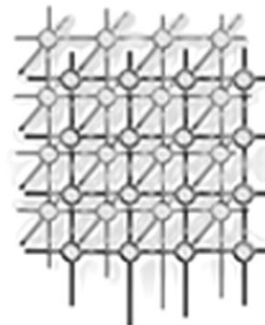


Measuring and modelling the performance of a parallel ODMG compliant object database server



Sandra de F. Mendes Sampaio¹, Norman W. Paton^{1,*},
Jim Smith² and Paul Watson²

¹*Department of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, U.K.*

²*Department of Computer Science, University of Newcastle upon Tyne, Newcastle upon Tyne NE1 7RU, U.K.*

SUMMARY

Object database management systems (ODBMSs) are now established as the database management technology of choice for a range of challenging data intensive applications. Furthermore, the applications associated with object databases typically have stringent performance requirements, and some are associated with very large data sets. An important feature for the performance of object databases is the speed at which relationships can be explored. In queries, this depends on the effectiveness of different join algorithms into which queries that follow relationships can be compiled. This paper presents a performance evaluation of the Polar parallel object database system, focusing in particular on the performance of parallel join algorithms. Polar is a parallel, shared-nothing implementation of the Object Database Management Group (ODMG) standard for object databases. The paper presents an empirical evaluation of queries expressed in the ODMG Query Language (OQL), as well as a cost model for the parallel algebra that is used to evaluate OQL queries. The cost model is validated against the empirical results for a collection of queries using four different join algorithms, one that is value based and three that are pointer based. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: object database; parallel databases; ODMG; OQL; benchmark; cost model

1. INTRODUCTION

Applications associated with object databases are demanding in terms of their complexity and performance requirements [1]. However, there has not been much work on parallel object databases,

*Correspondence to: Norman W. Paton, Department of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, U.K.

†E-mail: norm@cs.man.ac.uk

Contract/grant sponsor: Engineering and Physical Sciences Research Council; contract/grant number: GR/M76607



and few complete systems have been constructed. As a result, there has been still less work on the systematic assessment of the performance of query processing in parallel object databases. This paper presents a performance evaluation of different algorithms for exploring relationships in the parallel object database system Polar [2].

The focus in this paper is on the performance of Object Database Management Group (ODMG) Query Language (OQL) queries over ODMG databases, which are compiled in Polar into a parallel algebra for evaluation. The execution of the algebra, on a network of PCs, supports both inter- and intra-operator parallelism. The evaluation focuses on the performance of four parallel join algorithms, one of which is value based (hash join) and three of which are pointer based (materialize, hash loops and tuple-cache hash loops). Results are presented for queries running over the medium 007 database [3], the most widely used object database benchmark.

The experiments and the resulting performance figures can be seen to serve two purposes: (i) they provide insights on algorithm selection for implementers of parallel object databases; (ii) they provide empirical results against which cost models for parallel query processing can be validated (e.g. [4,5]).

The development of cost models for query performance is a well-established activity. Cost models are an essential component of optimizers, whereby physical plans can be compared, and they have been widely used for studying the performance of database algorithms (e.g. [6–8]) and architectures (e.g. [9]). However, cost models are only as reliable as the assumptions made by their developers, and it is straightforward to identify situations in which researchers make seemingly contradictory assumptions. For example, in parallel join processing, some researchers use models that discount the contribution of the network [10], while others pay considerable attention to it [11]. It is possible that in these examples the assumptions made by the authors were appropriate in their specific contexts, but it is easy to see how misleading conclusions could be drawn on the basis of inappropriate assumptions. This paper also presents cost models that have been validated against the experimental results presented, and can therefore also be used as a way of explaining where time is being spent during query evaluation.

2. RELATED WORK

2.1. Parallel database systems

The previous parallel real-time database management system (RDBMS) projects that have most influenced our work are EDS and Goldrush. In the 1980s, the EDS project [12] designed and implemented a complete parallel system including hardware, operating system, and a database server that was basically relational, but did contain some object extensions. This ran efficiently on up to 32 nodes. The ICL Goldrush project [13] built on these results and designed a parallel RDBMS product that ran parallel Oracle and Informix. Issues tackled in these two projects that are relevant to the parallel object database management system (ODBMS) include concurrency control in parallel systems, scalable data storage, and parallel query processing. Both of these projects used custom-built parallel hardware. In Polar we have investigated an alternative, which is the use of lower-cost commodity hardware in the form of a cluster of PCs.



Research in parallel object databases can probably be considered to belong to one of two principal areas—the development of object database models and query languages specifically for use in a parallel setting, and techniques to support the implementation of object models and query processors in a parallel setting. A thorough discussion of the issues of relevance to the development of a parallel object database server is given in [14].

An early parallel object database project was Bubba [15], which had a functional query language FAD. Although the Bubba model and languages probably influenced the later ODMG model, FAD provides more programming facilities than OQL. There has also been a significant body of work produced at the University of Florida [16,17], both on language design and query processing algorithms. However, the language on which this is based [16] seems less powerful than OQL, and the query processing framework is significantly different; it is not obvious to us that it can be adapted easily for use with OQL. Another parallel object-based database system is PRIMA [18], which uses the MAD data model and a SQL-like query language, MQL. The PRIMA system's architecture differs considerably from Polar's, as it is implemented as a multiple-level client-server architecture, where parallelism is exploited by partitioning the work associated with a query into a number of service requests that are propagated through the layers of the architecture in the form of client and server processes. However, the mapping of processes onto processors is accomplished by the operating system of the assumed shared memory multiprocessor. Translating the PRIMA approach to the shared-nothing environment assumed in Polar appears to be difficult.

There has been relatively little work on parallel query processing for mainstream object data models. The only other parallel implementation of an ODMG compliant system that we are aware of is Monet [19]. This shares with Polar the use of an object algebra for query processing, but operates over a main-memory storage system based on vertical data fragmentation that is very different from the Polar storage model. As such, Monet really supports the ODMG model through a front-end to a binary relational storage system.

There has also been work on parallel query processing in object relational databases [20]. However, object relational databases essentially operate over extended relational storage managers, so results obtained in the object relational setting may not transfer easily to an ODMG setting.

2.2. Evaluating join algorithms

The most straightforward pointer-based join involves dereferencing individual object identifiers as relationships are explored during query evaluation. This can be represented within an algebraic setting by the materialize operator [21]. More sophisticated pointer-based join algorithms seek to coordinate relationship following, to reduce the number of times that an object is navigated to. For example, six uni-processor pointer-based join algorithms are compared in [4]. The algorithms include value-based and pointer-based variants of nested-loop, sort-merge and hybrid-hash joins. This work is limited in the following aspects: (i) a physical realization for object identifiers (OIDs) is assumed, not allowing for the possibility of logical OIDs; (ii) in the assessments, only single-valued relationships are considered; (iii) it is assumed that there is no sharing of references between objects, i.e. two objects do not reference the same object; (iv) only simple queries with a single join are considered; and (v) the performance analysis is based on models that have not been validated through system tests.



In [8], the performance of sequential join algorithms was compared through a cost model and an empirical evaluation. The algorithms include the value-based hash-join, the pointer-based nested-loop, variations of the partition/merge algorithm which deal with order preservation, and other variations of these three which deal with different implementations of object identifiers. Results from the empirical evaluation were used to validate some aspects of the cost model, but most of the experiments were carried out using the model. The empirical results were obtained through experimentation on a prototype object-relational database system. The algorithms were tested by running navigational queries that require order preservation in their results, and different implementations to deal with logical and physical OIDs were tested for each algorithm. Thus, the scope is quite different from that of this paper. Running times for the different joins were considered in the measurements. The reported results show that the partition/merge algorithm applied to order preservation is superior to other traditional navigational joins. Furthermore, the results demonstrate that using logical OIDs rather than physical OIDs can be advantageous, especially in cases where objects are migrated from one physical address to another.

There has been considerable research on the development and evaluation of parallel query processing techniques, especially in the relational setting. For example, an experimental study of four parallel implementations of value-based join algorithms in the Gamma database machine [22] was reported in [23].

In [6], four hash-based parallel pointer-based join algorithms were described and compared. The comparisons were made through analysis, and the algorithms were classified into two groups: (i) those that require the existence of an explicit extent for the inner collection of referenced objects; and (ii) those that do not require such an extent, and which access stored objects directly. For the cases where there is not an explicit extent for the referenced objects, the proposed find-children algorithm is used with the algorithms of group (i) to compute the implicit extent. One of the joins of group (ii) is a version of the parallel hash-loops join. Single join tests were performed using a set-valued reference attribute, and it was shown that if data is relatively uniformly distributed across nodes, pointer-based join algorithms can be very effective.

In [24], the ParSets approach to parallelizing object database applications is described. The applications are parallelized through the use of a set of operations supported in a library. The approach is implemented in the Shore persistent object system, and was used to parallelize the 007 benchmark traversals by the exploitation of data parallelism. Performance results show the effectiveness and the limitations of the approach for different database sizes and numbers of processors. However, ParSets are considered in [24] for use in application development, not query processing, so the focus is quite different from that of this paper.

In [11], multi-join queries are evaluated under different parallel execution strategies, query plan tree shapes and numbers of processors on a parallel relational database. The experiments were carried out on the PRISMA/DB, and have shown the advantages of bushy trees for parallelism exploitation and the effectiveness of the pipelined, independent and intra-operator forms of parallelism. The results reported in [11] differ from the work in this paper in focusing on main-memory databases, relational query processing and alternative tree shapes rather than alternative join algorithms.

In [7], a parallel pointer-based join algorithm is analytically compared with the multiwavefront algorithm [25] under different application environments and data partitioning strategies, using the 007 benchmark. In contrast with [7], this paper presents an empirical evaluation as well as model-based results.



2.3. Cost models

This section describes several results on cost models for query evaluation, focusing in particular on results that have been validated to some extent.

In a relational setting, a cost model for analysing and comparing sequential join algorithms was proposed in [26]. Only joins using pre-computed access structures and join indices are considered, and the joins are compared by measuring only their I/O costs. The scope is thus very different from the experiments reported here. Certain aspects of the cost model were compared with experimental results, which showed that the analytical results were mostly within 25% of their experimental counterparts.

In [27], a cost model is proposed to predict the performance of sequential *ad hoc* relational join algorithms. A detailed I/O cost model is presented, which considers latency, seek and page transfer costs. The model is used to derive optimal buffer allocation schemes for the joins considered. The model is validated through an implementation of the joins, which reports positively on the accuracy of the models, which were always within 8% of the experimental values and often much closer. The models reported in [27] are narrower in scope but more detailed than those reported here.

Another validated cost model for sequential joins in relational databases is [28]. As in [27], a detailed I/O model was presented, and the models also considered CPU costs to be important for determining the most efficient method for performing a given join. The model was also used to optimize buffer usage, and examples were given that compare experimental and modelled results. In these examples, the experimental costs tend to be less than the modelled costs due principally to the presence of an unmodelled buffer cache in the experimental environment.

An early result on navigational joins compared three sequential pointer-based joins with their value-based counterparts [4]. The model takes account of both CPU and I/O costs, but has not been validated against system results. More recent work on navigational joins is reported in [8], in which new and existing pointer-based joins are compared using a comprehensive cost model that considers both I/O and CPU. A portion of the results were validated against an implementation, with errors in the predicted performance reported in the range 2–23%.

The work most related to ours is probably [6], in which several parallel join algorithms, including the hash-loops join used in this paper, were compared through an analytical model. The model in [6] considers only I/O, and its formulae have been adapted for use in this paper. As we use only single-pass algorithms, our I/O formulae are simpler than those in [6]. The model, however, has not been validated against system results, and a shared-everything environment is assumed. In our work, a more scalable shared-nothing environment is used.

In more recent work on parallel pointer-based joins, a cost model was used for comparing two types of navigational join, the hybrid-hash pointer-based join and the multiwavefront algorithm [7]. A shared-nothing environment is assumed, and only I/O is considered. The model is not validated against experimental results.

Another recent work that uses both the analytical and empirical approaches for predicting the performance of database queries is [29]. This work discusses the performance of the parallel Oracle Database System running over the ICL Goldrush machine. Instead of a cost model, a prediction tool namely STEADY is used to obtain the analytical results and those are compared against the actual measurements obtained from running Oracle over the Goldrush machine. The focus of the comparison is not on specific query plan operators, such as joins, but on the throughput and general response time of queries.

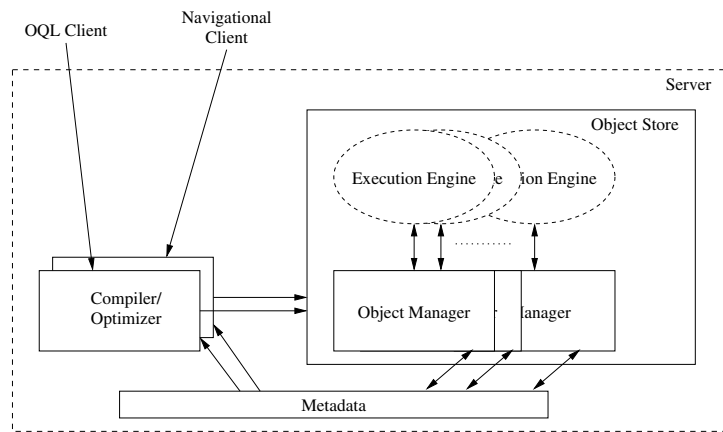


Figure 1. Polar architecture overview.

3. POLAR SYSTEM

Figure 1 presents an overview of the Polar architecture. An OQL client supports textual input of OQL expressions, forwarding each OQL expression to a query compiler and waiting for results. A navigational client executes an application written in a programming language, which makes an arbitrary traversal over the database, but may also employ embedded OQL expressions.

The fundamental source of parallelism is the partitioning of application data over separate physical disks managed by separate store units, and the partitioning of the computation entailed in its access across the processors of the server and clients. Typically, a store unit is mapped to a processor.

An OQL expression undergoes logical, physical and parallel optimization to yield a data flow graph of operators in a physical algebra (query plan), which is distributed between object store and client. The operators are implemented according to the *iterator model* [30], whereby each implements a common interface comprising the functions *open*, *next* and *close*, allowing the creation of arbitrary pipelines. The operators in a query plan manipulate generic structures, *tuples*, derived from object states. As described in [2], parallelism in a query is encapsulated in the *exchange* operator, which implements a partition between two threads of execution, and a configurable data redistribution, the latter implementing a flow control policy.

Figure 2 shows how the runtime services are used in the navigational client and store unit. At the lowest level there is basic support for the management of untyped objects, message exchange with other store units and multi-threading. On top of this, a storage service supports page-based access to objects either by OID directly or through an iterator interface to an extent partition. In support of inter-store navigation, the storage service can request and relay pages of objects stored at other store units. The other main services are the query instantiation and execution service and the support for communications within a query, encapsulated in *exchange*. The language binding and object cache

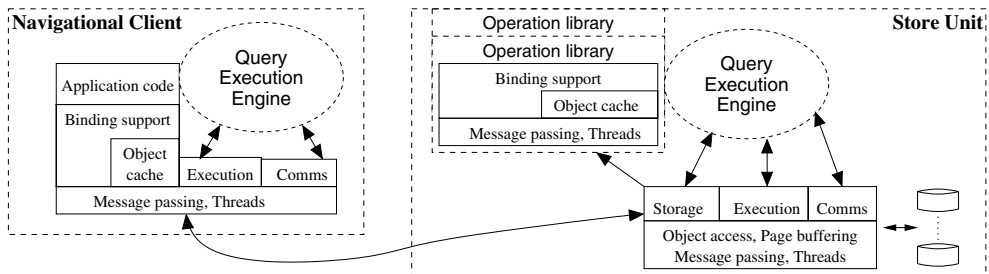


Figure 2. Runtime support services.

are employed by an application in a navigational client and an operation in a library of operations, but are also employed in a query compiler and an OQL client to support access to the distributed metadata.

The query execution engine implements the algorithms of the operators of the physical algebra. The four join algorithms within the physical algebra are as follows.

- *Hash-join*. The Polar version of hash-join is a one-pass implementation of the relational hash-join implemented as an iterator. This algorithm hashes the tuples of the smallest input on their join attribute(s), and places each tuple into a main memory hash table. Subsequently, it uses the tuples of the largest input to probe the hash table using the same hash function, and tests whether the tuple and the tuples that have the same result for the hash function satisfy the join condition.
- *Materialize*. The materialize operator is the simplest pointer-based join, which performs naive pointer chasing. It iterates over its input tuples, and for each tuple reads an object, the OID of which is an attribute of the tuple. Dereferencing the OID has the effect of following the relationship represented by the OID-valued attribute. Unlike the hash-join described previously, materialize does not retain (potentially large) intermediate data structures in memory, since the only input to materialize does not need to be held onto by the operator after the related object has been retrieved from the store. The pages of the related objects retrieved from the store may be cached for some time, but the overall space overhead of materialize is small.
- *Hash-loops*. The hash-loops operator is an adaptation for the iterator model of the pointer-based hash-loops join proposed in [6]. The main idea behind hash-loops is to minimize the number of repeated accesses to disk pages without retaining large amounts of data in memory. The first of these conflicting goals is addressed by collecting together repeated references to the same disk pages, so that all such references can be satisfied by a single access. The second goal is addressed by allowing the algorithm to consume its input in chunks, rather than all at once. Thus, hash-loops may fill and empty a main memory hash table multiple times to avoid keeping all of the input tuples in memory at the same time. Once the hash-table is filled with a number of tuples, each bucket in the hash table is scanned in turn, and its contents are matched with objects retrieved from the store. Since the tuples in the hash table are hashed on the page number of the objects specified in the inter-object relationship, each disk page is retrieved from the store only once within each *window* of input tuples. Once all the tuples that reference objects on a



particular page have been processed, the corresponding bucket is removed from the hash table, and the next page, which corresponds to the next bucket to be probed, is retrieved from the store. Thus, hash-loops seeks to improve on materialize by coordinating accesses to persistent objects, which are likely to suffer from poor locality of reference in materialize.

- *Tuple-cache hash-loops*. The tuple-cache hash-loops operator is a novel enhancement of the hash-loops operator that incorporates a tuple-cache mechanism to avoid multiple retrievals of the same object from the store and its subsequent mapping into tuple format. This is done by placing the tuple generated from each retrieved object into a main memory table of tuples, indexed by the OID of the object, when the object is retrieved for the first time. A subsequent request for the same object is performed by first searching the table of tuples for a previously generated tuple for the particular object. When the OID of the object is not found in the table, the object is retrieved from the store and tuple transformation takes place. As each bucket is removed from the hash table, the tuples generated from the objects retrieved during the processing of a particular bucket may be either removed from the table of tuples or kept in the table for reuse. If the hash table is filled and emptied multiple times, it may be desirable to keep the tuples generated within a window of input tuples for the next windows. Thus, tuple-cache hash-loops seeks to improve on hash-loops by decreasing the number of object retrievals and object-tuple transformations for the cases when there is object sharing between the input tuples, at the expense of some additional space overhead. The minimum additional space overhead of tuple-cache hash-loops relative to hash-loops depends on the number of distinct objects retrieved from the store per hash table bucket.

4. EMPIRICAL EVALUATION

This section describes the experiments performed to compare the performance of the four join algorithms introduced in the previous section. The experiments involve four queries with different levels of complexity, offering increasing challenges to the evaluator. The queries have been designed to provide insights on the behaviour of the algorithms when performing object navigation in parallel. In particular, the queries explore single and double navigations through single- and multiple-valued relationships over the 007 [3] benchmark schema.

4.1. The 007 database

Database benchmarks provide tasks that can be used to obtain a performance profile of a database system. By using benchmarks, database systems can be compared and bottlenecks can be found, providing guidelines for engineers in their implementation decisions. A number of benchmarks are described in the literature (e.g. [3,31]), which differ mainly in schema and sets of tests they offer, providing insights on the performance of various features of database systems.

The 007 database has been designed to test the performance of object database systems, in particular for analysing the performance of inter-object traversals, which are of interest in this paper. Moreover, it has been built based on reflections as to the shortcomings of other benchmarks, providing a wide range of tests over object database features. Examples of previous work on the performance analysis of query processing in which the 007 benchmark is used include [7,24,32].



Table I. Cardinalities of 007 extents and relationships.

Extent	Cardinality	Cardinality of relationships
<i>AtomicParts</i>	100 000	partOf: 1
<i>CompositeParts</i>	500	parts: 200, documentation 1
<i>Documents</i>	500	
<i>BaseAssemblies</i>	729	componentsPriv: 3

The 007 benchmark provides three different sizes for its database: small, medium and large. The differences in size are reflected in the cardinalities of extents and inter-object relationships. Table I shows the cardinalities of the extents and relationships used in the experiments for the medium 007 database, which is used here.

We have carried out our experiments using queries that were not in the original 007 suite, but which enable more systematic analysis of navigational queries with different levels of complexity than the queries included in the original 007 proposal.

To give an indication of the sizes of the persistent representations of the objects involved in 007, we give the following sizes of individual objects obtained by measuring the collections stored for the medium database: *AtomicPart*, 190 bytes; *CompositePart*, 2761 bytes; *BaseAssembly*, 190 bytes; *Document*, 24 776 bytes.

A companion paper on program-based (rather than query-based) access to object databases in Polar also presents results using the 007 benchmark [33].

4.2. Queries

The queries used in the experiments are described as follows. Aggregate and update operations are not used within the queries as the experiments aim to provide insights on the behaviour of the join algorithms with respect to object navigation.

(Q1) Retrieve the *id* of atomic parts and the composite parts in which they are contained, where the *id* of the atomic part is less than *v1* and the *id* of the composite part is less than *v2*. This query is implemented using a single join that follows the single-valued *partOf* relationship.

```
select struct(A:a.id, B:a.partOf.id)
from a in AtomicParts
where a.id <= v1
      and a.partOf.id <= v2;
```

(Q2) Retrieve the *id* and the *docId* of atomic parts, and the *id* of the documentations of the composite parts in which the atomic parts are contained, where the *docId* of the atomic part is different to the *id* of the documentation. This query is implemented using two joins, each of which follows a single-valued relationship.



```
select struct (A:a.id, B:a.docId,
              C:a.partOf.documentation.id)
from a in AtomicParts
where a.docId != a.partOf.documentation.id;
```

- (Q3) Retrieve the *id* of the composite parts and the atomic parts that are contained in the composite parts, where the *id* of the composite parts is less than *v1* and the *id* of the atomic parts is less than *v2*. This query is implemented using a single join that follows the multi-valued *parts* relationship.

```
select struct (A:c.id, B:a.id)
from c in CompositeParts,
     a in c.parts
where c.id <= v1
     and a.id <= v2;
```

- (Q4) Retrieve the *id* of the base assemblies and the atomic parts that are contained in the composite parts that compose the base assemblies, where the *buildDate* of the base assemblies is less than the *buildDate* of the atomic parts. This query is implemented using two joins, each of which follows a multi-valued relationship.

```
select struct (A:b.id, B:a.id)
from b in BaseAssemblies,
     c in b.componentsPriv,
     a in c.parts
where b.buildDate < a.buildDate;
```

The predicate in the where clauses in Q1 and Q3 is used to vary the selectivity of the queries over the objects of the input extents, which may affect the join operators in different ways. The selectivities are varied to retain 100%, 10%, 1% and 0.1% of the input extents.

Figures 3–6 show the parallel query execution plans for Q1–Q4, respectively. In each figure, two plans of different shapes are shown, plan (i) for the valued-based join (hash-join), and plan (ii) for the pointer-based joins (hash-loops, tc-hash-loops and materialise).

In the plans, multiple-valued relationships are resolved by unnesting the nested collection through the *unnest* operator. The key features of the plans can be explained with reference to Q1. The plan with the valued-based joins uses two *seq-scan* operators to scan the input extents. In turn, the plan with the pointer-based joins uses a single *seq-scan* operator to retrieve the objects of the collection to be navigated from. Objects of the collection to be navigated to are retrieved by the pointer-based joins.

The *exchange* operators are used to perform data repartitioning and to direct tuples to the appropriate nodes. For example, the *exchange* before the joins distributes the input tuples according to the reference defined in the relationship being followed by the pointer-based joins, or the join attribute for the valued-based joins. In other words it sends each tuple to the node where the referenced object lives. The *exchange* before the *print* operator distributes its input tuples using *round-robin*, but using a single destination node, where the results are built and presented to the user. The distribution policies for the two *exchanges* are *select-by-oid* and *round-robin*. Each *exchange* operator follows an *apply* operator which performs projections on the input tuples, causing the *exchange* to send smaller tuples through the network, thus saving communication costs.

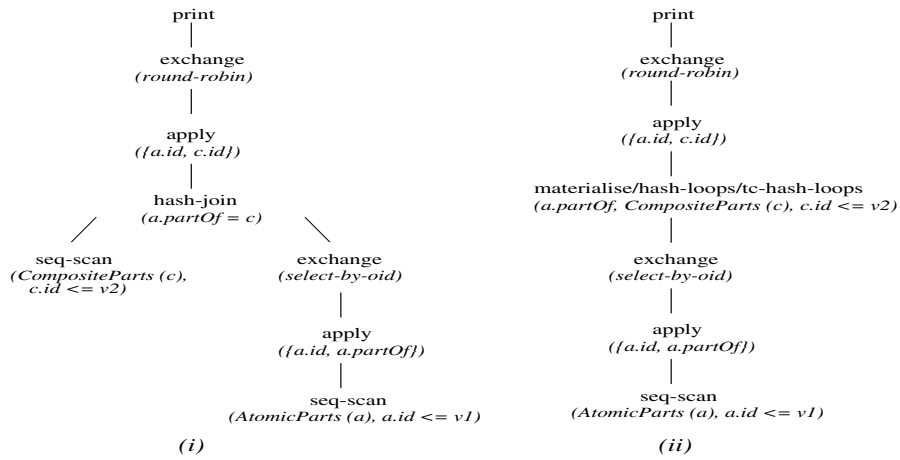


Figure 3. Parallel query execution plans for Q1.

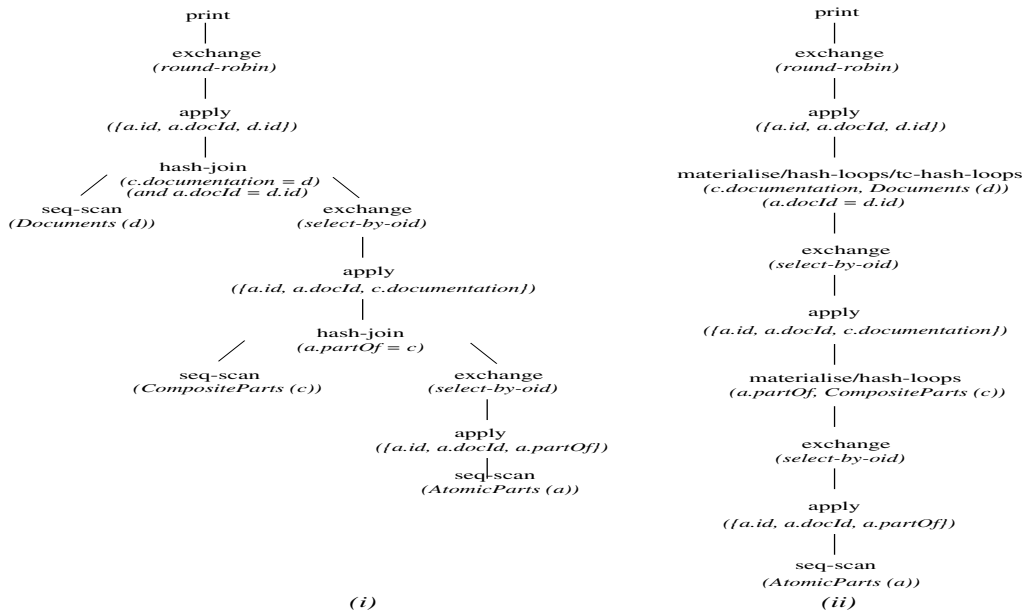


Figure 4. Parallel query execution plans for Q2.

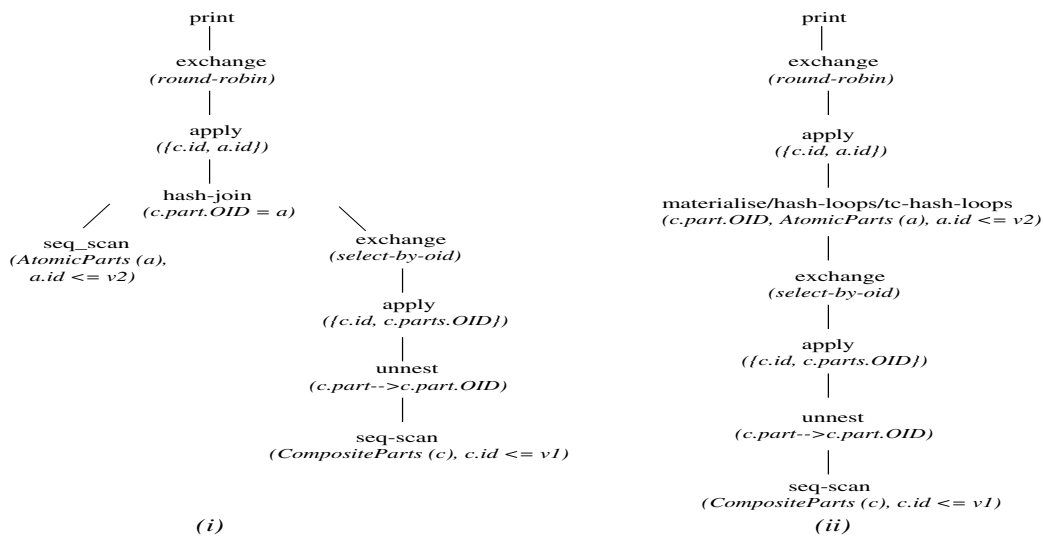


Figure 5. Parallel query execution plans for Q3.

4.3. Operator parameterization

Figures 3–6 show the parameters of most of the operators in the query plans for Q1–Q4. In the figures, the parameters for `seq-scan` specify the extent to be scanned and, in some cases, a selection predicate. The parameters for `unnest` specify the multiple-valued attribute or relationship to be unnested and, following symbol \rightarrow , the attribute to be added to the input tuples. The parameters for `apply` specify the list of attributes to be projected. The policy for distribution of tuples is specified as a parameter of `exchange`. For the pointer-based joins, the path expression, the target extent and the join predicate are specified in the figures. In the case of the value-based joins, only the join predicate is specified.

In the experiments, the `print` operator is set to count the number of tuples received, but not to print the results into a file. In this way, the amount of time that would be spent on the writing of data into a file is saved.

Some of the joins have tuning parameters that are not shown in the query plans, but that can have a significant impact on the way they perform (e.g. the hash table sizes for `hash-join` and `hash-loops`). In all cases, the values for these parameters were chosen so as to allow the algorithms to perform at their best. In `hash-join`, the hash table size is set differently for each join, to the value of the first prime number after the number of buckets to be stored in the hash table by the join. This means that there should be few collisions during hash table construction, but also that the hash table does not occupy excessive amounts of memory. In `hash-loops`, the hash table size is also set differently for each join, to the value of the first prime number after the number of pages occupied by the extent that is being navigated to. This means that there should be few collisions during hash table construction, but that

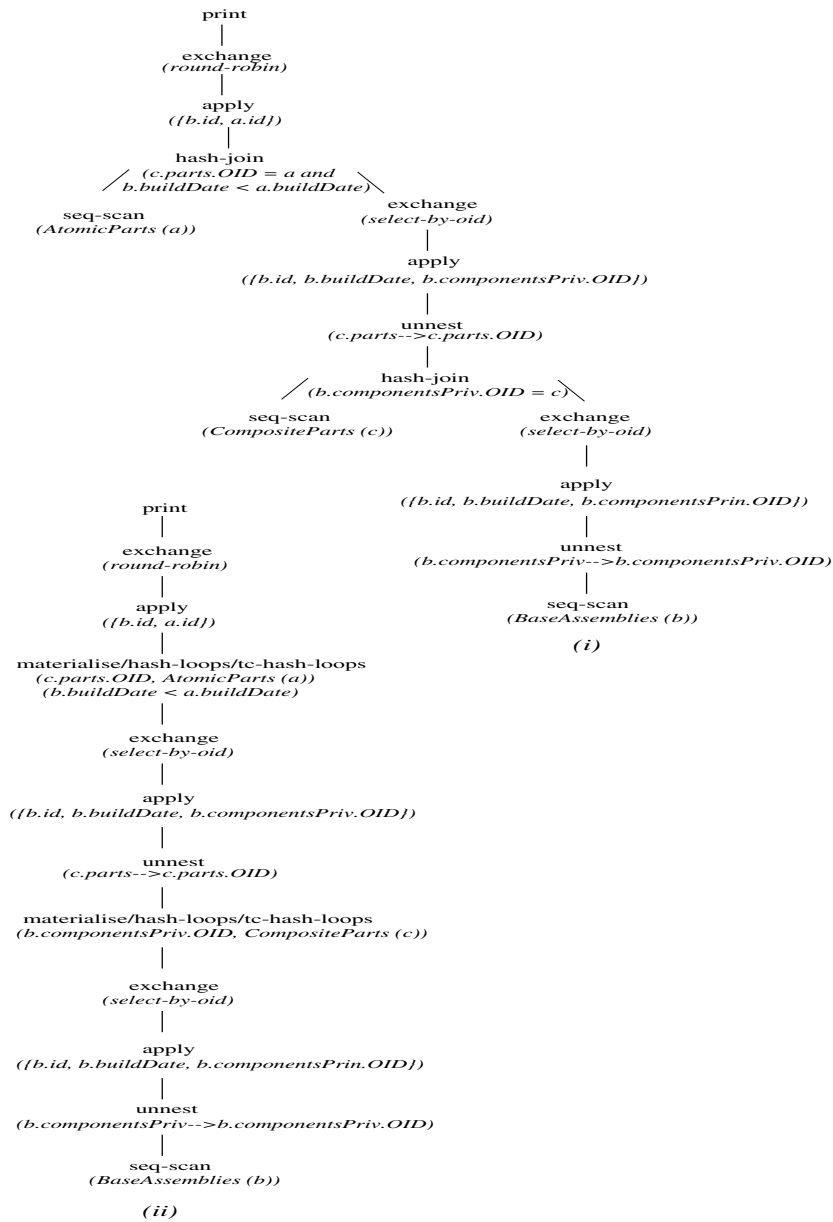


Figure 6. Parallel query execution plans for Q4.



the hash table does not occupy an excessive amount of memory. The other parameter for hash-loops is the window size, which is set to the size of the input collection, except where otherwise stated. This decision minimizes the number of page accesses carried out by hash-loops, at the expense of some additional hash table size. None of the experiments use indexes, although the use of explicit relationships with stored OIDs can be seen as analogous to indexes on join attributes in relational databases.

4.4. Experimental environment

The environment used in the experiments is a cluster of 233 MHz Pentium II PCs running RedHat Linux version 6.2, each with 64 MB main memory and a number of local disks, connected via a 100 Mbps Fast Ethernet hub. For each experiment, data is partitioned in 'round-robin' style over some number of disks, of which there is one MAXTOR MXT-540SL at each node. All timings are based on cold runs of the queries, with the server shut down and the operating system cache flushed between runs. In each case, the experiments were run three times and the average time obtained is reported.

4.5. Results and discussion

This section describes the different experiments that have been carried out using the experimental context described in Section 4.4. Each of queries Q1, Q2, Q3 and Q4 has been run on different numbers of stores, ranging from one to six, for each of the join operators. The graphs in Figures 7–13 show the obtained elapsed times (in seconds) against the variation in the number of stores, as well as speedup for each case. The speedup is obtained by dividing each elapsed time in the graph by the one-node elapsed time.

4.6. Following path expressions

Test queries Q1 and Q2 are examples of queries with path expressions, containing one and two single-valued relationships, respectively. Elapsed times and speedup results for these queries over the medium 007 database using 100% selectivity are given in Figures 7 and 8.

The graphs illustrate that all four algorithms show near linear speedup, but that hash-join and tc-hash-loops show similar performance and are significantly quicker throughout. The difference in response times between the four joins is explained with reference to Q1 as follows.

- (1) hash-join and tc-hash-loops. hash-join retrieves the instances of the two extents (*AtomicParts* and *CompositeParts*) by scanning. In contrast, tc-hash-loops scans the *AtomicParts* extent, and then retrieves the related instances of *CompositeParts* as a result of dereferencing the *partOf* attribute on each of the *AtomicParts*. This leads to essentially random reads from the extent of *CompositePart* (until such time as the entire extent is stored in the cache), and thus to potentially greater I/O costs for the pointer-based joins. However, based on the published seek times for the disks on the machine (an average of around 8.5 ms and a maximum of 20 ms), the additional time spent on seeks into the *CompositePart* extent should not be significantly more than 1 s on a single processor.

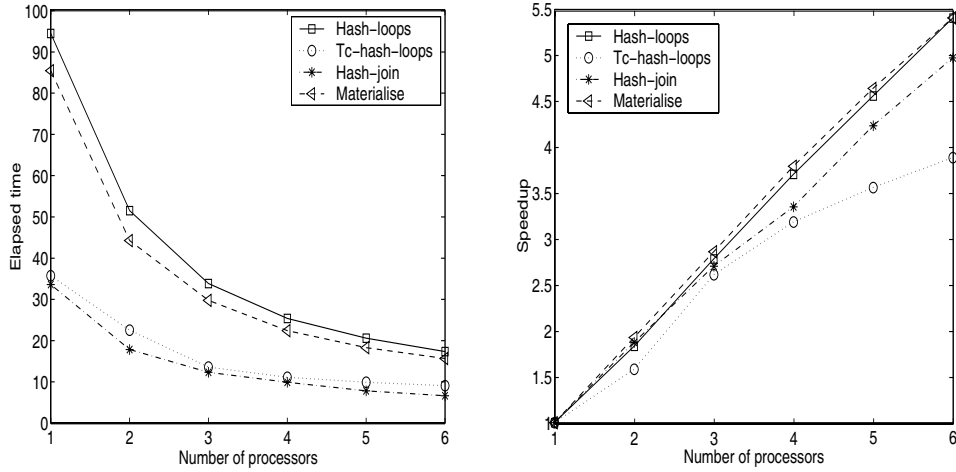


Figure 7. Elapsed time and speedup for Q1 on medium database.

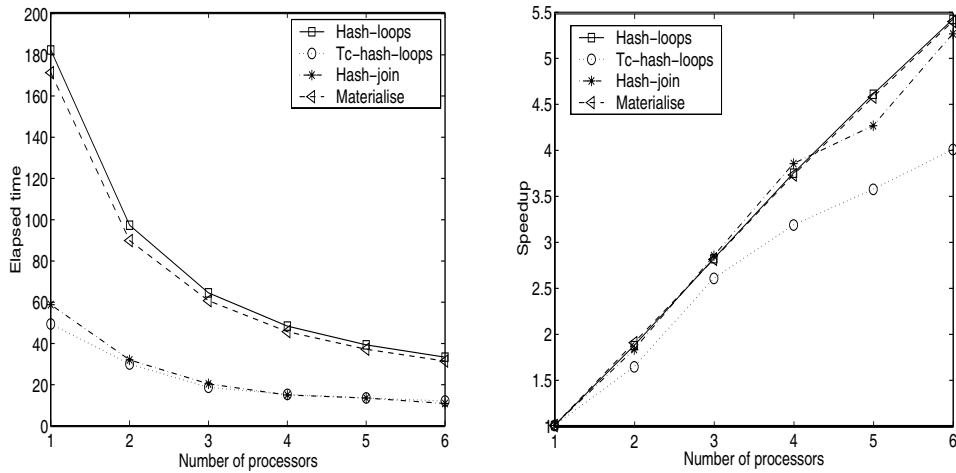


Figure 8. Elapsed time and speedup for Q2 on medium database.



- (2) **tc-hash-loops** and **materialise**. When an object has been read in from disk, it undergoes a mapping from its disk-based format into the nested tuple structure used for intermediate data by the evaluator. As each *CompositePart* is associated with many *AtomicParts*, the **materialise** join performs the *CompositePart* \rightarrow *tuple* mapping once for every *AtomicPart* (i.e. 100 000 times), whereas this mapping is carried out only once for each *CompositePart* (i.e. 500 times) for the **tc-hash-loops** join, as it keeps the previously generated *CompositePart* tuples in memory. The smaller number of *CompositePart* \rightarrow *tuple* mappings explains the significantly better performance of **tc-hash-loops** over **materialise** for Q1 and Q2 hash-join also performs only 500 *CompositePart* \rightarrow *tuple* mappings, as it scans extent *CompositeParts*.
- (3) **materialise** and **hash-loops**. Both perform the same number of *CompositePart* \rightarrow *tuple* mappings, i.e. one for every *AtomicPart*. It was anticipated that **hash-loops** would perform better than **materialise**, as a consequence of its better locality of reference, but this is not what Figures 7 and 8 show. The reason for the slightly better performance of **materialise** compared with **hash-loops** is the fact that, like **hash-loops**, **materialise** only reads each disk page occupied by the extents to be navigated to once for Q1 and Q2. In contrast to the **hash-loops** algorithm, which hashes the input tuples on the disk page of the referenced objects, **materialise** relies on the order in which disk pages are requested. In the case of Q1 and Q2, due to the order in which the input extents are loaded into and retrieved from disk, the accesses to disk pages performed by **materialise** are organized in a similar way to that brought about by the hash table of **hash-loops**. On the other hand, **hash-loops** has the additional overhead of hashing the input tuples. Additional experiments performed with **materialise** and **hash-loops**, randomizing the order in which the *AtomicPart* objects are loaded into Polar and thus accessed by Q1, have shown the benefit of the better locality of reference of **hash-loops** over **materialise**. Figure 9 shows the elapsed times obtained from these experiments for Q1, varying the number of stores.

4.7. Following multiple-valued relationships

Test queries Q3 and Q4 follow one and two multiple-valued relationships respectively. Response times for these queries over the medium 007 database using 100% selectivity are given in Figures 10 and 11, respectively.

These graphs present a less than straightforward picture. An interesting feature of the figures for both Q3 and Q4 is the superlinear speedup for **hash-join**, **hash-loops** and **tc-hash-loops**, especially in moving from one to two processors. These join algorithms have significant space overheads associated with their hash tables, which causes swapping during evaluation in the configurations with smaller numbers of nodes. Monitoring swapping on the different nodes shows that by the time the hash tables are split over three nodes they fit in memory, and thus the damaging effect of swapping on performance is removed for the larger configurations. The speedup graphs in Figures 10 and 11 are provided mainly for completeness, as they present distortions caused by the swapping activity on the one store configuration in the case of **hash-join**, **hash-loops** and **tc-hash-loops**.

Another noteworthy feature is the fact that, in Q3, **tc-hash-loops** presents similar performance to **hash-loops**, as there is no sharing of references to stored objects (*AtomicPart* objects) among the input tuples for both joins and, therefore, each *AtomicPart* object is mapped from store format into tuple format only once, offsetting the benefit of keeping the generated tuples in memory for **tc-hash-loops**.

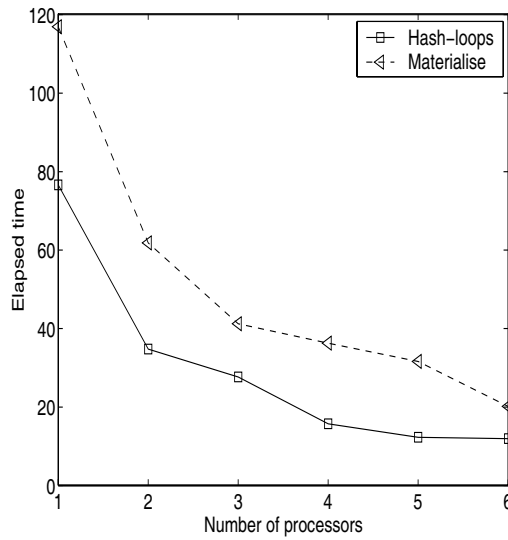


Figure 9. Elapsed time for Q1 on medium database, randomizing page requests performed by materialise and hash-loops.

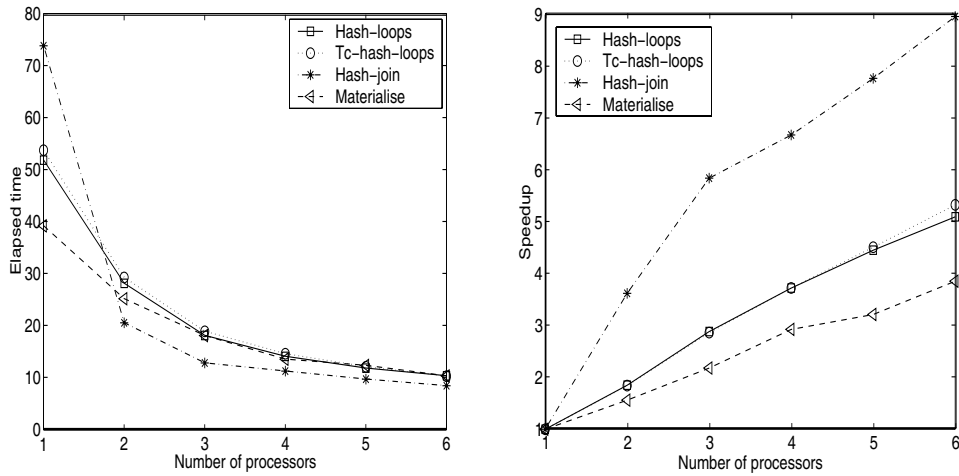


Figure 10. Elapsed time and speedup for Q3 on medium database.

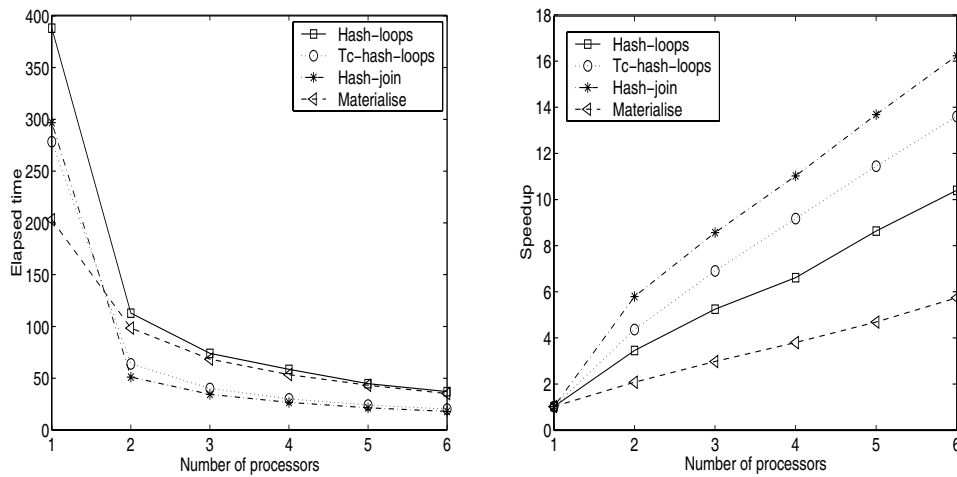


Figure 11. Elapsed time and speedup for Q4 on medium database.

Moreover, the relative performance of materialise and hash-loops compared with hash-join is better in Q3, than in Q1, Q2 and Q4, as in Q3 the total number of *CompositePart* \rightarrow *tuple* and *AtomicPart* \rightarrow *tuple* mappings is the same for all the join algorithms.

4.8. Varying selectivity

The selectivity experiments involve applying predicates to the inputs of Q1 and Q3, each of which carries out a single join. Response times for these queries over the medium 007 database running on six nodes, varying the values for $v1$ and $v2$ in the queries, are given in Figures 12 and 13, respectively. Note that what is being varied here is the selectivities of the scans of the collections being joined, not the *join selectivity* itself, which is ratio of the number of tuples returned by a join to the size of the Cartesian product of its inputs.

The experiments measure the effect of varying the selectivity of the scans on the inputs to the join as follows.

- (1) *Varying the selectivity of the outer collection ($v1$)*. The outer collection is used to probe the hash table in the hash-join, and is navigated from in the pointer-based joins. The effects of reducing selectivities are as follows.

- **hash-join**. The number of times the hash table is probed and the amount of network traffic caused by tuple exchange between nodes is reduced, although the number of objects read from disk and the size of the hash table remain the same. In Q1, the times reduce to a small extent, but not significantly, therefore it is the case that neither network delays nor hash table probing make substantial contributions to the time taken to evaluate the hash-join

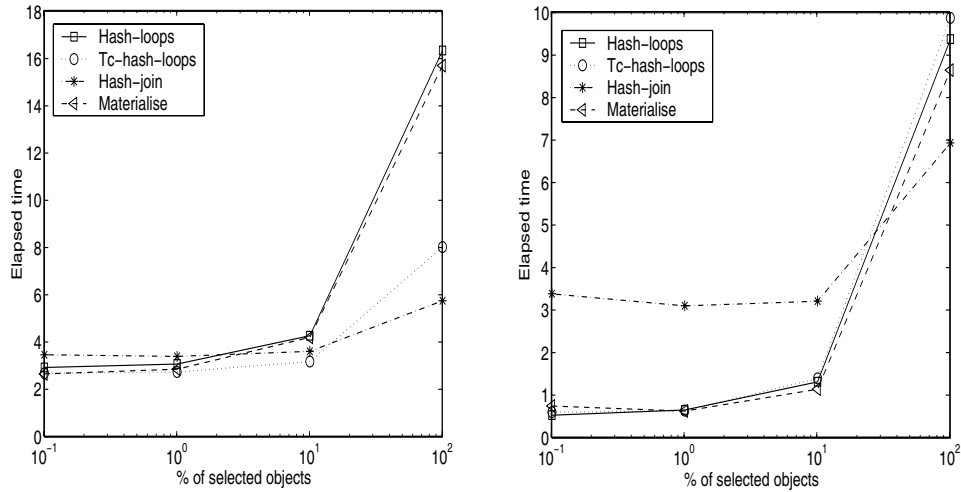


Figure 12. Elapsed times for Q1 (left) and Q3 (right), varying predicate selectivity using $v1$ on medium database.

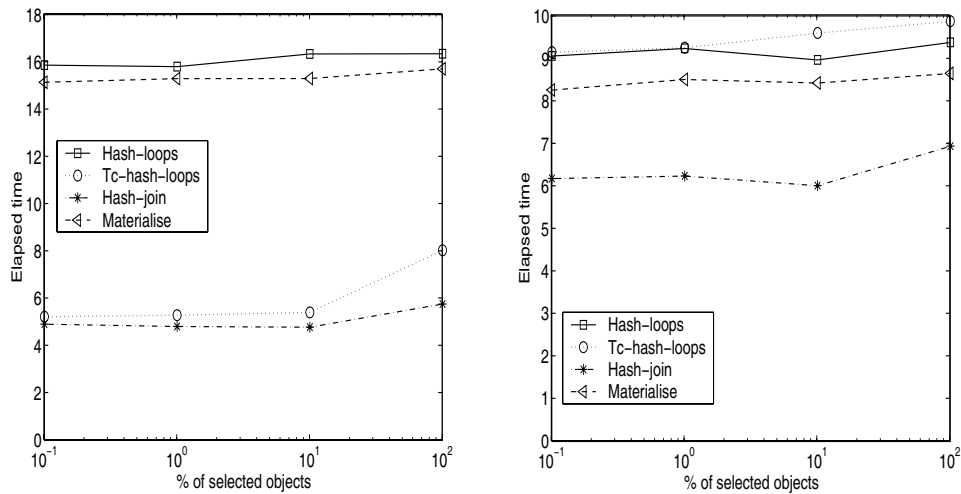


Figure 13. Elapsed times for Q1 (left) and Q3 (right), varying predicate selectivity using $v2$ on medium database.



version of Q1. As the reduction in network traffic and in hash table probes is similar for Q1 and Q3, it seems unlikely that these factors can explain the somewhat more substantial change in the performance of Q3. The only significant feature of Q3 that does not have a counterpart in Q1 is the unnesting of the *parts* attribute of *CompositeParts*. The *unnest* operator creates a large number of intermediate tuples in Q3 (100 000 in the case of 100% selectivity), so we postulate that much of the benefit observed from reduced selectivity in Q3 results from the smaller number of collections to be unnested.

- *Pointer-based joins*. The number of objects from which navigation takes place reduces with the selectivity, so reducing the selectivity of the outer collection significantly reduces the amount of work being done, e.g. fewer tuples to be hashed, fewer objects to be mapped from store format into tuple format, fewer disk pages to be read into memory, and fewer predicate evaluations. As a result, changing the selectivity of the scan on the outer collection has a significant impact on the response times for the pointer-based joins in the experiments. The impact is less significant for tc-hash-loops in Q1, as it performs much less work in the 100% selectivity case (e.g. *CompositePart* → *tuple* mapping) than the other pointer-based joins.
- (2) *Varying the selectivity of the inner collection (v2)*. The inner collection is used to populate the hash table in hash-join and to filter the results obtained after navigation in the pointer-based joins. The effects of reducing selectivities are as follows.
- *hash-join*. The number of entries inserted into the hash table reduces, as does the size of the hash table, although the number of objects read from disk and the number of times the hash table is probed remains the same. As shown in Figure 13 the overall change in response time is modest, both for Q1 and Q3.
 - *Pointer-based joins*. The amount of work done by the navigational joins is unaffected by the addition of the filter on the result of the join. As a result, changing the selectivity of the scan on the inner collection has a modest impact on the response times for the pointer-based joins in the experiments.

4.9. Summary

Some of the conclusions that can be drawn from the results obtained are as follows.

- The hash-join algorithm usually performs better than the pointer-based joins for 100% selectivity on the inputs, due to its sequential access to disk pages and the performance of the minimum possible number of object-tuple mappings (one per retrieved object).
- tc-hash-loops shows worst-case performance when there is no sharing of object references among the input tuples, making the use of a table of tuples unnecessary. In such cases, it performs closer to the other two pointer-based joins.
- materialise and hash-loops show similar performance in most of the experiments. However, hash-loops can perform significantly better for the cases where accesses to disk pages dictated by the input tuples are disorganized, so that hash-loops can take advantage of its better locality of reference.



- When applying higher predicate selectivities on $v1$ (outer collection), the pointer-based joins show a significant decrease in elapsed time, reflecting the decrease in number of objects retrieved from the inner collection and pages read from disk. On the other hand, hash-join does not show a significant decrease in elapsed time for the same case, reflecting the fact that no matter which selectivity is used on the outer collection, the inner collection is fully scanned.
- When varying the predicate selectivity on $v2$ (inner collection) the pointer-based joins are unaffected, as the amount of work performed does not decrease with the increase in selectivity. hash-join shows a small decrease in elapsed time, reflecting the reduction in number of entries inserted into the hash table.

5. COST MODEL

Among the fundamental techniques for performance analysis, measurements of existing systems (or empirical analysis) provides the most believable results, as it generally does not make use of simplifying assumptions. However, there are problems with experimental approaches. For example, experiments can be time-consuming to conduct and difficult to interpret, and they require that the system being evaluated already exists and is available for experimentation. This means that certain tasks commonly make use of models of system behaviour, for example for application sizing (e.g. [34]) or for query optimization. Models are partly used here to help explain the results produced from system measurements.

The cost of executing each operator depends on several system parameters and variables, which are described in Tables II and III, respectively. The values for the system parameters have been obtained through experiments and, in Table II, they are presented in seconds unless otherwise state.

The types of parallelism implemented within the algebra are captured as follows.

- *Partitioned parallelism.* The cost model accounts for partitioned parallelism by estimating the costs of the instances of a query subplan running on different nodes of the parallel machine separately, and taking the cost of the most costly instance as the elapsed time of the particular subplan. Hence $C_{subplan} = \max_{1 \leq i \leq N}(C_{subplan_i})$, where N is the number of nodes running the same subplan.
- *Pipelined parallelism.* In Polar, *intra-node* pipelined parallelism is supported by a multi-threaded implementation of the iterator model. Currently, multi-threading is supported within the implementation of `exchange`, which is able to spawn new threads. In other words, multi-threaded pipelining happens between operators running in distinct subplans linked by an `exchange`. *Inter-node* pipelined parallelism is implemented within the `exchange` operator. The granularity of the parallelism in this case is a buffer containing a number of tuples, and not a single tuple, as is the case for *intra-node* parallelism.

The cost model assumes that the sum of the costs of the operators of a subplan, running on a particular node, represents the cost of the subplan. We note that, due to the limitations of pipelined parallel execution in Polar, the simplification has not led to widespread difficulties in validating the models. Hence $C_{subplan_i} = \sum_{1 \leq j \leq K}(C_{operator_j})$, where K is the number of operators in the subplan.

In contrast with many other cost models, I/O, CPU and communication costs are taken into account in the estimation of the cost of an operator. Hence: $C_{operator_j} = C_{io} + C_{cpu} + C_{comm}$.



Table II. System parameters.

Name	Description	Default value (s)
C_{seek}	Average read seek time of disks.	0.0085
C_{latency}	Average latency time of disks.	0.0048
C_{read}	Average time to read a page.	0.0024
C_{eval}	Average time to evaluate a one-condition predicate.	7.0000×10^{-6}
C_{copy}	Average time to copy one tuple into another. The copy operation described within this variable only regards shallow copies of objects, i.e. only pointers to objects get copied, not the objects themselves.	3.2850×10^{-6}
C_{conv}	Average time to convert an OID into a page number.	3.8000×10^{-6}
C_{look}	Average time to look up a tuple in a table of tuples and retrieve it from the table.	3.6000×10^{-7}
C_{pack}	Average time to pack an object of type t into a buffer.	(depends on t)
C_{unpack}	Average time to unpack an object of type t from a buffer.	(depends on t)
C_{map}	Average time to map an attribute of type t from store format into tuple format.	(depends on t)
$C_{\text{hashOnNumber}}$	Average time to apply a hash function on the page number or OID number of an object and obtain the result.	2.7000×10^{-7}
C_{newTuple}	Average time to allocate memory space for an empty tuple.	1.5390×10^{-6}
C_{add}	Average time to add an attribute into a tuple.	2.9350×10^{-6}
C_{insert}	Average time to insert a pointer to an object into an array.	4.7200×10^{-7}
$\text{Net}_{\text{overhead}}$	Space overhead in bytes imposed by Ethernet related to protocol trailer and header, per packet transmitted.	18
Net_{band}	Network bandwidth in Mbps.	100

- *Independent parallelism.* This is obtained when two sub-plans, neither of which uses data produced by the other, run simultaneously on distinct processors, or on the same processor using different threads. In the first case, the cost of the two sub-plans is estimated by considering the cost of the most costly sub-plan. In the second case, the cost of the sub-plans is estimated as if they were executed sequentially.

The cost formula for each operator is presented in the following sections. A brief description of each algorithm is provided, to make the principal behaviours considered in the cost model explicit.

5.1. sequential-scan

Algorithm

```

1 for each disk page of the extent
  1.1 read page
  1.2 for each object in the page
    1.2.1 map object into tuple format
    1.2.2 apply predicate over tuple

```



Table III. System variables.

Name	Description
$I_{num_{left}}$	Cardinality of the left input.
$I_{num_{right}}$	Cardinality of the right input.
P_{len}	Length of the predicate.
O_{type}	Type of the object.
O_{num}	Number of objects.
$extent$	Extent of the database (e.g. AtomicParts).
$Page_{num}$	Number of pages.
$Bucket_{size}$	Size of a bucket (number of elements).
R_{card}	Cardinality of a relationship times a factor based on the selectivity of the predicate.
$O_{ref_{num}}$	Number of referenced objects.
W_{num}	Number of windows of input.
W_{size}	Size of a window of input (number of elements).
H_{size}	Size of the hash table (number of buckets).
Col_{card}	Cardinality of a collection.
$Proj_{num}$	Number of attributes to be projected.
T_{size}	Size of a tuple (in bytes).
T_{num}	Number of tuples.
$Pack_{num}$	Number of packets to be transmitted through the network.

CPU:

- (i) map objects from store format into tuple format (line 1 . 2 . 1);
- (ii) evaluate predicate over tuples (line 1 . 2 . 2).

I/O:

- (i) read disk pages into memory (line 1 . 1).

Hence

$$C_{cpu} = mapObjectTuple(O_{type}, O_{num}) + evalPred(P_{len}, O_{num}) \quad (1)$$

CPU (i). The cost of mapping an object from store format into tuple format depends on the type of the object being mapped, i.e. on its number of attributes and relationships, on the type of each of its attributes (e.g. string, bool int, OID, etc.), and on the cardinality of its multiple-valued attributes and relationships, if any. Hence

$$mapObjectTuple(typeOfObject, numOfObjects) = mapTime(typeOfObject) * numOfObjects \quad (2)$$

$$mapTime(typeOfObject) = \sum_{typeOfAttr \in \{int, \dots\}} C_{map_{typeOfAttr}} * typeOfObject.numOfAttr(typeOfAttr) \quad (3)$$

Experiments with the mapping of attribute values, such as longs, strings and references, have been obtained from experiments and used as values for $C_{map_{typeOfAttr}}$. Some of these values are shown in Table IV.



Table IV. Values for cost of mapping attributes.

Type	Default value (s)
Long	3.0930×10^{-6}
Reference	2.9480×10^{-6}
String (11 characters)	5.3651×10^{-6}

Table V. I/O interference costs.

Interfering operator(s)	Increase in I/O cost (%)
One other seq-scan	186.5
Two other seq-scans	226.3
A materialise	71.5

CPU (ii). The cost of evaluating a predicate depends on the number of conditions to be evaluated (length of the predicate), the average time to evaluate a condition, and the number of conjunctions and disjunctions. We only present the formula for conjunctions of conditions, which is the type of predicate relevant to the queries used in the experiments. We assume that at least one condition of the predicate is evaluated, and at most half of the conditions (if more than one) are evaluated before the predicate evaluation stops caused by a condition having evaluated to false. Hence, $1 + ((P_{len} - 1)/2)$ of the conditions are evaluated for any predicate.

$$evalPred(lengthOfPred, numTuples) = ((1 + ((lengthOfPred - 1)/2)) * C_{eval}) * numTuples \quad (4)$$

I/O (i). The I/O cost of seq-scan is associated with the sequential reading of all the pages occupied by a particular extent. The cost of reading each page sequentially, in turn, depends on the values for disk seek, latency and read times. As pages are read sequentially, seek and latency times are taken into account only once for all pages. Hence

$$C_{io} = readPagesSeq(extent.Page_{num}) \quad (5)$$

$$readPagesSeq(numOfPages) = C_{seek} + C_{latency} + (C_{read} * numOfPages) \quad (6)$$

When two or more operators compete for the same disk, overheads associated with the concurrent accesses to the disk are added to the I/O cost formulae. These overheads are represented as multipliers that were identified through experimental assessment, and are added to the values obtained from the I/O formulae. The experiments carried out to identify the effect of operator competition for the same disk take into account the number of operators accessing the disk and the type of access performed, e.g. random or sequential.

For the experiments, only seq-scan and materialise suffer from concurrent disk access in the experimental queries. hash-loops and tc-hash-loops, which also access stored data, are not affected in



the queries, because they consume all the tuples of their left input before retrieving data from the right input. In other words, taking Q1 as an example, by the time hash-loops or tc-hash-loops access the disk, seq-scan has finished scanning its input extent.

Some of the multipliers that have been identified to estimate the I/O cost of seq-scan considering that other operators compete for the same disk are described in Table V.

5.2. Unnest

Algorithm

```
1 for each input tuple
  1.1 for each element in the collection attribute
    1.1.1 replicate the tuple
    1.1.2 add attribute into the replica of the tuple
    1.1.3 evaluate predicate over the result
```

CPU:

- (i) replicate tuples (line 1.1.1);
- (ii) add an attribute into the tuples (line 1.1.2);
- (iii) evaluate a predicate over tuples (line 1.1.3).

Hence

$$C_{cpu} = ((C_{newTuple} + C_{copy} + C_{addAttr}) * Col_{card}) * I_{num_{left}} + evalPred(P_{len}, Col_{card} * I_{num_{left}}) \quad (7)$$

CPU (i) Replicating a tuple involves creating an empty tuple and copying the contents of the original tuple into the empty tuple. These two operations are performed as many times as the number of elements in the nested collection (multiple-valued attribute or relationship) times the number of input tuples to unnest.

CPU (ii) Adding an attribute into a tuple is performed as many times as the number of elements in the nested collection (multiple-valued attribute or relationship) times the number of input tuples to unnest.

CPU (iii) The cost of evaluating a predicate is described in Section 5.1.

5.3. apply

Algorithm

```
1 for each input tuple
  1.1 replicate tuple
  1.2 for each attribute in the list
    1.2.2 add attribute into the replica of the tuple
```

CPU:

- (i) replicate tuples (line 1.1);
- (ii) add each of the attributes in the list of attributes into the replicas of the tuples (line 1.2.2).



Hence

$$C_{cpu} = (C_{newTuple} + C_{addAttr} * Proj_{num}) * I_{num_{left}} \quad (8)$$

CPU (i) The cost of replicating a tuple is described in Section 5.2. This operation is performed as many times as the number of input tuples.

CPU (ii) The cost of adding an attribute into a tuple is described in Section 5.2. This operation is performed as many times as the number of input tuples times the number of attributes to be projected.

5.4. exchange

Algorithm

- 1 for each input tuple
 - 1.1 pack tuple into buffer
 - 1.2 if buffer is full
 - 1.2.1 send buffer
- 2 for each arriving buffer
 - 2.1 for each tuple in the buffer
 - 2.1.1 unpack tuple

CPU:

- (i) unpack tuples (line 2.1.1);
- (ii) pack tuples (line 1.1).

Communication:

- (i) receive buffers (line 2);
- (ii) send buffers (line 1.2.1).

Hence

$$C_{cpu} = (C_{pack_{O_{type}}} * T_{num_{pack}}) + (C_{unpack_{O_{type}}} * T_{num_{unpack}}) \quad (9)$$

$$C_{comm} = (dataSize(T_{size}, T_{num}) + netOverhead(Pack_{num}) * 8) / (10^6 * Net_{band}) \quad (10)$$

The cost of packing and unpacking tuples depends on the number of tuples to be packed ($T_{num_{pack}}$) and unpacked ($T_{num_{unpack}}$) by **exchange**, and the cost of packing ($C_{pack_{O_{type}}}$) and unpacking ($C_{unpack_{O_{type}}}$) a single tuple of a particular type (O_{type}). The number of packets depends on the data redistribution and the number of consumer and producer operators that **exchange** communicates with. Table VI shows values for $C_{unpack_{O_{type}}}$ and $C_{pack_{O_{type}}}$, which have been obtained through experiments.

The communication cost of **exchange** relates to the transmission of packets of data through the network. This cost is calculated by dividing the total amount of data to be transmitted in bits (amount of data in bytes times eight) by the network bandwidth in bps ($10^6 * Net_{band}$).

The total amount of data to be transmitted depends on the amount of data related to the tuples contained in the transmission packets and the space overhead imposed by the network per packet, e.g. protocol headers and trailers. The former depends on the average number of bytes contained in the



Table VI. Values for cost of packing and unpacking tuples.

Variable	Type of object	Default value (s)
$C_{\text{pack}_{\text{Otype}}}$	Long	1.9780×10^{-6}
$C_{\text{unpack}_{\text{Otype}}}$	Long	2.2930×10^{-6}
$C_{\text{pack}_{\text{Otype}}}$	Reference	3.4680×10^{-6}
$C_{\text{unpack}_{\text{Otype}}}$	Reference	3.2320×10^{-6}

input tuples, and the number of tuples. Hence

$$\text{dataSize}(\text{sizeOfTuple}, \text{numOfTuples}) = \text{sizeOfTuple} * \text{numOfTuples} \quad (11)$$

The latter depends on the number of packets to be transmitted and the overhead associated with each packet. Hence

$$\text{netOverhead}(\text{numOfPackets}) = \text{Net}_{\text{overhead}} * \text{numOfPackets} \quad (12)$$

The number of packets transmitted by **exchange** depends on the number of consumers to which **exchange** sends packets, and the maximum packet size. Each buffer of tuples to be sent to a particular destination may be divided into a number of packets, if its size exceeds the maximum packet size (1500 bytes). The maximum packet size is determined by the maximum transmission unit (MTU) of the network, which in the case of Polar is an Ethernet.

5.5. hash-join

Algorithm

- 1 for each tuple in the left input
 - 1.1 hash tuple on the join attribute(s)
 - 1.2 insert tuple into the hash table
- 2 for each tuple in the right input
 - 2.1 hash tuple on the join attribute(s)
 - 2.2 for each tuple in the corresponding hash table bucket
 - 2.2.1 concatenate tuple in the bucket with right input tuple
 - 2.2.2 apply predicate over resulting tuple

CPU:

- (i) hash tuples on the join attribute(s) (lines 1 . 1 and 2 . 1);
- (ii) insert tuples into the hash table (line 1 . 2);
- (iii) concatenate bucket tuples with right input tuples (lines 2 . 2 . 1);
- (iv) evaluate predicate over tuples (line 2 . 2 . 2).



Hence

$$C_{cpu} = hashTuples(I_{num_{left}} + I_{num_{right}}) + insertTuples(I_{num_{right}}) \\ + concatTuples(I_{num_{right}} * Bucket_{size}) + evalPred(P_{len}, (I_{num_{right}} * Bucket_{size})) \quad (13)$$

We assume that both left and right input tuples are hashed on an OID value. Since all right input tuples get concatenated with each tuple in one of the buckets of the hash table, the number of tuple concatenations and predicate evaluations is approximately the number of right input tuples times the average number of tuples in each bucket in the hash table.

CPU (i) When two tuples are joined on an explicit relationship, the join attribute is OID-valued[‡]. The cost of hashing thus depends on the cost of hashing one tuple on the appropriate OID value, and the number of tuples to be hashed. Hence

$$hashTuples(numOfTuples) = C_{hashOnNumber} * numOfTuples \quad (14)$$

CPU (ii) The cost of inserting tuples into a hash table is estimated by considering the number of tuples that are inserted and the cost of inserting a single tuple. The cost of inserting a tuple into one of the buckets of a hash table is the same as the cost of inserting a pointer to an element in memory into an array. Hence

$$insertTuples(numOfTuples) = C_{insert} * numOfTuples \quad (15)$$

CPU (iii) Concatenation of two tuples is done by copying the two tuples into a third tuple. This does not involve copying each object in the tuples, although, as tuples contain only pointers to objects in memory that are shared between the original tuples and their copies. The model accounts both for the allocation time for the new tuple and for the cost of copying pointers from one tuple to another

$$concatTuples(numOfPairs) = (2 * C_{copy} + C_{newTuple}) * numOfPairs \quad (16)$$

CPU (iv) The cost of evaluating a predicate is described under sequential-scan.

5.6. hash-loops

Algorithm

```

1 for each window of the left input
  1.1 for each tuple in the window
    1.1.1 convert reference to object into a page number
    1.1.2 hash tuple on the page number
    1.1.3 insert tuple into the hash table
  1.2 for each entry in the hash table
    1.2.1 read page
    1.2.2 for each tuple in the bucket
      1.2.2.1 map referenced object into tuple format
      1.2.2.2 concatenate tuples
      1.2.2.3 apply predicate over the result

```

[‡]We only consider OID-valued join attributes.

**CPU:**

- (i) convert OIDs into physical page numbers (line 1 . 1 . 1);
- (ii) hash tuples (line 1 . 1 . 2);
- (iii) insert tuples into the hash table (line 1 . 1 . 3);
- (iv) map objects from store format into tuple format (line 1 . 2 . 2 . 1);
- (v) evaluate predicate over tuples (line 1 . 2 . 2 . 3);
- (vi) concatenate pairs of tuples (line 1 . 2 . 2 . 2).

I/O:

- (i) read disk pages into memory (line 1 . 2 . 1).

Hence

$$\begin{aligned} C_{cpu} = & \text{convertOIDpage}(I_{num_{left}}) + \text{hashTuples}(I_{num_{left}}) + \text{insertTuples}(I_{num_{left}}) \\ & + \text{mapObjectTuple}(O_{type}, R_{card} * O_{ref_{num}}) + \text{concatTuples}(I_{num_{left}}) \\ & + \text{evalPred}(P_{len}, I_{num_{left}}) \end{aligned} \quad (17)$$

$$C_{io} = \text{readPagesRand}(\text{numReadPages}) \quad (18)$$

$$\text{readPagesRand}(\text{numReadPages}) = (C_{seek} + C_{latency} + C_{read}) * \text{numReadPages} \quad (19)$$

CPU (i) As Polar's OIDs are logical, they need to be mapped into the object's physical address. The cost of such mappings depends on the number of OIDs to be mapped (i.e. one for each input tuple) and the cost of converting a single OID. Hence

$$\text{convertOIDpage}(\text{numOfTuples}) = C_{conv} * \text{numOfTuples} \quad (20)$$

CPU (ii) The cost of hashing a tuple on an OID value is described in Section 5.5.

CPU (iii) The cost of inserting tuples into a hash table is described in Section 5.5.

CPU (iv) For hash-loops, the number of times a stored object is mapped into tuple format depends on the number of times the object is referenced by the input tuples. This corresponds, approximately, to the cardinality of the inverse relationship to the one being followed by hash-loops times a factor based on the selectivity of the predicate, R_{card} .

CPU (v) The cost of concatenating two tuples is described in Section 5.5.

CPU (vi) The cost of evaluating a predicate is described in Section 5.1.

I/O (i) To estimate the number of page reads performed per hash table, we use Yao's formula [35]. Yao calculates the expected fraction of pages for a collection with cardinality $collectionSize$ that must be read to retrieve a subset of this collection with cardinality $numRefObjects$, provided that $numObjPerPage$ objects fit on a page. The number of objects that are referenced within the input tuples, $numRefObjects$, can be estimated by a formula proposed in [6], which is described below:

$$\text{numReferencedObjects}(\text{windowSize}, \text{collectionSize}) = \min(\text{windowSize}, \text{collectionSize}) \quad (21)$$



Yao's formula is

$$Yao(collectionSize, numObjPerPage, numRefObjects) = 1 - \left(\prod_{i=1}^{numObjPerPage} (collectionSize - numRefObjects - i + 1) / (collectionSize - i + 1) \right) \quad (22)$$

$numObjPerPage$ can be obtained from the average size of the objects of the collection, and the size of a disk page:

$$numObjPerPage = pageSize/objectSize \quad (23)$$

The number of pages retrieved per window of input is estimated as the fraction of pages containing objects referenced by the input tuples, times the number of pages occupied by the whole collection.

$$numReadPages = Yao(collectionSize, numObjPerPage, numRefObjects) * numPages \quad (24)$$

5.7. tc-hash-loops

The CPU and I/O costs for tc-hash-loops are very similar to the costs described for hash-loops. Apart from the number of mappings of objects from store format into tuple format, and the extra cost of accessing the table of tuples, which are both CPU operations, all the other formulae described for hash-loops apply in the case of tc-hash-loops.

We assume that tuples generated from referenced objects are kept in the table of tuples only during the processing of the window of input tuples that caused them to be generated. In other words, each time a hash table is emptied and filled again with tuples, the table of tuples is also emptied and refilled during the processing of the next hash table. Therefore, supposing that the same object can be repeatedly referenced within all the windows of input tuples, in the worse case, an object is mapped as many times as the number of windows. Different from hash-loops, in which an object is mapped for each input tuple, for tc-hash-loops, each referenced object is mapped at most $nWindows$ times, if $W_{num} \leq R_{card}$. Hence, for tc-hash-loops, the function for mapping objects into tuples is parameterized as follows:

$$mapObjectTuple(O_{type}, W_{num} * O_{ref,num})$$

The cost of accessing the table of tuples depends on the average cost of one access, and the number of times the table is accessed, considering all the windows of input. This number is estimated as $W_{size} - H_{size}$, which represents the number of elements in the hash table (size of the window of inputs) minus the number of buckets in the hash table. This value is estimated based on the fact that the table of tuples is accessed each time a tuple in the hash table is processed, except when the tuple is the first in its bucket. In this case, there is no need to look for a tuple in the table of tuples, as no tuples have been generated from stored objects for the particular bucket. Hence

$$accessTableOfTuples(numOfTuples) = C_{look} * numOfTuples \quad (25)$$



5.8. materialise

Algorithm

- 1 for each input tuple
 - 1.1 read page where the referenced object lives
 - 1.2 map referenced object into tuple format
 - 1.3 concatenate input tuple with generated tuple
 - 1.4 apply predicate over the result

CPU:

- (i) map objects from store format into tuple format (line 1.2);
- (ii) evaluate predicate over tuples (line 1.4);
- (iii) concatenate pairs of tuples (line 1.3).

I/O:

- (i) read disk pages into memory (line 1.1).

Hence

$$C_{cpu} = mapObjectTuple(O_{type}, I_{num_{left}}) + concatTuples(I_{num_{left}}) + evalPred(P_{len}, I_{num_{left}}) \quad (26)$$

$$C_{io} = readPagesRand(Page_{num}) \quad (27)$$

The CPU cost for *materialise* is similar to the one for hash-loops, excluding operations that are particular to the hash-loops algorithm, such as hashing of tuples. Therefore, it is not described further.

I/O (i). Unlike hash-loops, *materialise* does not perform organized accesses to disk pages and suffers from poor locality of reference. In this case, multiple requests to the same disk page may cause the page to be read into memory multiple times. Thus, the I/O model for *materialise* extends the I/O model for hash-loops, so that repeated requests to the same page are considered.

In the hash-loops I/O model, Yao's formula is used to estimate the number of pages that are read within a window of input. For *materialise*, Yao estimates the minimum number of page reads, i.e. assuming that each requested page is read only once. In order to consider repeated reads of the same page, the following variables are described.

- P represents the number of pages that are requested (result of Yao).
- R represents the average number of times that a page is requested.
- D represents the *maximum* distance between each request to the same page in terms of requests to other pages. For example, if page p is now requested, the next time p will be requested again is after $D - 1$ pages other than p are requested. The value of D can be obtained from taking representative samples of the involved extent and estimating the value of D , which can be stored in the system's metadata.
- S represents the size of the cache *in pages*.

Based on the variables itemized above, the following dependencies can be defined.



Table VII. I/O interference costs.

Interfering operator(s)	Increase in I/O cost (%)
A seq-scan	33.1
Another materialise and a seq-scan	195.1

- (1) If $R = 1$, for any value of S , each requested page is read only once.
- (2) If $S \geq P$, each page is read only once, as they all fit together in the cache.
- (3) If $S \geq D$, each page is read only once, as, by the time a page is requested for the 2nd, . . . , or R th time, the page will be in the cache.
- (4) If $S < D$, multiple requests to the same page may incur multiple page reads to the same page.

For the case defined in dependency 4, we assume the worst scenario due to the unpredictability of page requests. We assume that for all pages in P , repeated requests to the same page are separated by a distance of D . Except when it is not possible to separate R requests to each of the pages in P pages by D , then a distance $D' < D$. D' represents the maximum distance, after D , that can be used to separate the pages in P which cannot be separated by a distance of D .

Note that by fixing the values for D and D' , we restrict the cost model to the worst-case situation for the values chosen. The advantage of this approach is the possibility of checking how the variation of D and D' affect the performance of *materialise*. An alternative approach is to use *probability*. For example, for the cases in which D is not defined, it is possible to estimate the probability that a requested page is not in the cache. As defined in [36], once the cache is filled with S pages, the probability that a requested page is not in the cache is $1 - (S/P)$. This probability, multiplied by the number of page requests, results in an approximation to the number of pages read.

As for *seq-scan*, multipliers that are used to estimate the increase in I/O cost caused by concurrent accesses to the same disk for *materialise* were identified through experimental assessment. Some of them are shown in Table VII.

5.9. Comparison with system results

This section presents the validation of the cost model against the system results presented in the previous section. The validation covers the experiments performed over the medium 007 database including the tests with single- and multiple-valued relationships (Q1, Q2 and Q3) and variation in selectivity of the predicates applied over the input extents (Q1 and Q3). Q4 is left out of the validation due to the swapping activity that is predominant for most of the join operators.

5.9.1. Elapsed time for queries

Figures 14–16 show experimental results using the Polar system and the corresponding cost model estimates for Q1, Q2 and Q3, respectively, using a predicate selectivity of 100%, varying the number of processors, and running the join operators hash-loops, tc-hash-loops, hash-join and *materialise*.

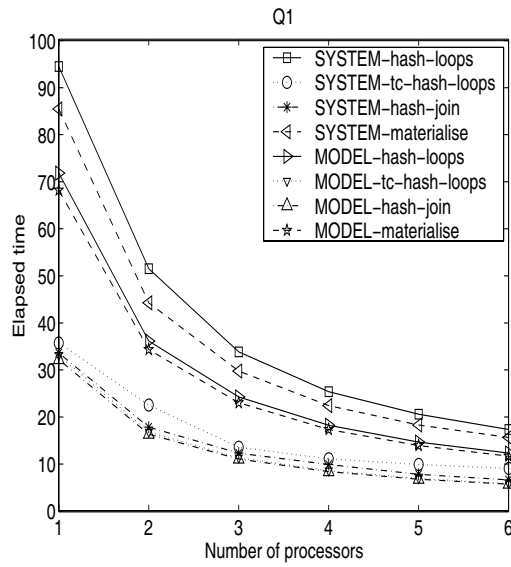


Figure 14. Elapsed time for Q1.

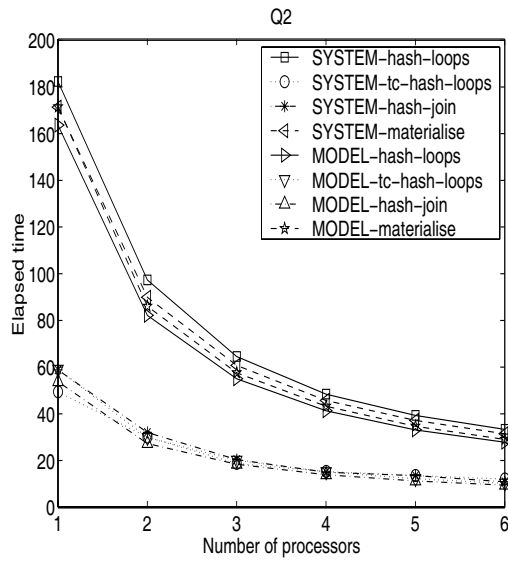


Figure 15. Elapsed time for Q2.

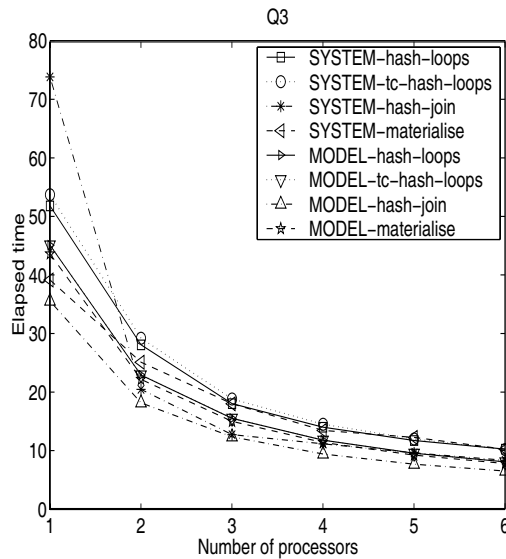


Figure 16. Elapsed time for Q3.

The results in Figure 14 show that hash-loops is the most costly operator, while hash-join is the fastest for all numbers of processors. tc-hash-loops stands between the two, but closer to hash-join than to hash-loops. materialise stands between hash-loops and tc-hash-loops, being closer to hash-loops than to tc-hash-loops.

As described in the experimental evaluation section, hash-loops is slower than tc-hash-loops because hash-loops performs 100 000 object-tuple mappings when retrieving *CompositePart* objects (200 mappings for each object), whereas tc-hash-loops only performs 500 (mapping each *CompositePart* once). The numbers of object-tuple mappings coincide for hash-join and tc-hash-loops. However, with hash-join, disk pages containing *CompositeParts* are retrieved sequentially, while in tc-hash-loops the pages are read randomly. The cost model captures the behaviours of the different joins reasonably well, although it is more precise for hash-join, as it performs fewer object-tuple mappings for *CompositePart* objects than most of the other joins and accesses disk pages sequentially.

The model slightly underestimates the mapping cost of *CompositePart* objects compared with the measured times obtained from experimentation with 007 object mappings—developing a single model for object-tuple mappings (as described in Section 5.1) that works for all shapes and sizes of objects has proved quite challenging. The model is generally more precise when estimating the cost of sequential accesses to disk pages, than random accesses, as the former tend to be more predictable.



The results in Figure 15 are similar to those in Figure 14. The model generally tracks the experimental results for hash-loops very closely, but provides a slightly misleading impression for hash-join and tc-hash-loops. The model predicts hash-join to be cheaper than tc-hash-loops, as for Q1. However, this is not what the system results show for the one and two processor configurations. We believe that this discrepancy can be explained by mild paging activity for the hash-join. In Q2, the two hash-join operators build large hash tables containing substantial *CompositePart* and *Document* tuples. In contrast, the hash tables for tc-hash-loops contain smaller input tuples resulting from the projections that are performed before exchanging data through the network. As a result, there is no paging activity in the evaluation of the plan for tc-hash-loops.

The models in this paper assume that the single-pass join algorithms can be evaluated without causing paging. It is also the case that the model predicts materialise to be more costly than hash-loops, while the system results show that hash-loops performs slightly worse than materialise. The I/O model for materialise seems to be more precise for Q2, where interference caused by the concurrent accesses to disk performed by the seq-scan and the two materialise operators is taken into account, than for hash-loops, where no interference is assumed, as both hash-loops operators consume their whole inputs before accessing the disk.

In Figure 16, tc-hash-loops and hash-loops present similar performance, as there is no sharing of *AtomicPart* objects among the objects of *CompositeParts*. This means that tc-hash-loops performs the same number of object-tuple mappings as hash-loops. hash-join suffers from paging when running Q3 on a single processor.

5.9.2. Varying predicate selectivity

Figures 17 and 19 show experimental results and the corresponding cost model estimates for Q1 and Q3, varying the selectivity of the predicate over the left input to the joins. Both model and experiments show that hash-join is more expensive than the two pointer-based joins for lower selectivities, as its whole right input is scanned no matter what selectivity is applied over the left input. For the pointer-based joins, a significant filtering on the left input causes fewer related objects to be referenced and retrieved from the store. For a selectivity of 100%, hash-join is the fastest operator, for reasons given in the discussion of Figure 14 above.

We note that in Figure 17, the model is successful at tracking the significant changes to the rankings of the plans as selectivity changes. At 1% selectivity, the cost ranking is hash-join > hash-loops > tc-hash-loops. At 10% selectivity, the ranking is hash-loops > hash-join > tc-hash-loops. At 100% selectivity the ranking is hash-loops > tc-hash-loops > hash-join. All these rankings are correctly predicted by the model.

Figures 18 and 20 show experimental results and the corresponding cost model estimates for Q1 varying the selectivity of the predicate over the right input to the join. As for the system results, the cost model predicts that the variation in the selectivity of the predicate does not substantially affect the join operators, both for the system results and the cost model estimates. For the pointer-based joins, the predicate on the right input to the join is applied after the join has taken place. For the hash-join, the predicate also has no effect on the amount of I/O carried out. The small increase in the elapsed times as the selectivity increases is mostly caused by the increase in the amount of data that is exchanged from the store processors to the coordinator and the tuple projections performed by apply.

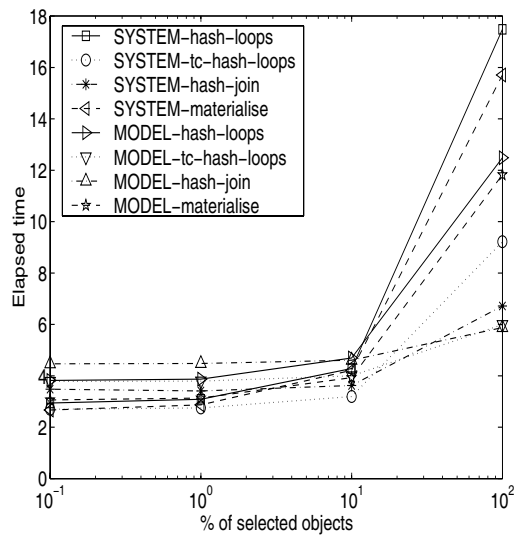


Figure 17. Selectivity tests for Q1 using v1.

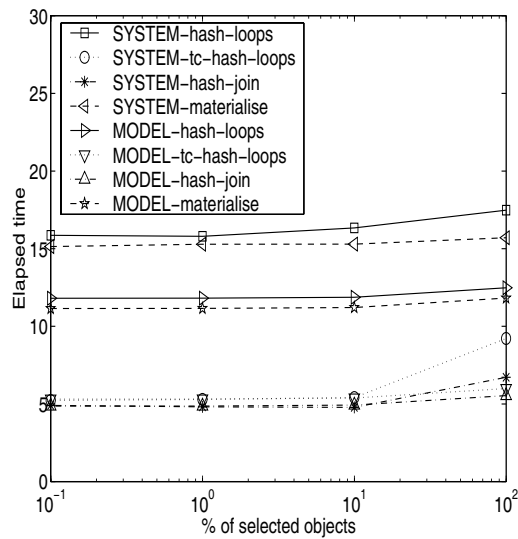


Figure 18. Selectivity tests for Q1 using v2.

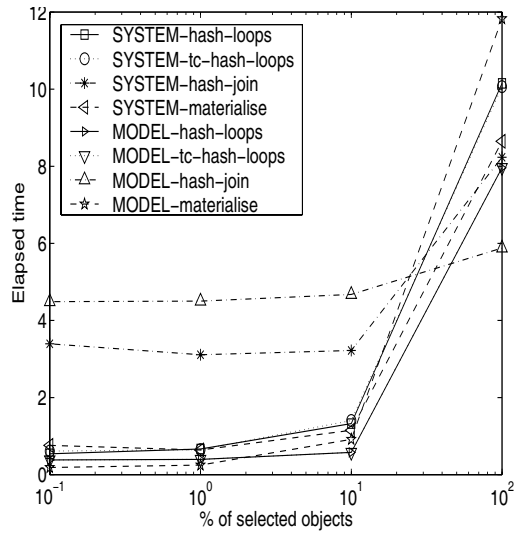


Figure 19. Selectivity tests for Q3 using v1.

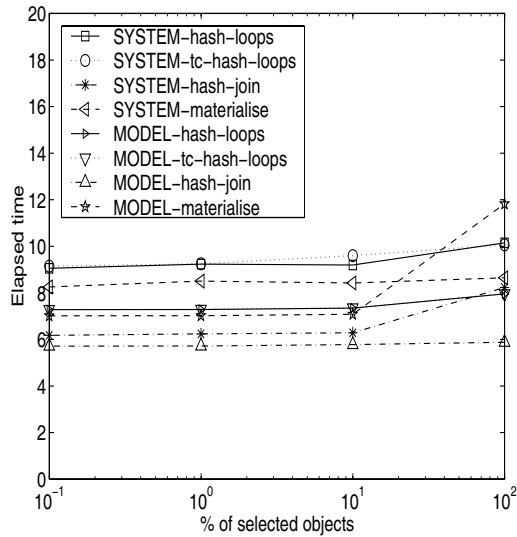


Figure 20. Selectivity tests for Q3 using v2.

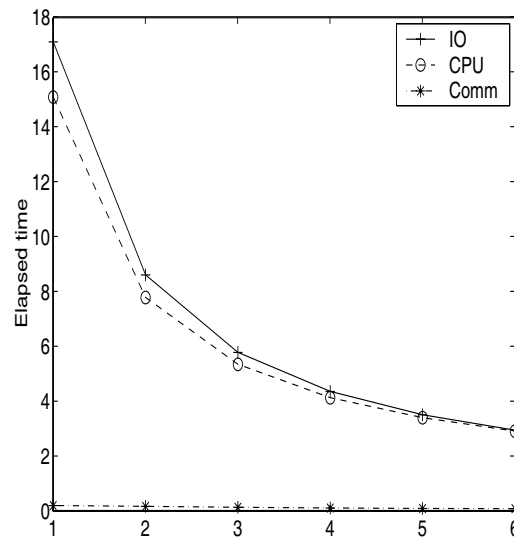


Figure 21. Q1 using hash-join.

5.9.3. *Intra-plan analysis*

The cost models described in this paper have been implemented using MATLAB, which provides facilities for more detailed study of the behaviour of the models. For example, Figures 21–24 show the distributions of the CPU, I/O and communication costs for Q1 with hash-join, hash-loops, tc-hash-loops and materialise, respectively. Note that the contribution of the communication costs to the response times are insignificant in the four figures, and decrease with an increase in the number of processors. This is because the costs included in the model for the exchange, such as for packing and unpacking of objects, benefit from parallelization. However, the most striking observation from these figures is the substantial contribution of CPU costs to all of the results, including its dominant contribution to the elapsed time for hash-loops and materialise.

Another way of breaking down the contributions to a plan focuses on the costs of individual operators. For example, Figures 25–28 show the contributions of the individual operators to the response times for Q1. One noteworthy feature from Figure 25 is the low CPU cost of hash-join, which suggests that building and probing the hash table imposes minimal overheads.

5.10. Summary

This section has described a cost model developed for the Polar system, which covers the operators of the parallel algebra used to implement the OQL queries. The section also describes the validation of the cost model against a wide range of experimental results. In contrast to many cost models found

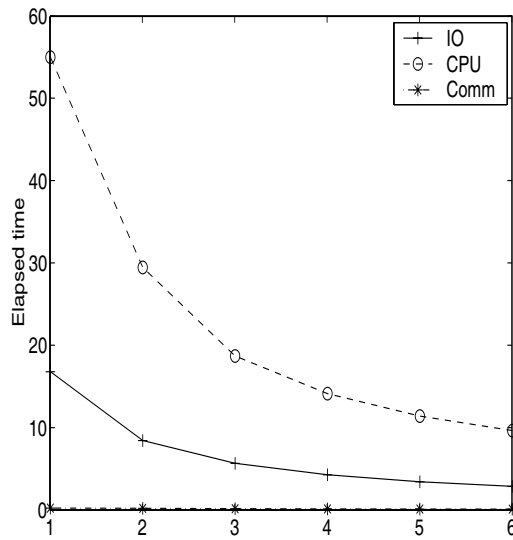


Figure 22. Q1 using hash-loops.

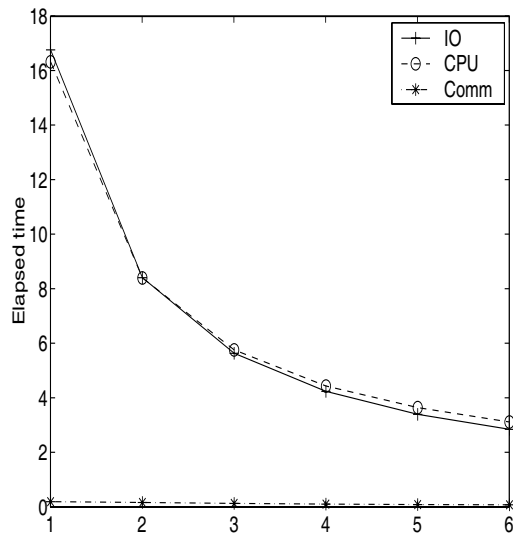


Figure 23. Q1 using tc-hash-loops.

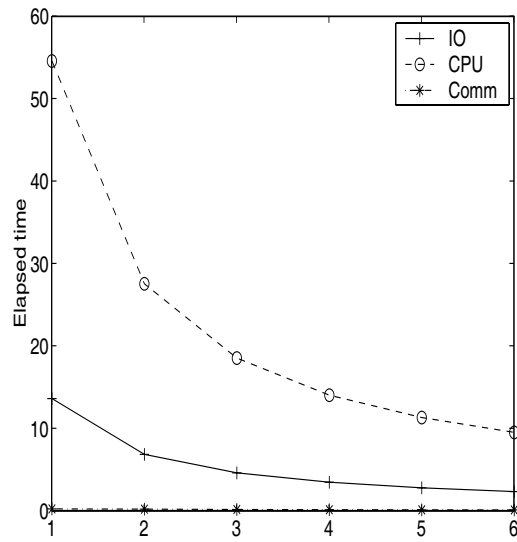


Figure 24. Q1 using materialise.

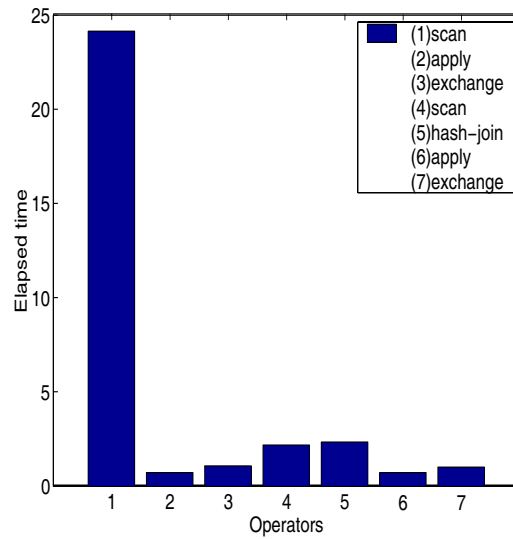


Figure 25. Operator timings for hash-join.

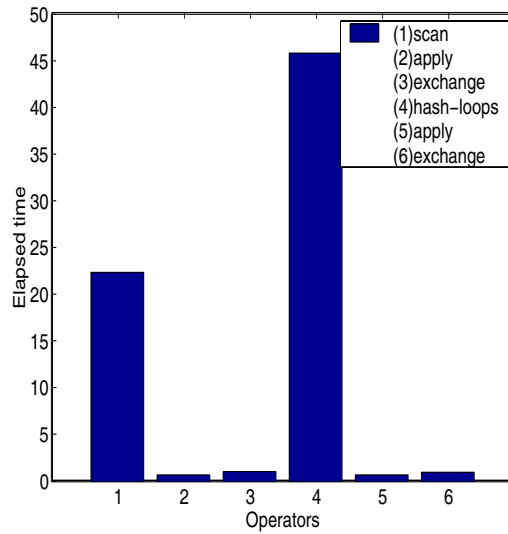


Figure 26. Operator timings for hash-loops.

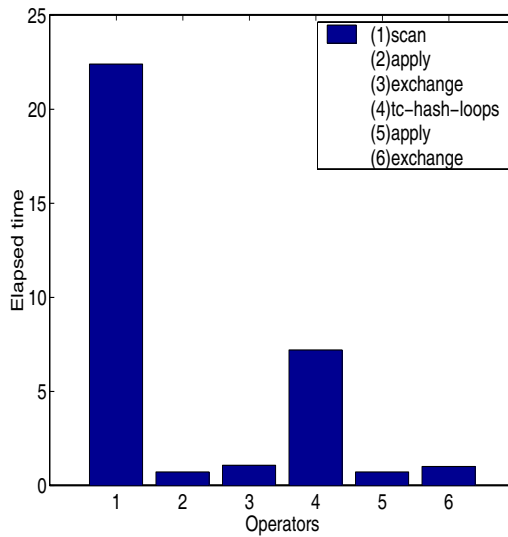


Figure 27. Operator timings for tc-hash-loops.

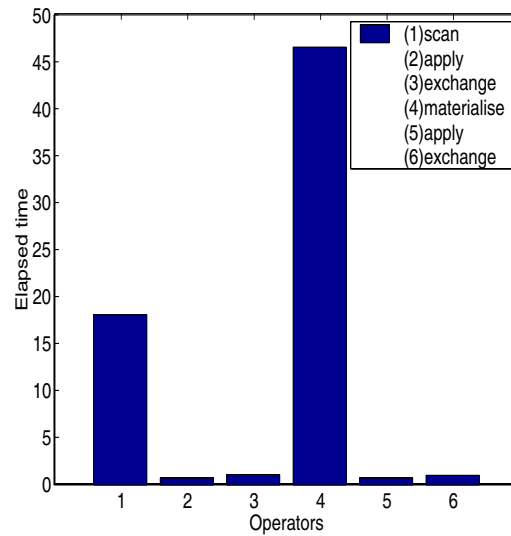


Figure 28. Operator timings for materialise.

in the literature, the Polar cost model: covers a complete parallel object algebra; takes CPU, I/O, and communication into account; takes different types of parallelism into account; and has been validated against system results.

The validation of the cost model is reasonably broad, covering most of the experiments in Section 4. The following table shows the percentage of errors when comparing the results obtained from the cost model with the system results.

Figure	hash-join	hash-loops	tc-hash-loops	materialise
14	10.6	27.8	22.6	22.6
15	11.4	14.4	8.6	6.7
16	14.4	16.9	19.2	17.3
17	24.7	23.3	34.4	13.8
18	9.1	26.6	9.4	26.1
19	37.6	36.9	38.9	48.3
20	13.0	20.5	21.4	21.2

These average error figures are generally larger than when validation results are reported by others referred to in Section 2, but the scope of both the cost model and the validation is relatively broader here compared with earlier examples of validation in the context of object queries and parallel database systems.

Models are less accurate than system implementations, being based on simplifying assumptions and capturing only a few aspects of a system's behaviour. Developing a cost model that is broad in scope



implies having more behavioural aspects of a system being modelled and more variables being taken into account within the model. It is a very challenging task to build such a model and obtain results that match tightly with reality in all the behavioural aspects of the system. Among the several behavioural aspects of the Polar system, those that have been shown to be hard to model are: (i) paging activity caused by exhaustive use of limited main memory by some operators, (ii) mapping of objects of various sizes and types into tuples, (iii) concurrent accesses to disk performed by operators running in the same node, and (iv) random disk accesses. The modelling of these four aspects present inaccuracies and are responsible for the error figures shown in the table above, as described in Section 5.9. The average errors are more significant for the experiments with variation of predicate selectivities, as the related queries have a short elapsed time and an error of one or two seconds represents a great percentage of the query's elapsed time.

6. COMPARING POLAR WITH OTHER DATABASE SYSTEMS

6.1. Commercial database systems

The main features provided by most commercial object database systems, such as Objectivity/DB, ONTOS DB, Versant, ObjectStore and GemStone, include mechanisms to support data persistency, client-server architecture and application interfaces based on object-oriented programming languages such as C++. Few of these systems, however, provide mechanisms for data querying based on SQL-like declarative query languages and none of them processes queries using any type of parallelism. Among the systems that provide some support for data querying through a declarative query language are Objectivity/DB, GemStone and O2.

A commercial parallel database management system is Teradata [37], which is a relational database system. Teradata runs on a shared-nothing machine and implements partitioned, pipelined and independent intra-query forms of parallelism, as well as inter-query parallelism. By its inter-query parallelism, Teradata allows several SQL queries to be bundled together and sent to the optimizer as if they were one. Any subexpressions that the different queries have in common will execute once and the results will be shared among them. Similar to Polar, Teradata performs automatic, compile-time optimization, where an operator tree is generated as a result of optimization.

6.2. Scientific database systems

Scientific parallel database systems, such as Volcano [38], MIDAS [39], XPRS [40], Gamma [23], Monet [32], Bubba [15], Cocoon [41], Goblin [42] and Prima [18], provide a declarative interface for querying data, which allows users to be insulated from details on the machine architecture and physical characteristics of data. In those systems, query optimization and parallelization are automatic, having as a result an operator tree that represents a query's evaluation plan. Although all of them implement some form of intra-transaction parallelism, few of them implement all forms of parallelism, e.g. Volcano, due to the complexity of combining all of them in a single optimizer.

Despite the fact that the shared-nothing approach for machine architecture is considered to be the most advantageous in terms of scalability, most of the mentioned systems are implemented over shared-memory machines, which provide ease of implementation. Among the systems that are implemented



over shared-nothing machines are Gamma, Shore [24], Bubba and Goblin. Among the mentioned parallel database systems, only Monet, Shore, Bubba, Cocoon, Goblin and Prima are object-based.

The Polar system uses mainstream query-processing technology, combining techniques that have been implemented with success in previous systems. In common with most of the mentioned systems, Polar provides a declarative query interface, performing automatic parallel optimization. The optimization is static, avoiding run-time overheads.

As in XPRS, Polar adopts two-phase optimization as a way of decreasing the complexity of parallel query optimization and allowing for modularity. The result of Polar's parallel optimization is an optimized parallel query evaluation plan, represented as an operator tree. Like Monet and Goblin, Polar implements join algorithms that do not write data to disk, having faster local processing than it would have if a disk-based approach was adopted, and allowing for assessment of limitations of join algorithms in relation to their memory usage.

Polar reuses implementation techniques developed within other systems, such as the operator model of parallelization, implemented in the Volcano system, and the iterator model of implementation, also used in the Volcano system. Polar is built on the top of Shore, using Shore as a storage system. Some of the advantages of Polar over most of the mentioned systems are itemized as follows.

- Use of a standard data model: Polar adopts a standard object data model, the ODMG data model and provides an interface to its query language, OQL.
- Use of commodity hardware: Polar is implemented over a shared-nothing machine architecture composed of commodity hardware, i.e. a cluster of PCs linked by an interconnection network.
- Implementation of all three forms of intra-transaction parallelism: Polar implements pipelined, independent and partitioned parallelisms.

7. CURRENT RESEARCH TRENDS AND USAGE OF OBJECT DATABASES

With the trend towards publishing data on and exchanging data through the Internet, languages for representing data on the Internet have been developed, and XML [43] has emerged as the dominant standard. In this context, there has been some work on combining the strengths of the XML data model with the maturity of database technology, such as [44,45]. The idea behind the combination of these two technologies is the following: with data stored in XML documents, it should be possible to query the contents of these documents. One should be able to issue queries over sets of XML documents to extract, synthesize, and analyse their contents. One of the approaches to achieve this is to load XML documents into a database system, translate semi-structured queries (specified in a language similar to XML-QL or Lorel) over XML documents in terms of the database query language and convert the results of queries back to XML. In the case of a relational database system, the documents are loaded into tuples, and semi-structured queries are translated into SQL queries over the corresponding relational data.

At the moment, however, there is little work on storing and querying XML data using object database systems (e.g. [46]), despite the fact that XML data have a tree-like structure that can be naturally mapped into linked objects of a object database. Nowadays object database systems are being used in production applications. Companies world wide have adopted commercial ODBMSs to fulfil their needs for high performance when handling complex data. Examples are itemized as follows [47].



- The Versant ODBMS is used by British Telecommunications (BT) for its integrated fraud management platform covering BT voice services.
- The Objectivity/DB ODBMS is used by Stanford Linear Accelerator Center (SLAC) for its flagship research project named B Factory. As of November, 2000, the SLAC stored 169 terabytes of production data using Objectivity/DB. The production data is distributed across several hundred processing nodes and over 30 on-line servers.
- The ObjectStore ODBMS is used in SouthWest Airline's Home Gate to provide self service to travellers through the Internet. This allows them to plan trips and purchase tickets on SouthWest Airlines.
- West McLaren Mercedes, developer of the MP4/13 Formula One racing car, is using Computer Associates' Jasmine ODBMS to enable its engineers to effectively monitor and track its Formula One car's performance.

It is hoped that the ODBMS market will become larger as more companies start adopting ODBMS technology to store and process complex data as well as XML data, overcoming the main barrier to the use of ODBMSs, which is the lack of familiarity by most programmers.

8. CONCLUSIONS

The paper has presented results on the evaluation of queries over a parallel object database server. To the best of our knowledge, the results reported are the most comprehensive to date on the experimental evaluation of navigational joins, and on the performance of parallel object databases in a shared-nothing environment. Furthermore, we know of no other validated results on a cost model for a parallel object algebra.

The principal conclusions have been presented above, at the ends of the sections on the experiments and the cost model. These have shown that query performance is affected by many factors, including the sizes of the collections participating in the join, the cardinalities of the relationships explored, the selectivities and locations of predicates on inputs to nodes, and the amount of memory available.

The cost models capture several features of the evaluated system that came as something of a surprise to us as systems developers, such as: (i) the substantial contribution of CPU to many of the response times; (ii) the minimal contribution of network costs to response times of the evaluated parallel queries; and (iii) the significant CPU cost of mapping data from disk format into the intermediate tuple format used by the query evaluator. The identification of (iii) led to the development of the *tc-hash-loops* algorithm as a variation of *hash-loops*, which was performing poorly in certain common circumstances.

What use are these results to others? We hope that the results could be particularly useful to three groups of people: developers of object database systems, who must select appropriate algorithms for inclusion in their systems; researchers working on analytical models of database performance, who stand to benefit from the availability of experimental results against which models can be validated; and the developers of query optimizers, who have to design cost models or heuristics that select physical operators based on evidence of how such operators perform in practice.

The MATLAB implementations of the cost models presented in this paper are available from <http://www.ncl.ac.uk/polar/models>.



REFERENCES

1. Chaudhri AB, Loomis M. *Object Databases in Practice*. Prentice-Hall: Englewood Cliffs, NJ, 1998.
2. Smith J, Sampaio SFM, Watson P, Paton NW. Polar: An architecture for a parallel ODMG compliant object database. *Proceedings of the Conference on Information and Knowledge Management (CIKM)*. ACM Press: New York, 2000; 352–359.
3. Carey M, DeWitt DJ, Naughton JF. The OO7 benchmark. *Proceedings of the ACM SIGMOD*. ACM Press: New York, 1993; 12–21.
4. Shekita E, Carey MJ. A performance evaluation of pointer-based joins. *Proceedings of the ACM SIGMOD*, Atlantic City, NJ, May 1990. ACM Press: New York, 1990; 300–311.
5. Sampaio SFM, Paton NW, Smith J, Watson P. Validated cost models for parallel OQL query processing. *Proceedings of the 8th International Conference on Object-Oriented Information Systems (Lecture Notes in Computer Science, vol. 2425)*. Springer: Berlin, 2002; 60–75.
6. DeWitt DJ, Lieuwen DF, Mehta M. Pointer-based join techniques for object-oriented databases. *Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems (PDIS)*. IEEE Computer Society Press: Los Alamitos, CA, 1993; 172–181.
7. Su SYW, Ranka S, He X. Performance analysis of parallel query processing algorithms for object-oriented databases. *IEEE Transactions on Knowledge and Data Engineering* 2000; **12**(6):979–997.
8. Braumandl R, Claussen J, Kemper A, Kossmann D. Functional-join processing. *VLDB Journal* 2000; **8**(3–4):156–177.
9. Buhr PA, Goel AK, Nishimura N, Ragde P. Parallel pointer-based join algorithms in memory-mapped environments. *Proceedings of ICDE*. IEEE Computer Society Press: Los Alamitos, CA, 1996; 266–275.
10. Metha M, DeWitt DJ. Data placement in shared-nothing parallel database systems. *VLDB Journal* 1997; **6**(1):53–72.
11. Wilschut AN, Flokstra J, Apers PMG. Parallel evaluation of multi-join queries. *ACM SIGMOD International Conference on Management of Data*. ACM Press: New York, 1995; 115–126.
12. Watson P, Townsend P. The EDS parallel relational database system. *The PRISMA Workshop on Parallel Database Systems*, Noordwijk aan Zee, 1990, America P (ed.) (*Lecture Notes in Computer Science*, vol. 503). Springer: Berlin, 1990.
13. Watson P, Catlow GW. The architecture of the ICL Goldrush MegaServer. *Proceedings of the 13th British National Conference on Databases*, Goble C, Keane JA (eds.) (*Lecture Notes in Computer Science*, vol. 940). Springer: Berlin, 1995.
14. Watson P. The design of an ODMG compatible parallel object database server. *International Meeting on Vector and Parallel Processing (VECPAR)* (*Lecture Notes in Computer Science*, vol. 1573). Springer: Berlin, June 1998.
15. Boral H, Alexander W, Clay L, Copeland G, Danforth S, Franklin M, Hart B, Smith M, Valduriez P. Prototyping Bubba, a highly parallel database system. *IEEE Transactions on Knowledge and Data Engineering* 1990; **2**(1):5–24.
16. Thakore AK, Su SYW. Performance analysis of parallel object-oriented query processing algorithms. *Distributed and Parallel Databases* 1994; **2**:59–100.
17. Chen Y-H, Su SYW. Implementation and evaluation of parallel query processing algorithms and data partitioning heuristics in object-oriented databases. *Distributed and Parallel Databases* 1996; **4**:107–142.
18. Gesmann M. Mapping a parallel complex-object DBMS to operating system processes. *International Euro-Par Conference*, Bougé L, Fraigniaud P, Mignotte A, Robert Y (eds.). Springer: Berlin, 1996; 852–861.
19. Boncz PA, Wilschut AN, Kersten ML. Flattening an object algebra to provide performance. *International Conference on Data Engineering*. IEEE Press: Piscataway, NJ, 1998; 568–577.
20. Olson MA, Hong WM, Ubell M, Stonebraker M. Query processing in a parallel object-relational database system. *Data Engineering Bulletin* 1996; **19**(4):3–10.
21. Blakelev JA, McKenna WJ, Graefe G. Experiences building the Open OODB query optimizer. *Proceedings of the ACM SIGMOD*, Washington, DC, 1993. ACM Press: New York, 1993; 287–296.
22. Schneider DA, DeWitt DJ. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, Portland, OR, 31 May–2 June 1989. ACM Press: New York, 1989; 110–121.
23. DeWitt DJ, Gerber RH, Graefe G, Heytens ML, Kumar KB, Muralikrishna M. Gamma—a high performance dataflow database machine. *Proceedings of the International Conference on Very Large Data Bases*, Kyoto, Japan, August 1986. Morgan Kaufmann: San Mateo, CA, 1986; 228–237.
24. DeWitt DJ, Naughton JF, Shafer JC, Venkataraman S. Parallelising OODBMS traversals: A performance evaluation. *VLDB Journal* 1996; **5**(1):3–18.
25. Thakore AK, Su SYW, Lam H. Algorithms for asynchronous parallel processing of object-oriented databases. *IEEE Transactions on Knowledge and Data Engineering* 1995; **7**(3):487–504.
26. Li Z, Ross KA. Fast joins using join indices. *VLDB Journal* 1999; **8**(1):1–24.
27. Haas LM, Carey MJ, Livny M, Shukla A. Seeking the truth about *ad hoc* join costs. *VLDB Journal* 1997; **6**(3):241–256.



28. Harris EP, Ramamohanarao K. Join algorithm costs revisited. *VLDB Journal* 1996; **5**(1):64–84.
29. Dempster EW, Tomov NT, Williams MH, Taylor H, Burger A, Trinder P. Modelling parallel oracle for performance prediction. *Distributed and Parallel Databases* 2003; **13**(3):251–269.
30. Graefe G. Query evaluation techniques for large databases. *ACM Computing Surveys* 1993; **25**(2):73–170.
31. Cattell RGG, Skeen J. Object operations benchmark. *ACM Transactions on Database Systems* 1992; **17**(1):1–31.
32. Boncz PA, Kwakkel F, Kersten ML. High performance support for OO traversals in Monet. *British National Conference on Databases*, Edinburgh, U.K., July 1996. Springer: Berlin, 1996; 152–169.
33. Smith J, Watson P, Sampaio SFM, Paton NW. Speeding up navigational requests in a parallel object database system. *Proceedings of the 8th International Euro-Par Conference (Lecture Notes in Computer Science*, vol. 2400). Springer: Berlin, 2002.
34. Williams MH, Dempster EW, Tomov NT, Pua CS, Burger A, Lu J, Broughton P. An analytical tool for predicting the performance of parallel relational databases. *Concurrency: Practice and Experience* 1999; **11**(11):635–653.
35. Yao SB. Approximating block accesses in database organizations. *Communications of the ACM* 1977; **20**(4):260–261.
36. Gesmann M. A cost model for parallel navigational access in complex-object DBMSs. *Database Systems for Advanced Applications '97, Proceedings of the 5th International Conference on Database Systems for Advanced Applications (DASFAA)*, Melbourne, Australia, 1–4 April 1997, vol. 6. World Scientific: Singapore, 1997; 1–10.
37. Ballinger C, Fryer R. Born to be parallel. *Data Engineering* 1997; **20**(2):3–12.
38. Graefe G. Volcano—an extensible and parallel query evaluation system. *ACM Transactions on Knowledge and Data Engineering* 1994; **6**(1):120–135.
39. Bozas G, Jaedicke M, Listl A, Mitschang B, Reiser A, Zimmermann S. On transforming a sequential sql-dbms into a parallel one: First results and experiences of the midas project. *Proceedings of the 2nd International Euro-Par Conference (Lecture Notes in Computer Science*, vol. 1124). Springer: Berlin, 1996; 881–886.
40. Hong W, Stonebraker M. Optimization of parallel query execution plans in XPRS. *Proceedings of the 1st International Conference on Parallel and Distributed Information Systems (PDIS 1991)*, Florida, December 1991. IEEE Computer Society Press: Los Alamitos, CA, 1991; 218–225.
41. Rys M, Norrie MC, Schek H. Intra-transaction parallelism in the mapping of an object model to a relational multi-processor system. *Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB'96)*. Morgan Kaufmann: San Mateo, CA, 1996; 460–471.
42. van den Berg CA. Dynamic query processing in a parallel object-oriented database system. *PhD Thesis*, Universiteit Twente, February 1994.
43. Extensible Markup Language (XML) 1.0, W3C Recommendation. <http://www.w3.org/tr/rec-xml> [March 2003].
44. Shanmugasundaram J, Tufte K, Zhang C, He G, DeWitt DJ, Naughton JF. Relational databases for querying XML documents: Limitations and opportunities. *Proceedings of 25th International Conference on Very Large Data Bases (VLDB'99)*. Morgan Kaufmann: San Mateo, CA, 1999; 302–314.
45. Zhang X, Pielech B, Rundesnteiner EA. Honey, I shrunk the xquery!: An XML algebra optimization approach. *Proceedings of the 4th International Workshop on Web Information and Data Management*. ACM Press: New York, 2002; 15–22.
46. Fegaras L, Levine D, Bose S, Chaluvadi V. Query processing of streamed XML data. *Proceedings of the 11th International Conference on Information and Knowledge Management (CIKM'02)*. ACM Press: New York, 2002; 126–133.
47. OODBMS FAQ. http://www.service-architecture.com/object-oriented-databases/articles/odbms_faq.html [March 2003].