POEMS: A Transformable Architecture for Managing System Overload

Wee Siong Ng[1], Panos Kalnis[2], Beng Chin Ooi[1,2], Kian-Lee Tan[1,2]

[1]Singapore-MIT Alliance
National University of Singapore
4 Engineering Drive 3, Singapore 117576

smangws@nus.edu.sg

[2]Department of Computer Science
National University of Singapore
3, Science Drive 2, Singapore 117543

{kalnis, ooibc, tankl}@comp.nus.edu.sg

ABSTRACT

In a typical organizational scenario, hundreds of personal computers (PCs) are used mainly for simple office tasks. Typically, a central database management system (DBMS) receives requests internally or externally through an Internet connection that serves as a backend of Web services. The unpredictability and fluctuations of requests could result in the overload of the DBMS. Existing load management systems assume nodes are fully dedicated to sharing loads, which could cause interruptions of the existing tasks running on the office PCs. In addition, data are statically partitioned and cached in the nodes permanently even though the system may be under-loaded. Moreover, the nodes that are involved in load-balancing are not allowed to dismiss the processes. In this paper, we describe a novel framework, POEMS, that is transformable between a client-server and Peer-to-Peer (P2P) architecture; it operates as a conventional DBMS under normal load condition without interrupting the nodes, and transforms to P2P operation mode for processing in a heavy-load condition. In contrast to traditional systems, all nodes in our proposed framework contribute the spare capacity of their hardware resources for data manipulation, and this is done without the need to install any DBMS at any of the nodes. Data are partitioned online and operators are distributed to nodes similarly. The effectiveness of POEMS query processing is achieved by node cooperation. POEMS allows processes or operators to be dismissed online, so a user can allocate more resources to his/her own operation tasks as and when the need arises. We evaluate the performance of our proposed system with a prototype implementation. The results suggest that POEMS is a feasible and effective approach for solving the system overload problem.

1. INTRODUCTION

Many organizations and the research community are heading, nowadays, toward distributed database technology. One of the major motivations behind the use of distributed computing is the desire to provide an economical method of harnessing more computing power by employing multiple processing elements. Significant advancements have taken place in the development and deployment of the distributed data management system (DDMS). These include mechanisms to provide transparency in accessing data from multiple servers [9, 24], and the support of distributed transactions which facilitate transparency and can execute queries over fragmented and heterogeneous data sources [13, 19].

In this paper, we investigate a common and practical problem: Imagine a typical business environment where a medium-size organization operates with hundreds of office personal computers (PCs) and a central DBMS. Most of the office-PCs are used mainly for simple word-processing and emailing, or as dumb terminals with no DBMS facilities installed at all. The central DBMS receives requests internally or externally through an Internet connection (e.g., it might serve as a backend of Web services). The unpredictability and fluctuations of the requests may overload the DBMS. In conventional DDMS approaches, overloading might be tackled with the introduction of additional database servers to handle the extra load. However, besides the additional costs, such approaches might not be flexible. This is especially true if the arrival rate of requests causing system overload is short and the fluctuations of requests are swift. In such a case, the server might end up under-loaded (and hence under-utilized) most of the time. The motivation of our work is that we can exploit the spare capacity of the hardware resources of the office PCs, in order to assist the DBMS to maintain its performance under fluctuating overload conditions. The intuition is to distribute some of the DBMS load to the PCs during the peak periods, while not interrupting them when the DBMS's workload is normal.

Peer-to-Peer (P2P) technology provides an attractive alternative for building distributed systems. The P2P environment is dynamic and sometimes *ad hoc*. Peers are allowed to join the network at any point of time and may leave at will. This results in an evolving architecture where each peer is fully autonomous. With such a dynamic environment, maintaining inter-operability among peers is a great challenge. In addition, finding ways to cope with databases that are incomplete, overlapping and mutually inconsistent is an exciting challenge. A considerable amount of research has been conducted on data integration, data mapping, data discovering and query processing in P2P environments. In [15], mapping tables are proposed for data integration in the P2P environment. The technique provides domain relation management by inferring new mapping tables and determining the consistency of mapping constraints. [11, 12] offer a schema mapping mechanism to capture the structures and terminologies between a source schema and a target schema. [17, 18] tackle the issues of data discovery and processing via an IR technique which allows SQL queries to be processed in a P2P environment without relying on a global schema.

These systems assume that all participating peers have a DBMS installed and are willing to share their data with other peers.

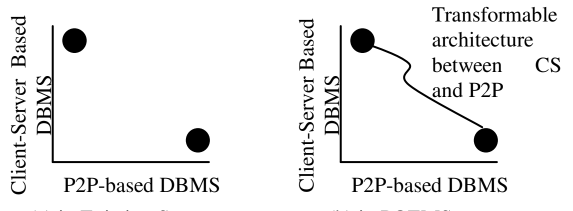
Moreover, each peer is expected to play the role of data provider and data consumer. DBMS facilities such as language parser, indexes, query optimizer and various operators are granted at will. We refer to this architecture as peer-based DBMS (PDBMS).

In contrast to DDMS, PDBMS offers a more cost effective solution where each of the peers in a PDBMS network has a DBMS installed and load or data is distributed among the peers in order to maintain system performance. The main concern in this case is not the additional costs, since the price of the peers (office PCs) are much cheaper than a server, but rather the complexity of maintaining the DBMS at each peer site. This involves several difficulties especially for novice users. First, a complex DBMS would affect the ad hoc tasks running on a typical PC, unless the PC is fully dedicated to load balancing purposes. Second, there is the issue of the complexity of interconnectivity and integration between one DBMS and others. Third, introducing new operators or functions would involve the upgrade of all the participants. Furthermore, both DDMS and PDBMS suffer from the drawback of not having the capability to handle dynamically changing user requests. This is due to the initial data placement that might not guarantee good load balancing for different kinds of access patterns in the future.

1.1 Our Proposal

Most existing systems employ either a client-server based or P2P-based architecture (Figure 1.1(a)), each having its pros and cons. A client-server architecture provides easy access to resources, and data control through a server; it also allows for new technology to be easily integrated into the system. These features are absent in P2P systems. A P2P architecture, on the other hand, provides scalability in harnessing processing power for solving a given task.

Here we describe POEMS (*P*eer-based *O*v*E*rload *M*anagement *S*ystem), a novel approach which handles the above-mentioned problems and inherits the advantages of both client-server and P2P systems. We propose an autonomic DBMS architecture with two operation modes: a centralized and a P2P mode. Under normal conditions, the DBMS operates in a client-server mode without interrupting other peers in the network. The administrator tells the system his expectation of performance, i.e., a minimum number of transactions it should be able to handle per minute. On overloading (exceeding the given threshold), the DBMS seeks ways to improve its performance, especially when the overload is caused by short and swiftly fluctuating requests. In such a condition, the DBMS transforms its query processing operation into P2P mode and harnesses more power from the peers to assist the DBMS during the peak period (Figure 1.1(b)).



(a) in Existing Systems (b) in POEMS

Figure 1.1: Transformable Architecture

When POEMS operates in a P2P mode, it treats each peer as an autonomic element (AE) [16] that will contribute only its processing power and memory resources without any DBMS capabilities (i.e., a scenario similar to SETI@Home [22] where each peer provides computation resources for discovering extraterrestrial intelligence without any need to know about the details of the operations). Operators have to be shifted on demand to AEs based on the query's execution plan. POEMS interacts continually with AEs to manage their current resource status. When the system becomes overloaded, a query optimizer partitions the data considering the available AEs and the amount of underutilized resources[1]. A root operator that consists of all the optimized sub-operators (e.g., scan, selection or join and projection) will be generated based on the local query execution plan and be sent together with the partitioned data to the AEs for processing. All processing to be carried out at an AE must be main memory based in order to reduce the effects (I/O operations) of interruption on the existing tasks running at the AE. Intuitively, data shifting may incur high communication overhead, but this concern can be easily resolved: by moving only selected operators to remote peers to cooperate with the existing operators, peers can process subsequent queries effectively without any further data shifting from the DBMS (i.e., PUSH-based prefetching).

(a)	**SELECT** CUSTOMER.name, CUSTOMER.address
	FROM CUSTOMER
	WHERE CUSTOMER.cid > 1000

(b)	**SELECT** CART.cartid, CART.status
	FROM CART

(c)	**SELECT** CUSTOMER.cid, CUSTOMER.firstname, CUSTOMER.sex, CART.cartid, CART.status
	FROM CUSTOMER, CART
	WHERE CUSTOMER.cid=CART.cid

Figure 1.2: Sample Queries

As an example of query processing, Figure 1.2 shows three queries on the CUSTOMER and CART relations. Assume Query (a), Query (b) and Query (c) arrive at the DBMS in this sequence. Query (a) is a *selection-projection* on CUSTOMER; Query (b) is a *projection* on CART; and Query (c) is a *projection-join* on CUSTOMER and CART.

Let X be a cluster of peers each of which receives a root operator consisting of *Project(Select(CUSTOMER, CUSTOMER >100))* and a fraction of non-overlapped data from the CUSTOMER relation. Similarly, let Y be another cluster of peers each of which receives a root operator consisting of *Project(CART)* and a fraction of non-overlapped data of the CART relation (see cluster X and Y in Figure 1.3). Incorporating the operators generated from Query (a) and Query (c) at multiple peers can effectively process the subsequent *projection-join* query that involves relation CUSTOMER and CART (cluster Z in Figure 1.3) without the need for any data from the DBMS.

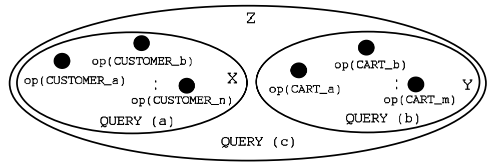


Figure 1.3: Query Processing

[1] We refer the resource in an autonomic element as the number of data pages that it can store in its main memory.

Recall that the term “peer” or autonomic element (AE) is used in this paper to identify office PCs that are dedicated to daily administration/office purposes (e.g., word processing) but contribute the spare capacity of their hardware resources to data manipulation. Therefore, an important property in this environment is that peers are allowed to dismiss a process at will during runtime when they need more resources for their local tasks. This is in vivid contrast to the traditional parallel or distributed database systems which are static.

In the following sections we present the details of POEMS. The contributions of our work include:

(i) The proposal of a dynamic framework that is transformable between the CS and P2P architecture, depending on the workload of the system.

(ii) The introduction of operator-based query processing where there is cooperation between different operators at different hosts for query answering.

(iii) The proposal of a movable operator architecture to handle the dynamism of peers where operators can freely move to another peer on request, e.g., when a peer is going offline or needs more resources for its local tasks.

(iv) The development of several optimization techniques that decrease data transfer among nodes.

(v) The evaluation of the proposed system in a real environment with 26 office PCs.

The rest of the paper is organized as follows: Section 2 provides some essential background. Section 3 describes the POEMS framework and its architecture, while in Section 4 we analyze the proposed techniques. Section 5 presents an extensive experimental evaluation of the system. Finally, Section 6 concludes the paper.

2. BACKGROUND AND RELATED WORK

POEMS can be categorized as an autonomic computing system. Kephart and Chess [16] present a vision of autonomic computing where self-management is an intrinsic property that deals with the complexity of modern computing systems. In autonomic computing, a system maintains and adjusts its operations in order to cope with changing workloads or conditions.

The goal of overload management is to maintain the system’s performance close to optimal under overload conditions. Conceptually, overload management is a special case of load balancing. The key difference between overload management and load balancing is that the former deals with a special set of load conditions (i.e., overload) whereas load balancing focuses on distributing equal loads among nodes even in under-load conditions. Load balancing in shared-nothing architecture [1, 7, 21] has been well studied and deployed in several well-known distributed computing projects such as NOW [1], Condor [10] and Beowulf [7]. The methods can be categorized into static [5, 8] and dynamic load-balancing [14, 23, 25]. These systems follow a common assumption that loads are distributed to a cluster of machines or processing elements that are fully dedicated to sharing loads. In the case of dynamic environments such as P2P, the notion of “virtual-server” has been used [20] to propose a load-balancing algorithm in Distributed Hash Tables (DHT). Each node in the DHT-based structured network can be assigned one or several “virtual-servers”. The “virtual-server” is responsible for a contiguous region of the identifier space. Therefore, a load can be easily split among “virtual-servers”. In this regard, DHT is an example of static load-balancing as it performs an initial de-clustering of key spaces with the intent of load balancing. As an alternative, [2] presents a dynamic load balancing P2P system, which explores balancing in terms of storage load and replication in the P-Grid [3] network.

The P2P systems surveyed above provide support for increases in workload by means of increasing the number of replicas. Our approach is different in several aspects: (i) Operators are distributed among peers and operators cooperate in response to a query, compared to existing techniques which route a query to a single source for processing. The proposed technique therefore offers the advantage of distributing the load of a single query to multiple nodes and consequently, minimizing the burden of each node. (ii) The granularity of data partition is finer and can be adjusted dynamically, allowing data to be partitioned online based on the available resources. (iii) Peers are involved in query processing when POEMS is overloaded, in contrast to existing techniques in which replicated data are cached permanently while waiting for requests, either in under-loaded or overloaded conditions.

Several techniques have been proposed to deal with issues concerning the management of data in the P2P environment [11, 17, 15, 4]. For example, [4] focuses on semantic interoperability in a P2P network with a gossiping technique. In [11], the *ppl*, a Peer-Programming Language combining both LAV- and GAV-style reformulation in a uniform way, is able to chain through multiple peer descriptions to reformulate a query. In [15], mapping tables are proposed for data mapping in the P2P environment. Bernstein et al. [6] introduced the Local Relational Model (LRM) as a data model specifically designed for P2P applications. LRM assumes a set of peers, each being a node with a relational database. A peer exchanges data and services with acquaintances, i.e., other peers. A peer is related to another by a logical acquaintance link. For each acquaintance link, domain relations are used to define translation rules between data items, and coordination formulas define semantic dependencies between two databases. PeerDB [17], on the other hand, employs Information Retrieval (IR) techniques to allow peers to share data without relying on a global shared schema. For each relation that is created by the user, meta-data are maintained for each relation name and attributes. These are essentially keywords/descriptions provided by users upon creation of the table, and serve as a thesaurus of synonyms. By matching keywords from the meta-data of the relations, PeerDB is able to locate relations that are potentially similar to the query relations.

Our proposed system differs from the above-described techniques in four ways: First the context is obviously different. The existing systems do not focus on overload management but on data management. Second, in our context, all peers contribute the spare capacity of their hardware resources for data manipulation without the need for any DBMS to be installed at any of the peers. Consequently, there are no DBMS facilities (i.e., operators, indexes, optimizer or data) at the peers. Third, unlike PDBMS systems which require a DBMS or data repository to be installed at each peer and thus preventing peers from dismissing processes easily, POEMS allows processes or operators to be dismissed online so that a user can channel more resources to his/her own operation tasks. Fourth, since POEMS focuses on a single DBMS with multiple peers (without DBMS), we are not restricted by certain constraints. For instance, we may assume the use of global schemas, which is not applicable to PDBMS.

It should be noted here that there are similarities between POEMS and the classical R* system [26], e.g., the mechanism that is used to minimize the cost of distributed joins. In contrast to POEMS, however, the R* system assumes the existence of multiple DBMSs.

3. PROTOTYPE DESIGN AND IMPLEMENTATION

POEMS is a Java-based implementation. It consists of two main components: the DBMS (we implemented a custom-made DBMS) which runs on a central server, and the AE component which runs on autonomic elements. We shall describe in detail the components of the architecture later in this section. Meanwhile, we shall define the *Processor,* which is a widely used object in our system. A root operator is an operator that encapsulates all optimized sub-operators. In order to distinguish it from operators commonly used in the DBMS, we name the root operator *Processor*. The *Processor* plays an important role in query processing in POEMS. It is a movable object that can be dispatched to AEs as requested or it can replicate itself at other AEs for load sharing.

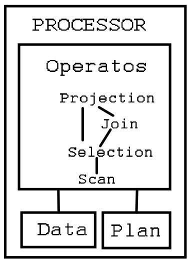


Figure 3.1: *Processor* Architecture

Each *Processor* consists of operators (all sub-optimized operators are encapsulated in a root operator), data and a *Plan* (Figure 3.1). Data are subsets of relations partitioned by the DBMS. We shall defer the discussion on *Plan* generating to a later section. It suffices to say here that the *Plan* is used to guide operators on how to process a query and where to retrieve other partitioned data.

Figure 3.2 depicts the architecture of POEMS. From the network point of view, it consists of a large number of AEs (i.e., AE$_1$ to AE$_7$) which offer their spare resources, and a central DBMS that offers services to internal or external users (outside the organization). The solid lines denote connections among AEs in the same cluster. For example, [AE_1,.., AE_5] and [AE_6, AE_7] are two different clusters that are responsible for storing different data/relations. Each AE in a cluster is responsible for storing part of the data. Continuing our previous example of Figure 1.3, [AE_1,…, AE_5] and [AE_6, AE_7] represent the clusters X and Y, respectively. In this case, each of AE_1 to AE_5 is responsible for storing the partial data of CUSTOMER and AE_6 to AE_7 store the partial data of the CART relation. Each AE also connects directly to the DBMS (denoted by dashed lines).

SQL requests are first directed to the central server. The DBMS has all the features of a traditional DBMS (i.e., query parser, query optimizer, various operators, etc.), which are subsystems kept in its *DBMS Modules*. In addition, the DBMS includes *Resource Inventories* and a *Plan and Processor Generator (PPG)*. *Resource Inventories* maintain an inventory of the location and characteristics of AEs (e.g., how many memory pages an AE contributes, etc.); this is essential for scheduling the execution of query plans.

A transition from the CS to the P2P mode occurs when the system is overloaded. For each query, the DBMS produces *Processors* to handle the task and dispatch the *Processors* to a set of AEs. This is the typical task of the *PPG*, which is also involved in data partitioning during the generation of *Processors*. *PPG* tries to partition and assign all data to the available AEs. If there are not enough AEs to handle the required amount of data, the remaining data are assigned to the DBMS. In such a case, the DBMS acts as an AE that is responsible for the remaining data.

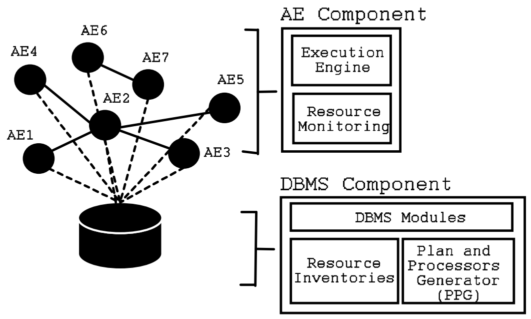


Figure 3.2: Architecture and a Typical POEMS Context Environment

AE$_i$ has a very simple architecture that consists of two major modules: the *Execution Engine* and the *Resource Monitor*. The main purpose of the *Execution Engine* is to execute the *Processor* that it receives without having to know what operators it consists of. Other AEs can connect to AE$_i$ and request data. AE$_i$ may answer a query by executing the operator (or part of it) locally, if it has the required data, or by acquiring data from other AEs. All processed results will be returned directly to the requester that initiated the query without going though the DBMS. We assume there is no firewall blocking the return of results to external users.

Since AEs are not fully dedicated to the task of load management and their load may vary from time to time depending on the tasks that they are currently running, a mechanism is necessary to provide the current resource status of AEs to the DBMS. In our prototype, we implemented a *Resource Monitor* (RM) module. The *RM* monitors the main memory usage of its AE periodically and calculates the number of available pages. This information is uploaded periodically to the DBMS.

4. QUERY PROCESSING

Users pose queries by means of an SQL statement to a central DBMS; incoming queries are put into a FIFO queue. In our current implementation, a query q has the following form:

SELECT *A*
FROM *R*
WHERE *C*

where *R* is a set of relations, *A* is the set of target attributes and *C* is the set of conditions. *C* in the WHERE clause supports the <*expression* **op** *expression*> form, where an expression is a column name, a constant or a string expression and **op** can be one of the comparison operators {<, <=, =, >=, >}.

Let ν_{sys} be a function of the system load, defined as:

$$\nu_{sys} = \frac{\text{Number of completed transactions per interval-}t}{\text{Number of incoming transactions per interval-}t}$$

where interval-*t* is a constant (eg., 1 min). For each interval-*t*, the size of the FIFO queue is checked and ν_{sys} is computed by dividing the number of completed transactions during the interval by the size of the FIFO queue.

Overload is interpreted as a condition where the $\nu_{sys} < \nu_{adm}$, where $\nu_{adm} \in (0,1]$ is a parameter set by the administrator. In this case, query processing operations switch to P2P mode to harness more computation power. Obviously, a large value of ν_{adm} causes the system to always face an overloaded condition, and consequently shifting the system towards P2P processing. In another extreme case, a very small value of ν_{adm} causes the system to retain a CS architecture. This tunable parameter that characterizes POEMS provides better adaptation to user needs based on different environment conditions.

We assume that the services offered by the DBMS are read-only, meaning that no update requests are allowed. This is in line with typical applications such as data warehousing (i.e., aggregation queries), genomic databases which allow the public to analyze existing data, or governmental service portals. In the following, we shall focus on query processing mechanisms under the overloaded condition.

4.1 Query Distribution

Assume that the DBMS notices an increase in load at an interval-*t*, and attempts to solve the problem by transforming the processing mechanism to P2P mode to achieve better performance through simultaneous operations. Let *firstQuery* be the first query that the central DBMS receives upon transformation to P2P mode or a query that requires a table or tables not requested by previous queries while the system was in an overloaded condition. Also, let *followingQuery* be a subsequent incoming request that is inserted into the DBMS FIFO queue waiting for processing. We shall focus on join or select-join queries (e.g., CUSTOMER.cid = CART.cid or CART.status = '1' AND CUSTOMER.cid = CART.cid) which require more processing resources; simple selections are processed similarly.

The DBMS generates a set of *Processors* to handle each incoming query. A *Processor* consists of three main components: data, operators and a *Plan*. First, we discuss the data partition mechanism. Let $R_{(\text{firstQuery})} = \{R_1,\ldots,R_n\}$ denote the cross-product of relations R_1 to R_n for *firstQuery*. For each R_i, data have to be partitioned and distributed to a set of AEs. The data size to be assigned to each AE has to be small enough to fit into its main memory in order to minimize any heavy I/O operation that may cause serious interruption to the user's currently running tasks. The procedure of assigning data based on AEs' available resources is summarized in Figure 4.1.

```
1:  Sort($R_{(\text{firstQuery})}$) in descending order based on relation size
2:  remaining pages = compute total size of $R_{(\text{firstQuery})}$
    // AEs in the Resource Inventories are sorted according to the
    available pages
3:  forward = 0;
4:  For each AE in Resource Inventories
    // $n$ is the number of relations in $R_{(\text{firstQuery})}$
5:    Partition size = AE.pageAvailable / ($n$ +2)
6 :   For each $R_i$
7:      Assign pages to peer while
8:        pages assigned to AE < Partition size + forward
9:      remaining pages--;
10:   Next $R_i$
11:   If pages assigned to AE< Partition size then
12:     forward = partition size – pages assigned to AE
13:   Break If remaining pages = 0
14: Next AE
15: If remaining pages > 0, let DBMS handle
```

Figure 4.1: Resource-based Data Assignment Algorithm

The relations in $R_{(\text{firstQuery})}$ are first sorted based on the size of each relation R_i. The information of each R_i size is collected from the DBMS system catalog. The total number of pages that are needed to store all the tuples of $R_{(\text{firstQuery})}$ is computed, too. For each AE that is registered in the *Resource Inventories*, its buffer (number of pages that it can contribute) is partitioned to n+2 segments, where n is the number of relations in $R_{(\text{firstQuery})}$. Notice that there are two additional segments; one is assigned as storage for intermediate results and another segment is used as the output buffer. Since all operations are performed in the main memory, pointers instead of real values are stored in the intermediate-result buffer. If the aggregate resources contributed by AEs happen to be less than the minimum storage required for storing all $R_{(\text{firstQuery})}$ tuples, the DBMS will be responsible for the remaining data. In such a case, the DBMS plays the role of an AE.

Figure 4.2 is a graphical view of how resources in AEs are partitioned and assigned data. Assume *R*, *S* and *T* are three relations of $R_{(\text{firstQuery})}$ and *Intm* is the intermediate result buffer and *Out* is the output buffer. The height of each rectangle is proportional to the amount of data that has been assigned to it. AE_1 is assigned more data than the others since it contributes more resources. No data from *S* has been assigned to AE_3 and no data from *S* or *T* has been assigned to AE_4 (the shaded-rectangles) as AE_1 AE_2 and AE_3 contribute enough resources to cover the entire *S* and *T*.

Apart from assigning partitioned data to each AE, there is a need to migrate operators to AEs. An AE produces final results solely from operators that are assigned to it by the DBMS. Although query optimization is critical in a relational DBMS, we concentrate in this work on a simple local optimization approach, since we focus more on distributed parallel processing; the optimizer can be replaced easily in the future without affecting the general framework. For the current prototype, we employ an iterative improvement algorithm for the randomized optimization of the query plan. The output of the optimizer is an operator consisting of many sub-operators, i.e., nested-join, selection, projection and scan operators. For each scan operator in an AE, a pointer of a corresponding set of partitioned data that have been generated previously is assigned to it.

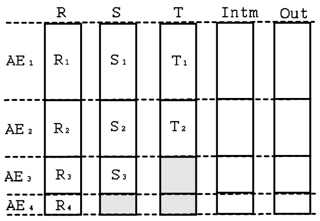


Figure 4.2: Data Assignment for AEs

The *Plan* is the last component that resides in a *Processor*. It is a simple data structure. A *Plan* has an *n* x *m* array, where *n* is the number of relations and *m* is the number of AEs that are participating in solving the given query (possibly from different clusters). Each cell (n_i, m_j), 0<*i*<n, 0<*j*<*m*, in a *Plan* has a value of 0 or 1. A cell is set if AE$_j$ is assigned data *Ri*, and it is reset otherwise. Referring to Figure 4.2, the information of data distribution among the AEs is transformed to a *Plan* as in Figure 4.3. The objective of a *Plan* is to assist query processing by providing information on how and where a query should be processed. In the following, we describe the mechanism of data retrieval and processing in a set of AEs.

	R	S	T
AE₁	1	1	1
AE₂	1	1	1
AE₃	1	1	0
AE₄	1	0	0

Figure 4.3: A *Plan* Structure

4.1.1 firstQuery Execution

On receiving a *Processor*, the AE starts processing the query. This is divided into two sub-processes: local processing (*LP*) and remote processing (*RP*). *LP* starts producing partial results immediately by executing the operators that are attached to the *Processor* if all the required data of $R_{(\text{firstQuery})}$ are locally available (e.g., in Figure 4.3, AE$_1$ and AE$_2$ can produce partial results locally since all the relations *R*, *S* and *T* are available, although the data are not complete). The results are returned to the requester immediately.

Remote processing (*RP*) involves retrieving data from other AEs to the peer for processing. One of the simplest ways to determine the order of data retrieval from different AEs is the following: AE$_j$ (0<j<*m*, *m* is the number of AEs) retrieves the data of relation R_i from AE$_{j+1}$ if and only if the cell of (*i*, *j*+1) in the *Plan* is set. For example, in Figure 4.3, AE$_1$ will retrieve R_2, S_2 and T_2 from AE$_2$, R_3 and S_3 from AE$_3$ and R_4 from AE$_4$. Similarly, R_3 from AE$_3$ and R_4 from AE$_4$ will be retrieved by AE$_2$. Observe that AE$_1$ performs more operations than AE$_2$; similarly, AE$_2$ performs more operations than AE$_3$ and so on. The advantage of this resource-based data and task assignment is that AEs with more resources are assigned more operations than others. As a result, the usage of resources at each AE is optimized and the overloading of AEs with low resources is avoided.

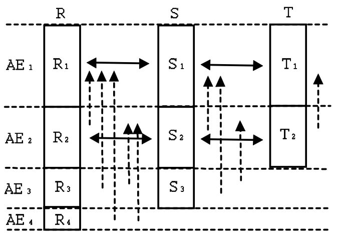


Figure 4.3: Data Retrieving and Processing. Double-arrows-solid-line denotes *LP* and dashed-arrow-line denotes *RP*.

Note that the operators of each AE are generated based on an execution plan produced by the DBMS query optimizer with only local information to estimate the local result size and cost. Therefore, strictly speaking, the plan is optimized if the query is run on the DBMS host. Given the *R*, *S* and *T* relations as an example, based on the associative property of joins, a query optimizer may produce any of these alternative plans: *R* @ (*S* @ *T*), (*R* @ *S*) @ *T* or (*R* @ *T*) @ *S*. Also, since the size of $R \geq S \geq T$, and assuming there are no indexes for the attributes to be joined and each of the relation embodies a uniform distribution of values, plan *R* @ (*S* @ *T*) is favored by the optimizer since *S* @ *T* has a higher probability to produce smaller intermediate results. Although *R* @ (*S* @ *T*) could be the most preferable plan, for *LP*, choosing any other of the execution plans might not affect performance since each AE is assigned an equal data size for each relation as the result of data partition in the earlier stage (e.g., *R*, *S* and *T* all have equal size in AE$_1$). Furthermore, there are no indexes in all the AEs and all the operations are performed in the main memory. Therefore, the processing cost of any of the above-mentioned plans is similar.

However, this might not be valid in *RP*. *RP* involves retrieving data from remote AE$_k$, where *j*<*k*<total AE, to AE$_j$, where the size of R_k in AE$_k$ is smaller than or equal to the size of R_j in AE$_j$. In this case, assigning a smaller set of data to the outer relation may involve higher processing cost. Although we do not have to consider any I/O operation since all operations are main memory-based, accessing memory entails cost-related characteristics that are similar to disk-based I/O operations [27]. We do not deal with main memory optimization issues in this paper, but we need to reduce L1 and L2 cache misses by avoiding the assignment of a larger set of data to an inner relation and a smaller set of data to an outer relation, as such assignments might cause more L1 and L2 cache misses since the size of the L1 and L2 caches is small compared to the main memory. Let Figure 4.4 be an example and let *R* @ (*S* @ *T*) be the plan produced by the DBMS. Assume that the right-most predicates are inner relations and the left-most predicates are outer relations.

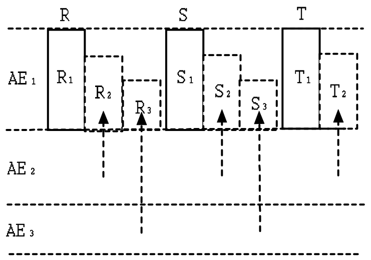


Figure 4.4: An Example of RP. A dashed box denotes a *Retrieve-from* Relationship

As noted, AE$_1$ retrieves R_2, S_2 and T_2 from AE$_2$ and R_3 and S_3 from AE$_3$ to be processed locally. Notice that since $R_3 \leq R_2 \leq R_1$, $S_3 \leq S_2 \leq S_1$ and $T_2 \leq T_1$, there are two possible cases where the plan R @ (S @ T) is inadequate for AE$_1$. First, AE$_1$ retrieves S_2 or S_3 and replaces the local S_1 with S_2 or S_3 for processing, R_1 @ ($\{S_2, S_3\}$@ T_1). In this case, since the size of $\{S_2, S_3\} \leq T_1$ and relation S is the outer relation, $\{S_2, S_3\}$@ T_1 would be more costly than T_1 @ $\{S_2, S_3\}$. Similarly, in the second case, AE$_1$ retrieves R_2 or R_3 and replaces the local R_1 with R_2 or R_3 for processing $\{R_2, R_3\}$ @ (S_1 @ T_1). In this case, the size of the intermediate results produced by (S_1 @ T_1) might be larger than the size of $\{R_2, R_3\}$ since the size of $R_1 = S_1 = T_1$ and $\{R_2, R_3\} \leq R_1$. Piping the larger intermediate results as an inner relation to its parent may affect the cost significantly due to the cache misses of L1 and L2.

One of the solutions is to let the optimizer consider the resources at different AEs while generating the execution plan. Different plans can then be tailored for the various AEs. However, this method adds extra load to the DBMS, which is especially undesirable under an overload condition. In POEMS, all AEs are assigned a single plan, which has the advantage of simplicity and scalability. Nonetheless, *RP* still suffers from the above-mentioned problem. We propose a simple yet effective mechanism to minimize the effect of assigning smaller sets of data to outer relations. Consider Figure 4.5 as an example. We remove S and T from the figure and insert the additional AE$_5$ for illustrative purposes. For each AEi, a window W$_j$ with a size that equals the segment size is created. The window W$_j$ is dispatched to AE$_{(i+1)}$ and is filled with the data of $R_{(i+1)}$. If W$_j$ is not full, it is dispatched to AE$_{(i+2)}$ and the filing process is repeated. The process will stop if W$_j$ is full or AE$_n$ is the last AE in the cluster participating in storing the data of R.

An exception happens when W$_j$ is full, AE$_n$ is the last AE and some data remain (e.g., in Figure 4.5, *W2* is full, AE$_5$ is the last AE and the dashed-box is the remaining data). In this case, the size of W$_j$ is expanded to capture the remaining data. Note, when expanding W$_j$, AEi would require more buffer than the initially assigned segment to store the additional data. The extra buffer that is needed can either be obtained from *Intm* buffer or *Out* buffer, or in the worst case, the additional data may be put on disk. In any case, this may not be an issue since the size of the remaining data is always small (i.e., recall that the arrangement is sorted).

4.1.2 *followingQuery Execution*

When the DBMS receives a *followingQuery*, it checks its FROM predicates. There are three possible cases: (i) all the required relations have been previously distributed to remote AEs in one cluster, (ii) the required relations are handled by two or more clusters and (iii) only parts of the required relations have been distributed to AEs. The DBMS determines the case by referring to the *Resource Inventories*. In the first case, the DBMS generates operators as usual based on its local cost information. The operators are then attached to a new *Processor*. Each of the AEs involved in the processing receives a similar *Processor*. Unlike the *Processor* defined in *firstQuery*, no data or *Plan* is assigned in the *Processor* of the *followingQuery*, since all such information can already be found at the AEs. This is desirable, as in practice, the workload of the DBMS can be reduced considerably.

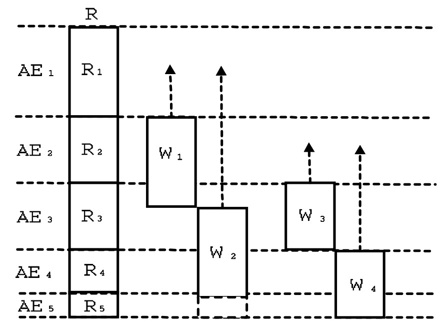


Figure 4.5: Fixed-Size Window for Data Retrieval

If some of the required data are not found in the AEs (i.e, case (iii)), a *followingQuery* will be processed in two steps. In the first step, each part of the missing data is obtained from the DBMS and then partitioned and distributed to a new set of AEs. The second step is the same procedure that applies to the case (ii): In contrast to *firstQuery* processing which involves a single cluster of multiple AEs, the major challenge in this case lies in the mechanism of joining relations from two or more clusters, each with a set of AEs. Assume there are two clusters: Cluster (R, S) consisting of {AE$_1$, AE$_2$, AE$_3$, AE$_4$} and Cluster (T) consisting of {AE$_5$, AE$_6$, AE$_7$} as shown in Figure 4.6. Cluster (R, S) stores the data of R and S, while Cluster (T) stores the data of T. Observe that there are several query processing issues. First, the cluster selection problem of defining where an execution should be performed (e.g., processing can be either running on Cluster (R, S) and fetching T from Cluster (T) or vice versa). Second, the size of the *to-be-fetched* relation in a cluster might be larger or smaller than the available memory resources of the requester AE. Fetching data larger than memory size incurs I/O operations since data have to be stored in the disk. At the other end, resources are underutilized when a small set of data is fetched.

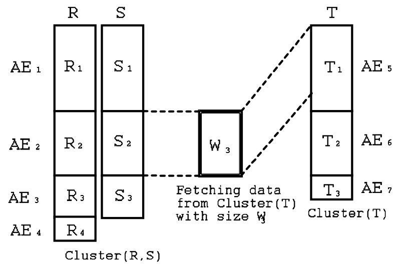


Figure 4.6: Processing with Two Clusters

Given a query R @ (S @ T) and two clusters as in Figure 4.6, POEMS selects a cluster that has the maximum number of FROM predicates stored, e.g., Cluster (R, S) has two predicates out of three of R @ (S @ T). If there is a tie, random selection is used. Notice that given two clusters, each of the AEs in the first cluster has to fetch data from the AEs of the second cluster. Therefore, in order to optimize the memory resources used, each AE in the first cluster will define a window of the size of the segment as described in *firstQuery* processing.

4.2 Reducing the Network Cost

There are two points that involve data transfer: (i) when the DBMS distributes data to a set of AEs, and (ii) during data fetching among AEs. We do not restrict data flow from the DBMS to AEs since any data missing from the AEs will entail reference to the DBMS again, which means an additional load on the already overloaded DBMS. On the other hand, data transfer among AEs should be reduced to minimize network cost. Even though the amount of data transfer from the DBMS could not be reduced, it is worthwhile grouping similar queries together and reducing the number of queries that will be posed to AEs for processing. Therefore, our optimization strategy is two-fold: to reduce the number of queries posed to AEs and to minimize data transfer among AEs.

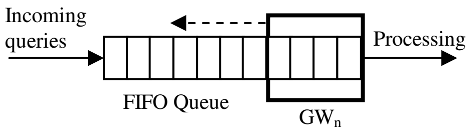


Figure 4.7: Windowed and Grouped Strategy

We propose a *Windowed and Grouped* (WG) algorithm to reduce the number of queries that will be posed to AEs that aims to minimize total execution cost by grouping concurrent queries that require similar resources. Recall that all incoming queries are first inserted into a processing FIFO queue in the DBMS. Let GW_n be a group window that takes *n* queries from the queue and groups similar queries together (see Figure 4.7). Given two queries, Q1, Q2, and their data sources $R_{(Q1)}$ and $R_{(Q2)}$ respectively, they can be grouped if and only if $R_{(Q1)} \subseteq R_{(Q2)}$ or $R_{(Q2)} \subseteq R_{(Q1)}$, and the data sources in $R_{(Q1)}$ are the same or a subset of the data sources in $R_{(Q2)}$ and vice versa. Freezing queries and grouping them later may not appear worthwhile at first glance because of the short lifetime of the query since the strategy introduces additional delay to query execution. However, there are two conditions that make WG a desirable algorithm in a situation of overload. First, the swiftness of requests enables a large number of candidates to be considered for grouping in a very short period. Second, a slowdown in the average query response time is unavoidable when system overload sets in. By sacrificing a short delay as a trade-off would reduce system workloads.

Two algorithms are used in POEMS to reduce the amount of data to be shipped among AEs. Semijoin [28] reduces the tuples to be shipped from site A to site B with the following steps (assuming relation *S* at site A is to be joined with relation *T* at site B on attribute *a*). First, compute the projection and sort *S* at site A on the join column '*a*' and then eliminate the duplicates. Let S' be the projection output of the tuples produced by this step. Next, send S' to site B and select the tuples of *T* that match S'.a, yielding a reduction *T*', and ship them to site A. Third, at site A, join *T*' with *S* and return the results to the user.

Bloomjoin [29] works similarly, but it uses Bloom filters to filter out non-matching tuples in a join. A Bloom filter is a bit-vector of size *k*. It works as follows: (i) It generates a *k*-bit-vector and set each bit to 0. Then, it hashes each of the value of *S*.a column at site A to a particular bit in the *k*-bit-vector and sets the bit to 1. (ii) It sends the vector to site B. It hashes each value of *T*.a at site B with the same hash function as in step (i). If the hashed bit is "1", the tuple can be a possible candidate. Assuming that T' is the reduction of all candidates, it is sent to site A. (iii) At site A, it joins *T*' with *S* and returns the results to the user. In our case, there is a one-to-many operation in which a processing initial site AE*i* (i.e., site A) sends projection tuples to a set of AEs $\{AE_{i+1},\ldots AE_{i+n}\}$, each of which stores the partial data of relation *T* (recall Figure 4.6) for requesting their reduction *T*'. Intuitively, we can send the request to each of $\{AE_{i+1},\ldots AE_{i+n}\}$ sequentially by ignoring the size of the reduction *T*'. However, this would not utilize the memory resources of AE_i, i.e., the returned *T*' might be larger or smaller than AE_i's memory resources. Therefore, we extend the algorithms as follows.

```
On RequestRemoteData
1: Generate v from the to-be-joined relation
   // v can be either a Bloom filter or a projection for bloomjoin
   //and semijoin respectively
2: If anchor is null send v and W to AE_data.
3: Else send v and W to anchor
   // W is a window of a size equal to AE_i segment size

On ReductionRequestReceived(v, W)
4: Generate reduction T'
5: Fill W with reduction T', starting from the last data accessed
6: If W is not full and AE_data+1 is not null, Send v and W to AE_data+1
7: Else set this as anchor and return W and anchor
```

Figure 4.8: Algorithm for Data Fetching among AEs

Let AE_{req} be the requester AE and AE_{data} be the AE having the data. AE_{req} generates a Bloom filter or a projection on the to-be-joined column, depending on its policy (**On** RequestRemoteData in Figure 4.8). An *anchor* is a pointer to the last item fetched. It consists of this information: the AE_{data} identity of the last accessed AE and its data pointer pointing to the last accessed data. Obviously, the *anchor* is null if this is a fresh operation. If the *anchor* is not null, AE_{req} starts fetching data from the AE_{data} that is defined in it. The AE_{data} that receives the request (**On** ReductionRequestReceived in Figure 4.8) then generates the reduction *T*' and fills *W* which is of the same size as the AE_{req} segment. If *W* is not full, AE_{req} forwards the operation to AE_{data+1}, where the identity of AE_{data+1} can be obtained from the AE_{data} *Plan*. The operation stops if AE_{data+1} is null or *W* is full, which causes *W* to be returned to AE_{req}.

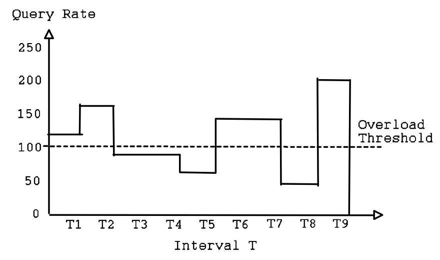
We also consider Hash-based join algorithm, instead of the Distributed Nested-Loop join which was mentioned previously. Hash-based method has the advantage that no data need to be transferred between AEs for processing the join. The disadvantage, however, is that if a subsequent query joins the datasets on a different attribute, all data must be re-hashed. Due to space constraint, we omit the algorithm here. Section 5.3.1 presents the experiment results of network load optimization based on the Hash-based distributed joins.

4.3 *Processor* Reallocation

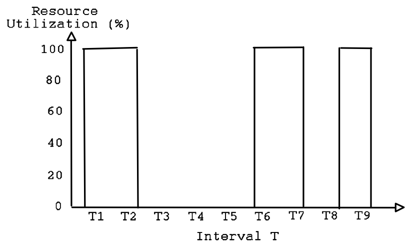
Recall that users are allowed to withdraw their contributed resources anytime. When a user withdraws his resources, the *Processor* (i.e., inclusive of data and operators) at the user AE has to be reallocated. The AE interacts with the DBMS to find an available AE' to handle the *Processor*. Checking its *Resource Inventories*, the DBMS informs the AE to migrate the *Processor* if there is (i) an AE' that has enough resources to take over the *Processor*, or (ii) a set of AEs whose aggregate resources are enough to take over the *Processor*. The first case is simple since it involves only migrating the *Processor* to AE'. In the second case, we need to split the *Processor* into several sub-*Processor*s. Still, this is straightforward as we only need to split the data in the *Processor* to several portions according to the new AEs' resources. If there are no AE' available the *Processor* will be relocated to the DBMS. All of these cases involve the updating of the *Plan* in order to reflect the new location of the data. The DBMS generates a new *Plan* and updates all other AEs who are members of the same cluster as the requester on the new location of the *Processor*. Notice that the above methods assume informed leave; handling node failures is outside the scope of this paper.

4.4 Optimizing the Utilization of Resources

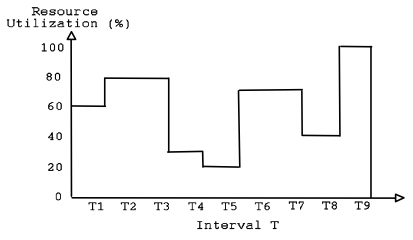
It should be obvious by now that the benefit of the system is improved if during an overload period many queries can be answered in the peers by data which are already cached there. However, the peers need their resources for their individual needs; ideally, they would like to contribute only in emergencies (i.e., server overload).



(a) Server workload



(b) Naïve resource utilization



(c) Microeconomics-based resource utilization

Figure 4.9: Server workload and Resource utilization at the peers

Figure 4.9a presents a scenario of server workload. When the query rate is more than 100 queries/period, the server is overloaded. During these periods, it requests resources from the peers. Figure 4.9b presents the resource utilization of a naïve algorithm, which uses 100% of the available peer resources during peak periods, and releases all of them when the peak ends. This is acceptable from the peers' perspective; however, when a new peak starts the server must retransmit all the required data.

To achieve a balance among the total amount of data transmitted by the server, the throughput of the system and the peers' desire to avoid contributing resources, we adopted an algorithm based on microeconomics (Figure 4.10). Each peer has a set of blocks which is its resource. Each of the blocks can be reserved for storing data by paying some amount of virtual currency. The block is reserved and valid for a period of an *interval-t*. On expiration, the peer has the right to regain it back if the requestor does not pay for it. In order to reserve the blocks, the requestor has to earn profit. The server earns profit for every query it manages to serve while exceeding its capacity. On each *interval-t*, the OnLoad function is called to compute the block reservation strategy. It first pays for all the reserved blocks if the account has enough currency, otherwise the peer will regain the blocks. After clearing the payment, it checks if the number of reserved blocks are enough to handle the incoming requests. If more blocks are needed and the account has enough currency, it buys extra blocks. Otherwise, it needs to loan extra blocks for storing the data. The loaned blocks have to be paid back once it has earned the profit. However, the profit may not always enough to pay for all blocks and unpaid blocks are returned to peers. In this case, some of the cache data are lost.

```
OnLoad(int load)
1: Pay for the reserved blocks;
2: If the amount of reserved blocks is not enough to handle the load then
3:         If has some currency then
4:                 buy blocks, up to the maximum of currency it has
5:         If the amount of reserved blocks is still not enough to handle
           the load then loan blocks (needBlk);
7: ComputeProfit(load);
8: If loaned then PayLoan;

PayLoan()
1: If account has enough currency then pay and clear loan;
3: Else If account > 0 but not enough to pay for the loan then
4:         pay whatever amounts exist in the account;
5:         return n number of reserved blocks to peer as loan payment,
           where n is the number of loan in balance.;
6: Else
7:         peer regains all resources;
8:         clear loan;
```

Figure 4.10: Resource allocation Algorithm

Figure 4.9c shows the behavior of the algorithm for the server workload of Figure 4.9a. It is evident that our algorithm follows closely the workload, and avoids using all the available resources on the peers if this is not necessary. Additionally, it keeps some resources in the peers even when the server does not need them at the time, since these will help increasing the query performance during the next peak.

5. EXPERIMENTAL EVALUATION

We tested our prototype on a network with 26 Pentium IV, 1.6MHz, WinXP PCs (AEs) with 256MB RAM, and a Sun Solaris server with 4GB RAM and two Ultra-SRARC processors 480MHz CPUs which served as the central DBMS. All machines were physically connected to a LAN. The server-to-AE and the AE-to-AE transfer rate was 10Mbps and 100Mbps, respectively.

We used the TPC-H schema to generate the dataset for our experiments. It consists of eight separate tables; the largest has 600572 tuples, while the total number of tuples in all tables is around 900,000. We defined a query set with a mixture of selection and join queries. We used two metrics to evaluate the system performance. (i) the number of completed transaction per interval (i.e., throughput), denoted as *CT(t)*, and (ii) the *effectiveness* of a system, defined as: 100**CP(t)/Q(t)*, where *Q(t)* is the number of incoming requests per interval *t*. The default value of *t* was 60min. Obviously, when the server is not overloaded, its *effectiveness* is 100%

5.1 Client-Server vs. POEMS

In the first set of experiments we compare a central DBMS that employs static client-server architecture (CSDBMS) against POEMS. For fairness, we do not employ any optimization strategies for POEMS in this state. The server is considered overloaded when the rate of incoming queries *Q(t)* ≥ 120. The results are presented in Figure 5.1.

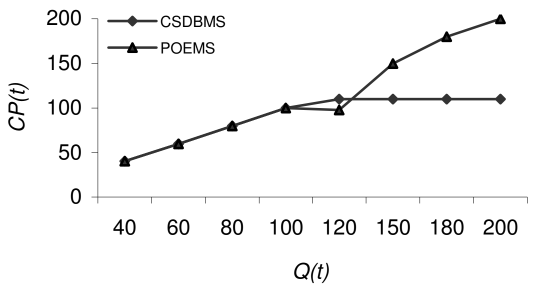


Figure 5.1: CSDBMS vs. POEMS

When *Q(t)* (less than 120 for this setting) is low, both systems achieve best results; notice that there is no overhead in POEMS when it works in the CS mode. However, when *Q(t)* increases to more than 120, the server cannot handle the additional load anymore. Therefore *CP(t)* for CSDBMS becomes constant, while the *effectiveness* drops, as shown in Figure 5.2.

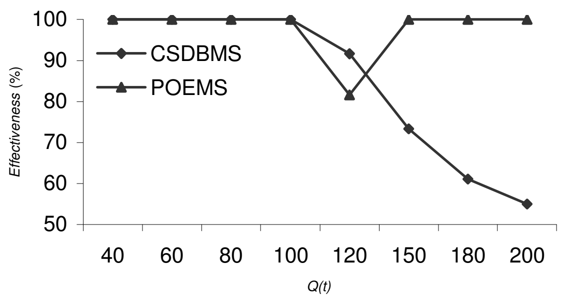


Figure 5.2: CSDBMS vs. POEMS in term of *Effectiveness*

POEMS, on the other hand, switches into the peer-to-peer mode. For *Q(t)*=120, the server is overloaded and additionally has to pay the cost of transforming to P2P mode. This involves data partitioning and data distribution, which are expensive processes since they involve scanning large portions of the data and preparing the *Processor* objects to be distributed to the remote AEs. For this reason, POEMS' performance at that point is slightly worse than CSDBMS.

After POEMS has paid the initial cost, however, it can utilize the peers' resources to increase its throughput. Since the data are already in the peers, only operators are sent to AEs for most subsequent queries, which are much smaller in size. Moreover the queries can be evaluated in parallel, from data which reside in the peers' main memory. These facts explain the performance boost of POEMS over CSDBMS for high query rates.

5.2 Optimized POEMS

In these experiments we evaluate the query grouping optimization, presented in Section 4.2. The settings are the same as the previous experiments and the results are presented in Figure 5.3 (number of completed queries) and Figure 5.4 (effectiveness). "Pure+Pure" represents a POEMS system with all the optimizations turned off (i.e., the same as in the previous section), while "FW+Pure" denotes POEMS with the query grouping optimization.

Query grouping identifies similar queries and shares the execution cost among them. Essentially, this optimization decreases the number of distinct queries running simultaneously; therefore it allows the system to complete more queries in a given period. This is evident from the figures, where query grouping increases the system's performance by around 50%

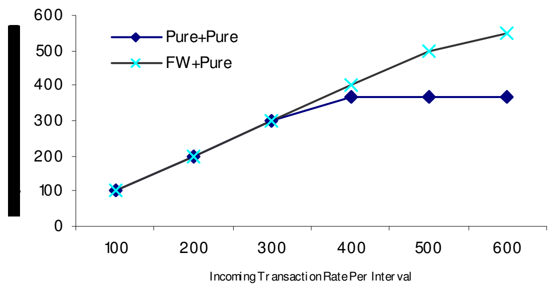


Figure 5.3: *CP(t)* for the query grouping optimization

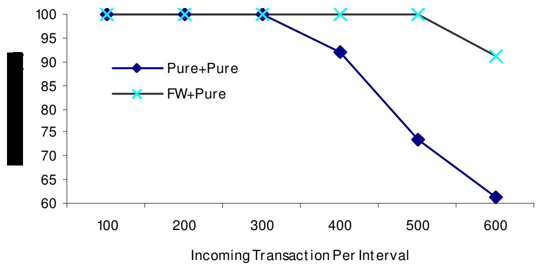


Figure 5.4: *Effectiveness* for the query grouping optimization

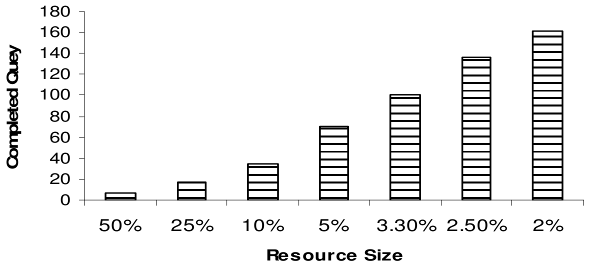
We also tested several other optimizations, including Semijoins and Bloomfilter joins. Both methods improved the results of pure POEMS. Due to space constraint, we omit the results here.

5.3 Network Load Optimization

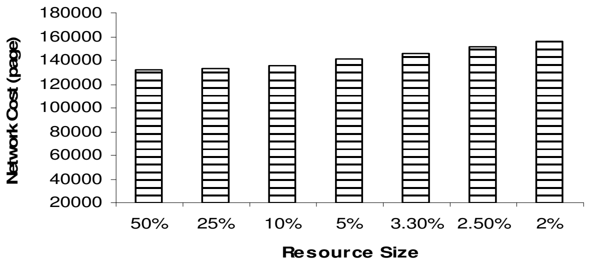
In this set of experiments, we implemented a simulator using the parameters from the prototype in order to evaluate the effectiveness of network load optimization on a larger network.

5.3.1 Effectiveness of Distributed Hash Joins

Figure 5.7a shows how the distribution of resources affects the query performance based on Distributed Hash Joins. We keep the total amount of resources constant, and vary the number of peers. The first column, for instance represents a network of 2 peers, each one having 50% of the available resources, while the last column is for a network of 50 peers, each having 2% of the total resources. Figure 5.7b shows the network cost for the same settings. From the results, it is obvious that hash-based join benefits much from the increased number of peers. This is due to the fact that the join can be performed in parallel with negligible communication overhead.



(a) Completed transactions (CT)

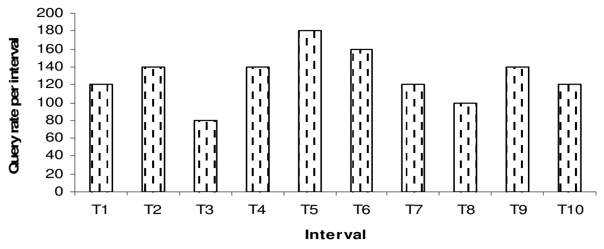


(b) Network cost (amount of transferred pages)

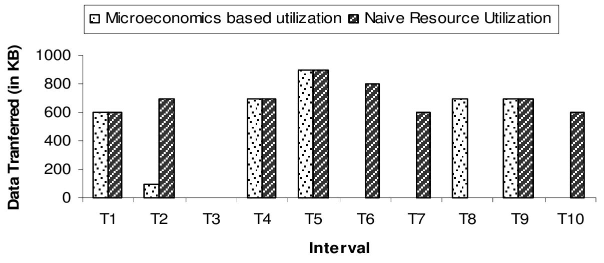
Figure 5.7: Hash-based Distributed Join

5.3.2 *Effectiveness of Microeconomics Algorithm*

In this experiment we compare the network load due to a naïve resource utilization algorithm with that of the microeconomics-based resource utilization introduced in Section 4.4. Figure 5.8a presents a scenario of server workload. When the query rate is more than 100 queries/period, the server is overloaded. During these periods, the server requests resources from the peers. When a naïve resource utilization algorithm is used, the server transmits all the required data to the peer and the data is discarded at the peer once the peak ends. When a new peak starts, the server must retransmit all the required data to the peer. When the microeconomics based method is used, some amount of data is retained at the peer even when the peak is over, so that it can be reused during the next peak. This avoids the need to retransmit all the data at each peak, thus reducing the load on the network. In the simulator, the network transfer rate is set to 10Mbps and a simple selection query is being used in this experiment. The workload scenario depicted in Figure 5.8a. We measured the data transferred in two different settings: one using a naïve resource utilization algorithm and the other one using the microeconomics based resource utilization. Figure 5.8b shows the results obtained. At intervals T6 and T7 the data cached at the peer is reused and hence no new data needs to be transmitted in the case of microeconomics based resource utilization, while all the data needs to be retransmitted when the naïve resource utilization algorithm is used. At T8 there has been a data transfer due to the microeconomics based utilization, because the resources at the peer are not enough to handle the load. Over the periods T1 to T10, the naïve resource utilization algorithm causes a transfer of a total of 5600 KB of data, while the microeconomics based utilization causes a transfer of 3700 KB of data. We see that there is about 33% of savings by the latter. Thus indicates that the microeconomics based algorithm optimizes network utilization more effectively.



(a) Server workload



(b) Data transferred from server to peer

Figure 5.8: Server workload and network utilization

6. CONCLUSION

In this paper we investigated the practical problem of dealing with overloads in enterprise database servers. Motivated by the fact that current solutions are either too expensive or too complicated to be deployed, we developed POEMS. Our system utilizes the available resources in numerous office PCs by asking them to perform some data manipulation in order to relieve the database server during the peak periods. The whole procedure is transparent to the user, and does not require the installation of any DBMS on the PCs, since all the data together with the required operator are sent from the server as Java objects. Furthermore the PCs contribute their resources only when it is absolutely necessary, thus minimizing the disturbance to the office user.

We implemented POEMS and deployed it on a network with 26 PCs and a UNIX server. The implementation illustrates the feasibility of the proposed framework, and the experimental results demonstrate its potential: it managed to increase the throughput by 400% based on the naïve methods and it achieved an additional boost of 200% by employing obvious optimizations. We also showed that under known query patterns, the system can be very scalable. We are confident that by incorporating sophisticated optimization methods, POEMS has the potential to achieve even better performance in a wide range of practical applications.

REFERENCES

[1] T. E. Anderson, D. E. Culler and D. A. Paterson, “A case for NOW (network of workstations)”, IEEE Micro, 15(1), 1994, pp. 54—64.

[2] K. Aberer, A. Datta, M. Hauswirth, “The Quest for Balancing Peer Load in Structured Peer-to-Peer Systems”, EFPL Technical Report IC/2003/32, 2003.

[3] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, R. Schmidt, J. Wu, “Advanced Peer-to-Peer Networking: The P-Grid System and its Applications”, PIK Journal Special Issue on P2P Systems, 2003.

[4] K. Aberer, P. Cudré-Mauroux, M. Hauswirth, “A framework for semantic gossiping”, SIGMOD Record, 31(4), 2002.

[5] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith and P. Valduriez, “Prototyping Bubba, a highly parallel database system”, IEEE TKDE, 2 (1), March 1990, pp 4—24.

[6] P. Bernstein, F. Giunchiglia, A. Kementsietsidis, J. Mylopoulos, L. Serafini and I. Zaihrayeu, “Data management for peer-to-peer computing: A vision”, WebDB, 2002.

[7] D. J. Becker, T. Sterling, D. Savarese, E. Dorband, U.A. Ranawake, and C.V. Packer, “BEOWULF: A Parallel Workstation for Scientific Computation”, ICPP, 1995.

[8] G. Copeland, W. Alexander, E. Boughter, and T Keller, “Data placement in bubba”, ACM SIGMOD, 1995, pp 99—108.

[9] M. J. Carey, L. M. Haas, P. M., et.al. “Towards heterogeneous multimedia information systems: The Garlic approach”, Int’l Workshop on Research Issues in Data Engineering (RIDE): Distributed Object Management, 1996.

[10] D. H. J. Epema, M. Livny, R. van Dantzig, X. Evers and J. Pruyne, “A worldwide flock of Condors: load sharing among workstation clusters”, Journal on Future Generations of Computer Systems, 1996.

[11] A. Y. Halevy, Z. G. Ives, P. Mork and I. Tatarinov, “Schema Mediation in Peer Data Management Systems”, ICDE, 2003.

[12] A. Y. Halevy, Z. G. Ives, P. Mork and I. Tatarinov, “Piazza: Data Management Infrastructure for Semantic Web Applications”, W3C Conf., 2003.

[13] L. M. Haas, R. J. Miller, B. Niswonger, M. T. Roth, P. M. Schwarz and E. L. Wimmers, “Transforming heterogeneous data with database middleware: Beyond integration”, IEEE Data Engineering Bulletin 22 no. 1, 1999, pp. 31—36.

[14] A. Helal, D. Yuan, H. El-Rewini, “Dynamic Data Reallocation for Skew Management in Shared-Nothing Parallel Databases”, Distributed and Parallel Databases vol 5 no. 3, 1997, pp. 271—288.

[15] A. Kementsietsidis, M. Arenas and R. J. Miller, “Mapping Data in Peer-to-Peer Systems: Semantics and Algorithmic Issues”, ACM SIGMOD, 2003, pp. 325—336.

[16] J. O. Kephart and D. M. Chess, “The vision of autonomic computing”, IEEE Computer, 36(1), January 2003, pp 41—50.

[17] W. S. Ng, B. C. Ooi, K. L. Tan and A. Y. Zhou, “A P2P-based System for Distributed Data Sharing”, ICDE, 2003, pp. 633–644.

[18] W. S. Ng, B. C. Ooi, K. L. Tan and A. Y. Zhou, “PeerDB: Peering into Personal Databases (Demo)”, ACM SIGMOD, 2003, pp. 659.

[19] C. Parent and S. Spaccapietra, “Database integration: an overview of issues and approaches”, Communications of the ACM 41 no. 5, 1998, 166—178.

[20] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp and I. Stoica, “Load Balancing in Structured P2P Systems”, International Workshop on Peer-to-Peer Systems, 2003.

[21] M. Stonebraker, “A Case for Shared Nothing”, Database Engineering, 9(1):4-9, 1986.

[22] SETI@Home Home Page, http://setiathome.ssl. berkeley.edu/.

[23] P. Scheuermann, G. Weikum and P. Zabback, “Data Partitioning and Load Balancing in Parallel Disk Systems”, VLDB Journal vol 7, no 1, 1998, pp. 48—66.

[24] A. Tomasic, L. Raschid, and P. Valduriez, “Scaling heterogeneous databases and the design of Disco”, ICDCS, 1996, pp. 449—457.

[25] R. Vingralek, Y. Breitbart and G. Weikum, “SNOWBALL: Scalable Storage on Networks of Workstations with Balanced Load”, Distributed and Parallel Databases vol 6, no 2, 1998, pp. 117—156.

[26] L.F. Mackert and G.M. Lohman, “R* Optimizer Validation and Performance Evaluation for Distributed Queries”, VLDB 1986.

[27] P. A. Boncz, S. Manegold, and M. L. Kersten., “Database Architecture Optimized for the New Bottleneck: Memory Access”, VLDB, 1999, pp 54–65.

[28] P.A. Bernstein and D.W. Chiu, Using semijoins to solve relational queries, Journal of the ACM 28,1, 1981, pp. 25-40.

[29] K. Bratbergsengen, Hanshing Methods and Relational Algebra Operations, VLDB, 1984, pp. 323-333