Benchmarking Queries over Trees: Learning the Hard Truth the Hard Way*

Fanny Wattez†‡, Sophie Cluet†‡, Véronique Benzaken§
Guy Ferran†‡, Christian Fiegel‡

contact author: Fanny Wattez, INRIA, BP 105, 78153 Le Chesnay, France
tel: (33) 1 39 63 51 46, fax: (33) 1 39 63 56 84, email: Fanny.Wattez@inria.fr

Abstract

In October 1997, we started what we thought would be a comprehensive benchmarking of OQL [6, 7] queries. Two years of hard work and many blunders later, we have tested one tenth of what we were shooting for. The first part of this paper should be useful to would-be benchmarkers: it explains how we could have avoided losing time. In the second part, we show that our few results are in fact extremely informative. Notably, we understand now why O_2 does not cope well with large associative accesses requiring duplicate I/Os and how this could be seriously improved. Also, we have a better grasp on the pros and cons of navigation versus joins in a real system and believe we know how to efficiently evaluate queries over hierarchical structures.

1 Introduction

Hierarchical and graph structures are very popular nowadays, thanks to XML and object-relational systems that broadened their range of applications. They are usually accessed in two fashions, depending on the applications: follow links from node to node (for instance, to access the title of the first section of a given XML document) or use associative accesses (e.g., to find the titles of a large collection of documents). In this paper, we present the results of a benchmark on the O_2 system[2] that shows, among other interesting things, that focusing on one kind of access may lead to overlooking the other, needlessly handicaping its performance.

Not surprisingly, these results were not the ones we were looking for when we started benchmarking. We were planning to run a huge set of tests on the O_2 OQL optimizer in the hope of (i) defining an accurate cost model and (ii) improving its search strategy. We failed on both points. In two years, we tested very few queries over few databases. We did not even make enough runs to use data analysis in any meaningful way. So much for self-flagellation. On the positive side, we discovered by chance some hard truths about O_2 that we believe are relevant beyond this particular system. Indeed, they should interest any developer of a system dealing with objects, navigation and associative accesses.

The contribution of this paper is threefold:

*This work was partially supported by the ANRT (French National Association for Technical Research)
†INRIA, BP 105, 78153 Le Chesnay Cedex, France
‡Ardent Software, 3 Place de Saverne, 92400 Courbevoie, France
§Université Paris XI, LRI, 91405 Orsay Cedex, France

- First, we give some sound and *a posteriori* obvious advice to would-be benchmarkers.
- Next, we explain why O_2 does not cope well with large associative accesses. There are mainly two reasons for this. (i) O_2 implements the full ODMG [6] data model. Notably, the system allows the manipulation of arbitrary complex values and manages indexes on arbitrary collections (i.e., not just class extents). This implies that some information is associated to each object (such as, is this object indexed?). This may have a serious cost when many objects move from disk to memory. (ii) Object systems are tested with object benchmarks against relational systems and are optimized accordingly. The main focus is on optimizing applications requiring random navigation within objects residing in memory, something that relational systems do not perform well. Little attention is given to queries (e.g., until recently, there was only one engineer working a small day out of five on O_2's OQL) whose performances suffer accordingly (e.g., they have never been seriously measured in O_2).

  Our conclusion is that O_2's performance on associative accesses could be greatly improved without hurting those of main memory navigation by (i) cleaning and compacting the information that is kept about each object and (ii) considering bulk allocation of this information rather than on a per object basis. Alternatively, it could provide less functionalities.
- Last, we give a performance analysis of queries over hierarchical structures in an object-oriented database featuring physical identifiers. Although far from complete, we believe this constitutes the first assessment of how a real object system deals with large associative queries.

  In [14, 4], the authors compare pointer-based against value-based algorithms and favors the former. In this paper, we build on these results. We focus on pointer-based algorithms on large collections (1, 2 and 3 million objects). We compare algorithms relying on pure navigation (no intermediate structure but potentially random disk accesses) against hash-based ones. We use a slight variation of the pointer-based join algorithm of [14] that allows sequential rather than randomized access to the outer collection. In contrast to [3], we focus on physical object identifiers rather than logical ones.

  Our tests indicate that, although pointer-based hash joins are certainly efficient, pure navigation is not as bad as usually believed. In a nutshell, when the number of children is large, child-to-parent pure navigation is always comparable to the best hash-join algorithms; it is better when the number is very large and worse when it is small. The second point indicates the need for hybrid hashing, which we did not test. In the parent-to-child direction, navigation beats hash-joins by far when the data is nicely clustered and is dreadful otherwise.

The paper is organized as follows. The next section succinctly describes the setup in which we started our benchmarking. In Section 3, we recount two years of hard and often unnecessary work. Section 4 tells how we found out that O_2's internal structures needed to be improved and how this could be done. Section 5 presents our results on various join algorithms. Section 6 concludes the paper.

2 Setting up

The OQL optimizer of the O_2 database management system [2] relies on heuristics to choose the "best" execution plans. As expected, this implies that "best" is sometimes rather bad. For lack of manpower and for some years, the optimizer and its search strategy have been improved according to a three steps algorithm. (i) A customer would complain of bad performance on a crucial query;

(ii) an analysis would be made to understand what could be done to improve the optimizer without breaking its genericity (and notably, the previous improvements); (iii) finally, either the optimizer would be "improved" or the customer be advised of a better way to formulate his/her query. In this manner, over the years, O_2 optimizer has been improved and its few and patient users satisfied[1]. Still, at the end of 1997, the O_2 development team benefited from an increase in manpower and we considered implementing a cost-based search strategy.

Our first task was to find out what statistics the system should maintain and how to incorporate them into a cost model. For this, we planned on running a large benchmark. Our hope was that, with the help of an expert in data analysis (Yves Lechevallier at INRIA), we could elicit a cost model from the results (in a manner similar to what [11] proposes).

To our knowledge, there does not exist an object query benchmark. The 007 benchmark [5] aims at comparing the performances of object-oriented systems, not the different strategies for object query evaluation. Notably, it considers navigation down hierarchical structures but not alternative join evaluation of this navigation. The reason for this is probably that very few commercial object database systems feature a reasonable query language (3?) and only one (O_2) features the full fledged OQL. Still, we had to start from somewhere and decided to work on the Derby schema [10] that represents doctors and patients. The next step was to choose "representative" queries and the physical/logical parameters we wanted to study.

In sections 4 and 5, we will see which queries finally won our two years attention (mainly simple selections and navigation over a hierarchical structure). In this section, we present the environment in which we conducted our experiments.

- The Derby schema of 1997 has been adapted and seriously reduced as shown in Figure 1.

  If we consider an average of 3 patients per doctor and 16 characters strings, each object of Class *Provider* is about 120 bytes (4 bytes per integer, 8 per address or object identifier plus some system overhead). The object size is slightly smaller if we consider 1000 patients since large collections are stored separately. *Patient* objects are about 60 bytes.

- We considered two databases: the first contains 2,000 doctors and an average of 1,000 patients per doctor, the second 1,000,000 doctors and 3 patients per doctor (on average).

- We studied three physical representation of the same databases.

  The first implements a class clustering and is illustrated on the left side of Figure 2 (only names and relationships are shown). All the objects of one class are grouped into a file. Physical addresses (prefixed by the @ character) are used to reference objects. The relationship between doctors and patients is randomized. I.e., the order of patients does not follow that of their providers. In Figure 2, we represented the 1-3 relationship. One can see that the values of the set attribute *clients* are stored in the same file as the providers they belong to (although, in reality, not always right next to them, depending on how we load objects - see Section 3). In the 1-1000 case, things are different: as stated before, collections whose size is over 4K (the size of a page) are always stored in a separate file. Note that with 4K pages, partially filled (the systems always leaves some extra space to deal with growing strings or collections), and a 10^6x3 database, this organization leads to about 33000 (resp. 49000) pages of providers (resp. patients).

  The second physical organization is randomized (central part of Figure 2). All objects reside in the same file but randomly.

[1] Note that database programmers have been re-formulating queries for years and that all are not O_2 customers

Classes	Types
Provider	name: string upin: integer address: string specialty: string office: string clients: set(Patient)
Patient	name: string mrn: integer age: integer sex: char random_integer: integer num: integer primary_care_provider: Provider

Names	Types
Providers	set(Provider)
Patients	set(Patient)

Figure 1: The database schema

The third physical organization (right part of Figure 2) follows the 1-n relationship: patients are stored next to their respective provider. In O_2 this kind of clustering can be specified, but is not guaranteed. It may be necessary to dump and reload the database once in a while to maintain a reasonable cluster.

- We ran the system with both server and client on the same machine. The size of the server cache was 4M and that of the client 32M (i.e., it can hold 8000 pages).
- The machine on which we ran the tests is a Sparc 20 with 128MB of memory and an SCSI 2GB disk. The operating system is Solaris 2.6. Apart from the database itself, all files within the system are managed by AFS (Andrew File System [1]). This has a non negligible cost that we could not evaluate. Also, the tests were launched from a *twm* window manager that consumed some memory.
- All queries were run twice on a cold system (the server was shutdown at the end of each evaluation). The only time we found significant divergence between two results was due to a bug in our launching program.

3 Tips to Benchmarkers

Our goal was to benchmark a piece of code that we knew by heart. It did not look too difficult, although we understood that it could be tedious. Well, it was hard, long and very tedious. Most of the unpleasantness is certainly due to our inexperience in large benchmarking. As a matter of fact, the advice we are about to give all look incredibly obvious. Still, we did not find them in the litterature ([9, 15, 16, 13]) and we wish someone had given them to us when we started.

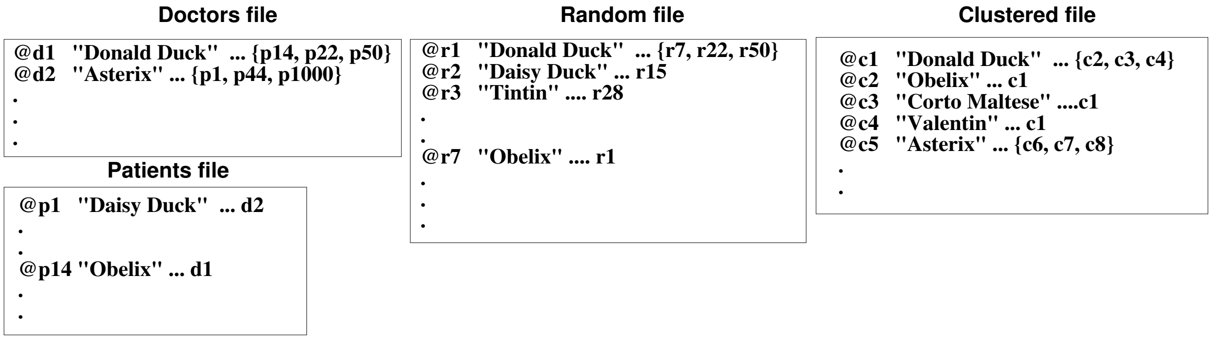

Figure 2: Three physical organizations

3.1 Buy Big!

We had a good workstation entirely devoted to our tests: a Sparc 20. Still, we did not have enough disk space and had to buy some more. We settled for an SCSI 2G bytes. It looked large at that time and definitely large enough to store 4 million objects (the largest of the database we planned to test).

Wrong arithmetic: when buying your disk think *sum*, not *max*. Each set of tests bring new questions and you need all your databases handy to answer them. For two years, we kept deleting/re-creating databases. Worst, since our creation programs kept changing (hopefully, less bugs and better performances), so did the databases they generated. Thus, not only did we have to re-create databases, sometimes we even had to re-run tests!

3.2 Get System Gurus Involved (Soon) Alternatively, Read the Documentation (Carefully)

We now count two system gurus in our ranks, for nearly two years we had none. The query optimizer is not among the top priorities of the O_2 development team and we had difficulties convincing people to do more than just listen sympathetically to our troubles and give us a little tip once in a while over a cup of coffee. We had never created a large database and were under the assumption that it was similar to creating small ones. Big mistake!

Our first problem was to avoid the "out of memory" message that occurs when you create too many objects within one transaction. One reason why want to have a system guru handy is to understand, given your system configuration, how many objects you can create before you have to spend time committing. In our case, we settled for 10.000.

At some point, we had created a database of 4M objects. We were rather proud of ourselves. Still, we were worried about the time it took (especially since, if you remember well, we kept deleting and creating databases): 12 hours. We asked a guru how long should a 1GB creation take. The answer was 1 hour! Obviously, we had a problem. After much effort, we managed to understand the various mistakes we were making and how we could improve things.

- We have always heard that it is more efficient to create an index once the collection is populated, rather than to update one at each object insertion. This is often true, but, as you can find out if you read carefully the O_2 system documentation, not for the first index.

  In order to maintain indexes properly, the O_2 system records, for each object, the indexes it belongs to (see Section 4 for an explanation). This information is stored on disk in the object header. When an object becomes persistent, if it is part of some indexed collection the system creates a header allowing to store information about 8 indexes (it can be extended if required). If it is not indexed, the header does not contain space for any index information. Thus, by indexing our collections after having created all objects, we forced the system to reallocate all objects on disk so as to add index information in their header. This is a rather expensive operation when you deal with collections containing 1 to 3 millions of objects. Further, this destroys the physical organization that you managed to impose to your system.

- When you load a large database from scratch, you are usually working by yourself. You do not need to worry about other programs updating the data you are working on and you do not care so much about loosing the data you are creating (you can always re-run the program). This is why, most systems provide the ability to run loading programs without transactions. By removing the need to manage a log and read/write locks, the O_2 transaction-off mode allows to load large databases faster (note that we used this mode only for loading, not for running our tests). This information, part of the system document, is one of the many we missed.

- Obviously, the size of client and server caches grandly influence the loading time. By default, these two parameters are set to 4MB. Asking the right people, you understand that with (i) 128MB of RAM, (ii) only one client running and (iii) no log maintained, a good configuration is 4MB for the server cache, and 32MB for the client. It is simple enough to understand. When no log is maintained, the only information that navigates from disk to memory is that needed by the applications. When the system serves only one client, the number of IOs depends on the largest cache size, independently of its function (client or server). Thus, by giving more memory to the client, you reduce both IOs and RPCs.

After having performed these various improvements, our loading process still required 5 hours, not 1. We found three explanations to this.

- Hardware is the first explanation. Our Sparc 20 is a bit dated nowadays as compared to the workstation our informant was using.

- We coded our loading program in O_2C. We chose this language because it is the most user-friendly among the three languages proposed by O_2 (Java and C++ are the other two). But it is an old language (O_2's first) whose compiler is certainly not optimal. We would get better performances with the C++ interface, or even better, by using the O_2 engine API.

- Imposing a specific data physical organization is important (see previous section) but can also be costly. Let us consider the database with 1M doctors, 3 patients per doctor and a class clustering (left part of Figure 2). The relationship between doctors and patients is randomized. In order to achieve that, we need to create all doctors and all patients before we can update the doctor-patients relationship (objects are located on files according to their creation time). The *upin* attribute of doctors is an integer representing the relative position of the object on disk. The *random_integer* attribute of clients is an integer whose value is comprised between 1 and 1M (the number of doctors) and that is assigned randomly (using the Unix lrand48 function). A join between the two collections over these attributes is used to update the association. Note that since we cannot perform too many updates within the same transaction, some optimization is needed in order to avoid performing the same and very large join too many times. Still, the operation takes time.

3.3 Large Benchmark Equals Many Numbers: Why Not Use a Database?

Although we are far from the goal we wanted to achieve when we started benchmarking, we are still playing with 5000 numbers (after having lost some). We started storing results in files and exploiting them by hand. On top of not being reliable, this is time consuming. You end up with files whose names are all but clear, you have to edit or grep in order to find the information you are looking for, your curves look sometimes rather strange, etc. After messing around in this fashion for some time, we realized that a database was a very reasonable place to store information. From then on, our results were automatically stored in a database whose object schema is illustrated in Figure 3.3. An object of class Stat is created each time an experiment is done. An experiment consists of (1) a query (attribute *query*) on (2) a database (attribute *database*) with (3) a clustering strategy (attribute *cluster*), (4) an algorithm (attribute *algo*) and (5) some parameters about the system configuration.

Once in a database, benchmark results are really easy to process. Notably, a query language can be used to extract the information you are looking for. Also, you can build easily automatic translation tools to create input files for data analysis softwares.

3.4 Do Not Think You Know What Your Tests Will Be

Two years ago, we thought about the OQL queries we wanted to test and the parameters we wanted to study. Then, we tested one canonical query extensively. There is nothing left of this first test that still took a good month of our time. There are many reasons for that: the database we tested was too small, there was a bug in our loading program, the physical organization was not the one we wanted, etc. In any case, we know now that testing a lot without analyzing intermediary results is a mistake. Apart from the fact that you may use a lot of your resources on worthless results, our experience is that by running tests in small increments, you get many interesting insights and a better inkling of what the next step should be.

3.5 Good News, Measuring is Easier Than You Think

O_2 relies on a client-server architecture. In that context, we did not really know what should be measured and how to measure it. We never found out how to evaluate the part of the CPU consumed by O_2 client and server or how to exclude from our measures the jobs performed by the AFS routines running regularly on our machines. Still, after many tests, we discovered that elapsed time was as good a measure as anything else. In most cases, it evolved similarly to the number of RPCs and IOs. When this was not the case, there was always a good reason, e.g., a hash on a very large table implying a lot of memory swap.

4 On Object Databases and Associative Accesses

Two experiences led us to think a little more about the way O_2 manipulates objects in main memory. We relate these experiences before analyzing the hard truth they taught us. Finally, we study how things could be improved.

4.1 Hash table: Rids or Handles?

Rid and **Handle** are two internal types of the O_2 system [2]. Rids (for Record identifiers) correspond to physical addresses on disks (e.g., the **@pis** in Figure 2), Handles point to the structure

```
class Stat
    numtest: integer,
    query: Query,
    database: {(Extent)},
    cluster: string,
    algo: string,
    system: System,
    CCPagefaults: integer, /*number of page faults in the client cache*/
    ElapsedTime: real,
    /*elapsed time between the beginning and the end of the query*/
    RPCsnumber: integer,
    /*number of RPCs between the client cache and the server cache*/
    RPCstotalsize: integer,
    /*total size (in Mb) of the messages between the client and the server*/
    D2SCreadpages: integer,
    /*number of pages that are read from the disk to the server cache*/
    SC2CCreadpages: integer,
    /*number of pages that are read from the server cache to the client cache*/
    CCMissrate: integer, /*miss rate (in percentage) in the client cache*/
    SCMissrate: integer /*miss rate (in percentage) in the server cache*/

class Query:
    cold: boolean, /*is the query evaluated after a server shutdown?*/
    projectiontype: string, /*projection type*/
    selectivity: integer, /*selectivity on queried extents*/
    text: string /*the text of the query*/

class Extent
    classname: string, /*the extent is on this class*/
    size: integer, /*cardinality of the extent*/
    associations: { [extent: Extent,
        linkratio: integer] }
    /*all the associations between the extent and the other extents in the base*/

class System
    servercachesize: integer, /*the server cache size*/
    clientcachesize: integer, /*the client cache size*/
    sameworkstation: boolean
    /*do the client and the server run on the same device?*/
```

Figure 3: A Schema to Store Benchmark Results

```
Get the Rids of patients whose mrn<k1 using a clustered index

For all returned Rids

   get an Handle and read the corresponding object

   hash the object by its primary care provider attribute
      /* the table contains entries of the form (doctor, {patient, ...}) */

Get the Rids of providers whose upin < k2 using a clustered index

For all returned Rids

   probe the hashtable to get the corresponding patients

   add to the resulting set
```

Figure 4: Pseudo-code of one hash-join algorithm

representing an object in main memory. When we started comparing join algorithms, we wonder about storing Handles or Rids in the hash table. So, we compared two versions of the hash-join algorithm whose pseudo-code is given in Figure 4. This algorithm joins patients whose $mrn < k1$ with providers whose $upin < k2$ and returns a set of tuples associating patients and providers. The algorithm is a variation of the pointer-based join algorithm of [14] (see next section for a detailed discussion).

As we will see in the next section, this is not the optimal way of performing this join. However, this algorithm requires the largest hashtable since the collection Patients, i.e. the largest collection, is hashed. That is why it illustrates well the difference between using Rids and Handles to represent hashed objects.

In Figure 5, one sees the result of running this algorithm on a base containing 2000 providers, and 2 million clients, all objects randomly stored on disk. *HJRID* (resp. *HJHD*) corresponds to the version with Rids (resp. Handles) in the hash table. Selectivities are given in percentage, the time ratio for one algorithm is the ratio between its elapsed time and that of the best algorithm. All times are given in seconds. For instance, consider the two first lines: when the selectivity on patients and providers is equal to 10%, HJRID is the best algorithm and HJHD is 1.1 times slower. The join returns a collection whose size varies between 20000 and 1620000 according to the selectivity. Note that, when 90% of the patients are selected, we are hashing 1.8 million objects and the performance of the algorithm is dreadful (we are not using hybrid hashing).

As can be noticed, storing objects as Rids in the hash table improves performances by a factor of two to three when the hashed collection is large (90% of the patients are selected). When the collection is small, the difference is not really important. Obviously, handles are larger than Rids (which is reasonable) thus imply more swapping when the number of entries in the hash table increased.

4.2 Indexes are Too Good to be True

At some point, we realized that a set of algorithms was running much faster without indexes than with. Since some indexes were used with a selectivity factor of 10% on a large collection, we were rather puzzled and decided to investigate this more seriously. Looking more carefully at the test, we realized we were using an unclustered index to perform the selection (i.e., the order of the keys is not that of the elements within the collection). Thus, we used a similar index and performed

Selectivity on patients	Selectivity on providers	Algorithm	Time ratio	Time (in sec)
10	10	HJRID	1	206.07
		HJHD	1.10	226.85
10	90	HJHD	1	351.28
		HJRID	1.03	364.57
90	10	HJRID	1	1964.28
		HJHD	2.82	5501.74
90	90	HJRID	1	16233.79
		HJHD	2.11	34311.17

Figure 5: Hash-Join with Rids or Handles

Selectivity	Time (sec) with index	Pages read with index	Time (sec) no index	Pages read no index
0.1	27.24	1961	802.15	54618
0.5	101.84	8680	814.90	54612
1	147.26	16451	815.17	54615
5	947.72	77431	821.41	54610
10	1824.83	153709	1352.79	54608
30	5376.24	458383	1467.75	54610
60	10724.57	915233	1641.24	54613
90	16177.83	1372629	1908.24	54612

Figure 6: Testing Unclustered Indexes

the simple selection-projection query given below on a set of 2 million patients (*num* is a random attribute of patients whose value ranges from 1 to 2M). Figure 6 shows the resulting numbers.

select pa.age
from pa **in** Patients
where pa.num > k

From the results without index, we can determine the cost of constructing a collection of 1.8 millions integers (selectivity of 90%). Indeed, when no index is used, the number of I/Os for performing a selection does not depend on the selectivity. It is the number of I/Os needed to scan the whole collection, object per object (refer to figure 6, last column). Moreover, the time required to perform the selection is the sum of the time to scan the whole collection and the time to build the result. When the selectivity is very high, since the result size is rather small (2000 clients when the selectivity is 0.1%), the time to build the result is negligible. Thus, 802.15 seconds can be considered as the time required to scan the collection Patients. Thus, the cost of constructing a collection of 1.8 millions integers is the difference between 1908.24 seconds (time for the selection without index and when the selectivity is 90%) and 802.15 seconds (time for the selection without index and when the selectivity is 0.1%), that is, about 1100 seconds or 18 minutes. Note that we are using here a standard transaction mode, i.e., the system creates this collection as if it could become persistent. Furthermore, note that the unclustered index increases the number of pages that have to be read once we reach a threshold selectivity situated between 1 and 5%. Especially, look at the number of I/Os when the selectivity is 5% and when an unclustered index is used. Since it is bigger than the number of I/Os required to perform the selection whithout index, many pages

Selectivity on Patients	Unclustered index + Sort	No index
10	343.49	1352.99
30	591.49	1467.75
60	1015.52	1641.24
90	1648.62	1908.24

Figure 7: Comparing Sorted Unclustered Index with No Index (time in sec)

```
open scan on Patients
for each Rid r returned by the scan
    get Handle h
    if get_att(h, num) > k
       add get_att(h, age) to the result
    unreference h
```

```
open index scan on (Patients, num > k)
for each Rid r returned by the index scan
    add r to Table T
sort T on Rids
for each r in T
    get Handle h
    add get_att(h, age) to the result
    unreference h
```

Figure 8: Standard Scan and Sorted Index Scan

are read more than once. This means that objects are accessed truly randomly and this is rather standard. Still, we were impressed. Since an index may be clustered or not and since we believe indexes to be important, we decided to see if a preliminary sort of the elements returned by an index could improve things. It did and exceeded our expectations by far. The results are shown on the figure 7.

Although we never ran this kind of test on another database system, our belief was that with 90% of selectivity, even clustered indexes were worthless (and could be bad). Now, not only our indexes were still very good when their use potentially augmented the number of I/Os (because we read all the collection pages but also those of the index structure - remember *num* is a random key), but even after adding the cost of sorting 1.8 millions of addresses (in the 90% case), they remained good. We then re-checked the results of the test on the non indexed selection. Let us concentrate on the first line: assuming 10ms per page read, there are about 250 seconds that are not spent on reads. Even knowing that some writes may have occurred and that constructing a collection of 2000 integers is not costless, this number seems rather large.

4.3 Handles are not Optimal

The only explanation we found to that phenomena requires that we look more deeply into the code involved. Figure 8 shows the pseudo-code executed when one selects patients without an index (on the left hand side) and with a "sorted" unclustered index scan (on the right hand side). If the object referenced by a Handle is not needed elsewhere, the "*unreference*" instruction frees the structure pointed by that Handle (in fact, it is sometimes delayed). In the current case, this means that there should not be swapping during the execution of any of the two given algorithms.

Figure 9 analyses the differences between the two algorithms when the selectivity factor is equal to 90% (i.e., all pages containing patients are read). Note that the standard scan needs to create and unreference handles for all the collection whereas the index scan needs only handles for the

	Standard Scan	Sorted Index Scan
Input/Output		+ Read index pages
CPU	+ get&unref 200,000 Handles	+ sort 1.8M Rids (8 Bytes each)
CPU	+ compare 2M integers	

Figure 9: Standard Scan or Sorted Index Scan: Cost Difference

selected elements. Obviously, the above comparison between the two algorithms and the numbers of Figure 7 indicate that Handles are expensive. At this point, we could not ignore that issue anymore and decided to investigate.

4.4 On Improving the Management of Objects in Memory

O_2 provides functionalities that other object database systems do not. Notably: (i) the full ODMG model, including arbitrary complex values and persistence by attachement, (ii) indexes on arbitrary collections (i.e., not only extents), (iii) object versioning and (iv) dynamic class evolution. These various features and the fact that objects can be shared imply that some system information be stored within each object.

To understand this, let us consider indexes on arbitrary collections. Suppose that we have an collection containing all patients living in Paris, indexed by their *primary_care_provider* attribute. Now, suppose that one doctor retires and that we want to assign "nil" to all his/her patients (some of whom live in Paris). How will the system know which index to update unless each patient carries that information (or we scan all indexes containing patients, but that is obviously not a reasonable solution)?

Indexes are just one example of the kind of information each object must carry along. Others are versions, exact type (because of inheritance), persistence (because of persistence by reachability), etc. Some of this information is stored on disk, some is needed only when the object is in memory. As an example of the latter, consider an application algorithm with two variables pointing to the same object. There is obviously no need to have two structures representing this object in memory. Thus, O_2 allocates only one and keeps a record of the number of pointers to this structure (to know when to delete it).

Now, let us take a non exhaustive look at some of the information stored within O_2 objects representatives (Handle) in memory (some information is common to other object systems).

- A pointer to the object in memory (if loaded) or on disk (if not loaded or swapped out).
- Some flags (bits) to indicate whereas the object is indexed, persistent, in memory, under schema modification, deleted, etc.
- A pointer to a structure giving information on the object exact type, structure that is shared by all objects of that type.
- A pointer to the list of indexes containing this object. This is needed because, as explained before, an object may be indexed as part of different collections and updated from many different places.
- The number of pointers to the object in memory. When this number is equal to zero, the memory space used by this representative can be recovered.

- A pointer to some structure representing the version to which the object belongs.
- Some information about the schema update history of the object class

All in all, the structure takes 60 Bytes of memory that have to be allocated, updated and freed whenever necessary. The above list is non exhaustive and has been growing over the years. The impact on performance went undetected. The reason for this is simple. Large Handles are bad essentially for cold associative accesses. Object benchmarks focus on applications requiring random navigation within objects residing in memory, applications for which Handles allocation has been optimized. Notably, the structure is large so that the system can perform management tasks required by object modifications fast and without having to fix the object in memory. Furthermore, the destruction of Handles is delayed as much as possible so as to avoid unnecessary free/allocate. Still, the system could be improved in different ways.

- The records associated to each Handle contains a large part of the information that is stored within the Handle. We evoked the fact that duplicating this information was good in order to have it handy at all time without fixing the object in memory. Still, we believe that a good benchmarking would tell us that some information could be removed at a reasonable cost for standard object applications.
- Handles do not only represent object, they are also given to complex values (literals or immutable objects) and strings (in fact, all elements whose size can evolve and that are represented as separate records in O_2). Most of the information stored in Handles is absolutely irrelevant to literals (no sharing, versioning, etc.). Obviously, a first simple task would be to give smaller Handles to literals by e.g., creating a class hierarchy of Handles. As a matter of fact, some Handle information concerns only some objects, yet is given to all. This hierarchy could thus also benefits objects.
- There exist some literals whose size never change on disk: tuples. Indeed, they have a fixed number of attributes whose values are either constant in size (e.g., integers) or stored in separate records (e.g., strings). Accordingly, there is no need to create separate records (and thus handles) for tuple literals that are part of an object. Note that, with C++ member classes, this case may occur rather frequently.
- A tighter connection between query optimizer and system engine would allow to manage handles in a more efficient way by considering allocating/updating/freeing handles for bulks of objects/literals rather than on a per object basis.
- Another more abrupt way of optimizing performance would be to reduce O_2 functionalities and forbids indexing arbitrary collections, versioning of objects, etc. Not surprisingly, we do not like this solution. Moreover, we note that O_2 is comparable in terms of performance to other object systems providing less functionalities.

5 On Join and Pure Navigation over Hierarchical Structures

In this section, we consider a typical OQL query that involves going down over a tree structure and study the performance of various algorithms in different situations (evolution of the database size and physical organization). Although far from complete, we believe this constitutes the first assessment of how a real object system deals with large associative queries (note that our results were obtained with a bad management of Handles). Indeed, the 007 benchmark [5] aims at comparing the performances of object-oriented systems, not the different strategies for object query evaluation.

The query is defined on the schema given in Figure 1 and is the following:

select f(p,pa)
from p **in** Providers,
pa **in** p.clients
where pa.mrn < k1 **and** p.upin < k2

where $f(p, pa)$ denotes a subquery involving providers and patients.

Before we introduce the algorithms we tested and compare their performance, we need to make some general remarks.

- Remember from Section 2 that queries are executed after an O_2 system shutdown, i.e., both client and server caches are empty (cold situation).
- We always store in the hash tables the elements needed to construct $f(p, pa)$, either objects, integers or strings.
- Whenever possible, we use two indexes to evaluate the conditions on $patient.mrn$ and $provider.upin$. Both indexes are clustered and store only object identifiers in their leaves (i.e., no object properties).
- In our tests, $f(p, pa)$ corresponds to the construction of a tuple with two attributes. Sometimes, the attributes are assigned the scanned objects (i.e., p and pa), other times properties of these objects (e.g., $p.name$, $pa.age$). Using one or the other influences the algorithm evaluation time in different ways. For instance, consider that we return objects (i.e., not objects properties). A navigation algorithm does not need to read patients (resp. providers) if it uses an index to evaluate the condition on providers (resp. patients) whereas hash-join algorithms do. The following results are obtained with $f(p, pa) = [p.name, pa.age]$, i.e., they require that all selected objects be loaded at least once.

5.1 The Algorithms

We compare four algorithms, two are based on navigation and two on hashing. We started testing sort-based algorithms but they proved to be worse than hash-based ones and we dropped them. The algorithms are the following ones :

Parent-to-child navigation (NL)
The algorithm is :

For all providers p whose $upin < k2$ /* index scan */
  For all clients pa of p /* navigation */
    if $pa.mrn < k1$
      add $f(p, pa)$ to the result

Note that only one index can be used for this algorithm since patients are accessed through their doctors. This is a big handicap since the collection of patients is the largest of the two (can be a 1000 times bigger). Also, note that whereas providers are accessed sequentially at all time, patients are accessed (i) randomly if we consider a class or random clustering physical organization and (ii) sequentially otherwise. Obviously, (i) constitutes another large handicap (especially since the random accesses are performed on the largest collection) whereas (ii) is a real plus.

Child-to-parent navigation (NOJOIN)

The algorithm is :

For all patients whose $mrn < k1$ /* index scan */
    get the patient primary_care_provider p /* navigation */
        if $p.upin < k2$
            add $f(p, pa)$ to the result

Again, only one index can be used. However, this time, it is that of the largest collection so the handicap is less. However note that, since a doctor belongs to 3 (resp. 1000) patients, we may test the condition on $p.upin$ up to 3 (resp. 1000) times. Concerning the way collections are accessed, patients (the large collection) are always accessed sequentially. For providers, we consider three cases. (i) The random or class clustering strategy implies random accesses. (ii) When doctors are clustered with three patients, we are accessing doctors sequentially. (iii) When doctors are clustered with 1000 patients, the page containing one doctor and that containing its 1000th patient are different (there are about 50 patients per page). Still, the chances are that one doctor will remain in memory the whole time its 1000 patients are considered since they are accessed in a row.

We call this algorithm $NOJOIN$ because the join is hidden within the navigation pattern.

Hash the parents and join (PHJ)

The algorithm is :

hash all providers whose $upin < k2$ by their identifiers /* index scan */
        /* An entry in the hash table = (providerid, provider information) */
For all patients whose $mrn < k1$ /* index scan */
    get the information about its primary_care_provider by probing the hash table
    add $f(p, pa)$ to the result

In any case, the algorithm allows to use index on both collections and access them in a sequential way. Yet note that, in the composition clustering case, accessing all patients leads to accessing all doctors and vice versa.

Naturally, the size of the hash table varies according to the selectivity on patients. Figure 10 gives an approximation of the hash table size for this algorithm and the next. Recall that we have a RAM of 128MB, 36MB of which are used by the O_2 caches. Some other non evaluated MB are consumed by the window manager. Given that, one can see that swapping will occur in the 1:3 case, when 90% of the providers are selected. We did not consider hybrid hashing [17] to optimize this. Finally, note that this algorithm requires more instructions than the previous ones.

Hash the children and join (CHJ)

The algorithm is :

hash all patients whose $mrn < k1$ by their primary_care_provider /* index scan */
        /* An entry in the hash table = (provider, {patient1, patient2, ...}) */

Algorithm	Number of providers	Number of children	Selectivity on patients	Selectivity on providers	Hash table size (MB)
PHJ	2000	1:1000		10	0.0128
PHJ	2000	1:1000		90	0.1152
PHJ	1000000	1:3		10	6.4
PHJ	1000000	1:3		90	57.6
CHJ	2000	1:1000	10		1.72
CHJ	2000	1:1000	90		14.52
CHJ	1000000	1:3	10		62.4
CHJ	1000000	1:3	90		81.6

Figure 10: Approximation of the hash table sizes

For all providers whose $upin < k2$ /* index scan */
  get the corresponding patient information in the hash table
  add $f(p, pa)$ to the result

This algorithm is a slight variation of the pointer-based join of [14]. Because we do not rely on hybrid hashing, we are able to scan the provider collection sequentially rather than access it randomly, according to the objects occurences in the hash table.

The algorithm has the same characteristics than the previous one concerning indexes or sequentiality of accesses. The difference is the size of the hash table that contains children rather than parents (i.e., potentially 3 to 1000 times more elements). As a consequence, we see that this table is too large in the 1:3 case whatever the selectivity on Patients is.

5.2 Results with Class Clusters

Figure 11 (resp. 12) shows the result we obtained with a 1-1000 (resp. 1:3) relationship on a database that stores objects of the same class together in one file.

Let us consider first the 1:1000 relationship. Not surprisingly, hash join algorithms are the most performant and *NL* is dreadful except when very few providers are selected. What is more surprising is the fact that *NOJOIN* is comparable to hash joins. The reason is that the access and index overhead of this algorithm depends on the small collection and is thus not that significant. Things are evidently very different in the 1:3 case where we are dealing with 1 million providers (and 3 million patients). *NOJOIN* becomes dreadful and becomes comparable to the hash join algorithms only when these require too much memory. As a matter of fact, we note that *CHJ* and *PHJ* evolve respectively and naturally according to this factor.

Although not illustrated here, storing objects randomly on disk multiplies evaluation time by a factor of 1.5 to 2 (not really a surprise) but favours the same algorithms (see Figure 15). This type of physical organization may look silly. Yet, in a model where many data types do not have a fixed size it is important to know the price one will pay by performing updates resulting in size increase of certain properties, properties that will have to be moved (maybe far from their owner).

Selectivity on patients	**Selectivity on providers**	**Algorithm**	**Time ratio**	**Time (sec)**
10	10	PHJ	1	89.83
		CHJ	1.12	101.05
		NOJOIN	1.40	125.90
		NL	15.79	1418.56
10	90	CHJ	1	154.09
		PHJ	1	154.57
		NOJOIN	1.24	191.51
		NL	80.03	12331.96
90	10	PHJ	1	925.07
		NOJOIN	1.36	1266.31
		CHJ	1.42	1320.69
		NL	1.63	1509.19
90	90	PHJ	1	1913.80
		CHJ	1.02	1956.35
		NOJOIN	1.20	2315.62
		NL	7.01	13423.38

Figure 11: One file per Class, 2x10^3 Providers, 2x10^6 Patients

Selectivity on patients	**Selectivity on providers**	**Algorithm**	**Time ratio**	**Time (sec)**
10	10	PHJ	1	365.72
		CHJ	1.10	402.38
		NOJOIN	9.70	3550.62
		NL	12.48	4566.06
10	90	CHJ	1	1286.18
		NOJOIN	2.93	3777.10
		PHJ	4.44	5723.28
		NL	31.96	41119.29
90	10	PHJ	1	2676.37
		NL	1.77	4738.09
		CHJ	3.53	9457.91
		NOJOIN	11.70	31318.05
90	90	NOJOIN	1	34708.13
		NL	1.26	43850.03
		PHJ	1.27	44188.33
		CHJ	1.69	58963.71

Figure 12: One file per Class, 10^6 Providers, 3x10^6 Patients

Selectivity on patients	Selectivity on providers	Algorithm	Time ratio	Time (sec)
10	10	NL	1	92.78
		NOJOIN	10.36	961.88
		CHJ	10.47	971.84
		PHJ	10.56	980.42
10	90	NL	1	923.84
		PHJ	1.12	1042.16
		CHJ	1.16	1078.47
		NOJOIN	1.18	1090.98
90	10	NL	1	155.17
		PHJ	7.50	1164.97
		CHJ	7.87	1221.29
		NOJOIN	8.40	1303.90
90	90	NL	1	1665.51
		PHJ	1.14	1898.97
		CHJ	1.19	1993.88
		NOJOIN	1.20	2006.76

Figure 13: Composition Cluster, 2x10^3 Providers, 2x10^6 Patients

5.3 Results with Composition Clusters

Figure 13 (resp. 14) shows the result we obtained with a composition clustering (see right part of Figure 2) and a 1-1000 (resp. 1:3) relationship.

This time, we note that navigation is by far the most advantageous situation. Furthermore, comparing these results with those of the previous section, we see that composition clustering improves performance seriously except in one case: when many providers and very few patients are selected. This is not really a surprise since this means that each page read to access the many providers will contain many non selected patients (especially in the 1:1000 case).

For similar reasons, our composition clustering may have a serious cost on simple selections, especially those requiring that we read only parents. However, note that this is due mainly to the way O_2 is implementing composition clustering, an implementation that aims at optimizing random navigation through some particular objects of an application rather than associative accesses. Another alternative, as in [4], would be to store patients and doctors separately, but according to the way they are associated to each other (i.e., the first objects in the patients file would be patients of the first doctor in the providers file). In this fashion, simple selections and hash-joins would perform as in the class clustering case while the performance of $NOJOIN$ and NL algorithms would remain the same.

As a conclusion, Figure 15 gives the best algorithm and corresponding evaluation time corresponding to the different cases we studied here.

6 Conclusion

Relational database systems are known to be efficient on associative queries, object systems on navigation. We think that there is no objective reason not to be good on both kinds of accesses. Naturally, since we are object-oriented, we believe it may be easier for object systems to achieve

Selectivity on patients	Selectivity on providers	Algorithm	Time ratio	Time (sec)
10	10	NL	1	165.97
		NOJOIN	8.82	1465.20
		PHJ	9.43	1566.68
		CHJ	9.84	1634.72
10	90	NOJOIN	1	1572.40
		NL	1.11	1749.50
		CHJ	2.02	3181.43
		PHJ	5.14	8090.45
90	10	NL	1	280.53
		PHJ	6.88	1932.78
		NOJOIN	7.08	1988.82
		CHJ	17.79	4993.11
90	90	NL	1	2709.16
		NOJOIN	1.22	3332.08
		PHJ	3.78	10251
		CHJ	3.97	10761.14

Figure 14: Composition Cluster, 10^6 Providers, $3x10^6$ Patients

Rel prov: pat	Sel. pat.	Sel. prov.	Best Algo Rand. Org.	Time For Rand. Org.	Best Algo Class Cluster	Time For Class Cluster	Best Algo Comp. Cluster	Time For Comp. Cluster
1:1000	10	10	PHJ	158.67	PHJ	89.83	NL	92.78
	10	90	CHJ	279.88	CHJ	154.09	NL	923.84
	90	10	PHJ	1419.87	PHJ	925.07	NL	155.17
	90	90	CHJ	2617.10	PHJ	1913.80	NL	1665.51
1:3	10	10	PHJ	277.24	PHJ	365.72	NL	165.97
	10	90	CHJ	1884.61	CHJ	1286.18	NOJOIN	1572.40
	90	10	PHJ	2216.87	PHJ	2676.37	NL	280.53
	90	90	NL	41954.19	NOJOIN	34708.13	NL	2709.16

Figure 15: Summarizing Results: Winning Algorithms

that, and, even easier if the system features physical identifiers (see [14, 4]). We believe this to be especially true when there is a need to manipulate hierarchical structures. In this paper, we did not prove anything, merely explored the matter from the O_2 perspective. Still, we hope that the experiences we related will be useful to others.

Acknowledgments

We would like to deeply thank Yves Lechevallier for helping us trying to make sense of our figures and Jérôme Siméon for giving us a hand in using YAT[8] to convert data from O_2 to Gnuplot [12].

References

[1] The Andrew File System, 1997.
`http://www.alw.nih.gov/Docs/AFS/AFS_toc.html/` .

[2] The o_2 database system.
`http://www.ardentsoftware.fr/`.

[3] R. Braumandl, J. Claussen, and A. Kemper. Evaluating functional joins along nested reference sets in object-relational and object-oriented databases. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, New-York, 1998.

[4] M. J. Carey and G. Lapis. An incremental join attachment for starburst. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, Brisbane, Australia, 1990.

[5] M. J. Carey, D. J. De Witt, and J. F. Naughton. The 007 benchmark. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, 1993.

[6] R.G. Cattell. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann, 1997.

[7] S. Cluet. Designing OQL: Allowing Objects to be Queried. *Information Systems*, 23(5), 1998.

[8] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your Mediators Need Data Conversion! In *Proc. of the ACM SIGMOD Conf. on Management of Data*, Seattle, Washington, June 1998.

[9] C. Turbyfill D. Bitton, D. De Witt. Benchmarking database systems : a systematic approach. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, Florence, Italy, 1983.

[10] The Derby Schema, 1997.
`http://www.odbmsfacts.com/derby/1997.html/` .

[11] J. Fedorowicz. Database evaluation using multiple regression techniques. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, Boston, Massachusetts, 1984.

[12] The Gnuplot Software Reference Manual.
`http://www.cs.dartmouth.edu/gnuplot/gnuplot.html/`.

[13] J. Gray, editor. *The benchmark handbook for database and transaction processing systems.* Morgan Kaufmann, 1993.

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

[15] M. Stonebraker. Tips on benchmarking data base systems. *Database Engineering Bulletin 8(1)*, 1985.

[16] E. Cortes-Rello S.W. Dietrich, M. Brown and S. Wunderlin. A practitioner's introduction to database performance benchmarks and measurements. *The computer Journal 35(4)*, 1992.

[17] D. De Witt, R. Katz, and F. Olken. Implementation techniques for main memory database systems. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, Boston, Massachusetts, 1984.