ELSEVIER

Information and Software Technology 38 (1996) 103–110

INFORMATION AND SOFTWARE TECHNOLOGY

DITSE: an experimental distributed database system

Cheng-Yao Ni, Shyan-Ming Yuan*

Department of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan, Republic of China

Abstract

DITSE (DIstributed Transaction Services Environment) is a distributed heterogeneous multi-database system which provides SQL to allow users to transparently access multiple databases on different machines. It also provides OLTP (On-Line Transaction Processing) services which contribute to develop reliable distributed applications. DITSE follows the most popular commit protocol, two-phase commit protocol, to reach an agreement between different transaction managers. In this paper, we describe the architecture and some important features of DITSE. Relevant design issues are also addressed.

Keywords: Transaction; DBMS; Concurrency control; SQL; Client-server

1. Introduction

The basic premise of a distributed database system is to make data stores spread over a collection of machines in a wide or local area network such that they appear to the users as a single machine. Although this premise is really attractive, there are many technical problems behind it. According to Stonebraker [1], a distributed database system must have the following transparencies: location transparency, performance transparency, copy transparency, transaction transparency, fragment transparency, schema change transparency, and local DBMS transparency. Many research prototypes have been published since 1970s, such as SDD-1, Distributed INGRES, and R* [1]. Unfortunately, these prototypes cannot meet all these transparencies; neither can our system. The goal of our system is to try to build a distributed database system in an alternative direction, not to beat other prototypes.

What is the difference between our system and others? Normally, a database system, whether centralized or distributed, is built on top of a file system or raw disk. However, we do not build it from scratch. Instead, we build it on top of a distributed transactional file system (DTFS) [2]. DTFS is a distributed file system which supports distributed transaction facilities, including concurrency control and recovery. It is standalone and provides an OLTP [3] library. A client program links the library and then it can access files spread over DTFS. Actually, you will see in the latter that our database server is also a client of DTFS. The benefit of separating DTFS from database systems is that we offer transaction services at two different levels—file level and database level. To the best of our knowledge, no system uses this approach. We think this is an interesting experiment, which is why we call DITSE an *experimental* distributed database system.

There are many different types of distributed database system (DDBMS). To avoid ambiguity, we classify DDBMS more formally. DDBMS can be classified in different ways [4,5]. In this paper, we follow the classification in Ozsu and Valduriez [4]. DDBMS can be classified with respect to: (1) the autonomy of local systems; (2) distribution; and (3) their heterogeneity. Autonomy refers to the distribution of control, not of data. It indicates the degree to which individual database systems can operate independently. The distribution dimension of the taxonomy deals with data. Heterogeneity may occur in various forms, ranging from hardware heterogeneity to variations in data models.

From this classification, we find that our system is a distributed heterogeneous multidatabase system [6,7]. Currently, we have integrated one heterogeneous DBMS—Sybase *SQL* server. There is no technical problem to integrate other DBMS. However, we haven't done it yet owing to resource constraints. To integrate a heterogeneous DBMS, we built a front-end software (also termed agent or gateway) which operates on top of each local DBMS. Different DBMSs use different gateways. The advantage of this front-end approach is that each individual DBMS does not need to make any change. Moreover, this approach makes it possible to gain access to a variety of DBMSs in a uniform way. By the way, a *common* high level language (SQL) is provided to prevent heterogeneous query languages from annoying users.

This paper is organized in five sections. In the next section, we take an overview of the DITSE. The following section describes some design issues relevant to DDBMS and the choices we have made. The next section explains some technical details, algorithms, and problems that we have met, and finally, we make a conclusion in the last section.

*Corresponding author. This paper is a partial result of the project No. I83001 conducted by the ITRI under sponsorship of the Minister of Economic Affairs, ROC.

2. System architecture

The overview of the DITSE system architecture is shown in Fig. 1. An arrow is a communication link, either by sockets or by remote procedure calls. An ellipse represents a process. In the following, we first take a glance at each component of our system.

2.1. Distributed Transactional File System (DTFS)

The DTFS consists of the OLTP library and a set of servers. Its architecture is shown in Fig. 2. There is only one name server (NS) in the DTFS. The name server is analogous to the directory assistance operator in the telephone system. Just as in the telephone system, it is essential that the address of the NS is well known. Name server provides the following services:

(1) Registration. Every transaction manager (TM) must register themselves to the name server.
(2) Query address. *SQL* clients can query the name server to know where a specific table is.
(3) Location allocation. When an *SQL* server creates a new table, it asks the NS to assign a location to place it. Because every TM has registered itself, the NS has the configuration of the whole system, such as the number of disks and where these disks are located. Hence, it can find a way to balance the capacity of each disk. This can prevent the overrun of some TMs while the others are idle. Thus, the transaction throughput may be increased.

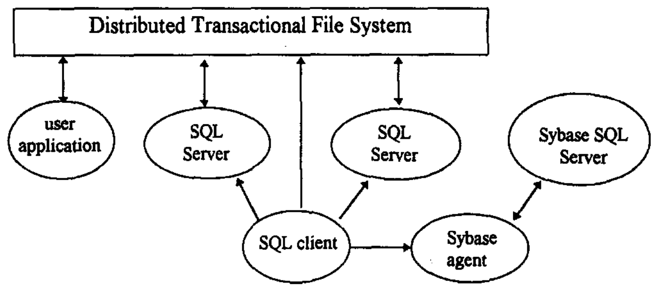

Fig. 1. The overview of the DITSE structure.

In addition, there is one lock manager (LM) and one TM in each machine. The LM handles concurrency control. As its name suggests, it uses locking-based concurrency control algorithm to synchronize all local transactions. Upon receiving a transaction request, the TM generates (forks) a process at each transaction. It depends on the type of the request to generate either a coordinator process or a cohort process. When a TM receives an OpenTransaction operation, it creates a new transaction and generates a coordinator process to coordinate the transaction. If a TM receives an access operation of an unseen transaction, it generates a cohort process to handle that operation and all subsequent operations of the transaction. Thus, any transaction has only one cohort process at one site, but it may have several cohort processes at different sites at the same time. A coordinator and all its cohorts use two-phase commit protocol to ensure atomicity property of the transaction. Also, a cohort communicates with local LM via message queues to acquire lock permissions before accessing the local database. The message queue is one of the System V IPCs in UNIX.

2.2. SQL server

An *SQL* server is a LAN-based relational database server that stores client data and provides other services such as data update and retrieval. It provides database engine that processes *SQL* statements submitted by various clients. In general, a database server should also provide other services such as concurrency control and transaction processing. However, in DITSE, concurrency control is handled by LMs and transaction processing is handled by TMs. There can be more than one *SQL* server in DITSE. A user can query tables on different *SQL* servers transparently.

2.3. Sybase Agent

Different DBMSs may differ in several aspects. For example,

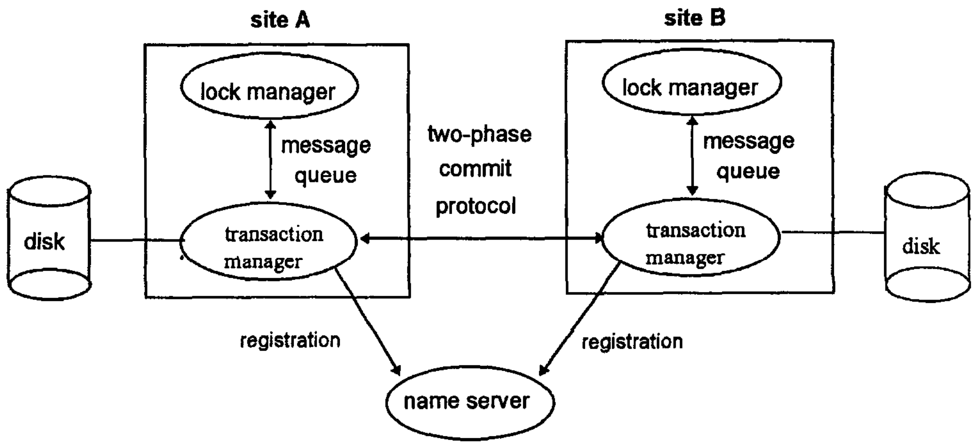

Fig. 2. Distributed transactional file system.

they may use different data models, query languages, or data types. Thus, a local DBMS may not understand the *SQL* commands submitted by our *SQL* clients. To solve this problem, we build an agent on top of each local DBMS to transform our clients' *SQL* commands to the native *SQL* commands of the DBMS. The only task of an agent is the transformation. Our Sybase agent is implemented by the Sybase's DB-Library. It is a set of C routines and macros that allow users to send Transact-SQL commands to Sybase *SQL* servers. Since both Sybase and our *SQL* server are relational DBMSs, only data type transformation and some query language transformation are needed for our Sybase agent.

3. Design issues

3.1. Distributed query processing

In traditional database systems, a major aim is to hide structural details of the data from the users as much as possible. In distributed database systems, one of the main goals is that the distribution details should also be hidden. To meet these requirements, database systems often provide a high level query language. The main function of a query processor is to transform this high-level query into equivalent lower-level one. The most important module of a query processor is query optimization. Query optimizer tries to optimize a cost function. In distributed database systems, this cost function should be a weighted combination of CPU, I/O, and communications costs. However, for simplification, we make an assumption that communication cost is a dominant factor.

There are two types of distributed query optimization: join and semijoin based algorithms. Semijoin operator [8] can reduce network transfer cost if it is beneficial. A semijoin operator is beneficial only if the cost of projecting and sending join attributes to the other site is less than the cost of sending the whole operand relation and doing the actual join. Finding a beneficial semijoin ordering must count on database statistics. Maintaining database statistics is a burden and the precision of these statistics is still a question. Therefore we use join operator.

Fig. 3 shows the layering scheme of our distributed query processing. Some implementation details can be found in the section below headed 'Implementation'. The first layer translates the distributed query into an intermediate language. Then global optimization tries to find an execution plan which can minimize network-transfer cost. The last layer is performed by all the sites which have relations involved in the distributed query. Each subquery is then optimized by the traditional query optimization algorithm used in the centralized DBMS.

3.2. Concurrency control

A simple and elegant locking based algorithm to ensure

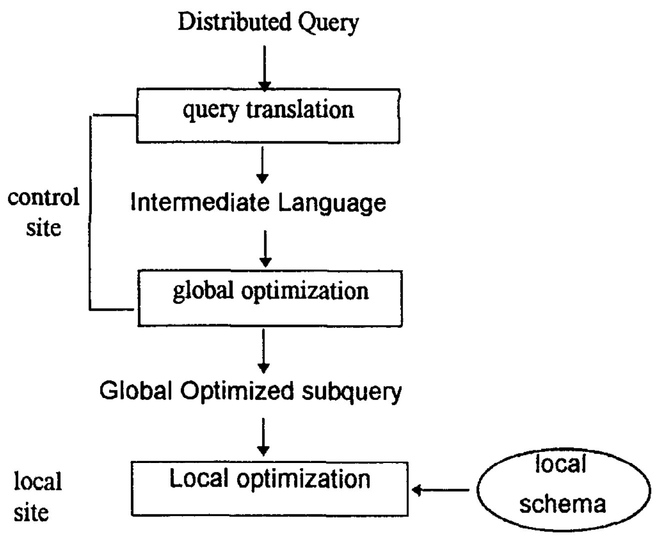

Fig. 3. Layering scheme for distributed query processing.

transaction serializability is two-phase locking (2PL) [9]. Our concurrency control algorithm, which is called *distributed strict* two phase locking algorithm, is one variation of 2PL. Here, *distributed* means that lock management duty is shared by all sites of the network, and *strict* implies that all locks hold by one transaction are released together as the transaction terminates (commits or aborts).

Table 1 shows lock compatibility followed by lock manager. It was first proposed in Gifford [10]. In general, a **read** lock prevents any other transaction from writing the locked data item. This reduces concurrency more than is necessary. Since we use intention list as the database recovery scheme, a transaction that only reads an item (acquires a **read** lock) should allow another transaction with its tentative writes (acquire a **I-write** lock) until the writing transaction commits. Thus, **read** lock is compatible with **I-write** lock. When the transaction is committing, the **I-write** lock is converted to a **commit** lock. **Commit** lock is incompatible with any other locks.

3.3. Deadlock management

As we know, any locking-based concurrency control algorithm may result in deadlock. Deadlock may occur in DITSE when transactions wait for each other. There are three well-known methods for handling deadlocks: prevention, avoidance, and detection-and-resolution. Detection-

Table 1
Lock compatibility

Acquired lock	Lock to be acquired		
	Read	I-write	Commit
None	OK	OK	OK
Read	OK	OK	WAIT
I-write	OK	WAIT	WAIT
Commit	WAIT	WAIT	WAIT

and-resolution is the most popular and best-studied management method. Detection is done by studying the global wait-for graph (GWFG). Resolution of deadlock is typically done by selecting one or more victim transaction(s) to be preempted and aborted in order to break the cycles in the GWFG. The main difficulty of deadlock detection is to gather the GWFG for the whole system. For this consideration, we use a simple deadlock avoidance method which bases on a time-out protocol. This method avoids the trouble of assigning a unique time stamp to each transaction and comparing time stamps of transactions in run-time. Because we do not use the GWFG, this method seems to abort more transactions than are actually needed to be aborted. Implementation details can be found below under 'Lock manager'.

3.4. Object naming

To reference data objects at multiple sites, each data object must be uniquely named. One way to guarantee that all names are unique is to use a name server that provides a global name space. Alternatively, the name space can be partitioned by sites so that each site can generate unique names without conflict. Although using a name server reduces site autonomy, it does increase location transparency. Without a name server, one must specify table location when he or she refers to this table. R* and ORACLE use synonyms to solve this problem [11]. DBA must create a synonym for each remote table. Even though synonyms can avoid table location to be exposed to users or application programs, DBA still have to know where the remote table is. Besides, if programmers use the OLTP library to develop their own applications, nobody can create synonyms for them. For these considerations, we use a name server to achieve location transparency.

4. Implementation

4.1. The OLTP library

The OLTP library provides distributed, transaction-semantic file access operations. It communicates with the transaction manager via socket interface. The syntax of each function provided by the OLTP library is the same as corresponding traditional UNIX file system call. So programmers who are familiar with UNIX would not feel strange. Programmers can transparently access any file spread over DTFS. When a programmer opens a file, the OLTP library use the file name and the user name to query the name server. Name server checks its database and then answers where the file is. Also, the OLTP library guarantees that all of the file operations in a transaction will be atomic and serializable. The following are the functions provided by the OLTP library:

(1)	*Open_Trans:*	open a new transaction
(2)	*Close_Trans:*	close a transaction
(3)	*t_open:*	open a file, like UNIX *open* system call
(4)	*t_close:*	close a file, like UNIX *close* system call
(4)	*t_read:*	read data, like UNIX *read* system call
(5)	*t_write:*	write data, like UNIX *write* system call
(6)	*t_lseek:*	move file pointer, like UNIX *lseek* system call

The OLTP library is implemented by means of transaction service operations. Implementation details of transaction service operation can be found in Yuan et al. [2].

4.2. Client-server database architecture

The deficiencies of file-server technology have led to the development of client-server database management system [12]. Client-server configurations attempt to make the best use of both hardware and software resources by separating functions into two parts: the front-end portion running on the client computer, and the back-end database server, which stores and manages data. Before we look at any detail of our client-server database architecture, let us review general architecture of centralized DBMSs as shown in Fig. 4. There is no general guideline about how to divide these layers. To adapt this central DBMS architecture to client-server environment, an intuitive approach is that the client handles the first two layers, and the server handles the other layers. Also, recall that in the previous section we mentioned that our system based on the OLTP library and the OLTP library is a client of the transaction manager. The architecture of our client-server database system is shown in Fig. 5.

4.3. SQL client

One of the responsibilities of an *SQL* client is handling user

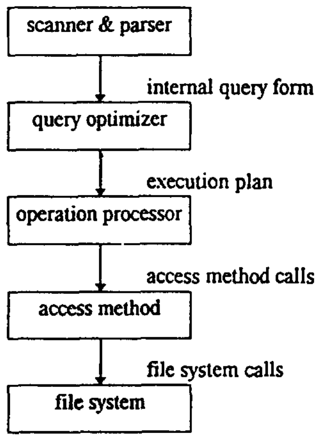

Fig. 4. Layers of traditional DBMS.

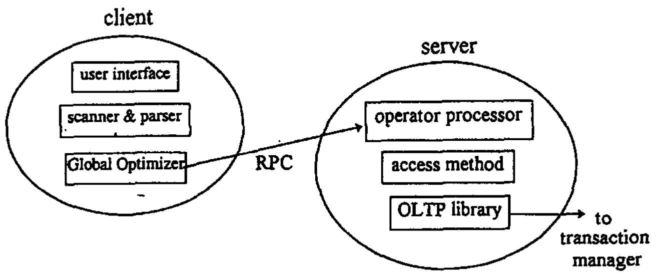

Fig. 5. Our client-server database system architecture.

interactions. We provide a graphic user interface (GUI) to interact with users. This interface is implemented by XView (OPEN LOOK toolkits for X11) [13]. Whenever GUI receives an *SQL* program submitted by the user, it first calls the scanner and parser module. The parser module will check syntax of the *SQL* program. If there is no syntax error, it translates the *SQL* program to an intermediate form to facilitate further processing by the global query optimizer.

4.4. Global query optimization algorithm

Most query optimization algorithms except R* algorithm [14], which makes an exhaustive search on all join permutations, are heuristic and heavily dependent on database statistics. Our algorithm does not use prediction statistics. The following is our algorithm.

```
begin
        /* phase 1: generate subqueries */
        for each relation R involved in the query do
          generate optimized local subquery SQ for R
          size of R = retrieve(SQ) /* retrieve is a RPC */
        endfor
        /* phase 2: find join ordering */
        while there are more than one relation do
          for each pair of relations R and S do
            calculate cost C of joining these two relations
            if cost C less than optimal cost O than O = C
          endfor
          size of result = join(R,S)
                        /* R and S with minimum join cost */
                        /* join is a RPC */
          remove(R)  /* destroy relation R and S */
          remove(S)  /* remove is a RPC */
        endwhile
        /* phase 3: retrieve final result */
        copy(final result)
        remove(final result)
end
```

The algorithm contains three phases. Phase 1 generates globally optimized subqueries. Phase 2 tries to find a join ordering which has minimum network transmission cost. Phase 3, of course, retrieves final query result.

This algorithm is a greedy algorithm. It is similar to SDD-1 algorithm [15]. However, there is one important difference. SDD-1 algorithm is static but our algorithm is dynamic. Optimization can be done dynamically as the query is executed or statically before executing the query. At any point of execution, dynamic optimization can choose the best next operation based on accurate knowledge of the results of the operation executed previously. Therefore, database statistics are not needed to estimate the size of intermediate results, thereby minimizing the probability of a bad choice. Of course, dynamic optimization is not perfect. The main shortcoming is that query optimization, an expensive task, must be repeated for each execution of the query.

4.5. Example

Here, we use a simple example to illustrate the whole flow of the query processing. In this example, we consider three relations, namely, relation S, P, and SP, located at site S1, S2, and S3 respectively. Relation S has three attributes, s#, sname, and city. Relation P has two attributes: p# and pname. Relation SP has three attributes: s#, p# and quality. We assume that an *SQL* query is submitted at site S4 as follows:

```
select S.sname, P.pname, SP.quality
from S,P,SP
where S.s# = SP.s# and SP.p# = P.p# and
S.city = 'London' and SP.quality > 100
```

The parser module in the *SQL* client translates the query into an intermediate language and then passes it to the global optimizer module. The intermediate language is as follows:

```
ATTRIBUTES
    S.sname
    P.pname
    SP.quality
CONDITIONS
    S.city = 'London'
    SP.quality > 100
JOIN
    S.s# = SP.s#
    SP.p# = P.p#
```

There are three phases involved in the optimization process. In the first phase, the optimizer sends three optimized subqueries, SQ1, SQ2 and SQ3, to site 1, site 2, and site 3 respectively (via remote procedure calls). These subqueries are as follows:

```
SQ1:
select S.sname, S.s#
from S
where S.city = 'London';

SQ2:
select P.pname, P.p#
from P;

SQ3:
select SP.quality, SP.s#, SP.p#
from SP
where SP.quality > 100;
```

In the second phase, the optimizer tries to find an 'optimal' join ordering which minimizes data transfer cost. In the first round, it chooses to join relation SP and S into one relation, say RESULT0. So it sends the following commands (via RPC):

```
join SP and S into RESULT0              --> site 3
remove SP                               --> site 3
remove S                                --> site 1
```

In the second round, it chooses to join RESULT0 and P into relation RESULT1. Again, it sends the following commands:

```
join RESULT0 and P into RESULT1         --> site 3
remove RESULT0                          --> site 3
remove P                                --> site 2
```

Finally, it copies RESULT1 from site 3 and then removes it:

```
copy RESULT1                            --> site 3
remove RESULT1                          --> site 3
```

4.6. Callback procedure

Although our query optimization algorithm can find the best join ordering based on the accurate table size gathered from phase 1, it is quite time consuming. This is because the remote procedure calls are synchronous. To get the accurate table sizes, the optimizer must block to wait for the *retrieve* remote procedure returns. That means a subquery cannot be issued before the previous one has completed. The synchronous behaviour leads phase 1 to be time-consuming, especially when retrieving a large relation. So, we must try to execute subqueries in parallel.

To execute subqueries in parallel, the optimizer cannot wait to hear back from the server. To deal with this, we use callback procedures. Upon receiving a *retrieve* request, an *SQL* server first acknowledges the request and then uses callback procedure to reply after it really completes the task. Callback procedure mechanism changes the role of clients and servers temporarily. After the optimizer issues all subqueries, it changes its role from the client to the server, and then waits for callback procedures. Via callback procedures, waiting time of phase 1 becomes the maximal executing time among all subqueries, not the sum of them.

4.7. SQL server

There are two stages to implement an *SQL* server. First, we develop an RPC server with Sun RPC [16]. Second, we use Sun lightweight process [17] to develop a multithread version of the RPC server.

The RPC server provides two categories of remote procedure calls. One category of procedures handles data definition operations. The others handle data manipulation operations. Data definition remote procedure calls include:

create__rel:	create a new table
drop__rel:	drop a table

Data manipulation remote procedure calls include:

retrieve:	retrieve a portion of the relation
remove:	destroy the relation
join:	join two relations
copy:	copy the relation from one site to the other site
update:	update specific tuples
insert:	insert one tuple into the relation
delete:	delete the tuple from the relation

Standard SUN remote procedure call paradigm is synchronous. In other words, a server can only handle one request at a time. During the server handling a request of one client, the other clients must wait (if it issues a request at this moment). However, a database server must handle a client's request concurrently (asynchronous). There are two ways to solve this problem. The first is using the UNIX fork system call. Each time when a server gets a request, it forks a child process to handle this request. The parent process then keeps waiting for other requests. This method is simple but not suitable for a database server. A database server normally contains some global information in its address space. If it forks new processes, the consistency of global information between these processes must be maintained by other system facilities, such as shared memory IPC. The better way is to let parent and child processes share some portion of address space. This consideration leads us to use threads.

We use Sun lightweight process library to implement our multithread *SQL* server. Sun lightweight processes (threads) operate more efficiently than ordinary SunOS processes because threads communicate via shared memory instead of a file system. Because threads can share a common address space, the cost of creating tasks and inter-task communication is substantially less than the cost of using more heavyweight primitives.

Two alternatives to manage threads are possible: static threads and dynamic threads. With a static design, the choice of how many threads needed can be made either when the program is written or when it is compiled. This approach is simple, but inflexible. A more flexible approach is to allow threads to be created and destroyed during the execution time. Thus, we choose to use the dynamic approach.

The structure of our multithread server is shown in Fig. 6. Whenever a multithread server boots up, it first creates two threads: scheduler and daemon thread. Scheduling in Sun lightweight process is by default, priority-based, nonpreemptive within a priority. But we want each thread equally to share CPU time. So we must use round-robin scheduling. Fortunately, Sun lightweight process provides sufficient primitives to do this work. Scheduler thread owns the highest priority. It sets up a timer and then goes to sleep. When it is sleeping, a lower priority thread can be executed. When the timer expires, it reshuffles the queue of time sliced threads which are at a lower priority and then goes to sleep again. Daemon thread is a watchdog waiting for the request of *SQL* clients. If an *SQL* client makes a remote procedure call, daemon thread will dynamically create a new thread which executes the specific procedure. When this procedure eventually returns, the thread kills itself.

Because there are multiple threads executing concurrently, we must have a synchronization mechanism. Sun lightweight process library provides monitor mechanism as the synchronization mechanism. All of the global information, such as system catalog, must be protected by a monitor.

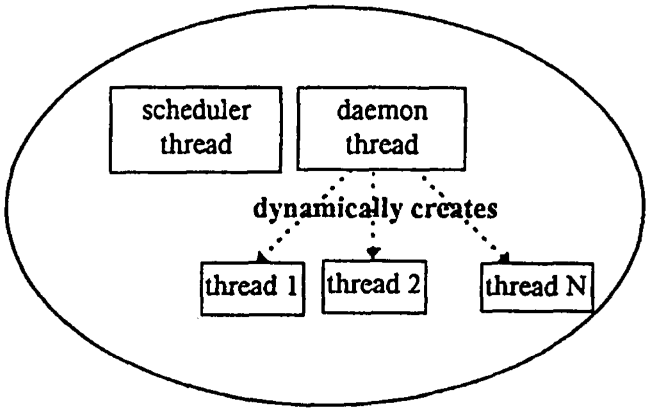

Fig. 6. The multithread SQL server.

The main problem of Sun lightweight process library is that threads lack kernel supports, so system calls are handled serially. Although the non-blocking I/O library mitigates this problem somewhat. The performance of a multithread server is still not very good. In the future, the *SQL* server will be ported on Sun Solaris operating system which has kernel supported threads.

4.8. Lock manager

We have already mentioned that we use a simple deadlock avoidance method which is based on a time-out protocol. The basic idea is that each transaction can at most own a lock for a default period of time. After the timer expires, the transaction must be aborted mandatorily. Since circular waiting is impossible, deadlock never happens. Obviously, DITSE cannot support long-lived transaction [18] because a transaction cannot own the lock for a long period of time.

Before looking at the scenario of the lock manager, we first introduce two special data structures: data item waiting queue (DIWQ) and lock manager waiting queue (LMWQ). Each locked data item has its own DIWQ. DIWQ element records transaction identifier and lock type which the transaction acquires. Each lock server has one LMWQ. LMWQ element records transaction identifier, how long to wake up the transaction (Time__to__WakeUp), and some information of lock request which the transaction makes.

When a transaction cannot acquire a lock on a data item, the LM performs the following actions:

(1) Puts the transaction to the data item's DIWQ.
(2) Calculates Time__to__WakeUp.
(3) Puts the transaction to LMWQ.
(4) Sets an alarm clock according to Time__to__WakeUp of the transaction.

In action (3), Time__to__WakeUp is calculated by the formula:

$$\mathit{Time\_\_to\_\_WakeUp} = \mathit{Lock\_\_Time} + \mathit{Time\_\_Out} - \mathit{current\ time}.$$

We use an example to explain this formula. If a transaction *A* cannot acquire a lock, there must be another transaction *B* that has acquired an incompatible lock before transaction *A*. *Lock__Time* indicates when transaction *B* acquires the lock. *Time__Out* is a default value which specifies the longest time that a transaction can own a lock. *Current time* is time that we are calculating this formula. From the above description, you can see that *Time__to__WakeUP* is how long to wake up transaction *A* since it was put into LMWQ.

You may be curious about when to invoke the transactions in either DIWQ or LMWQ. There are two possibilities. First, when a transaction releases the lock, the lock manager checks the data item's DIWQ. If DIWQ is empty, nothing will be done. Otherwise, lock manager will wake up one or more transactions in DIWQ. Second, when the alarm clock expires, the transaction in LMWQ will be woken up. Note that no matter in which case, the awaken transaction can acquire the lock successfully. Therefore, our time-out protocol not only ensures no deadlock but also ensures no starvation.

5. Summary and future work

In this paper, we have given an overview of the DITSE. It is a distributed database system which was built on top of a distributed transactional file system, not directly on top of a traditional file system. A global query optimization algorithm which does not rely on prediction of join selectivity is also proposed. We follow the client-server computing model to build an autonomous multithread *SQL* server. Keeping *SQL* server's autonomy has an advantage that DITSE can dynamically integrate other *SQL* servers. All communication links between *SQL* client and *SQL* server are implemented with Sun remote procedure call.

There are still a lot of efforts that should be made in the future. The most important thing is performance enhancement of transaction manager and *SQL* server. The *SQL* server currently lacks fast access methods and buffer management. We believe that adding this stuff can greatly enhance performance. Also, a PC version of *SQL* client should be provided.

In recent years, there have been ever-increasing DBMS vendors supporting Microsoft ODBC (Open Database Connectivity). The ODBC interface is a C-programming language interface for database connectivity. It provides a common API (Application Programming Interface) which allows applications to access diverse DBMSs. In the future, we will use the ODBC to implement our database agent. There is then no need to develop a new database agent in order to integrate other DBMSs. For different DBMSs, we just need to link our database agent with different ODBC drivers supported by database vendors.

References

[1] Stonebraker, M *Readings in database systems* Morgan-Kaufmann (1994) pp 507–509

[2] Yuan, S-M, Lin, J H and Ni, C-Y 'The design and implementation of a distributed transaction processing system' *J. Inf. Science and Eng.* Vol 9 No 4 (1993) pp 1–22

[3] Andrade, J M, Carges, T and MacBlane, M 'Open on-line transaction processing with the TUXEDO system' *IEEE Computer Society Int. Conf.: COMPCON-92* San Francisco, CA, USA (1992) pp 366–371

[4] Ozsu, M T and Valduriez, P *Principles of distributed database systems* Prentice-Hall (1991)

[5] Sheth, P and Larson, A 'Federated database systems for managing distributed heterogeneous, and autonomous databases' *ACM Comp. Surv.* Vol 22 No 3 (September 1990)

[6] Templeton, M et al. 'Mermaid–a front-end to distributed heterogeneous databases' *Proc. of the IEEE* Vol 75 No 5 (May 1987) pp 695–707

[7] Cardenas, A F 'Heterogeneous distributed database management:

the HD-DBMS' *Proc. of the IEEE* Vol 75 No 5 (May 1987) pp 588–600

[8] Bernstein, P A and Chiu, D M 'Using semi-joins to solve relational queries' *J. ACM* Vol 28 No 1 (January 1981) pp 25–40

[9] Eswaran et al. 'The notions of consistency and predicate locks in a database system' *Comm. ACM,* Vol 19 No 11 (November 1976) pp 624–633

[10] Gifford, D K 'Violet: an experimental decentralized system' *ACM Operating Systems Review* Vol 13 No 5 (1979)

[11] Khoshafian, S et al. *A guide to developing client/server SQL applications* Morgan-Kaufmann (1992)

[12] DeWitt, S et al. 'A study of three alternative workstation-server architectures for object-oriented database systems' *16th Int. Conf. on Very Large Data Bases* Brisbane Australia (1990) pp 107–121

[13] Heller, S *XView Programming Manual* Op'Reilly & Associates (1990)

[14] Kim, W, Reiner, S and Batory, D S *Query Processing in Database Systems* Springer-Verlag (1985)

[15] Bernstein, P A et al. 'Query processing in a system for distributed database (SDD-1)' *ACM Trans. Database System* Vol 6 No 4 (December 1981) pp 602–625

[16] *Network programming guide* Sun Microsystems (1990)

[17] *Programming utilities and libraries* Sun Microsystems (1990)

[18] Gray, J and Reuter, A *Transaction processing: concepts and techniques* Morgan-Kaufmann (1993) pp 210–219