Xiaofang Zhou Stanley Su
Mike P. Papazoglou Maria E. Orlowska
Keith G. Jeffery (Eds.)

Web Information Systems – WISE 2004

5th International Conference on
Web Information Systems Engineering
Brisbane, Australia, November 22-24, 2004
Proceedings

Springer

Table of Contents

Session 3: Information Extraction

Session 4: Advanced Applications

Session 5: Performance Issues

Session 6: Linkage Analysis and Document Clustering

Session 7: Web Caching and Content Analysis

Session 8: XML Query Processing

Session 9: Web Search and Personalization

Session 10: Workflow Management and Enterprise Information Systems

Session 11: Business Processes

Session 12: Deep Web and Dynamic Content

Session 13: Web Information System Design

Session 14: Ontology and Applications

Session 15: Multimedia, User Interfaces, and Languages

Session 16: Peer-to-Peer and Grid Systems

What Does It Mean to "Measure Performance"?

Alistair Moffat[1] and Justin Zobel[2]

[1] Department of Computer Science and Software Engineering
The University of Melbourne, Victoria 3010, Australia
[2] School of Computer Science and Information Technology
RMIT University, Victoria 3001, Australia

Abstract. The purpose of much computer science research is to invent algorithms, and generate evidence to convince others that the new methods are worthwhile. All too often, however, the work has no impact, because the supporting evidence does not meet basic standards of rigor or persuasiveness. Here the notion of "experiment" is explored, with reference to our current investigation into distributed text query processing on a cluster of computers. We describe some of the issues that we encountered, and lessons that can be applied by researchers designing experiments in other areas of computing.

1 Introduction

Research into practical systems is often intended to yield what the investigators believe to be new structures and algorithms. Having established via the research literature that an invention is indeed novel, the investigators seek to establish its worth. In most papers, the demonstration of worth is by some combination of experiment, simulation, or analysis. Sometimes the demonstration is entirely absent.

For the great majority of practical algorithms the most reliable form of demonstration is by experiment. The alternatives are interesting, but not compelling. In some papers, the "proof" of validity or superiority rests entirely on rhetoric – known in less intellectual circles as "hand-waving". Such papers have little impact in either an academic or practical sense.

In other papers, the demonstration is by mathematical analysis, or simulation using a model, or a combination of both. But such approaches typically involve simplifications such as representing data by statistical distributions, and can fail to capture the complexity of a realistic computing environment. Asymptotic superiority can also be illusory. The difference between a $\log n$ factor and a $\log\log n$ factor in an analysis can be completely swamped by constant factors, and many algorithms that appear strong analytically have not survived the test of a practical implementation.

In the traditional sciences, theories are working hypotheses whose applicability is validated by experiments designed to distinguish between competing proposals. An experiment that confirms a theory does not prove that theory to be true – it merely adds weight to the likelihood that it is true. This "accepted until demonstrated incorrect" methodology has evolved over hundreds of years, and has itself stood the test of time.

The same scientific method underlies algorithmic computer science [Tichy, 1998]. The algorithms that are regarded as significant are those that have been shown to work in practice, with evidence strong enough to convince skeptics. And because computing

X. Zhou et al. (Eds.): WISE 2004, LNCS 3306, pp. 1–12, 2004.

is a discipline in which innovations are valued as much for their economic merits as for their intrinsic elegance, new techniques tend to be regarded as of only curiosity value until they have been carefully evaluated in realistic settings.

After inventing a new indexing structure or query processing algorithm, we should, therefore, seek to implement it and measure its behavior. First, we form a hypothesis, that is, make a statement of belief about the algorithm, such as identifying what it is expected to be superior to and in what circumstances it is expected to be superior. Such hypotheses are often highly general. Rather than make claims about behavior on specific hardware, for example, we might claim that one algorithm is always faster than another for sufficiently large volumes of input data. Second, we design an experiment to distinguish between our hypothesis and previous ones. Third, we impartially carry out the experiment. The final step is to communicate the structure of the experiment, the outcomes of the experiment, and the conclusions we draw from those outcomes, usually as a written paper or report. Importantly, that description should allow an independent expert to undertake similar experiments and validate our claims.

In the long term, this scientific model is effective, and only the most successful algorithms are remembered and used. The work in individual computer science papers, however, is often remote from the ideal. In the specific areas of indexing and searching – used in this paper to illustrate the difficulties of experimentation – testing an algorithm often consists of implementing the simplest baseline that the investigators think is worth considering; implementing the "improved" algorithm; running both on some test data; and measuring the amount of CPU or elapsed time that was consumed. Using this evidence, the researchers draw both graphs and conclusions, often rather more of the former than the latter.

Such experiments almost always demonstrate improvements, and researchers rarely admit to having invented inferior techniques. In computing, reviewers tend to react negatively to papers that have as their rationale a further verification of the status quo, and an experimental refutation of the implicit hypothesis that "the new idea is better" is generally deemed to be not worthy of communication. As a result, researchers all too often construct experiments in which their new method is identified as a success.

Many research papers fail to earn any citations. A key reason, we believe, is that the evidence does not meet basic standards of rigor or persuasiveness, or is simply inadequate [Tichy et al., 1995]. Perhaps the experiments were flawed, the data inadequate, the baselines inappropriate, or, in extreme cases, the investigators deliberately chose to overlook the parts of the results that shed a bad light. Another issue is that, often, insufficient thought is given to the experimental design. In many cases it is far from clear how improvements in performance should be measured. Such failings mar many experimental papers [Johnson, 2002].

In this paper the notion of "experiment" is explored, with reference to the task of distributed text search on a tightly-coupled cluster of computers. Our intention is, as a case study, to explore the rigor that we believe is necessary in experimental computer science, in the hope that the lessons learnt in our experiments will be helpful to others.

It is also worth reflecting on what can happen when rigor is neglected. Physical scientists know well how costly careless experiments can be – thousands or millions of dollars worth of effort wasted, and possibly reputations destroyed. Consider for example the cold fusion saga, in which hundreds of scientists unsuccessfully attempted to reproduce work that had been announced without appropriate experimental validation. More

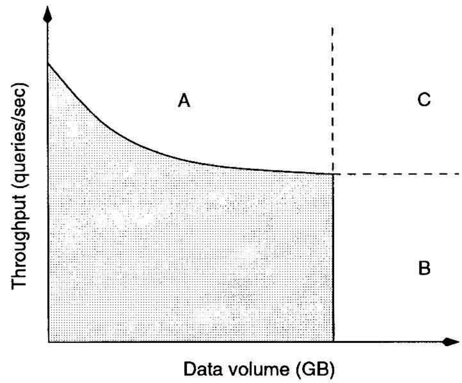

Fig. 1. The relationship between data volume and query throughput rate for a standard "unit cost" computer. Points in the shaded region are feasible; points above and to the right are not. The regions labelled A, B, and C are discussed below.

recently, a claimed link (based on flimsy and now disproven evidence) between autism and a common childhood vaccine has led to unnecessary suffering. Arguably the Y2K "bug", whose effects were largely unsubstantiated yet cost many billions of dollars, is an example of similar carelessness in computing. Competent scientists applying robust methods in an open manner are unlikely to cause such incidents.

2 Storing and Accessing Data

Consider Figure 1, which depicts a computer – with a processor, main memory, and disk storage – applied to a data management task. The horizontal axis shows schematically the size of problem that can be handled on this machine, with the vertical line crossing it representing the physical storage capacity of the given hardware. For example, on a machine with 1 TB of disk storage, we might (certainly with a straightforward implementation) suppose that problems involving more than 1 TB of data are not feasible.

The vertical axis in Figure 1 shows a processing rate – measured in queries per second – and is an independent measure of the system. The curve represents the peak processing load for that hardware combination. In most retrieval applications, the effort per query increases as the data volume increases, hence the downward trend.

The shaded region bounded by the two lines is the *feasible* zone. Combinations of data volume and query arrival rate that fall within the feasible zone are sustainable on the hardware, whereas combinations outside the region are not. In practice the exact positions of the lines depends on many factors, not all of which are easy to quantify:

- the processor speed and other characteristics, such as cache size and type;
- the amount of main memory available;
- the amount of disk storage available, and its access speed;
- the quality of the compiler;
- the skill of the programmer; and
- the fundamental nature of the algorithm and of the task that it supports.

When circumstances dictate a combination of data volume and query arrival rate that lies outside the feasible region, we must shift the bounding lines so that the feasible region includes the desired performance point. Reflecting the list above, this might be done by increasing the processor speed or cache speed or size; increasing the amount of main memory; increasing the disk capacity or speed; using a better compiler; improving the quality of the implementation; or by using a better algorithm.

The goal of research in areas of this type is almost always to achieve the last of these: we seek recognition by devising better techniques, and use our science to recognize circumstances in which adding more resources is unlikely to be helpful. In contrast, a supplier of commercial solutions more often than not simply scales the hardware.

Figure 1 is simplistic in several ways, not the least of which is the lack of scale on either axis. Another way in which it is potentially misleading is that the bounding curve is portrayed as a sharp line, whereas in practice there is a blurred transition between the feasible region and the infeasible region. When the query load is close to the maximum that can be supported, the system may begin to thrash or behave chaotically, and average query response time will increase markedly. In practice, systems are typically operated well below their peak capability, so as to meet a quality-of-service guarantee.

As an example of the type of improvements that can alter the feasible region, consider the role of compression. If we suppose that the data being manipulated can be stored compressed, then the data volume limit (the vertical boundary in Figure 1) shifts to the right, widening the feasible region. On the other hand, depending on the nature of the transformation used, and the type of processing required for each query, compression might slow query response rates rates, and lower the horizontal boundary of the feasible region. Note, however, that in some circumstances compression can boost query processing rates, thereby extending the feasible region in both dimensions [Zobel and Moffat, 1995, Williams and Zobel, 1999].

3 Distribution

Another way to extend feasibility is via distribution – by harnessing the power of multiple computers, we are able to tackle problems that are orders of magnitude larger than might be handled on a single machine. For example, much of the impressive performance achieved by the Google search engine arises as a consequence of the more than 10,000 computers used to support querying operations [Barroso et al., 2003].

Looking again at Figure 1, several distinct situations might occur. If the data volume fits the machine profile, but the anticipated query volume is too high, then we are in region A. To meet the required query load, the correct response is *replication* of the system, since a single computer is able to support the data volume, and, assuming that queries are read-only, multiple computers operating independently in parallel with mirrored data sets have no need for any synchronization overhead. In this scenario, a query is routed to a server by a *receptionist* process, and the server that resolves the query communicates the output directly back to the initial client.

On the other hand, in region B of Figure 1, one server could possibly carry the query load, but is unable to hold the necessary volume of data. Multiple computers are again required, but now it is the data that must be divided into manageable chunks, rather than the workload. Systems that partition the data are inherently more complex than replicated

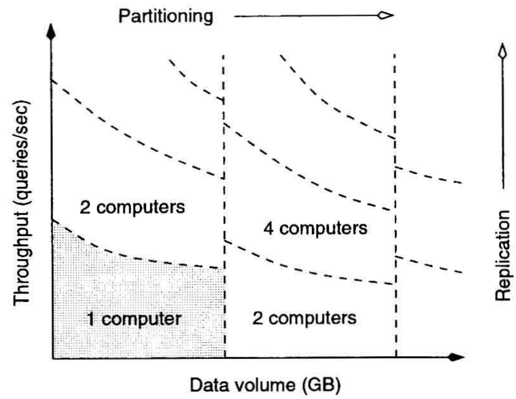

Fig. 2. Use of multiple unit machines to handle data volumes or query loads larger than can be handled by a single unit machine.

systems – the receptionist process must split each query according to the data distribution, invoke multiple servers on sub-queries, and then combine the partial answers that they return. There are two places in this arrangement where *drag*, or redundant computation, might be introduced. First, it might not be possible to perfectly split the query or data, making the sum of the effort involved in the sub-queries greater than the cost of executing the original query. Second, there may be non-trivial effort involved in combining the partial answers to make a global answer.

Finally, in region C of Figure 1, the data must be partitioned across several machines, and then replicated to obtain the required query processing capability.

One interesting question is whether it is more effective to double the volume of disk and memory on a given machine or to add another machine with the same memory and disk capacity. If the aim is to increase throughput, disk is unhelpful but memory could make a massive difference; if the aim is to increase the volume of data supported, further disk and memory can do so but at some cost in throughput and response time. If the aim is to fully explore the space depicted in Figure 1, all three resources must be added – hence our interest in distribution using idealized unit computers.

Figure 2 shows, again for a hypothetical unit computer of some fixed configuration, the zones that describe the resources required when data partitioning and replication must be combined. The shapes of the regions are a direct consequence of the factors discussed. Twice the number of machines used to manage twice the volume of data result is unlikely to result in the same query throughput rate being possible. On the other hand, a data split across multiple machines reduces the load on each, so total query load can, in theory, also increase – but not beyond what could be attained by placing a proportionate fraction of the data on a single machine.

For these various reasons, any algorithmic challenge lies in the area of partitioning rather than of replication. An immediate consequence is that experiments to test a distributed mechanism must be on a scale that warrants data partitioning. Taking any fixed volume of data and executing a software system using $k = 1, 2, 3, \ldots$ computers does not supply any inherent evidence as to the scalability of the technique in question. Such an experiment represents an implausible situation.

Given a new algorithm, then, the question arises as to how the benefit it offers might be measured. A typical experiment involves selecting some sample data sets and measuring, say, query processing times as the number of processors used is varied. But do changes in processing time tell us anything about the position of the feasible regions (Figure 2), or about potential throughput? And do the results have any lessons for other hardware, or other data, or other implementations? We explore such questions below, after reviewing issues and challenges for experimental design.

4 Designing an Experiment

The purpose of an experiment is to seek confirmation of a hypothesis. Having accepted that experimentation is essential for robust science [Tichy, 1998], there are many issues that an investigator needs to confront [Zobel et al., 1996, Zobel, 2004, chapter 11]. The following paragraphs summarize a set of minimum standards that a skeptical reader (and a good reviewer) will apply before accepting the validity of any experiments.

Baselines. There needs to be a clear, interesting hypothesis. It isn't particularly meaningful, for example, to claim that an algorithm is "fast" or "efficient". Efficient compared to what? Such a claim implies the existence of a baseline – a method against which the new algorithm is being compared. It follows that the baseline should be the best previous method for computing the same task. Comparison against a poor baseline makes little sense, since there are many different ways of gaining initial improvements. And is the new method faster on all kinds of data, on all machines, at all scales?

More typically, it is the trend of behavior that is relevant. A technique is most interesting if, as the data volume increases, the improvement offered by the new algorithm increases by a greater fraction.

The use of baselines presents other challenges. If the implementations are not of similar standard, then the results are meaningless; yet, during a research program, investigators tend to focus on developing their contribution and not on reimplementing the work of others. In some cases, other investigators have made their code available – allowing results directly comparable to those previously published. (Note that if you do use the source code or data of others, you should both cite the paper in which the techniques are described and also as a separate note indicate the source and authorship of the software you have made use of. The two works may involve quite different sets of people, and both sets should have their work acknowledged.)

An experiment might also be used to show that an algorithm provides a feasible method for solving a problem, for example when expected asymptotic costs have been reduced from quadratic to linear. In such cases, a baseline might be of less interest, and absolute performance sufficient to defend the hypothesis. Nevertheless, measuring performance across a range of scales of data is crucial. Performance at a single data point is (singularly) uninformative.

Data and software. Measurement of performance requires use of input data. It is important for sensible data to be used, and explained carefully. If the data is artificial in some way, the justification for using it needs to be very careful indeed – results on synthetic data rarely generalize to realistic cases.

Once you have finished with your data and software, it is good practice to make them publicly available. By all means add any caveats that you wish – "may be used for research purposes only", "no warranty", and so on – but do allow them to be used by your colleagues. The field that you are a member of will benefit, and your work is more likely to be accurately recognized. Being overly protective of your intellectual property hints that there are limitations in your code or data that would undermine or contradict your published results.

In some fields, there are data sets that have become so widely used that they have the status of "gold standards". Even if you wish to give results for, and then make public, more specific data sets that show the behavior you wish to comment on, you should also give comparative values for previous data sets that have been widely quoted.

Occasionally an investigator argues that, because the cost of undertaking a full-scale experiment is prohibitive, a limited or artificial experiment should be used. Perhaps two computers running multiple processes are assumed to resemble a grid; or a gigabyte of data is used where a terabyte would be the likely volume in practice; or querying is simulated by random accesses into an index; or documents are simulated by random strings of fixed length; or queries are randomly-sampled strings from documents. Doing so is perfectly reasonable, but only if the conclusions make it clear that the results are no more than preliminary and may have no implications for realistic implementations.

Any extrapolations from results on a limited data set are likely to be flawed. For example, search amongst a few thousand documents does not present the challenges of search across the web. Similarly, performance on a data set that fits into a CPU cache has no implications for performance on a memory-sized data set. Nor does performance on data that fits in memory have any lessons for larger data sets on disk.

Measurement. Another important issue is deciding what to measure. Candidates include – but are not limited to – response time, CPU time, query throughput, memory usage, number of simultaneous users, network activity, disk volume, and, in the case of text search, effectiveness as indicated by the ability to retrieve relevant documents.

These various facets are often in tension, and can be traded against each other. Giving an overall feel as to the viability of a technique is just as useful as giving precise results in one axis of measurement. For example, a new data compression technique that obtains improved compression effectiveness may be of considerable interest, but the excitement would be greatly tempered if the computational costs and memory requirements mean that in practice it can only be applied to small files.

Measurement is rarely straightforward. Even simple cases present issues, such as startup costs and questions such as whether it makes sense to average costs over widely varying inputs. A common error is to report reductions in size and gains in speed, but to fail to note that they could not be achieved at the same time.

Another common error is to introduce a large number of variables, then fail to determine which are responsible for the observed performance. Fixing key variables while exploring others is essential for thorough analysis of behavior. However, be aware that variables may be correlated with each other; for example, considering the example in the previous section, fixing the size of the data set while varying the number of machines may lead to meaningless results.

A key issue is that any training must be separated from testing. It is enticing to play around with your implementation until you are sure it is properly tuned, and then report a "best" run and claim it as evidence of success; but in a production environment such settings might be impossible to determine. In contrast, if tuning or training on one set of data leads to excellent performance on another, it is clear that the results are indeed strong. Even better is if a wide range of training data gives rise to consistent and stable parameter settings. That is, the more independent cross-checks that can be performed, the more robust the claims. Conversely, results from a single data set should be treated with suspicion – and if there is any doubt about replicability, statistical tests should be applied to the output of multiple independent experiments, and confidence estimates reported to lend credence to claimed relativities.

Finally, it is perhaps worth commenting that it can also be valuable to carry out experimentation on a tractable rather than industrial scale, or using assessment methodologies that have known flaws but still provide some useful guidance. Not everyone has the resources to operate – referring to our distributed text querying case study – a network of dozens of computers and terabytes of data.

Reporting. A final aspect of measurement is considering what to report. For example, what units will convey the most general information? How might the numbers be standardized across different data sets to eliminate unimportant information? And are they presented as graphs, or tables, or both? This aspect of experimental design is beyond the scope of this paper, and the reader is instead referred to the advice given by, for example, Johnson [2002], and Zobel [2004].

5 Distributed Text Querying

We now return to our case study: distributed text retrieval. We suppose that there are k processors, that the data is being split k ways, and that the data is sufficiently voluminous that a k-way split is sensible on the target hardware.

In a text retrieval system, queries are sets of words or phrases, and the records being searched are unstructured. For example, in a web search engine, a large number of web pages are indexed, and users supply queries typically containing just a few words. To answer a query, the retrieval system identifies the r documents in the collection that are assigned the highest scores by a similarity heuristic. In web applications $r = 10$ or perhaps 100, and in other document search environments $r = 1{,}000$ is a reasonable upper limit. In both situations, the system is implemented using an *inverted index*, consisting of a *vocabulary*, and a set of *inverted lists* [Witten et al., 1999].

Two different types of data partitioning have been proposed for text searching. In a *document-partitioned* index, each server contains an index for a fraction of the documents, and can answer queries for that subset of the documents. Queries for the collection as a whole are only answered when all servers have computed their top r answers, at which time the top r of those rk answers can be identified by the receptionist. Document-partitioned systems have been described by, amongst others, Harman et al. [1991], Tomasic and García-Molina [1993], and [Cahoon et al., 2000]. Google uses both document-partitioning and replication [Barroso et al., 2003].

In an *index-partitioned* index, each server contains the full index information for a subset of the terms. Queries for the collection as a whole can only be answered when all servers that store information for query terms have returned the corresponding inverted lists to the receptionist. Index-partitioned indexing has been considered by, among others, Jeong and Omiecinski [1995], Ribeiro-Neto and Barbosa [1998], and Badue et al. [2001]. In effect, an index-partitioned system uses the additional computers as data stores only,

Table 1. Comparison of costs associated with document-partitioned text retrieval and index-partitioned text retrieval, when a query of q terms is processed with a k-way data partition, to determine a ranked list of r answers. Note that q is assumed to be less than k. Quantity I is the sum of the lengths of the inverted lists for the query terms, counted in pointers.

Performance indicator	Monolithic system	Index partitioned	Document partitioned
Number of servers active on query	1	q	k
Per processor			
Disk seeks and transfers	q	1	q
Index volume transferred from disk	I	I/q	I/k
Number of documents scored	r	0	r
Plus			
Computation load at receptionist	n/a	$I + r$	kr
Network volume	n/a	I	kr
Total cost	$I + q + r$	$I + q + r$	$I + kq + kr$

rather than as auxiliary computation devices. In the event of two terms being stored on the same machine, local operations can take place, but for large text searching systems the usual relationship is $k \gg q$, and only one term per processor is active in any given query.

Each of the alternatives has disadvantages. In a document-partitioned system, each query is executed in full on each of the machines, including the disk seeks to retrieve inverted lists. Over the k processors, a query of q terms involves kq seeks and data transfers. Moreover, if the top-ranking r documents for the whole collection are to be correctly determined, each of the subcollections must also identify its top r documents. In total, more computation is performed than in a monolithic system. Conversely, in an index-partitioned system, all of the computation is undertaken on the receptionist, which is then a potential bottleneck.

Table 1 summarizes the relative performance of document- and index-partitioned systems, and compares them to a non-partitioned system (assuming it to be feasible) for the same volume of data. In a non-partitioned system, there is one seek per query term, and a total computation load dominated by the processing of I pointers, and then extracting the top r answers. Note that, in practice, I is directly proportional to the volume of indexed data.

An interesting aspect of performance not directly captured in the table is elapsed time per query, which is best approximated as the sum of the per processor section of the table, plus the receptionist cost. In the case of the index-partitioned system, the sum is proportional to $I + q + r$ still, as the receptionist must collate all inverted lists to determine the final answer. On the other hand, for a document-partitioned system, the elapsed sum time is proportional to $I/k + q + kr$, and it is clear that the elapsed time will be less, since $r \ll I$ for a typical query.

Table 1 suggests that index-partitioning is more *scalable* than document-partitioning, because the per-query cost using the latter is dependent on k. In this argument, as the data is split over more and more machines, the drag associated with document-partitioning becomes greater, and index-partitioning is more efficient.