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

Fanny Wattez[†] Sophie Cluet, Véronique Benzaken

INRIA, BP 105, 78153 Le Chesnay, France U. Paris XI, LRI, 91405 Orsay, France

Fanny.Wattez@inria.fr Sophie.Cluet@inria.fr Veronique.Benzaken@lri.fr

Guy Ferran Christian Fiegel

Ardent Software, 3 place de Saverne, 92400 Courbevoie, France

Guy.Ferran@ardentsoftware.fr Christian.Fiegel@ardentsoftware.fr

1 Introduction

Hierarchical and graph structures are very popular nowadays, thanks to XML and object-relational systems that broadened their range of applications. They can be accessed in two fashions, depending on the applications: follow links from node to node (e.g., DOM-like [5]) or use associative accesses. A benchmark we ran on the O_2 system[1] showed, 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. Still, we believe they are relevant beyond this particular system and should interest any developer of a system dealing with objects, navigation and associative accesses.

In [8], we give some advice to would-be benchmarkers, explain why O_2 does not cope well with large associative accesses and give a performance analysis of queries against hierarchical structures in an object-oriented database featuring physical identifiers. This paper summarizes these three contributions.

2 Tips to Benchmarkers

The O_2 database management system [1] supports OQL [3,2]. The current optimizer relies on heuristics. At the end of 1997, we benefited from an increase in manpower and considered implementing a cost-based search strategy. So, we started benchmarking in order to better understand what a good cost model should be. We understood that benchmarking could be useful. Well, it was hard, long and very tedious. Most of the unpleasantness is certainly due to our inexperience. As a matter of fact, the advice we are about to give look incredibly obvious. Still, we did not find them in the litterature (e.g., [6]) and we wish someone had given them to us when we started.

Buy Big! We bought a 2G bytes disk. It looked large enough to store the largest of the databases we planned to test. Wrong arithmetic: when buying your disk, think *small*. Each set of tests bring new questions and you need all your databases handy to answer them.

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 fun to code and only benefited from a little guru tip once in a while. We were under the assumption that creating a large database was similar to creating a small one. Big mistake! Among the many problems that slowed our work considerably, let us simply mention this one. We had 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 usually true, but, as you can find out if you read carefully the O_2 system documentation, not for the first index. For various good reasons [8], the creation of a first index on a collection entails a physical move of all its objects and that takes some time.

Why Not Use a Database to store your results? We started storing results in files. On top of

*This work was partially supported by the ANRT (French National Association for Technical Research)

†This work was done while the author was an employee of Ardent Software

not being reliable, this is time consuming. You end up with files whose names are all but clear, have to use low level tools, etc. After messing around in this fashion for some time, we realized that a database was a very reasonable place to store information.

3 On Object Databases and Associative Accesses

O_2 provides functionalities that other object database systems do not. Notably: (i) the full ODMG model [2], 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 information be associated to each object. As an example, consider indexes. An object may belong to several collections, not all of which are extents, and that may be indexed or not. Thus, when an object is updated, there is a priori no way to know which index should be updated, unless the object has the appropriate information.

Thus, when O_2 loads an object into memory, it creates a structure called DV (for Dope Vector) that contains information about indexes, versions, types, etc. At some point in our experiments, we ran into results that clearly showed that DVs were large, considerably slowing down algebraic operations requiring some buffering (e.g., most join algorithms, grouping). We realized then that DVs had been growing over the years to reach 60 Bytes. At first, we thought it strange that no impact on performance was ever detected. But actually, DVs growed to improve performance. Indeed, large DVs are bad essentially for cold associative accesses requiring buffering. But, object benchmarks focus on applications requiring random navigation within objects residing in memory, applications which benefit a lot from a structure that allows to perform management tasks fast and without having to fix the object in memory. In [8], we propose various ways to solve this apparent conflict of interest.

4 On Join and Pure Navigation over Hierarchical Structures

We tested extensively a typical object query on collections of 1 to 3 million objects. The query is characterized by a dependency between variable definitions as in the following OQL **from** clause:

from o1 **in** Collection1,
o2 **in** o1.children

We compared algorithms relying on pure navigation (no intermediate structure but potentially random disk accesses) against hash-based ones, considering each time parent-to-child and child-to-parent accesses. The hash-join algorithm we used is a slight variation of the pointer-based join algorithm of [7] that allows sequential rather than randomized access to the outer collection.

Our tests indicate that, although pointer-based hash joins are certainly efficient, pure navigation is not 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. This seems to indicate that object database systems should be natural candidates to provide DOM interface on persistent XML data [5]. Our results also seriously suggest that, given the appropriate manpower, object database systems could support OQL as efficiently as relational systems support SQL.

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[4] to convert data from O_2 to Gnuplot.

References

[1] The o$_2$ database system. `www.ardentsoftware.fr`.

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

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

[4] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your Mediators Need Data Conversion! In *Proc. ACM SIGMOD*, 1998.

[5] World Wide Web Consortium. The document object model. `www.w3.org/DOM`.

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

[7] E. J. Shekita and M. J. Carey. A performance evaluation of pointer-based joins. In *Proc. ACM SIGMOD*, 1990.

[8] F. Wattez, S. Cluet, V. Benzaken, G. Ferran, and C. Fiegel. Benchmarking queries over trees: Learning the hard truth the hard way. `www-rocq.inria.fr/~wattez`.