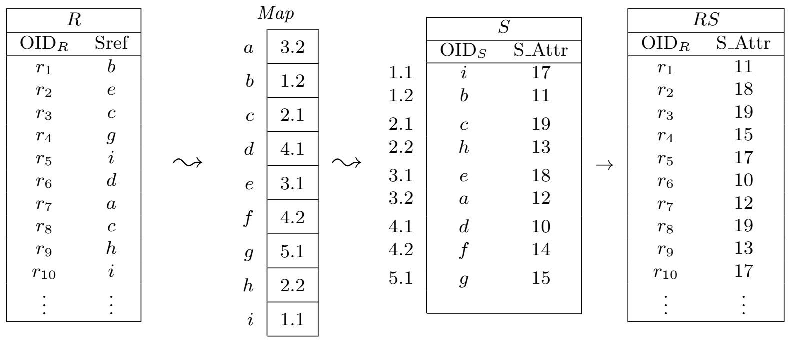


Fig. 5. Naïve functional join with direct mapping

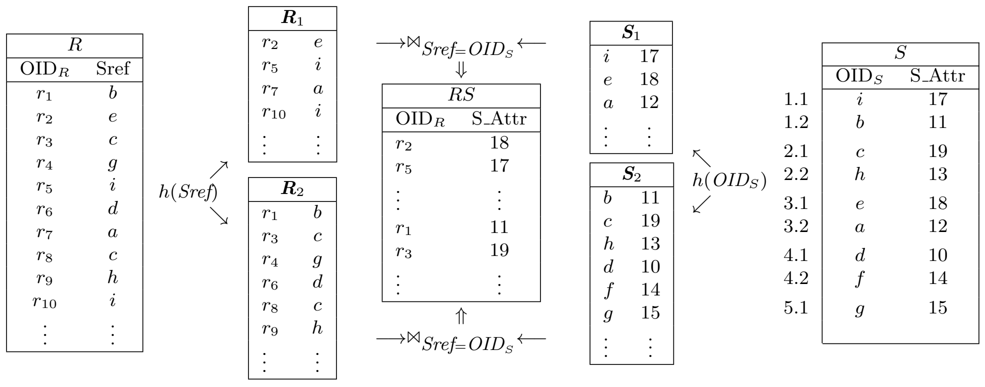


Fig. 6. Value-based functional join (hash join)

3.4 Special functional-join techniques for physical OIDs

As we will see in Sect. 6, both the naïve approach and value-based functional joins show poor performance in many situations: the naïve approach can be the cause of excessive random disk I/O and value-based joins do not exploit the address information contained in OIDs and/or the mapping structures. In this sub-section, we will describe approaches that often do better and can be used by systems that employ physical OIDs. We will first describe techniques which were pioneered by Shekita and Carey [SC90]. Their work, however, ignored the presence of *forwards*, so that we will describe how their techniques can be extended to deal with *forwards* at the end of this sub-section.

3.4.1 Sort-based functional-join evaluation

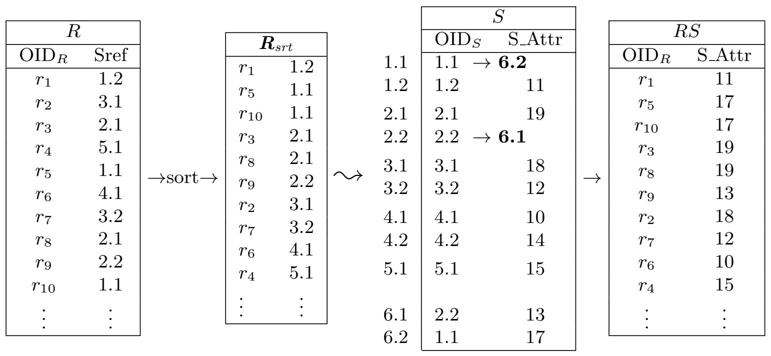
The idea of this technique is to *sort* the objects of table R in ascending order of the *page number* of the physical OIDs stored in the *Sref* fields. After this sorting step, the matching S objects can be read from disk sequentially. The whole process is shown in Fig. 7. (The bold typed **6.1** and **6.2** in the figure represent forwards. Note that we only sort by the *page number*, so r_1 and r_5 need not be swapped in table R_{srt}.)

3.4.2 Partition-based functional-join evaluation

The idea of this technique is to range-partition the R objects in such a way that all the S objects referenced by the R objects of a single partition fit into main memory. The query result is then obtained by joining (with the help of an in-memory hash table) the first partition of R objects with the first chunk of S objects, then joining the second partition of R objects with the second chunk of S objects, and so on. Figure 8 illustrates this process. As an alternative to range partitioning, it is also possible to partition the R objects using hashing. In this particular case, however, range partitioning is more attractive because it guarantees that all the S objects referenced by the objects in a single partition of R objects fit into main memory. Choosing the partition size for range partitioning should be supported by database statistics. If some partitions do not fit into memory, they have to be re-partitioned.

3.4.3 Dealing with forwards

There are two different ways to handle the occurrence of forwards while evaluating functional joins based on sorting or partitioning. The first approach is to immediately chase every forward, thereby accepting the penalty that additional page faults may cause page thrashing. This approach is only efficient if there are very few forwards. Alternatively, instead of immediately chasing the forwards, all forwards can be collected during the initial algorithm evaluation and processed in a separate pass afterwards. This way we avoid random I/O because we can sort or partition the forwards at the end. On the negative side, collecting forwards requires additional disk I/O and/or additional main-memory buffers.



R

OID_R	Sref
r_1	1.2
r_2	3.1
r_3	2.1
r_4	5.1
r_5	1.1
r_6	4.1
r_7	3.2
r_8	2.1
r_9	2.2
r_{10}	1.1
⋮	⋮

→sort→

$\boldsymbol{R}_{srt}$

r_1	1.2
r_5	1.1
r_{10}	1.1
r_3	2.1
r_8	2.1
r_9	2.2
r_2	3.1
r_7	3.2
r_6	4.1
r_4	5.1
⋮	⋮

⇝

S

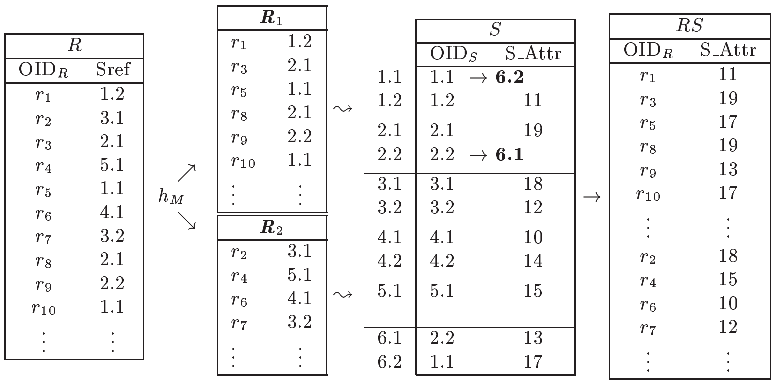
	OID_S	S_Attr
1.1	1.1	→ **6.2**
1.2	1.2	11
2.1	2.1	19
2.2	2.2	→ **6.1**
3.1	3.1	18
3.2	3.2	12
4.1	4.1	10
4.2	4.2	14
5.1	5.1	15
6.1	2.2	13
6.2	1.1	17

→

RS

OID_R	S_Attr
r_1	11
r_5	17
r_{10}	17
r_3	19
r_8	19
r_9	13
r_2	18
r_7	12
r_6	10
r_4	15
⋮	⋮

Fig. 7. Functional join with physical OIDs based on sorting



R

OID_R	Sref
r_1	1.2
r_2	3.1
r_3	2.1
r_4	5.1
r_5	1.1
r_6	4.1
r_7	3.2
r_8	2.1
r_9	2.2
r_{10}	1.1
⋮	⋮

h_M ↗ ↘

$\boldsymbol{R}_1$

r_1	1.2
r_3	2.1
r_5	1.1
r_8	2.1
r_9	2.2
r_{10}	1.1
⋮	⋮

$\boldsymbol{R}_2$

r_2	3.1
r_4	5.1
r_6	4.1
r_7	3.2
⋮	⋮

⇝

S

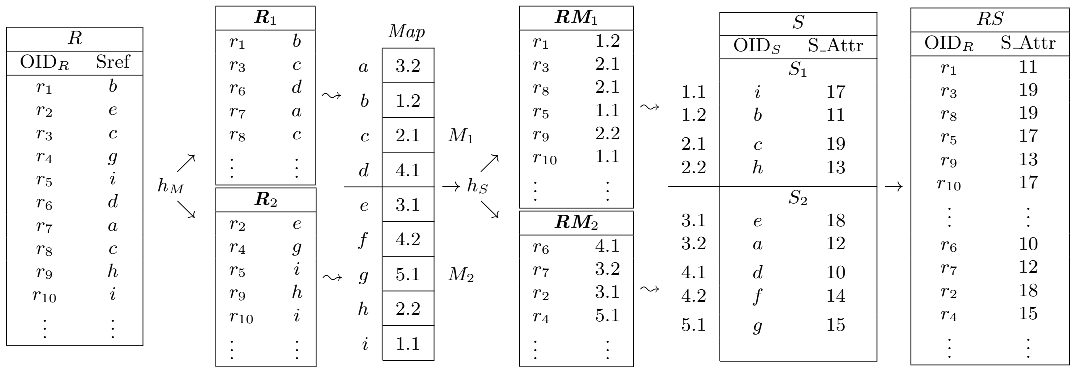
	OID_S	S_Attr
1.1	1.1	→ **6.2**
1.2	1.2	11
2.1	2.1	19
2.2	2.2	→ **6.1**
3.1	3.1	18
3.2	3.2	12
4.1	4.1	10
4.2	4.2	14
5.1	5.1	15
6.1	2.2	13
6.2	1.1	17

→

RS

OID_R	S_Attr
r_1	11
r_3	19
r_5	17
r_8	19
r_9	13
r_{10}	17
⋮	⋮
r_2	18
r_4	15
r_6	10
r_7	12
⋮	⋮

Fig. 8. Functinal join with physical OIDs based on partitioning



R

OID_R	Sref
r_1	b
r_2	e
r_3	c
r_4	g
r_5	i
r_6	d
r_7	a
r_8	c
r_9	h
r_{10}	i
⋮	⋮

h_M ↗ ↘

$\boldsymbol{R}_1$

r_1	b
r_3	c
r_6	d
r_7	a
r_8	c
⋮	⋮

$\boldsymbol{R}_2$

r_2	e
r_4	g
r_5	i
r_9	h
r_{10}	i
⋮	⋮

⇝

Map

a	3.2
b	1.2
c	2.1
d	4.1
e	3.1
f	4.2
g	5.1
h	2.2
i	1.1

M_1 M_2 → h_S ↗ ↘

$\boldsymbol{RM}_1$

r_1	1.2
r_3	2.1
r_8	2.1
r_5	1.1
r_9	2.2
r_{10}	1.1
⋮	⋮

$\boldsymbol{RM}_2$

r_6	4.1
r_7	3.2
r_2	3.1
r_4	5.1
⋮	⋮

⇝

S

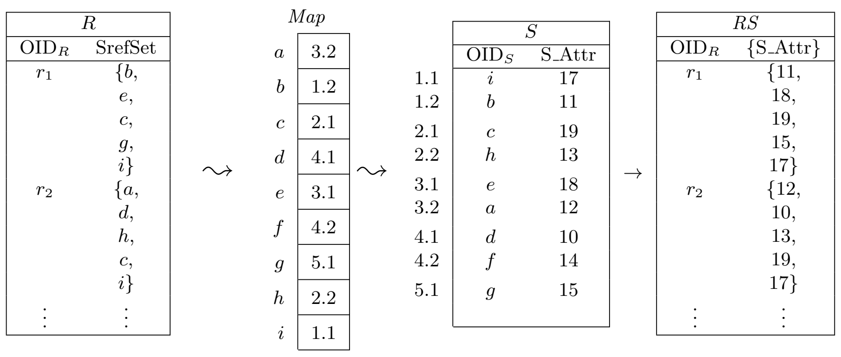
	OID_S	S_Attr
	S_1	
1.1	i	17
1.2	b	11
2.1	c	19
2.2	h	13
	S_2	
3.1	e	18
3.2	a	12
4.1	d	10
4.2	f	14
5.1	g	15

→

RS

OID_R	S_Attr
r_1	11
r_3	19
r_8	19
r_5	17
r_9	13
r_{10}	17
⋮	⋮
r_6	10
r_7	12
r_2	18
r_4	15
⋮	⋮

Fig. 9. Functional join with logical OIDs based on partitioning



R

OID_R	SrefSet
r_1	$\{b, e, c, g, i\}$
r_2	$\{a, d, h, c, i\}$
⋮	⋮

⇝

Map

a	3.2
b	1.2
c	2.1
d	4.1
e	3.1
f	4.2
g	5.1
h	2.2
i	1.1

⇝

S

	OID_S	S_Attr
1.1	i	17
1.2	b	11
2.1	c	19
2.2	h	13
3.1	e	18
3.2	a	12
4.1	d	10
4.2	f	14
5.1	g	15

→

RS

OID_R	{S_Attr}
r_1	{11, 18, 19, 15, 17}
r_2	{12, 10, 13, 19, 17}
⋮	⋮

Fig. 10. Naïve pointer chasing

3.5 Functional-join techniques for logical OIDs

In this sub-section, we will show how sort-based and partition-based functional-join-processing work in the presence of logical OIDs. The key observation is that we must carry out two functional joins in the presence of logical OIDs. That is, we can represent our example query as follows:

$$(R \bowtie \mathit{Map}) \bowtie S\,.$$

The second join (i.e., the join with S) can be executed using exactly the same sort-based and partition-based techniques as with physical OIDs (see the previous sub-section). To implement the first join with the mapping structure *Map* (i.e., $R \bowtie \mathit{Map}$), we can also apply sorting and partitioning, but we must respect the following restrictions which depend on the mapping technique.

Hashing. Sorting R directly by *Sref* and then probing the hash table in ascending *Sref* order does not make sense, because probing a hash table with ordered *Srefs* is just as expensive as with unordered *Srefs*. To take advantage of sorting and (hash or range) partitioning, we must first apply the hash function of the mapping structure to the *Sref* values in order to find the page numbers of the corresponding buckets of the hash table, and then sort or partition these page numbers.

B$^+$-trees. Here, directly sorting R by *Sref* is useful because sorting allows to read the leaves of the B$^+$-tree sequentially. Range partitioning is also viable, but it is less effective than sorting for physical OIDs because of the complex internal structure of a B$^+$-tree which makes it difficult to predict the right range predicates. Hash partitioning does not make sense for B$^+$-tree mapping.

Direct mapping. Sorting, range, and hash partitioning can naturally be applied. With direct mapping, a logical OID can be interpreted as a physical OID of a *handle* of an object.

Figure 9 illustrates how a functional join with directly mapped logical OIDs can be carried out. In this particular scenario, direct mapping is used, the first join, $R \bowtie \mathit{Map}$, is implemented using range partitioning of R, and the second join, the join with S, is also carried out using range partitioning. Comparing physical and logical OIDs, the price for logical OIDs is an additional join with the mapping table. On the positive side, systems with logical OIDs need not worry about *forwards* during functional-join processing. We will study this trade-off in more detail in Sect. 6.

4 Functional joins along nested reference sets

We now turn to a discussion of functional-join-processing techniques in the presence of multi-valued reference attributes. Multi-valued references are either present in the tables of the database – as assumed here – or created on the fly by special operators like nestjoin [SABdB94] or the binary grouping operator Γ [CM95]. We will extend our example from the previous section and then describe alternative functional-join techniques that exploit the pre-grouping given by the nested reference sets, assuming that the nested reference sets are clustered as done by all database systems we are aware of.

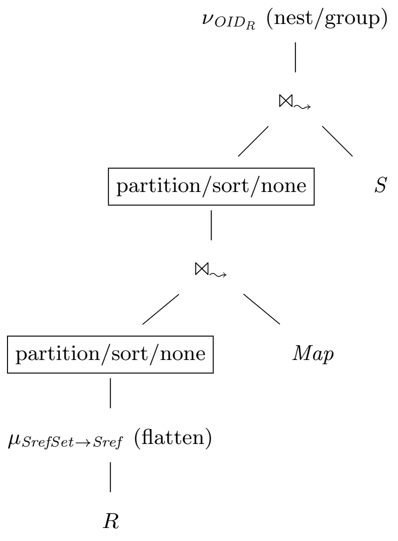


Fig. 11. Evaluation based on flattening/grouping

4.1 Example schema and queries

For the subsequent discussion of the algorithms, we change the example schema of Sect. 3 to include a set-valued reference attribute *SrefSet*:

```
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:[1]

```
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;
```

In both queries, the grouping that is given by the nested reference set *SRefSet* has to be maintained during (or restored after) the functional-join evaluation. In the query on the left-hand side, the elements of the nested sets are aggregated. In the query on the right-hand side, all elements of the nested sets are output. Asking for the sum of the prices of *Lineitems* of every *Order*, one of the queries mentioned in the introduction, is a more intuitive example that follows the pattern of the left query.

4.2 The naïve pointer-chasing algorithm

The naïve, pointer-chasing algorithm scans R and traverses every reference stored in the nested set *SrefSet* individually. For logical OIDs, again, the mapping structure must be probed to obtain the address of the referenced S object. If the combined size of the *Map* and S exceeds the memory capacity, this algorithm performs very poorly because its execution involves a great deal of random disk I/O.

[1] 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 query, and in OQL this query is also expressible (in a slightly different syntax, though).

4.3 The flatten algorithms

Flatten algorithms work in three steps. First, they flatten (unnest) the set-valued *SrefSet* attribute to obtain single-valued reference attributes. Second, a single-valued functional join is performed, using one of the special-purpose algorithms based on sorting or partitioning (Sects. 3.4 and 3.5) or a value-based functional-join method (Sect. 3.3). In the third step, the initial grouping is re-established by means of an (expensive) sorting or hashing operation. This three-step process is shown for logical OIDs in Fig. 11 for a sorting, partitioning, or do-nothing approach to implement the single-valued functional joins. If a value-based functional-join method is used, the join with the *Map* is not needed. Figure 12 shows an example evaluation using partitioning for both the *Map* and S.

4.4 The partition/merge algorithm $P(PM)^*M$ for logical OIDs

The naïve algorithm obviously suffers from a lack of locality, resulting in random I/O. The flatten algorithms incur high costs for restoring the grouping at the end. The question arises whether an algorithm can be designed that achieves locality *and* maintains the grouping. For this purpose, we proposed the partition/merge algorithm in [BCK98]. This algorithm 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 unary functional join under logical OIDs. More intermediate *PM*-steps are needed when longer functional-join chains are evaluated (cf. Sect. 4.5).

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.[2] 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., pre-fetched) 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:

$$\begin{array}{c} \text{flatten and } \textit{p}\text{artition} \\ R\ N\text{-way} \end{array} \rightarrow \begin{array}{c} \text{join with } \textit{Map} \text{ to obtain } \textit{RM} \\ \text{and } \textit{p}\text{artition } RM \\ (N*K)\text{-way} \end{array} \rightarrow \cdots$$

$$\cdots \rightarrow \begin{array}{c} \text{re-}\textit{m}\text{erge } \textit{RM} \text{ to } K \text{ partitions} \\ \text{and join with } S \end{array} \rightarrow \textit{m}\text{erge}$$

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 $[r, \{\textit{Sref}_1, \ldots, \textit{Sref}_l\}] \in R$ generate the l flat tuples $[r, \textit{Sref}_1], \ldots, [r, \textit{Sref}_l]$ and insert these tuples into their corresponding partitions $h_M(\textit{Sref}_1), \ldots, h_M(\textit{Sref}_l)$, 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. Sect. 4.6). The partitions are written to disk.
2. For all $1 \leq i \leq N$ 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 $[r, \textit{Sref}] \in R_i$, 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 $[r, \textit{Saddr}]$ into the partition RM_{ij}, where $j = h_S(\textit{Saddr})$.

   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 $1 \leq j \leq K$, 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 $[r, \textit{Saddr}]$, 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 $[r, \textit{S\_Attr}]$ into partition $\textit{RMS}_j$.

     All S objects referenced in this step belong to the j-th partition S_j of S which is pre-fetched or faulted

[2] 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.

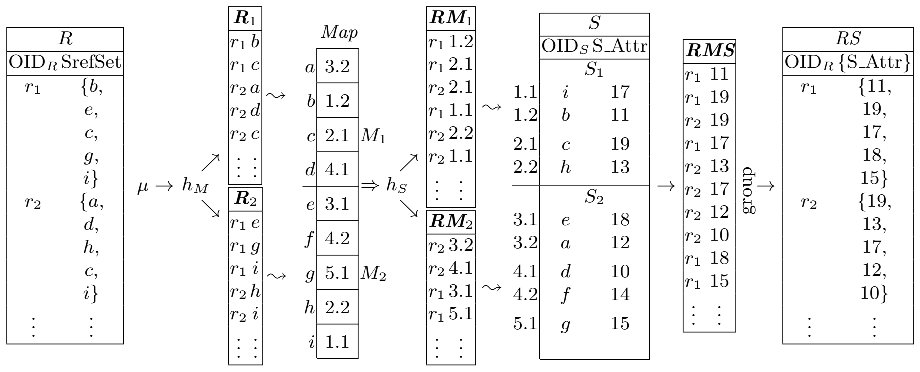


Fig. 12. Flattening, partitioning and (re-)grouping

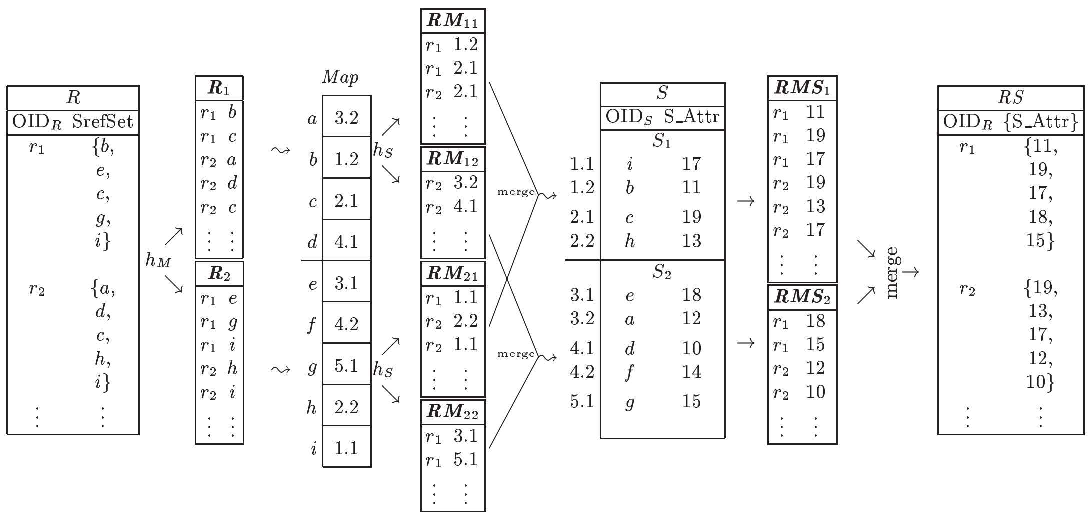


Fig. 13. Partition/merge-join

into the buffer – again, the partitioning ensures that the entire S_j fits into memory.

After completion of step 3, the K partitions *RMS*$_1$, ..., *RMS*$_K$ are on disk.

4. Scan all partitions *RMS*$_1$, ..., *RMS*$_K$ simultaneously and re-assemble the flat tuples into the nested representation, i.e., group the tuples according to R-OIDs.

For $N = K = 2$, the partition/merge algorithm is exemplified in Fig. 13, using range-partitioning functions for h_M and h_S. As emphasized in Fig. 14, 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. It is essential that the fine-grained partitions are generated first and that the re-merge is performed afterwards, as highlighted in Fig. 15.

In comparison, the partition/merge algorithm induces the same I/O-overhead as the basic flatten algorithms of Sect. 4.3. 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. Furthermore, the $P(PM)^*M$ algorithm gives room for optimizations based on the retained grouping that are not applicable to other algorithms (cf. Sect. 4.6).

4.5 $P(PM)^*M$ for physical OIDs and multi-way functional joins

The partition/merge algorithm is applicable for systems employing physical OIDs in the same way as for logical OIDs. In the simple case of a one-step functional join the variant $P(PM)^0M$ is applied, i.e., the plan then consists of a single partition and a single merge step, and no interleaved partition/merge operation is applied. However, the full-fledged $P(PM)^*M$ algorithm is necessary if the query traverses a longer path expression. Consider, for example, an additional type T (with attribute *T_Attr*) that is referenced by *S.Tref*. Then, a query may traverse the path expression $R \xrightarrow{SrefSet} S \xrightarrow{Tref} T$ as follows:

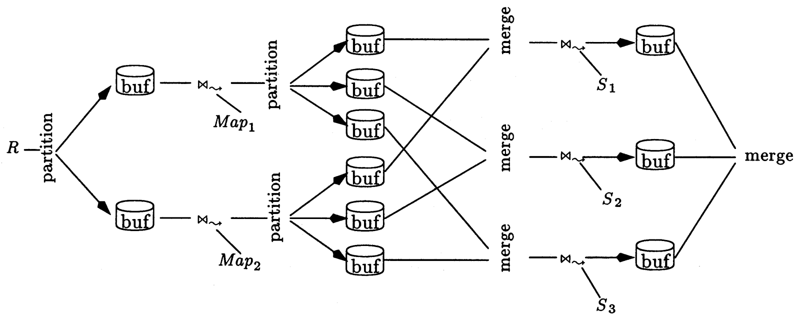


Fig. 14. Disk writes of the partition/merge algorithm

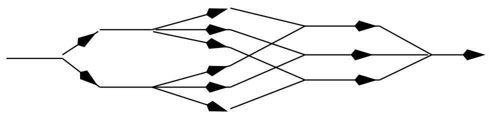


Fig. 15. The partition/merge-pattern of the $P(PM)^{*}M$ algorithm

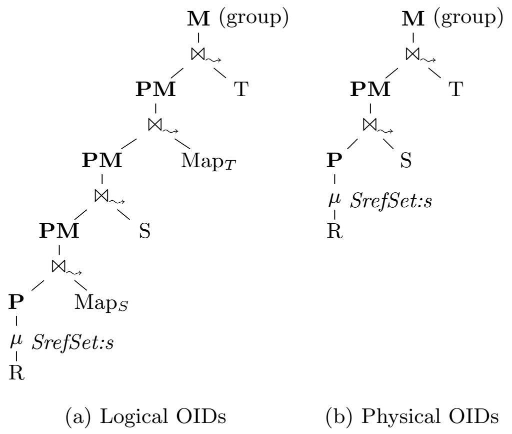


Fig. 16a,b. Plans for a query with a path expression

```
select r.∗, ( select sum(s.Tref.T_Attr)
              from r.SrefSet s )
from R r
```

Such a query may, for example, sum up the base prices of the *Products* of the *Lineitems* of all *Orders*.

The $P(PM)^{*}M$ evaluation plans for logical OIDs and physical OIDs are outlined in Fig. 16a and b, respectively. Both plans unnest the *SrefSet* – but in intermediate stages they retain the grouping of the same R objects by interleaved fine-grained partition/merge operations. When comparing the two plans, they differ mainly in the higher number of functional joins needed for mapping logical OIDs. We assume separate mapping structures Map_S and Map_T for *S* and *T*, 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 by R and S objects is retained across the successive functional joins. Therefore, the final grouping operation is carried out as a (very cheap) merge, in both plans.

4.6 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 would like to address in the following.

Obtaining an order on R. The algorithm requires an order on the R objects for the merge iterators. When comparing objects from different partitions – e.g., tuple [r_2,3.2] from RM_{12} and [r_1,3.1] from RM_{22} in Fig. 13 – it has to be determined in what order r_1 and r_2 were stored in the original R. If there is no such order given by the key of 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.

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 re-use in the final merge.

This procedure is illustrated in Fig. 17. Bypassing the bulky R attributes around the functional joins saves considerably in terms of I/O volume of the intermediate results (i.e., the writing of the *buf* operator – denoted as disk icons – and the subsequent reading from disk by the *merge* operator).

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

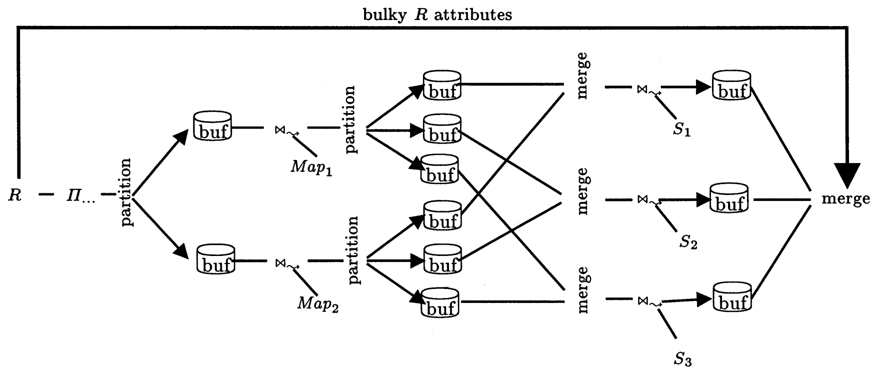


Fig. 17. Bypassing bulky R attributes around functional join processing

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 Fig. 14: 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.

5 Order-preserving functional joins

In this section, we apply the partition/merge algorithm to a wider range of queries that do not necessarily traverse along nested reference sets. The queries we optimize are those that require ordered results – as they occur very often in decision support systems.

5.1 Exploiting a physical order

The key idea in optimizing queries with ordered results follows from the observation that the partition/merge algorithm, when applied to an ordered (source) object extension R, will always preserve this order in the intermediate partitions. That is, the partitions generated by the interleaved partition/merge operations constitute valid runs with respect to the original sorting of R. In [CKK98], we exploited this idea in the context of the pure (flat) relational model to design the order-preserving hash join (OHJ) algorithm in order to optimize decision support queries that require sorting or flexible grouping (e.g., **cube** and **roll-up** aggregation). Here, we concentrate on this run-preserving invariant of the partition/merge algorithm for order-preserving functional joins.

In order to simplify this presentation we switch back to the example schema with single-valued references, as presented in Sect. 3. However, the order-preserving functional-join algorithm is equally applicable to queries traversing nested reference sets. Consider the example query of Sect. 3 with an additional **order by** clause:

select r.*, r.Sref.S_Attr
from R r
order by $r.\cdots$

First, we assume that the order required in the query is "physically" given by, e.g., a cluster index on R. The partition/merge algorithm, as illustrated in Fig. 19, evaluates the query and retains the initial order (indicated by ascending OID_R values) on R. All intermediate partitions R_1, R_2, …; RM_{11}, RM_{12}, …; RMS_1, RMS_2, … constitute runs with respect to the sort criterion of R. Then, the overall order is (cheaply) restored via a final merge operation. Note that, except for the naïve functional-join evaluation algorithm (cf. Fig. 5), all other functional-join evaluation algorithms (cf. Figs. 6–9) "lose" the original order of R. They all require a final costly sort operation on the result RS.

Assuming that there is a physical ordering (i.e., by a cluster index) on R, the partition/merge evaluation of this query is shown in Fig. 18a for logical OIDs and in Fig. 18b for physical OIDs. Of course, the order-preserving functional join is applicable to arbitrarily long functional-join chains (path expressions) which may also contain nested reference sets.

5.2 Sorting ahead

One might argue that our order-preserving functional-join technique is only efficient if there is a clustered index on R. Fortunately, however, we can generate the desired order on the fly during the initial partitioning step. This way we entirely avoid any additional I/O cost for sorting, and

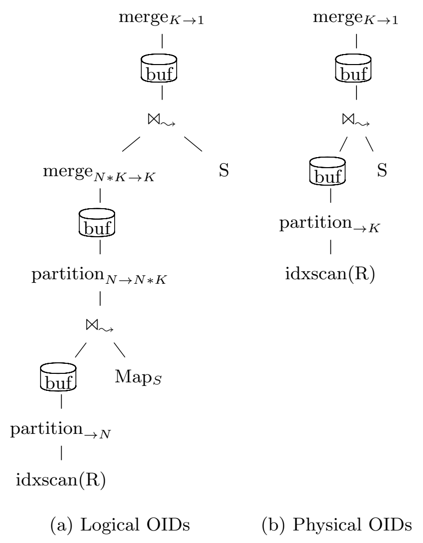


Fig. 18a,b. Order-preserving functional-join evaluation

therefore, as we will show in the performance section, we get (almost) the same performance in the presence as in the absence of a clustered index; that is, we get sorting (almost) for free [CKK98].

The trick is to combine the initial partitioning step of the order-preserving functional-join plan with sorting runs. That is, we sort memory-sized runs of the probe input and partition each run individually. The partitions of every run are then re-merged during the processing of the first join. Assume that R is M times bigger than the available main memory and, to perform the functional join with the *Map*, R has to be partitioned N-way using the hash function h_M. Then, for each $1 \leq i \leq M$, do:

1. Load the (next) memory-sized chunk R_i into memory and sort it according to attribute A.
2. Partition R_i into N partitions $R_{i1}, \ldots R_{iN}$ by applying h_M on the reference attribute. Each partition constitutes a valid run according to the sort attribute. The partitioning can be done in a single linear iteration through the main-memory-resident run R_i – see below.
3. Write the partitions $R_{i1}, \ldots R_{iN}$ sequentially to disk.

Having finished this combined sort/partitioning step, there are $M * N$ partitions $R_{11}, \ldots, R_{1N}, \ldots, R_{MN}$ – each constituting a valid sort run – stored on disk. Then, while evaluating the functional join with the i-th partition of the *Map*, the M runs $R_{1i}, \ldots, R_{Mi}$ are merged to obtain the i-th partition of R. From there on, the algorithm works just like the order-preserving partition/merge join algorithm.

The algorithm is illustrated for $M = 2$ and $N = 2$ in Fig. 20. Of course, these early sort plans can, therefore, also be applied to longer functional-join chain queries in the same way as described in the previous sub-section, and they can also be applied in the presence of nested reference sets. Tracing again the object extension R, the following pattern of operators are applied (here, $S\&P$ denotes the combined sorting and partitioning step):

$S\&P\ M\ (P\ M)^*\ M$

Figure 21 illustrates the combined sorting/partitioning phase of the algorithm. A memory-sized chunk of the relation is loaded. Sorting is done via a vector that maintains pointers to the objects being sorted; that is, only this vector is sorted, whereas the individual objects need not be moved. Once the sorting is complete, we linearly scan this vector and determine the partition to which every object belongs. Hereby, we chain objects that belong to the same partition together (i.e., we keep the index of the next object of the same partition in an additional field within the vector) and we keep a separate vector, called the *partition anchors*, in order to keep the heads and the tails of every of the N sorted "partition lists" (in the example of Fig. 21, $N = 2$). Once this partitioning is complete (i.e., the chaining is done and the heads and tails of the partition anchors are set), the objects can be written sequentially to disk: partition by partition following the heads of the partition anchors one at a time and in the right sort order. All partitions could, for example, be written into a single temporary file by inserting markers at partition boundaries, thereby avoiding overhead for allocating multiple temporary files. Note that Fig. 21 shows, in fact, the generation of the partitions R_{11} and R_{12} for run R_1 of Fig. 20.

With respect to run-time complexity, it would be cheaper to first partition each complete memory chunk and then sort the individual partitions. Assuming $m = |R|/M$ records fit into one memory chunk, first sorting and then partitioning a memory chunk takes $m \log m + m$ abstract "operations." The opposite order, i.e., first partitioning a memory chunk and then sorting each of the N partitions requires only $m + N \cdot m/N \cdot \log(m/N) = m + m \cdot \log(m/N)$ operations. However, memory management for the partition/sort variant is more complex than for the sort/partition algorithm, because several sort vectors of unknown size have to be allocated. We have implemented both variants and our performance experiments have shown that the difference in run time is only marginal for the investigated configurations.

Sorting ahead of the functional join is especially beneficial if in the course of evaluating the join the result size is increasing. This happens if R contains a nested set of references (i.e., *SrefSet*). A concrete example is a query that computes the *Order* values sorted by order date and summing up the product prices. Under the assumption that no physical order can be exploited, the evaluation plan of Fig. 22a shows the early sort plan for logical OIDs and Fig. 22b shows the analogous plan for physical OIDs.

6 Performance analysis

In this section, we will present the results of performance experiments that study the trade-offs of the alternative functional-join-processing techniques. We will first describe our experimental environment, including a prototype implementation and a detailed cost model that accurately models the behavior of the techniques in a standard database system. After that, we will present results for single-valued functional joins, multi-valued functional joins, and results that demonstrate the benefits of our order-preserving functional-join techniques.

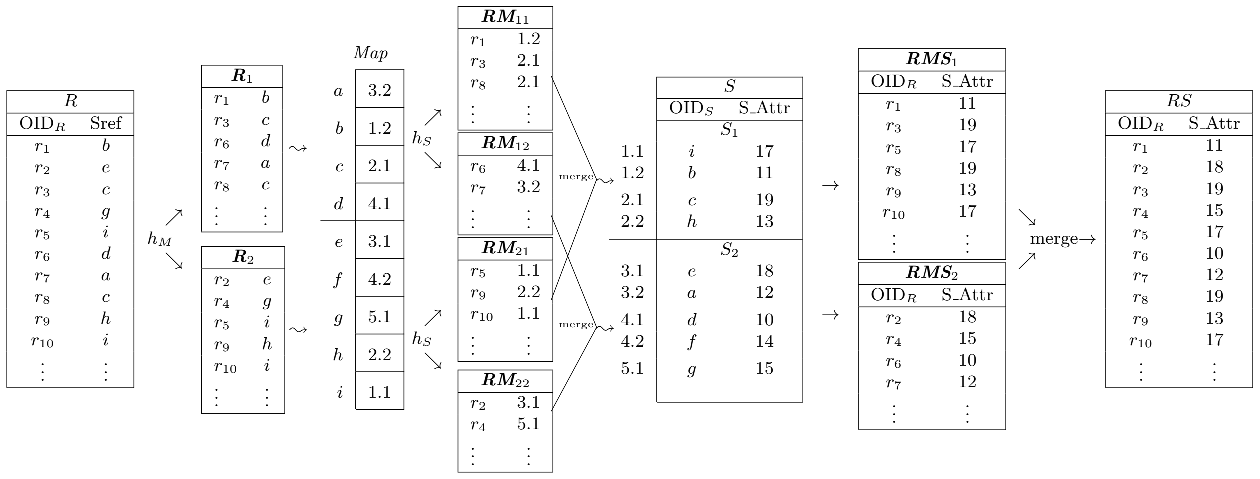


Fig. 19. Order-preserving partition/merge-join

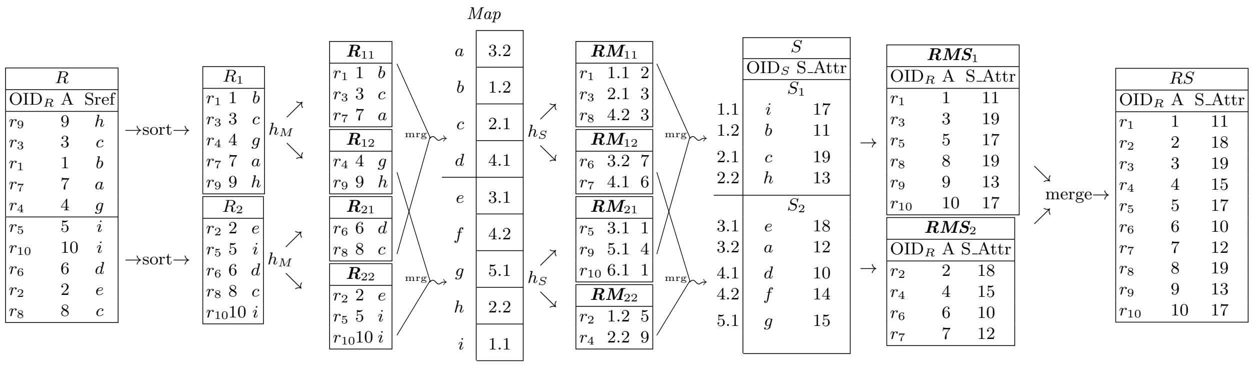


Fig. 20. Sorting on the fly

6.1 Experimental environment

6.1.1 The cost model

The design of our cost model is strongly influenced by the structure of modern query engines implementing the iterator model. This means that cost estimations are calculated on a per-iterator basis. I/O costs are modeled 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 formulae. The cost formulae 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 modeling 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.

For space limitations and ease of presentation, we only describe the cost formulae of the iterators needed for implementing the partition/merge algorithm as shown in Fig. 13. That is, we only present the formulae for direct mapping and if range partitioning on the references to the *Map* and S and prefetching for reading these sets into main memory are used. For other strategies, other formulae apply. We would like to emphasize, however, that we did use the right formulae in order to obtain performance results.

Table 3. Parameters describing the hardware

T_S	average seek time	10.2 ms
T_L	average latency time	5.54 ms
T_T	transfer time for a page (4 KB)	1.7 ms
T_{IO}	time to initiate an I/O operation	1.21 ms
T_{hash}	time to execute a hash function	0.285 ms
T_{add}	time to add two integers	0.00719 ms
T_{probe}	time to test a hash table	0.02339 ms
T_{copy}	time to copy a byte	0.000115 ms
T_{comp}	time to compare two OIDs	0.00719 ms

The cost model parameters for modeling the CPU and I/O costs are described in Table 3. Note that the constants regarding CPU costs include all instructions related to the operations, e.g., T_{comp} involves pointer arithmetics, etc. and not only a single CPU instruction. The cost model variables that describe characteristics of the database are described in Table 4.

6.1.2 Analysis of I/O cost

The $P(PM)^*M$ algorithm with the above-mentioned premises has very similar I/O access patterns throughout all its phases. Therefore, we describe the patterns and list

Table 4. Variables used in the cost model formulae

P_R	number of pages in table R (equivalently for *RM* and *RMS*)
$\|R\|$	cardinality of table R
r	average size of an R object
b	read/write buffer size (in pages)
N	number of partitions
$\|SrefSet\|$	average number of elements in the nested reference set

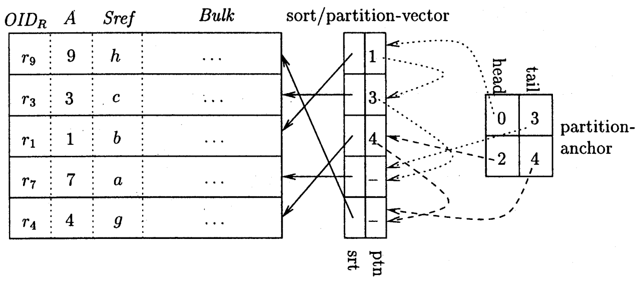


Fig. 21. Sorting and partitioning "in one go"

the phases in the algorithm where the pattern shows up. In the cost formulae, it is assumed that no inter-operator interference occurs. The number of additional seeks caused by interference would be calculated separately and added to the cost of the algorithm. Up to now, we are only able to model interference if just one disk is used at all. In our benchmarks, however, we used two disks and – although three disks would be necessary to avoid all interference effects in the investigated algorithms – we decided to neglect interference. Furthermore, we assume constant seek times here.

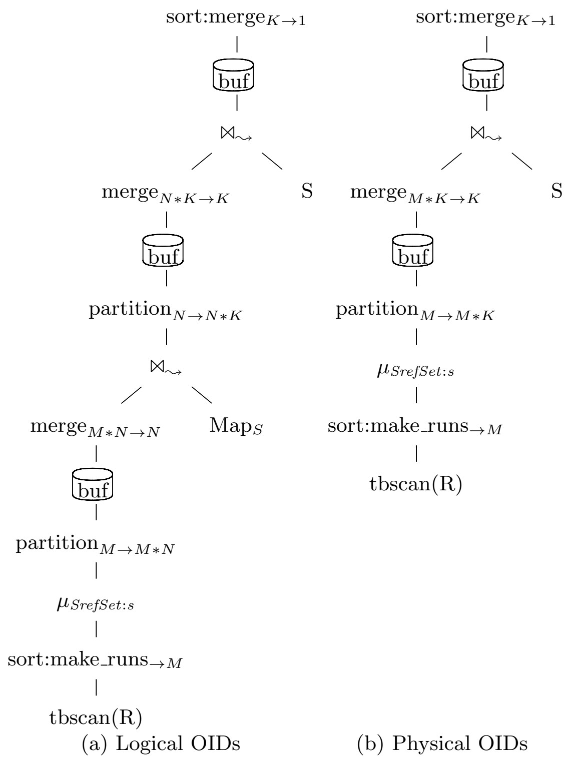


Fig. 22a,b. Sorting on the fly during functional-join processing

Reading from disk. We denote the number of pages read in one I/O operation as b. The merge operator uses a buffer of b pages for each input partition. The cost for reading *RM* (and also analogously for reading *RMS*) by the merge operator is then given by the following formula:

$$\left\lceil \frac{P_{RM}}{b} \right\rceil \cdot (T_S + T_L + T_{IO}) + P_{RM} \cdot T_T \,.$$

For the initial reading of R, the cost can be computed as

$$T_S + \left\lceil \frac{P_R}{b} \right\rceil \cdot (T_L + T_{IO}) + P_R \cdot T_T \,.$$

For the scan operator reading the first set of partitions R_i ($1 \le i \le N$), we get

$$N \cdot T_S + \left\lceil \frac{P_R}{b} \right\rceil \cdot (T_L + T_{IO}) + P_R \cdot T_T \,.$$

Here N is the number of partitions generated by the preceding partition operator. For the join operator[3], the same formula can be used, except that R has to be replaced by *Map* or S, respectively.

Writing to disk. We use the same variable b for the buffer size as before. The cost for the write operations of the partition iterator for partitioning R (and also analogously for partitioning *RM*) can be computed by the following formula:

$$\left\lceil \frac{P_R}{b} \right\rceil \cdot (T_S + T_L + T_{IO}) + P_R \cdot T_T \,.$$

The cache iterator which is applied on *RMS* produces smaller cost with its writing operations:

$$T_S + \left\lceil \frac{P_{RMS}}{b} \right\rceil \cdot (T_L + T_{IO}) + P_{RMS} \cdot T_T \,.$$

6.1.3 Analysis of CPU cost

Again the actions consuming CPU time are listed together with their cost formulae and the iterators performing those actions.

Copying of elements. The cost formula for copying all elements of a set X in main memory:

$$|X| \cdot x \cdot T_{copy} \,.$$

The small x denotes the size of an element in a set $X \in \{R, RM, RMS\}$. This action is performed by all the iterators writing temporary sets to disk, especially the partition and the cache iterator, and by operators using in-memory working areas like sort and hash.

Comparing elements. The merge iterator has to compare sequence numbers attached to each element for detecting those stemming from the same element of an initial input set. The cost for such an operation is

[3] The join operators read *Map* and S.

$|R| \cdot |\mathit{SrefSet}| \cdot \log_2(N) \cdot T_{comp}$.

Here, the variable N denotes the number of partitions merged into one partition by the merge iterator. Since the ordering of elements is done by a tournament tree, we only have to perform $\log_2(N)$ comparisons for each element in R.

Computing hash functions. For each join attribute in its input set, the partition iterator has to call a hash function:

$|R| \cdot |\mathit{SrefSet}| \cdot T_{hash}$.

Performing aggregation. For each element in R, we have to add an integer value for every element in the nested set:

$|R| \cdot |\mathit{SrefSet}| \cdot T_{add}$.

Testing the buffer. Each join iterator in a partition/ merge algorithm uses a buffer for reading S and *Map*. For each join attribute in the set R, the join iterator has to look up the buffer for the appropriate page. We assume that this lookup is done by accessing a hash table. Then the cost can be computed by

$|R| \cdot |\mathit{SrefSet}| \cdot T_{probe}$.

6.1.4 Prototype implementation

Most of our performance experiments were carried out using the cost model. To validate the cost model and get a feeling for the trade-offs of the algorithms in a real system, we also carried out certain experiments using an experimental object-relational database system. (Since we carried out a great deal of experiments with many different database configurations, we were not able to carry out all experiments with this prototype.) The prototype database system we used for these experiments is very much a textbook database system in which we integrated all the different functional-join algorithms (e.g., P(PM)*M). The experiments with this experimental database system were carried out on a Sun SparcStation 20 running under Solaris 2.6. There was one disk that stored the database and all the software needed, and there was another disk which was used for temporary files. In order to avoid side effects due to file system caching, we made use of Solaris' direct I/O option, so that all disk I/O was carried out bypassing the cache of the file system. Displaying the result tuples was suppressed for all queries in order to study the sheer performance of the functional-join-processing algorithms. The database buffer cache was segmented and configured individually for every query plan according to the estimates of our detailed cost model.

6.1.5 Test database and test queries

Unless stated otherwise, the analyses are based on a simple database with tables R and S, which we have been using in our examples throughout this paper; that is, the objects in R contain references to objects in S. The cardinalities of the two tables are shown in Table 5. In the first set of analyses, we investigated functional joins along single-valued reference attributes; in this case, $|$*SrefSet*$|$ was fixed at 1. (In fact, R objects only had a single-valued attribute called *Sref* in these experiments; see Sect. 3.1.) In the second and third set of analyses, we investigated functional joins along nested reference sets; in those experiments, we varied $|$*SrefSet*$|$. (In these experiments, R objects had a nested reference set called *SrefSet*; see Sect. 4.1.)

Table 5. Database cardinalities

object type	cardinality	data pages	*Map* pages	object size
$\|R\|$	100,000	9933	–	228 + $\|\mathit{SrefSet}\| * 12$
$\|S\|$	100,000	6667	591	228

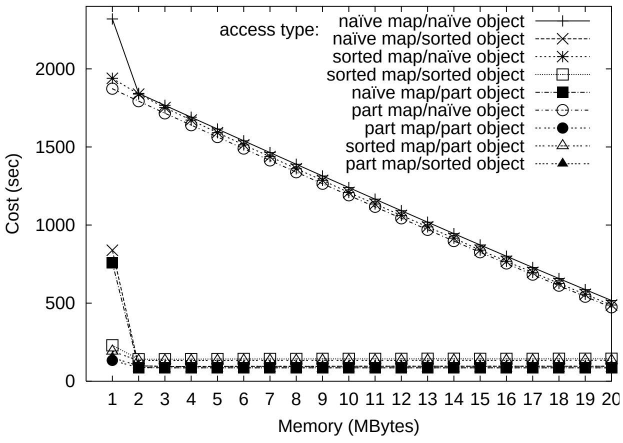


Fig. 23. Pointer-based joins with direct mapping

The benchmark queries we used are, also, those that we have been using in the examples throughout this paper. For the first set of analyses, the one that studies the trade-offs of the single-valued functional-join techniques, we used the query of Sect. 3.1. For the second set of analyses, the one that studies the trade-offs of the functional-join techniques along nested reference sets, we used the "aggregation" query of Sect. 4.1. For the third set of analyses, the one that demonstrates the benefits of order-preserving functional-join techniques, we used the query of Sect. 5.1, with the only difference that the functional join was carried out along a nested reference set (i.e., r.SrefSet.S_Attr was retrieved and ordered).

6.2 Functional joins along single-valued reference attributes

Figures 23–25 show the running times of the different functional-join-processing strategies for single-valued attributes with logical OIDs and direct mapping, a B$^+$-tree, and a hash table, respectively. The size of the available main memory is varied from 1 MB to 20 MB. For each curve, the strategy to evaluate the functional joins is given. For example, "part map/sorted object" in Fig. 23 means that the $R \bowtie$ *Map* join is carried out using (range-) partitioning and that

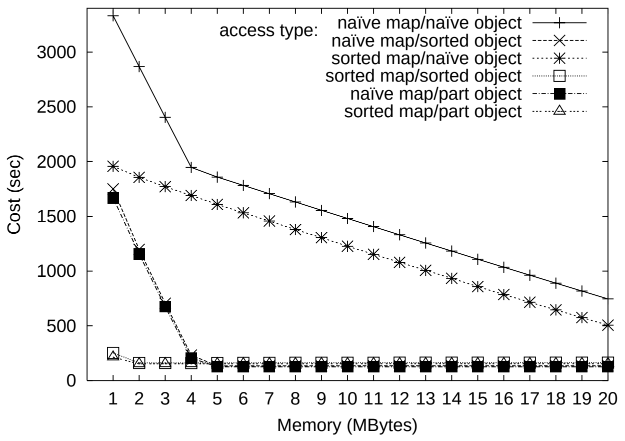


Fig. 24. Pointer-based joins with a B^+-tree mapping

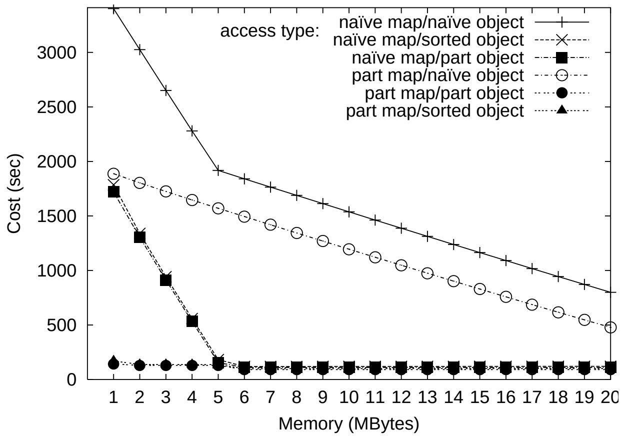


Fig. 25. Pointer-based joins with hash table mapping

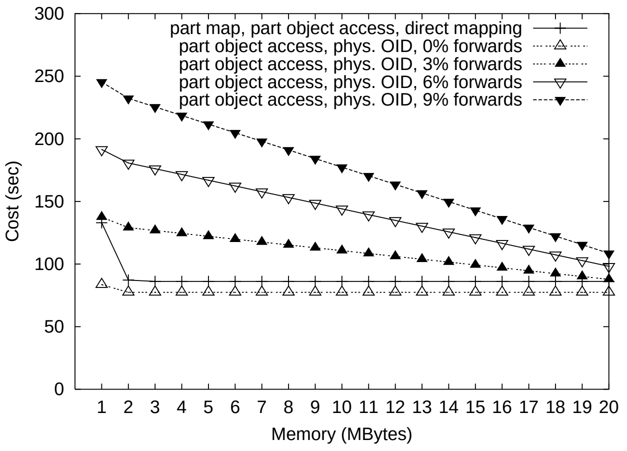


Fig. 27. Logical OIDs versus physical OIDs

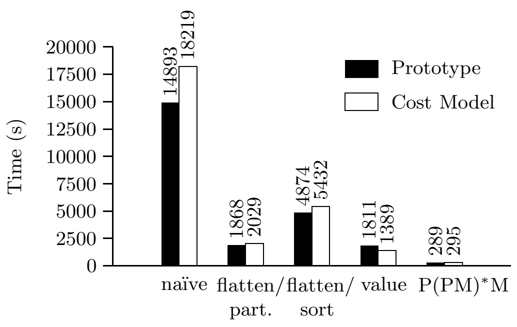


Fig. 28. Running times in seconds (2 MB memory, avg. |*SrefSet*| = 10, direct mapping)

sorting is employed for the join with S. We use short forms like "part/sort" in the following. What we can see, in all, is that sorting and/or partitioning are crucial techniques to carry out functional joins with logical OIDs efficiently, regardless of whether direct mapping, B^+-trees, or hash tables are used. The "do-nothing" reference traversal approach is only viable if the functional join can be carried out in main memory. With direct mapping, the $R \bowtie$ *Map* join can be carried out in memory if more than 2 MB are available. With a B^+-tree and a hash table, the $R \bowtie$ *Map* join can be carried out in memory at 4 MB and 5 MB, respectively, because in these cases the mapping structures are larger.

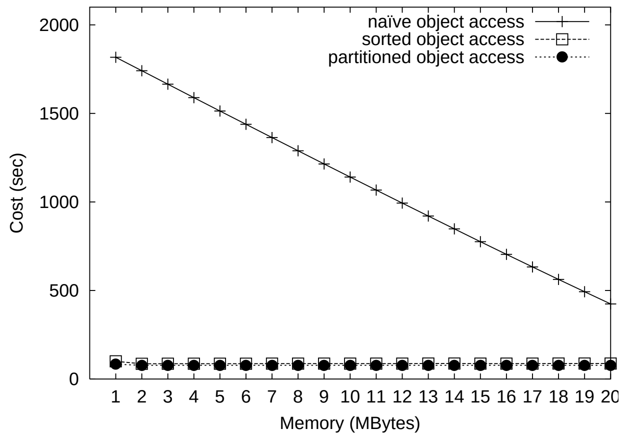


Fig. 26. Pointer-based joins with physical OIDs

Figure 26 shows the functional-join performance with physical OIDs in the absence of forwards. Again, we observe the same effect: partitioning and/or sorting are crucial in order to achieve acceptable response times. In this particular case, partitioning is slightly better than sorting.

Figure 27 shows the response time of the partition-based functional-join evaluation with physical OIDs in the presence of forwards. We observe that with an increasing degree of forwards, the performance of physical OIDs gets worse, in particular for small memory sizes. As a baseline, the figure also shows the running times of the query for a double partitioning strategy with logical OIDs and direct mapping, the best strategy for logical OIDs. Obviously, physical OIDs outperform logical OIDs in the absence of forwards (the 0% curve). However, the performance of logical OIDs is not far behind if more than 2 MB of memory are available (i.e., if the $R \bowtie$ *Map* join can be carried out in memory), and the performance of logical OIDs is better even if as few as 3% of the S objects have been migrated (the 3% forwards curve).

6.3 Joins along nested reference sets

We now turn our attention to evaluating functional joins along nested reference sets. First, we describe some cost measures we obtained from our prototype system. Thereafter, we investigate a broader range of database configurations using our cost model.

6.3.1 Results with the prototype implementation

The benchmarks were performed with the prototype system described above. The database buffer cache was segmented and configured according to the optimizer (cost model) estimation individually for each query plan. The total amount of memory available to a query did not exceed 2 MB at any time. Direct mapping was employed to resolve logical OIDs.

Recall that we concentrate on the "aggregation" query of Sect. 4.1 in our presentation. For the other example query of Sect. 4.1, we observed similar effects and trade-offs. In our prototype, we restricted the size of the main-memory buffer pool to 2 MB in order to run these queries using the alternative algorithms presented in Sect. 4. Figure 28 gives an overview of the running times for each algorithm. For comparison, the predictions of our cost model are also shown (again, limiting the buffer size to 2 MB). When comparing the $P(PM)^*M$ running time to the naïve algorithm, there is a performance gap of more than an order of magnitude: The absolute running time of the naïve algorithm amounts to more than 5 h, while our $P(PM)^*M$ algorithm requires only less than 5 min. The $P(PM)^*M$ algorithm also significantly outperforms all the flatten algorithms, the state-of-the-art approaches for this purposes. The flatten plans all suffer from the expensive "re-grouping": the sort-based flatten plan suffers from high CPU cost for sorting and from small run files due to the restricted amount of memory. The partition-based flatten plan and the value-based join cannot keep its complete build input in memory and, as a consequence, has to perform an expensive hash aggregation at the end.

There is a small deviation between the cost model figures and the running times observed with the prototype implementation. This is mostly due to the fact that some cost model constants are hard to calibrate. They have been measured by profiling; profiling, however, changes the total running time of the queries. Most cost model estimations are therefore slightly higher than the observed running time. The running time of the value-based hash join implementation is slightly higher than predicted by the cost model since the I/O operations of our hash join are currently not implemented as efficiently as assumed by the cost model.

6.3.2 Varying the memory size

Figure 29 shows the running times (using the cost model) of the various algorithms under varying memory sizes. The naïve plan (denoted NN) does not even show up in the plot due to its running time of 6'20 h for 1 MB to 4'10 h for a 6-MB buffer. The naïve/sort plan uses a naïve *Map* lookup, but sorts the physical addresses before accessing S; it, therefore, requires flattening and grouping. The sort/sort plan uses sorting for both the *Map* lookup and for the join with S; it also

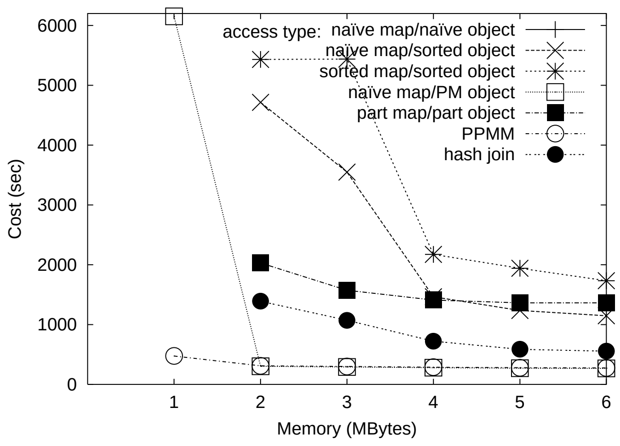


Fig. 29. Cost model results (direct mapping, |*SrefSet*| = 10)

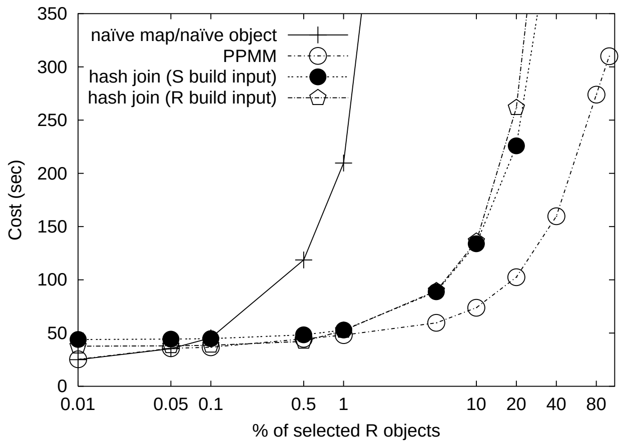


Fig. 30. Selection on R (direct mapping, |*SrefSet*| = 10, 2 MB memory

involves flattening and re-grouping. Comparing naïve/sort with sort/sort, 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 variants suffer from high CPU costs for sorting. The part/part plan which is also

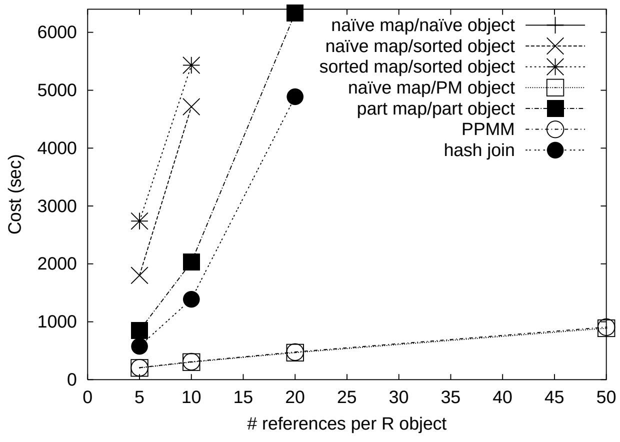


Fig. 31. Varying the cardinality of *SrefSet* (2 MB memory, direct mapping)

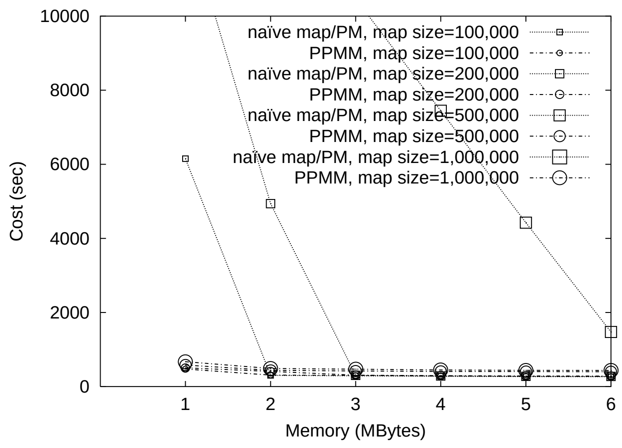


Fig. 32. Inflating the OID map under varying memory sizes (direct mapping)

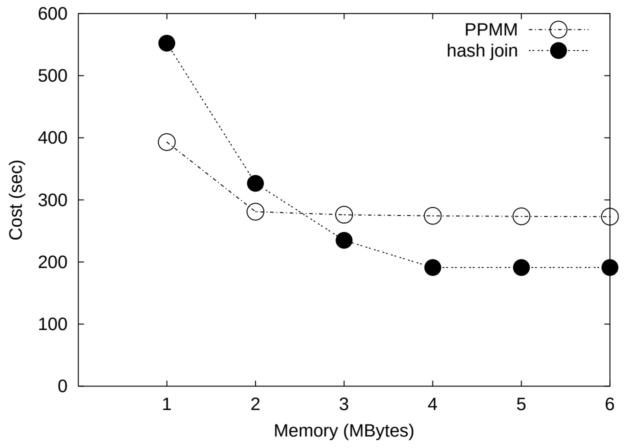


Fig. 33. Value-based vs. $P(PM)^*M$ pointer join: |*SrefSet*| = 3, direct mapping, 1,000,000 map entries

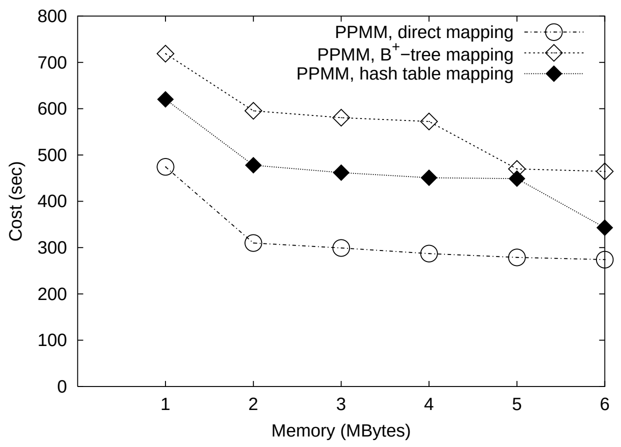


Fig. 34. Comparison of different OID mapping techniques, all with $P(PM)^*M$ algorithm

a flatten plan and uses partitioning for both the *Map* lookup and the join with S yields significantly better performance than both sort-based flatten 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 plan which also involves flattening and re-grouping performs even better than part/part, but is still quite costly compared to the winners PPMM (=$P(PM)^1M$) and naïve/PM (=$P(PM)^0M$). 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 running time, since PPMM uses only one partition for the *Map* access anyway and, therefore, coincides with naïve/PM. 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.

6.3.3 Varying the selectivity on *R*

In Fig. 30, the percentage of R objects taking part in the functional joins is varied on the (logarithmically scaled) x-axis; that is, in this experiment, we studied a variant of our benchmark query with a **where** clause and a selection predicate filtering out tuples of R. 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 naïve algorithm. However, for a high selectivity (e.g., 0.01% corresponding to 10 R objects), they have nearly the same running 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 naïve 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 naïve plan.

6.3.4 Varying the set cardinality

In the previous experiments, the number of elements in *SrefSet* was constantly 10. Figure 31 shows running times of the algorithms with different set sizes. While the $P(PM)^*M$ algorithm scales linearly, the running times for all others explode. The flatten variants behave poorly. The naïve plan suffers from an enormous amount of random I/O (up to $50 * 100,000$ references, calculated running time of roughly 25 h and therefore not shown) and the flatten plans suffer from large temporary files.

6.3.5 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

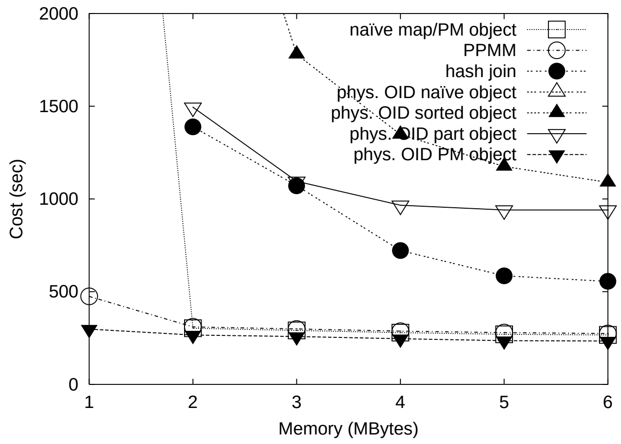


Fig. 35. Physical OIDs vs. logical OIDs with direct mapping

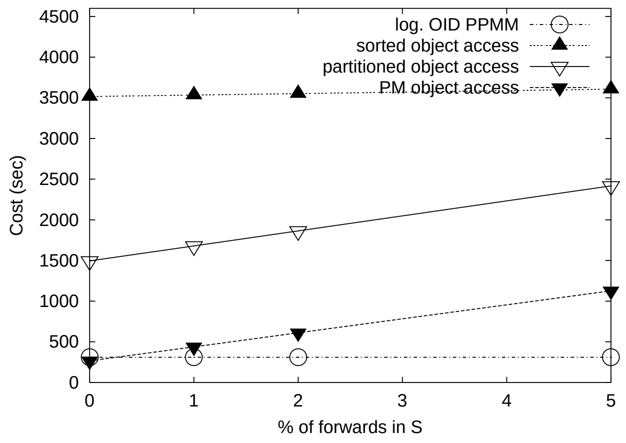


Fig. 36. Effect of forwards

million entries. The naïve/PM 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 Fig. 32 indicates the size of the *Map* (100,000, ..., 1,000,000). The smallest symbols denote the configuration that was used in Fig. 29, i.e., the *Map* was optimally clustered. For larger *Maps*, the PPMM plan shows only a slight running 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, naïve/PM 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.

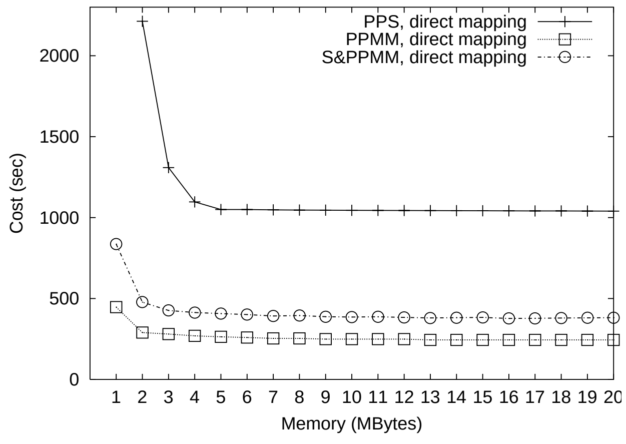


Fig. 37. Order-preserving functional joins

Figure 33 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 approximately 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.

6.3.6 Comparing different OID mapping techniques

Figure 34 compares the three OID mapping techniques that we have discussed in Sect. 2.2 for our application, i.e., in combination with the $P(PM)^*M$ algorithm. Both B$^+$-tree and hash table mapping 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,[4] this point is reached with a smaller memory size for the B$^+$-tree curve. In addition, B$^+$-trees are generally more expensive due to higher CPU cost for the tree lookup. The direct mapping 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 direct mapping than for the other two mapping techniques.

6.3.7 Logical OIDs in comparison to physical OIDs

So far, we have assessed our set-aware algorithms using logical OIDs only. Next, we turn to physical OIDs and see how the performance of functional joins with physical OIDs compares to that of functional joins with logical OIDs. The use of physical OIDs simplifies all algorithms, since the extra

[4] Due to prefix compression and the specialized splitting procedure described in Sect. 2.2.1 the B$^+$-tree contains more entries per page than the hash table.

Map lookup operation is omitted. Thus, the functional-join algorithms for physical OIDs are no partitioning (denoted as "phys. OID naïve object access"), sorting, partitioning, and $P(PM)^0M$ (labelled PM). The value-based hash join is independent of the kind of OID used. For comparison, Fig. 35 additionally includes the naïve/PM and PPMM plans for logical OIDs realized with direct mapping. The naïve plan does not show up in the plot, since it ranges between 4 and 5 h. The running time of the partition plan 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 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 without forwards, they incur a severe performance penalty in the presence of forwards. Again (as in the experiments shown in Fig. 27), we studied database instances with a varying degree of S objects that were migrated in the range of 0–5%. Figure 36 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 (using a chase-forward-immediately approach, see Sect. 3.4), whereas partition-based plans need to allocate more buffer for a partition containing forwards (using a "collect-forward" approach which is better in this case). Partitioning 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.

6.4 Order-preserving functional joins

Figure 37 shows the performance of running a functional-join query which returns its results ordered by an attribute of R. We consider three different evaluation strategies:

– PPS. This query evaluation plan flattens the nested *SrefSet*, performs the functional join with the *Map* and with S by partitioning, and then sorts the result.
– S&PPMM. This plan combines the first partitioning with sorting runs, then performs the functional join using the partition/merge algorithm which, in its final merge step, generates the desired order of the result.
– PPMM. This query evaluation plan requires the extent R to be pre-ordered (i.e., via a cluster index) and merely performs the order-preserving partition/merge algorithm, which automatically delivers the tuples in the desired order.

We should emphasize that the PPMM plan requires a different database configuration than the other two plans, i.e., a clustered index on the sort attribute is required. Therefore, it is impressive, that the sort-ahead evaluation plan (S&PPMM) is only about 20% slower. On the other hand, the conventional evaluation plan which performs the functional join first and then sorts the result takes a factor 2.5 longer than the sort-ahead plan. The difference to the order-preserving plan PPMM is even more pronounced – but keep in mind that the order-preserving plan relies on an existing physical ordering, which the sort-ahead and the conventional plans do not require.

7 Summary

This paper gives a comprehensive overview and assessment of alternative query-processing techniques for functional joins. First, the implementation techniques for logical OIDs were contrasted with the physical OID realizations. Then, the alternative functional-join evaluation techniques for single-valued reference attributes were described. 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. For this purpose, a new algorithm that is based on successively partitioning and merging was developed. This so-called partition/merge algorithm retains the grouping within the partitions and restores the overall grouping by (efficient) merge operations. The partition/merge algorithm could be adapted to become an order-preserving functional-join algorithm. This new order-preserving functional-join evaluation allows to exploit an existing ordering of the object extent or to push-down the sorting in the evaluation plan. This proves to be a very effective optimization if the join result's cardinality is larger than the sort relation's cardinality – as it is the case when evaluating functional joins along nested reference sets. Further enhancements of the partition/merge functional join plans reduce the size of intermediate results. The bulk-bypassing technique allows to bypass the large attributes of the sort relation around the join processing and early aggregation is applicable in group-by queries. Our quantitative assessment based on a detailed cost model and a prototype implementation proves that the partition/merge algorithm applied to group preservation as well as to order preservation is superior to other, traditional functional-join methods. Furthermore, our experiments demonstrate that the penalty for using logical OIDs in an object-oriented or object-relational database system is very low as compared to the use of physical OIDs, and that logical OIDs are significantly better than physical OIDs, even if only a small percentage of objects are migrated.

Acknowledgements. We thank the anonymous referees for helpful suggestions to improve the paper. Stefan Börszönyi pointed out the performance enhancement of the partition/sort sequence.

References

[BCK98] Braumandl R, Claussen J, Kemper A (1998) Evaluating functional joins along nested reference sets in object-relational and object-oriented databases. In: Proc. of the Conf. on Very Large Data Bases (VLDB), August 1998, New York, N.Y., pp 110–121

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

[BM72] Bayer R, McCreight EM (1972) Organization and maintenance of large ordered indices. Acta Informatica 1(3): 173–189

[BP95] Biliris A, Panagos E (1995) A high performance configurable storage manager. In: Proc. IEEE Conf. on Data Engineering, March 1995, Taipeh, Taiwan, pp 35–43

[BR90] Brown A, Rosenberg J (1990) Persistent object stores: An implementation technique. In: Dearle A, Shaw G, Zdonik S (eds) Implementing Persistent Object Bases, Principles and Practice. Morgan Kaufmann, San Mateo, Calif., pp 199–212

[CD92] Cluet S, Delobel C (1992) A general framework for the optimization of object-oriented queries. In: Proc. of the ACM SIGMOD Conf. on Management of Data, June 1992, San Diego, Calif., pp 383–392

[CDF+94] Carey M, DeWitt D, Franklin M, Hall N, McAuliffe M, Naughton J, Schuh D, Solomon M, Tan C, Tsatalos O, White S, Zwilling M (1994) Shoring up persistent applications. In: Proc. of the ACM SIGMOD Conf. on Management of Data, May 1994, Minneapolis, Mich., pp 383–394

[CDRS86] Carey M, DeWitt D, Richardson J, Shekita E (1986) Object and file management in the EXODUS extensible database system. In: Proc. of the Conf. on Very Large Data Bases (VLDB), August 1986, Kyoto, Japan, pp 91–100

[CKK98] Claussen J, Kemper A, Kossmann D (1998) Order-preserving hash joins: Sorting (almost) for free. Technical Report MIP-9810. University of Passau, 94030 Passau, Germany

[CM95] Cluet S, Moerkotte G (1995) Classification and optimization of nested queries in object bases. Technical Report 95–6. RWTH Aachen, Germany

[Com79] Comer D (1979) The ubiquitous B-tree. ACM Comput Surv 11(2): 121–137

[CSL+90] Carey MJ, Shekita E, Lapis G, Lindsay B, McPherson J (1990) An incremental join attachment for Starburst. In: Proc. of the Conf. on Very Large Data Bases (VLDB), August 1990, Brisbane, Australia, pp 662–673

[DLM93] DeWitt D, Lieuwen D, Mehta M (1993) Parallel pointer-based join techniques for object- oriented databases. In: Proc. of the Int. IEEE Conf. on Parallel and Distributed Information Systems, January 1993, San Diego, Calif., pp 172–181

[ED88] Enbody RJ, Du HC (1988) Dynamic hashing schemes. ACM Comput Surv 20(2): 85-113

[EGK95] Eickler A, Gerlhof C, Kossmann D (1995) A performance evaluation of OID mapping techniques. In: Proc. of the Conf. on Very Large Data Bases (VLDB), September 1995, Zurich, Switzerland, pp 18–29

[EKK97] Eickler A, Kemper A, Kossmann D (1997) Finding data in the neighborhood. In: Proc. of the Conf. on Very Large Data Bases (VLDB), August 1997, Athens, Greece, pp 336–345

[GGT96] Gardarin G, Gruser J-R, Tang Z-H (1996) Cost-based selection of path expression processing algorithms in object-oriented databases. In: Proc. of the Conf. on Very Large Data Bases (VLDB), September 1996, Bombay, India, pp 390–401

[GKG+97] Grust T, Kröger J, Gluche D, Heuer A, Scholl MH (1997) Query evaluation in CROQUE - calculus and algebra coincide. In: Proc. British National Conference on Databases (BNCOD), July 1997, London, UK, pp 84–100

[GR93] Gray J, Reuter A (1993) Transaction Processing: Concepts and Techniques. Morgan Kaufmann, San Mateo, Calif.

[Här78] Härder T (1978) Implementing a generalized access path structure for a relational database system. ACM Trans Database Syst 3(3): 285–298

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

[HR96] Harris E, Ramamohanarao K (1996) Join algorithm costs revisited. VLDB J 5(1): 64–84

[HZ87] Hornick M, Zdonik S (1987) A shared, segmented memory system for an object-oriented database. ACM Trans Off Inf Syst 5(1): 70–95

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

[KC86] Khoshafian SN, Copeland GP (1986) Object identity. In: Proc. of the ACM Conf. on Object- Oriented Programming Systems and Languages (OOPSLA), November 1986, Portland, Or., pp 406–416

[KGM91] Keller T, Graefe G, Maier D (1991) Efficient assembly of complex objects. In: Proc. of the ACM SIGMOD Conf. on Management of Data, May 1991, Denver, Colo., pp 148–158

[KM90] Kemper A, Moerkotte G (1990) Access support in object bases. In: Proc. of the ACM SIGMOD Conf. on Management of Data, April 1990, Atlantic City, N.J., pp 364–374

[Lit80] Litwin W (1980) Linear hashing: A new tool for file and table addressing. In: Proc. of the Conf. on Very Large Data Bases (VLDB), October 1980, Montreal, Canada, pp 212–223

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

[LMB97] Leverenz L, Mateosian R, Bobrowski S (1997) Oracle8 Server - Concepts Manual. Oracle Corporation, Redwood Shores, Calif.

[LR99] Li Z, Ross KA (1999) Fast joins using join indices. VLDB J 8(1): 1–24

[MGS+94] Maier D, Graefe G, Shapiro L, Daniels S, Keller T, Vance B (1994) Issues in distributed object assembly. In: Özsu T, Dayal U, Valduriez P (eds) Distributed Object Management (International Workshop on Distributed Object Management), May 1994, Morgan Kaufmann, San Mateo, Calif., pp 165–181

[MS87] Maier D, Stein J (1987) Development and implementation of an object-oriented DBMS. In: Shriver B, Wegner P (eds) Research Directions in Object-Oriented Programming. MIT Press, Cambridge, Mass., pp 355–392

[O2T94] O2 Technology (1994) A Technical Overview of the O2 System. O2 Technology, Versailles Cedex, France

[Obj96] Objectivity, Inc (1996) Objectivity Technical Overview, Version 4, June 1996. Objectivity, Inc; http://www.objectivity.com/

[PCV94] Patel J, Carey M, Vernon M (1994) Accurate modeling of the hybrid hash join algorithm. Proc. of the ACM SIGMETRICS, May 1994, Santa Clara, Calif., pp 56–66

[SABdB94] Steenhagen HJ, Apers PMG, Blanken HM, By RA de (1994) From nested-loop to join queries in OODB. In: Proc. of the Conf. on Very Large Data Bases (VLDB), September 1994, Santiago, Chile, pp 618–629

[SC90] Shekita E, Carey M (1990) A performance evaluation of pointer-based joins. In: Proc. of the ACM SIGMOD Conf. on Management of Data, May 1990, Atlantic City, N.J., pp 300–311

[SG89] Segev A, Gunadhi H (1989) Event-join optimization in temporal relational databases. In: Proc. of the Conf. on Very Large Data Bases (VLDB), 1989, Amsterdam, The Netherlands, pp 205–215

[Sto96] Stonebraker M (1996) Object-Relational DBMSs: The Next Great Wave. Morgan Kaufmann, San Mateo, Calif.

[Val87] Valduriez P (1987) Join indices. ACM Trans Database Syst 12(2): 218–246

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

[WW90] Williams I, Wolczko M (1990) An object-based memory architecture. In: Dearle A, Shaw G, Zdonik S (eds) Implementing Persistent Object Bases, Principles and Practice. Morgan Kaufmann, San Mateo, Calif., pp 114–130

[XH94] Xie Z, Han J (1994) Join index hierarchies for supporting efficient navigations in object- oriented databases. In: Proc. of the Conf. on Very Large Data Bases (VLDB), September 1994, Santiago, Chile, pp 522–533