Modern Client–Server DBMS Architectures*

Nick Roussopoulos and **Alexis Delis**

Department of Computer Science
University of Maryland
College Park, MD 20742

Abstract

In this paper, we describe three Client–Server DBMS architectures. We discuss their functional components and provide an overview of their performance characteristics.

1 Introduction

In the eighties, a lot of research was geared towards the design and realization of database machines. Special purpose, expensive hardware and software were designed and attempted to provide high transaction throughput rates utilizing parallel processing and accessing of multiple disks. However, in a world of continuously evolving hardware technology, designing and prototyping such systems proved to be extremely difficult.

In recent years, we have observed different trends. Research and technology in local area networks have matured, and workstations have become very fast and inexpensive while data volume requirements continue to grow rapidly [1]. In light of these developments, computer systems—and DBMSs in particular—in order to overcome long latencies have adopted software client–server architectures to improve their performance. These software solutions are not only less costly, but also much more durable as the hardware evolves.

Client–Server architectures originated in engineering application environments where data is mostly processed in powerful workstations with CAD/CAM or other special purpose software, while centralized repositories with check in and check out protocols are predominantly used for maintaining data consistency. However, many more applications demand exactly the same or very similar access and process patterns, including those dealing with large scientific databases and archival databases.

The following are the basic database capabilities needed today:

1. *data staging*: The volume of operational databases keeps growing and, in several applications, has already passed well beyond the effective range of query optimization and processing. Therefore, data caching mechanisms are necessary for extracting chunks from these "massive" databases and using them in a user–tailored operational region.
2. *database interoperability*: Multiple already deployed databases must be made accessible through a cooperative envi-

*This research was sponsored partially by the National Science Foundation under Grant IRI–8719458, by the Air Force Office of Scientific Research Grant AFOSR–89–0303, and by the University of Maryland Institute for Advanced Computer Studies (UMIACS).

ronment for exchanging data, bindings, and control on a continuous basis. This will allow invaluable data correlation and smooth update propagation.

The purpose of this short paper is to present three Client-Server DBMS architectures and provide an overview of their performance characteristics. The three DBMS configurations are: the standard Client–Server (CS) [3, 10], the RAD–Unify type of DBMS (RU)[8], and the ADMS± Enhanced Workstation–Server architecture (EWS)[5, 7]. These three Client–Server architectures cover the whole spectrum of query processing and data distribution; from no query processing on the clients and no data distribution (CS), to only some query processing on the clients but still no data distribution (RU), all the way to most of the query processing on the clients with almost all page access from downloaded and cached data (EWS). [1]

In order to evaluate the performance characteristics of these three architectures under a wide variety of parameters, we developed three closed queuing network models and implemented a simulation package for each of the architectures. Our experiments included several different workloads expressed in the context of continuous transaction streams[2] in which we varied query-update mix and the selectivity of update transactions from 0%–8% of the tuples of the base relations. We measured several key parameters including I/O and query processing distribution, parallelism achieved, network overhead, and total transaction throughput. The last one captures the real power of each architecture because it combines all other measurements. Finally, we measured the scale up behavior of each architecture as the number of workstations increased.

In section 2, we describe the three Client-Server DBMS architectures and identify their specific functional components. In section 3, we discuss the method of evaluation and their performance. Conclusions can be found in section 4.

2 Modern Client–Server DBMS Architectures

Although the architectures are general enough to be applicable for many data models, they are described here for the relational model only.

2.1 The Standard Client–Server Architecture (CS)

In the CS architecture [3, 10], each client runs an application on a workstation but does database access from the server. This process is depicted in Figure 1(a). Only one client is shown for the sake of presentation. The communication between server and clients is done through remote calls over a local area network (LAN). Applications processing is carried out at the client sites, leaving the server free to carry out database work only. The same model is also applicable for a single machine in which one process runs the server and others run the clients. This configuration avoids the transmission of data over the network but obviously puts the burden on the system's load. In this paper, we assume that server and client processes run on different hardware.

Despite the distributed fashion of the applications, this architecture retains most of the main resource bottlenecks of a centralized DBMS, e.g., blocking, I/O, and CPU. Furthermore, application programs, despite the fact that they are distributed, are totally dependent on a typically overloaded process.

[1] Note that although a Client–Server architecture can distribute both query processing and data, it is not a distributed DBMS architecture because the server acts as the primary data and control site.

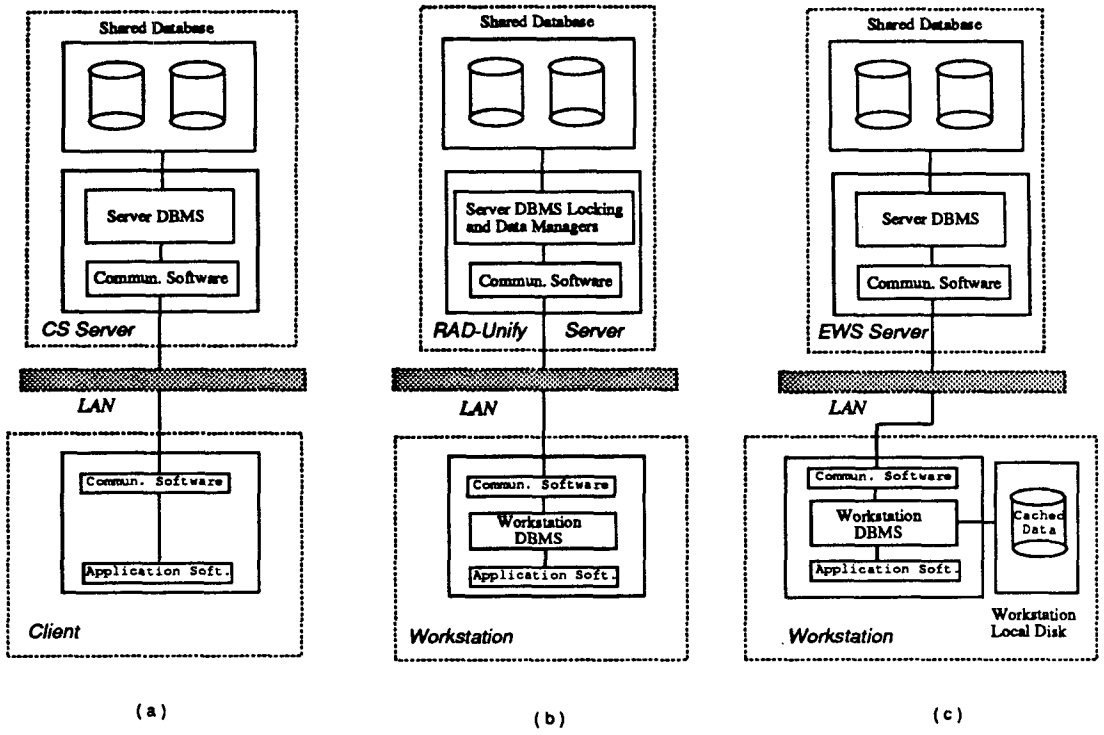

Figure 1: CS, RU, and EWS architectures

2.2 RAD–Unify Type of DBMS Architecture(RU)

The broad availability of high speed networks and fast processing diskless workstations were the principal reasons for the introduction of the RAD–Unify type of DBMS architecture(RU) presented by Rubinstein et al. in [8]. The main objective of this configuration is to improve the response time by utilizing both the workstation processing capability and its virtual memory. This architecture is depicted in Figure 1(b). The role of the Server DBMS is to execute low level operations such as locking and page reads/writes. The workstation performs query optimization, i.e., for each query it selects the plan of execution, and then carries it out. As data pages are needed during the query processing, they are requested from the server. The server's data manager retrieves and sends data pages to the diskless workstation virtual memory for carrying out the rest of the processing.

This architecture was shown to perform well for mostly look–up operations, much better than the traditional client server configurations [8]. However, the authors acknowledge that the architecture may perform well only under the assumption of light update load. More specifically in the system's prototype, it is required that only one server database writer is permitted at a time. The novel feature of this architecture is that unlike distributed systems it does not have to support a front end/back split at the query level, and that the transfer of pages to the appropriate workstation memory improves response time at least in the case of small to medium–sized retrieval operations.

Although this architecture extends distribution with query and application processing performed on the workstations, it is still very dependent on the server's I/O band-

width and the load of the data manager. In other words, it does not improve the biggest bottleneck in DBMSs. Furthermore, it decreases the autonomy of the workstations which have to make a huge number of I/O requests to the server, while at the same time causing a much higher network load than the CS architecture.

2.3 The ADMS± Extended Workstation Server Architecture (EWS)

The ADMS± Extended Workstation Server Architecture (EWS) distributes most of the query processing and the I/O load to the workstation clients by incorporating in each workstation a full–fledged DBMS which cooperates with the server DBMS, and by downloading and caching query results in the workstations' databases. [5, 7]. The workstation and the server DBMSs process queries in synergy utilizing as much as possible downloaded and cached data which is maintained using incremental access methods [4]. Figure 1(c) depicts the EWS architecture.

Initially, the workstations start off with an empty database. Caching query results over time permits a user to create a *local* database subset which is pertinent to the user's application. Furthermore, a user can integrate into her/his local database *private data* for added-value. The caching of query results offers two major advantages: it eliminates requests for the same data from the server and boosts overall performance because workstation DBMSs access in parallel local data copies. In the presence of updates, the system needs to ensure that, before a workstation DBMS uses one or more cached results derived from some global relation(s), relevant updated tuples of these relations are propagated and incorporated into the locally cached results.

Updates are transmitted to the server before they are performed on the local copies Thus, the server acts as the the *primary dat site.* Logs are utilized as the recording mech anism of committed updates. Every serve relation is associated with an update log an every tuple in it is timestamped with a *up date operation number.* During a query o an update request from a workstation, th server, along with the query result or incre mental update of one, also returns to th workstation DBMS the last update times tamp for each of the deriving relations. Thi information is stored on the workstation and relieves the server from keeping trac of the timestamps whose number can be ar bitrarily large; there can be a large numbe of cached data subsets with independent up date timestamps which multiply very quickl with the number of workstations.

Query transactions involving server relation are transmitted to and processed by th server. When the result of a query is cache into a local relation for the first time, thi local relation gets *bound* to the server rela tions used in extracting the result. Quer processing against bound data is precede by a request for an incremental update c the cached data. The server is required t look up the portion of the log that maintain timestamps greater than the one seen by th submitting workstation. Only relevant frac tions (*increments*) of the modifications ar propagated to the workstation's site. The a gorithms that carry out these tasks are base on the *Incremental Access Methods* for rela tional operators described in [4] and involv looking at the update logs of the server an transmitting *differential files* [9]. This sig nificantly reduces data transmission over th network as it only transmits the incremen affecting the bound object, unlike the trad tional CS architectures, in which query r sults are continuously transmitted in the entirety.

It is important to point out some of the characteristics of the concurrency control mechanism assumed. First, since in each of these three architectures, updates are done on the server, a 2–phase locking protocol is assumed to be run by the server DBMS. Second, we assume that the update logs on the servers are not locked and, therefore, the server can process multiple concurrent workstation requests for incremental updates.

There are two major advantages of the EWS architecture over the other two architectures. The first is its ability to perform concurrent and parallel access of cached data from multiple disks of workstations. This brings a significant reduction of the load on the server I/O manager and the blocking factor. The net result is much higher transaction throughput. The second advantage is the increase of the autonomy of the workstations. Lazy or periodic update propagation even allows the workstations to disconnect themselves temporarily, and work with cached data; even when the server is down. Furthermore, the incremental (lazy or periodic) update strategy avoids the enormous network traffic and workstation data filtering caused by broadcasting, i.e., transmission of all (relevant and irrelevant) updates to all workstations.

3 Simulation Experiments

In this section, we give a small set of simulation experiments comparing the three architectures. The main performance criteria for our evaluation are the average transaction throughput and extensibility. A more thorough evaluation of the three architectures can be found in [2].

The database for our experiments consists of eight base relations. Every relation has cardinality of 10000 and maintains two indexes (one is clustered). The database is resident on the server's disk. Before running the experiments in the case of the EWS architecture, we let the system run for some initial period so that all the workstations cache the data of their interest (locale). Once this initial phase has ended, identical query streams are submitted to each of the architectures. The average page disk access time is set to 15msec for the server and 20msec for the clients. The server CPU rate is set to 11MIPS and to 20MIPS for the workstations.

A Query–Update Stream(QUS) is a sequence of query and update transactions mixed in a predefined ratio. In this paper's experiments, the mix ratio is 1:10 that is each QUS includes one update for every ten query transactions. QUS transactions are randomly selected from a predetermined set of operations which characterizes the workload. Every client/workstation submits for execution its QUS (one transaction at a time) and terminates when all its transactions have completed successfully.

The used QUSs for the experiment are made up by the following transactions: eight selections with tuple selectivity of 5% (one per relation–two of them are done on clustered attributes), four 2–way join operations with join selectivity 0.4, and finally eight modifications to the base relations with varying update selectivity rates (two use clustered attributes). The update selectivity rates give the percentage of tuples changed in a relation during an update transaction. For our experiments, update selectivity rates are set to the following values: 0% (no modifications), 2%, 4% and 8%. Other experiments designed around the Wisconsin benchmark can be found in [5].

In Figure 2, the throughput rates for both CS and RU architectures are presented. The number of clients varies from 4 to 56. There are clearly two groups of curves: those of the

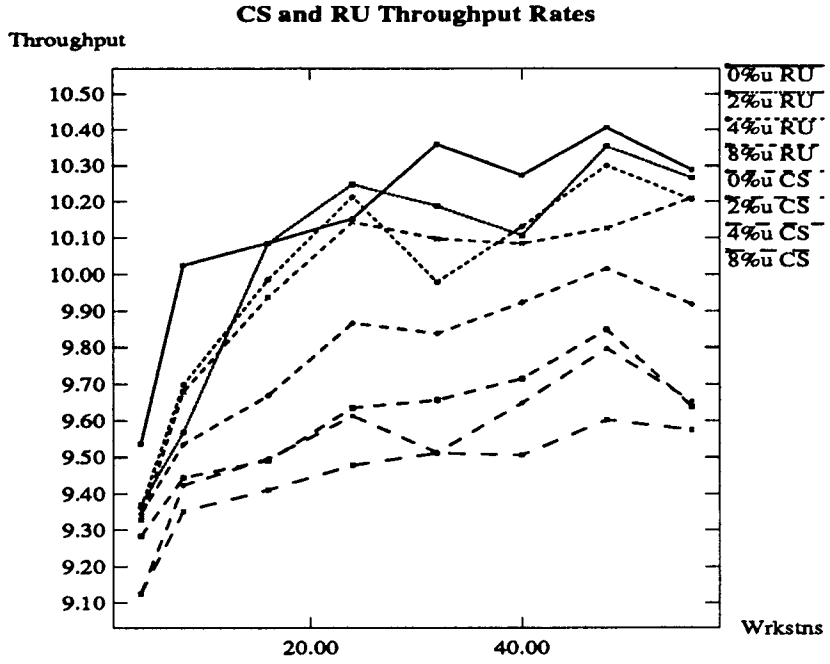

Figure 2: CS and RU Throughput

CS (located in the lower part of the chart) and those of the RU (located in the upper part of the chart). In general, we could say that the throughput averages at about 9.5 transactions per minute for the CS and 10.1 for the RU case. From 4 to 24 clients, throughput increases as the configurations capitalize upon their resources. It is evident from the figure that since RU utilizes the CPU of its workstations for DBMS page processing, it performs slightly better than its CS counterpart. The average RU throughput improvement for this experiment is 5.4% higher than that of CS. For more than 48 clients, we observe a throughput decline attributed to high contention and the higher number of aborted and restarted transactions.

Figure 3 shows the EWS throughput for the very same streams. The 0% update selectivity curve shows that the throughput of the system increases almost linearly with the number of workstations. As the update selectivity rate increases (2%, 4%) the level o the achieved throughput rates remains higl and increases almost linearly with the num ber of workstations. This holds up to 32 sta tions. After that, we observe a small declin that is due to higher conflict rate caused b the increased number of updates. Similar de clines are observed by all others (except th 0% update curve). From Figures 2 and 3 we see that the performance of EWS is sig nificantly higher than those of CS and RU For EWS the maximum transaction process ing rate is 392 jobs/min (all 56 workstation attached to a single server with 0% updates while the maximum throughput value of th CS is about 10.01 jobs/min and of the RI 10.42 jobs/min. This increased performanc is attributed to three reasons: 1) the client use their local disks and achieve maxima parallel access to already cached data, 2) th server carries out less disk operations (onl updates) and handles only short message and acknowledgments routed through th network, and 3) significantly less data mov

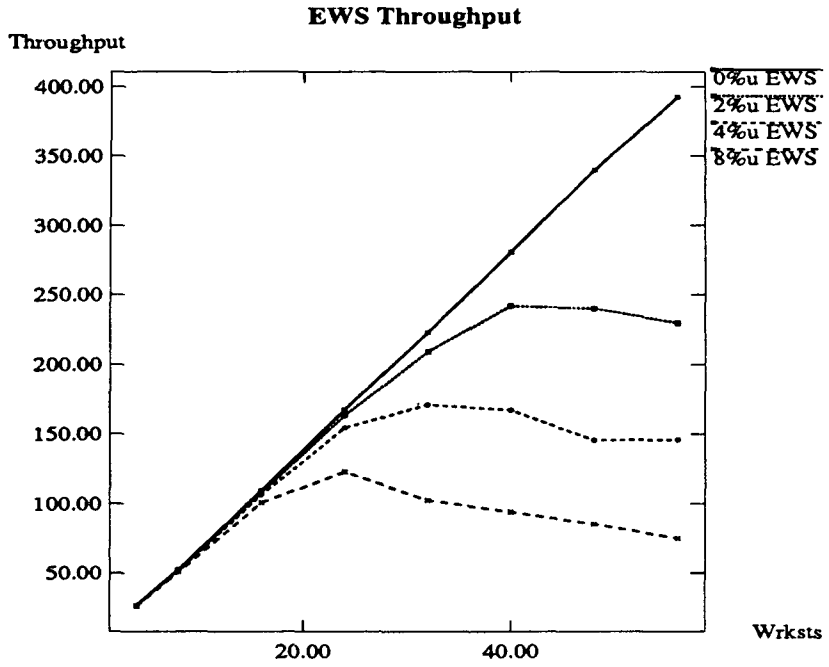

Figure 3: EWS Throughput

ment across the network is observed. The number of workstations after which EWS observes a decline in transaction throughput is termed "maximum throughput threshold" (*mtt*) and varies with the update rate. For instance, for the 2% curve it comes to about 40 workstations and for the 4% and 8% curves about 24 workstations. We believe that *mtt* greatly depends on the server concurrency control manager, and a more sophisticated manager (such as that in [11]) than the one used in our simulation package would further increase the *mtt* values.

Figure 4 depicts the throughput speedup of RU and EWS architectures over CS (y axis is depicted in logarithmic scale). At the lower part of Figure 4, the RU over CS speedup is shown. Although RU performs better than CS under the current workload for the reasons given earlier, its corresponding speedup is bearly noticeable. Thus, only the 0% update selectivity rate experiment is shown in Figure 4. The rest of the curves map very closely to the 0% curve. The speedup for EWS is almost proportional to the number of workstations at least for the light update streams (0%, 2%, and 4%) in the range of 4 to 32 stations. For the 4% update stream, the relative throughput for EWS is 15 times better than its CS–RU counterparts (at 56 workstations). It is worth noting that even for the worst case (8% updates), the EWS system performance remains 7 times higher than that of CS.

We were also interested in the scale–up behavior of all architectures under the presence of a large number of workstations. For this purpose, we ran experiments where the number of clients/workstations ranged from 4 to 200 per server. Figure 5 depicts the resulting curves for the CS and RU experiments and indicates that both curve groups (for CS and RU) beyond 32 clients give approximately stable throughput rates as it is expected. Figure 6 shows the curves for the EWS experiment. The graph indicates that

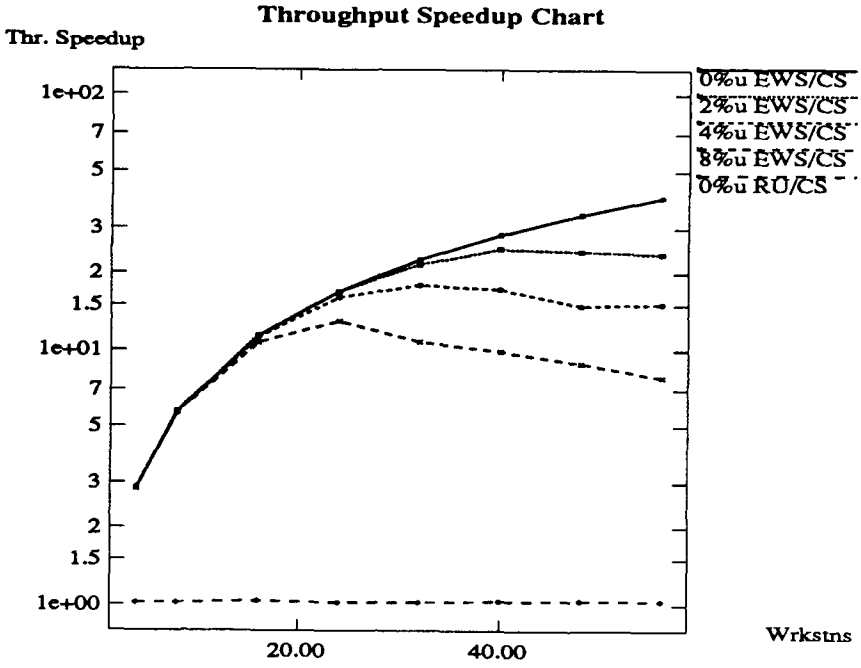

Figure 4: EWS/CS and RU/CS Throughput Speedup

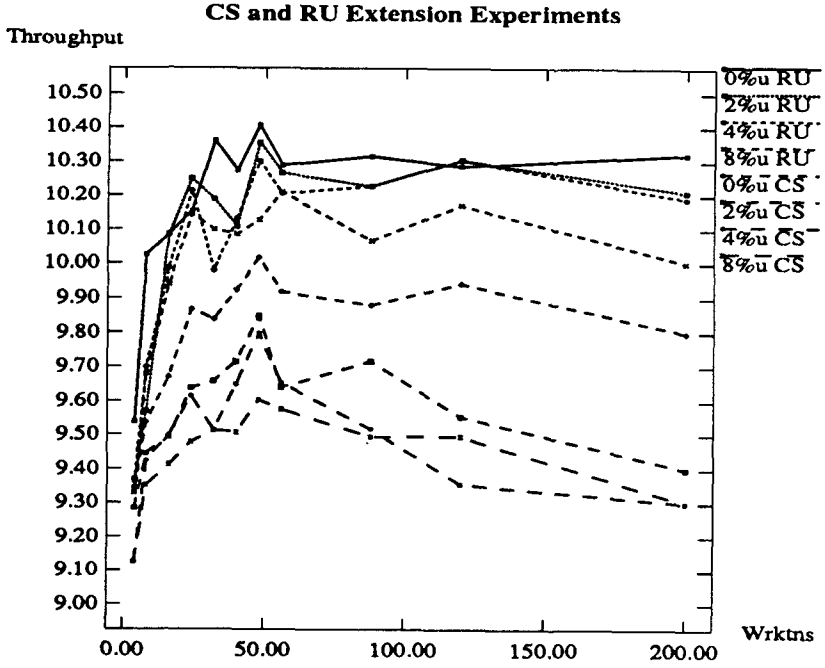

Figure 5: CS and RU Extensibility Experiment

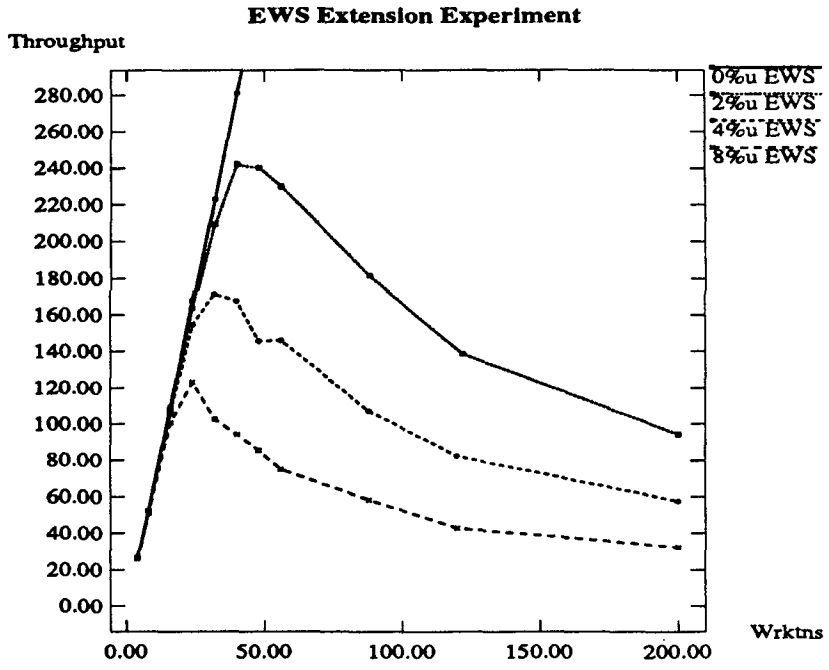

Figure 6: EWS Extensibility Experiment

beyond the *mtts* region the throughput of the EWS is almost "stabilized" (beyond 100 workstations) regardless of the update intensity. This stability is due to the saturation of the server's concurrency control manager. Note however, that even after the saturation point, the EWS architecture can process many more jobs than the CS and RU architectures. The 0% update curve shows no saturation and continues to increase almost linearly.

4 Summary

The simulation experiments comparing the three Client–Server architectures showed that the RAD–Unify performs almost always slightly better than the standard Client–Server architecture. However, the improvement is rather negligible because the experiments that we ran—continuous streams of relational queries and updates, were mostly I/O bound. On the other hand, the ADMS± Enhanced Workstation–Server architecture offers performance which is at least an order of magnitude higher than that of RAD–Unify architecture. Most of this improvement is attributed to the parallel access of the staged data cached on the local disks.

It should be pointed out that a limited number of our simulations for the ADMS± EWS architecture were verified by real experimental runs on our prototype [6]. These runs covered the lower end of our graphs, up to 10 workstations, and displayed behavior identical to the simulations. These experimental runs are reassuring about the correctness of the assumptions of the simulation models.

5 Acknowledgement

The authors would like to thank Timos Sellis for the many useful discussions and suggestions.

References

[1] R. Alonso, D. Barbara, and H. Garcia-

Molina. Data Caching Issues in an Information Retrieval System. *ACM–Transactions on Database Systems*, 15(3):359–384, September 1990.

[2] A. Delis and N. Roussopoulos. Performance Comparison of Three Modern DBMS Architectures. Technical report, University of Maryland, College Park, MD, 20742, May 1991. CS–TR–2679, UMIACS–TR–91–75.

[3] D. DeWitt, D. Maier, P. Futtersack, and F. Velez. A Study of Three Alternative Workstation–Server Architectures for Object–Oriented Database Systems. In *Proceedings of the 16th Very Large Data Bases Conference*, pages 107–121, 1990.

[4] N. Roussopoulos. An Incremental Access Method for Viewcache: Concept, Algorithms, and Cost Functions. *ACM–Transactions on Database Systems*, 1991. To appear.

[5] N. Roussopoulos and A. Delis. Evaluation of an Enhanced Workstation–Server DBMS Architecture. Technical report, University of Maryland, College Park, March 1991. UMIACS–TR–91–30.

[6] N. Roussopoulos, N. Economou, and A. Stamenas. ADMS: A Testbed for Incremental Access Methods. Technical report, University of Maryland, College Park, July 1990. UMIACS-TR-90-103, CS-TR-2514.

[7] N. Roussopoulos and H. Kang. Principles and Techniques in the Design of ADMS±. *Computer*, December 1986.

[8] W. Rubenstein, M. Kubicar, and R. Cattell. Benchmarking simple database operations. In *ACM–SIGMOD–Conference on the Management of Data*, pages 387–394, 1987.

[9] D. G. Severance and G. M. Lohman. Differential Files: Their Application to the Maintenance of Large Databases. *ACM–Transactions on Database Systems*, 1(3), September 1976.

[10] M. Stonebraker. Architectures of Future Data Base Systems. *IEEE–Data Engineering*, 13(4), December 1990.

[11] K. Wilkinson and M.A. Neimat. Maintaining Consistency of Client–Cached Data. In *Proceedings of International Conference on Very large Data Bases*, August 1990.