Office de la Propriété Intellectuelle du Canada
Un organisme d'Industrie Canada

Canadian Intellectual Property Office
An agency of Industry Canada

CA 2484694 A1 2006/04/14

(21) 2 484 694

(12) DEMANDE DE BREVET CANADIEN
CANADIAN PATENT APPLICATION

(13) A1

(22) Date de dépôt/Filing Date: 2004/10/14

(41) Mise à la disp. pub./Open to Public Insp.: 2006/04/14

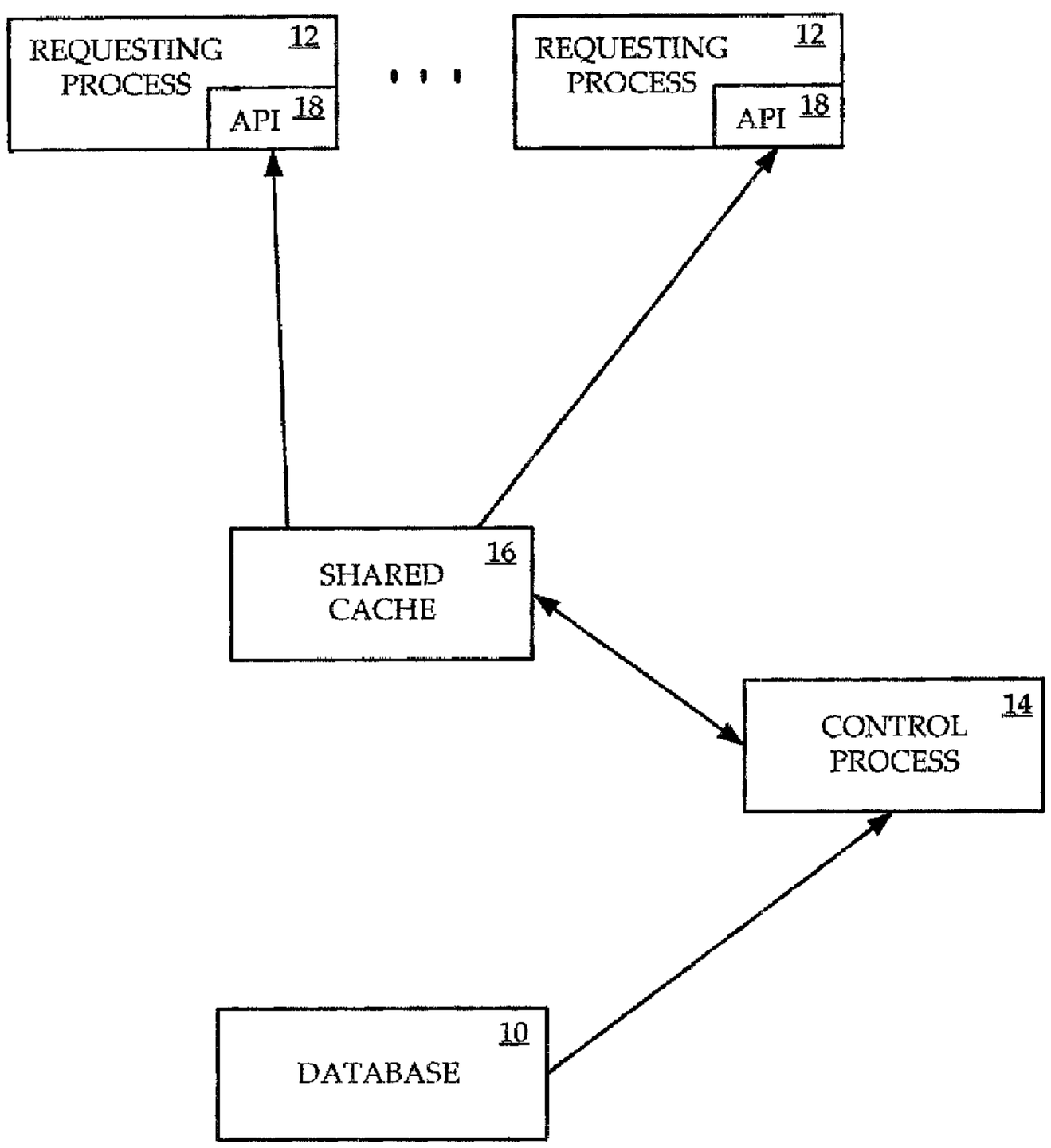
(51) Cl.Int./Int.Cl. *G06F 12/02* (2006.01), *G06F 17/30* (2006.01), *G06F 13/38* (2006.01)

(71) Demandeur/Applicant:
ALCATEL, FR

(72) Inventeurs/Inventors:
PIPER, RICH, CA;
PILON, MARK, CA;
LANDRY, FELIX, CA

(74) Agent: MARKS & CLERK

(54) Titre : MEMOIRE CACHE RAM POUR BASE DE DONNEES
(54) Title: DATABASE RAM CACHE



(57) Abrégé/Abstract:
A system and method are provided for providing a shared RAM cache of a database, accessible by multiple processes. By sharing a single cache rather than local copies of the database, memory is saved and synchronization of data accessed by different

CA 2484694 A1 2006/04/14

(21) 2 484 694

(13) A1

(57) Abrégé(suite)/Abstract(continued):
processes is assured. Synchronization between the database and the shared cache is assured by using a unidirectional notification mechanism between the database and the shared cache. Client APIs within the processes search the data within the shared cache directly, rather than by making a request to a database server. Therefore server load is not affected by the number of requesting applications and data fetch time is not affected by Inter-Process Communication delay or by additional context switching. A new synchronization scheme allows multiple processes to be used in building and maintaining the cache, greatly reducing start up time.

Abstract

A system and method are provided for providing a shared RAM cache of a database, accessible by multiple processes. By sharing a single cache rather than local copies of the database, memory is saved and synchronization of data accessed by different processes is assured. Synchronization between the database and the shared cache is assured by using a unidirectional notification mechanism between the database and the shared cache. Client APIs within the processes search the data within the shared cache directly, rather than by making a request to a database server. Therefore server load is not affected by the number of requesting applications and data fetch time is not affected by Inter-Process Communication delay or by additional context switching. A new synchronization scheme allows multiple processes to be used in building and maintaining the cache, greatly reducing start up time.

137988

DATABASE RAM CACHE

Field of the invention

[01] The invention relates to database caches, and more particularly to the use of shared caches in multi-threaded processes.

Background of the invention

[02] In multi-process applications which require access to a shared database, a requesting process requiring database access makes a request to a central memory sharing process, such as a database server. The database server retrieves the required data and copies it to an Inter-Process Communication (IPC) mechanism, from where the requesting process can access the data. However, this requires synchronization between the database server and the requesting processes, which leads to delays and time inefficiencies.

[03] One solution would be to have each requesting process cache its own copy of the database. Although memory intensive, this approach may be feasible for small databases. However for large databases, such as on the order of 4 GB as is typical for network management systems, this approach is clearly not realistic.

Summary of the invention

[04] In accordance with one aspect of the invention, a method is provided for providing multiple processes with access to data stored in a database. Data is copied from the database to a cache in shared memory, the shared memory being accessible by each process. Synchronicity between the database and the cache is maintained. Each process is provided with an Application Program Interface (API) containing instructions for accessing the data within the cache.

[05] In accordance with another aspect of the invention, a cache is provided for storing data from a database storing tables. The cache has a data structure which includes, for each table, at least one linked list of data segments storing data from the table. For each table, the data structure includes a control segment storing an address of the first data segment in each linked list. The data structure includes a master segment storing an address for each control segment.

[06] In accordance with yet another aspect of the invention, a method is provided for providing job synchronization. For each job, the job is divided into tasks. The tasks are grouped into one or more task groups, each task group being a single operation for synchronization purposes and comprising at least one of the tasks. Each of the tasks is executed on a worker thread belonging to a specific thread server pool. All waiting client applications are notified of completion of the job only upon completion of the last task of the job.

[07] Apparatus are provided for carrying out the methods of the invention. The methods may be stored in the form of software instructions stored on computer-readable media.

[08] The methods and apparatus of the present invention allow multiple processes to access a common database with minimal memory usage, while also keeping synchronization times to a minimum. By sharing a single cache, memory is saved and synchronization of data accessed by different processes is assured. Synchronization between the database and the shared cache is assured by using a unidirectional notification mechanism between the database and the shared cache. Since the APIs searches the data within the shared cache directly, rather than by making a request to a database server, the server load is not affected by the number of requesting applications, and data fetch time is not affected by Inter-Process Communication delay or by additional context switching. A new synchronization scheme allows multiple processors to be used in building and maintaining the cache, greatly reducing start up time.

137988

Brief description of the drawings

[09] The features and advantages of the invention will become more apparent from the following detailed description of the preferred embodiment(s) with reference to the attached figures, wherein:

FIG. 1 is a block diagram of a shared RAM cache system according to one embodiment of the invention;

FIG. 2 is a block diagram of messaging during the maintenance of the shared RAM cache of FIG. 1 according to one embodiment of the invention;

FIG. 3 is a flowchart of a method by which the worker threads of FIG. 2 maintain the shared cache according to one embodiment of the invention;

FIG. 4 is a block diagram of objects used for synchronizing requests according to one embodiment of the invention;

FIG. 5 is a block diagram of messaging during creation of a notification group;

FIG. 6 is a block diagram of messaging during creation of a request; and

FIG. 7 is a block diagram of messaging when a request is completed.

[10] It will be noted that in the attached figures, like features bear similar labels.

Detailed description of the embodiments

[11] Referring to FIG. 1, an example system of implementing a shared RAM cache according to one embodiment of the invention is shown. A database 10 stores data which is of interest to one or more requesting processes 12. A control process 14 has read-write access to the database 10. The control process 14 also has read-write access to a shared cache 16 stored in RAM. Each

requesting process 12 has an API 18, which has read-only access to the shared RAM cache 16. Broadly, in operation the shared cache 16 stores copies of data stored in the database 10, for direct access by each of the requesting processes 12 through their respective APIs 18. Creation of the shared cache 16 and maintenance of the shared cache 16 (synchronization between the shared cache and the database 10) is carried out by the control process 14.

[12] When the control process 14 is first turned on or created, the control process 14 determines whether the shared cache 16 exists. If the shared cache 16 does not exist, the control process 14 creates one or more worker threads, which copy the contents of the database to the shared cache 16. The worker threads preferably use the synchronization scheme described below, in order to accelerate the build time of the shared cache 16.

[13] The shared cache 16 is structured in a way that allows the worker threads to build the shared cache 16 in parallel. The control process 14 creates a master segment within the shared cache 16. The master segment contains a plurality of entries. Each master segment entry stores the location of a different control segment within the shared cache 16, each control segment corresponding to a unique table within the database 10. Each control segment contains at least one entry. Each control segment entry stores the location of a different first data segment, each first data segment corresponding to a unique worker thread. When a worker thread responsible for particular data within a particular table within the database 10 begins copying data from the table to the shared cache 16, the worker thread is therefore able to locate the first data segment to which the data is to be copied. Any first data segment may have as its last entry the address of a second data segment, to which the corresponding worker thread copies data once the first data segment is full. This may be repeated, until a data segment stores an invalid address in its last entry. In effect, each control segment lists the starting address of a linked list of data segments to be filled by a corresponding worker thread with data from the table corresponding to the control segment.

[14] Referring to FIG. 2, messaging during synchronization of the database 10 and the shared cache 16 is shown according to one embodiment of the invention. The database server includes a function for initializing an IPC for communicating with the control process 14. Each table within the database 10 includes a trigger. When the contents of a table within the database 10 are altered, by adding a record, deleting a record, or updating a record, the trigger is activated and a stored procedure within the database server of the database 10 is called. The stored procedure creates a message 30, and appends the message 30 to a first queue 32 of messages. The message 30 includes a table identifier (ID), a row ID, and an action. The message is preferably three integers in length, one integer for each of the three fields. The action is a flag signifying one of "addition", "deletion", and "update".

[15] The control process 14 includes a controller 34 and preferably at least one redundancy 36 for the controller 34. If the controller 34 fails, the redundancy 36 takes over the actions of the controller described below. The controller 34 listens to the IPC, created by the database server, for database table updates in the form of messages from the first queue 32. For each message, the controller 34 determines the table ID and the row ID. If the action is a "deletion" or an "update", the controller 34 sets a row status flag 38 to the appropriate table and row within the shared cache 16, thereby locking the row and preventing any of the requesting processes 12 from accessing any record which has just been updated or deleted. The controller 34 then forwards the message to a second queue 40 of messages within the control process 14, and deletes the message from the first queue 32.

[16] The control process 14 also includes at least one worker thread 42. The worker threads form a multi-thread process responsible for the actual update of the shared cache 16. Each worker 42 thread reads messages from the second queue 40, using a synchronization scheme described below. Referring to FIG. 3, a flowchart of a method by which a worker thread 42 updates the shared cache 16 according to one embodiment of the invention is shown. The worker thread

42 reads a message from the second queue 40, then at step 50 the worker thread determines the action of the message. If at step 52 the worker thread determines that the action is a "deletion", then at step 54 the worker thread consults the database 10 to verify that the row identified by the row ID and the table ID within the message has actually been deleted from the database. If the deletion is verified, then at step 56 the worker thread deletes the corresponding row from the shared cache 16. If the deletion is not verified at step 54, then at step 58 the worker thread discards the message and creates an error report, and changes the row status flag to a value signifying that the row is to audited. The row remains accessible to requesting processes, but will be re-examined for possible deletion by an audit portion of the control process (not shown in FIG. 2).

[17] If at step 52 the worker thread determines that the action is not a "deletion", then at step 60 the worker thread determines whether the action is an "addition". If the action is an "addition", then at step 62 the worker thread consults the database 10 to verify that the row identified by the row ID and the table ID within the message has actually been added to the database. If the addition is verified, then at step 64 the worker thread copies the contents of the row within the database 10 to the shared cache 16. If the addition is not verified at step 62, then at step 65 the worker thread copies the contents of the row to the shared cache 16, but discards the message and creates an error report at step 66 and marks the row as to be audited as in step 58.

[18] If at step 60 the worker thread determines that the action is not an "addition", then at step 70 the worker thread determines whether the action is an "update". If the action is an "update", then at step 72 the worker thread consults the database 10 to verify that the row identified by the row ID and the table ID within the message has actually been updated. If the update is verified, then at step 74 the worker thread copies the contents of the row within the database 10 to the shared cache 16 in order to update the shared cache 16. At step 75 the worker thread resets the row status flag to remove the rowlock that

had been set by the controller 34. If the update is not verified at step 72, then at step 76 the worker thread discards the message and creates an error report, and marks the row as to be audited as in step 58.

[19] If at step 70 the worker thread determines that the action is not an "update", then the action is of an unknown type. The worker thread generates an error report at step 80, and discards the message.

[20] The shared cache 16 has no equivalent to a database server. Rather, the API 18 within each requesting process 12 accesses data within the shared cache 16 directly. Each API 18 accesses the data in the shared memory segments of the shared cache 16 through the master segment and does not directly interact with the server process. If the row status flag of a row being accessed by an API 18 is marked as update pending or deletion pending, the API 18 does not retrieve the data for the row. Synchronization is thereby achieved by use of row status flags, set by the controller 34 (or the redundancy 36) when a row is being updated or deleted and by the worker threads 42 once the update or deletion is verified.

[21] In order to accelerate creation and maintenance of the shared cache by the worker threads, a synchronization schemed described with reference to FIG. 4 to FIG. 7 is used. Referring to FIG. 4, a block diagram of objects used for synchronizing requests according to one embodiment of the invention is shown. An application thread 90 communicates with a thread server manager 92. The application thread 90 sends requests for notification groups and task requests to the thread server manager 92. In the case of building and maintaining the shared cache 16, the application thread would be an application within the control process 14, such as the controller 34. The thread server manager 92 is responsible for creating and destroying one or more worker threads 94, which may be grouped into one or more worker thread groups 96. The thread server manager 92 also serves as the worker server main command loop. Although only one application thread 90 is shown in FIG. 4, it is to be understood that

there will generally be more than one application thread 90 requesting task requests and notification groups.

[22] The thread server manager 92 sends requests for task requests to a request manager 100. The request manager 100 creates task requests 102, which are sent to the work threads 94 on internal request manager queues 104, one request manager queue 104 per worker thread group 96. The worker threads 94 continuously attempt to read task requests 102 from the corresponding request manager queue for execution, access to the each request manager queue being restricted by use of counting semaphores to prevent access when the request manager queue is empty.

[23] The thread server manager 92 also sends requests for notification groups (described in more detail below) to a notification manager 110. The notification manager 110 is responsible for task synchronization, and creates and deletes notification groups. The notification manager 110 stores a global list 112 of all open and/or executing notification group objects 114 which it has created. The notification manager 110 also stores one or more thread specific lists 116 of handles 118 to the notification group objects 114. Each thread specific list 116 lists handles 118 to notification group objects for a particular application thread.

[24] Referring to FIG. 5, a block diagram of messaging when requesting a notification group according to one embodiment of the invention is shown. The application thread 90 sends a request for a notification group to the thread sever manager 92. The thread server manager 92 forwards the request to the notification manager 110. In response to the request, the notification manager 110 creates a notification group object (NGO) 114. The NGO 114 allows the application thread 90 to be informed when all the tasks that the application thread 90 has requested (between creation of the NGO and the NGO being set to inactive) have been completed. The NGO is an object-oriented programming object, and includes as values a group status, a request count, a completed

request count, a blocked status, and a condition variable used for blocking application threads.

[25] The group status has a value of 'active' if the notification group is still accepting requests, and the group can not complete until the group status is set to 'inactive'. The group status is set to 'inactive' when the application thread 90 closes the group via the handle of the NGO. When the NGO 114 is first created, the group status is set to 'active'.

[26] The request count is a count of the number of requests that have been added to the group since the NGO was created. The initial value of the request count is '0'. The completed request count is a count of the number of requests that have been added to the group and that have been completed. The initial value of the completed request count is '0'.

[27] The blocked status is used to bypass the condition wait if the notification group does not currently warrant being blocked. This may arise only if the group status is set to 'inactive', and either no requests have been made to the group or all requests have been completed. The initial value of the blocked status is 'blocked'. The blocking condition variable is used in a condition wait call. The blocking condition variable will block all threads that perform a condition wait on the variable. This block will continue until a signal or a broadcast is performed on the variable. A blocked status of 'blocked, signifies that the group has not completed, and any thread that blocks on this group will not bypass the condition variable block. Once a thread reaches the condition variable, it is blocked until it receives a continue signal from another thread. The block status also acts as a guard that will not allow the thread to continue, even if released by signal, until the blocking status is 'unblocked'. This prevents inadvertent release of the condition variable. However, once the group status is closed and the number of completed requests equals the number of sent requests (the request count equals the completed request count), it should be impossible to reach a blocked state. Under these conditions the blocked status

prevents the thread from blocking on the condition variable and never being woken up.

[28] More particularly, an application thread makes a call to wait until a specific NGO completes. If the NGO has not already completed then the internal routine in the NGO calls a function pthread_cond_wait(), passing in the address of the condition variable that exists within the NGO. This effectively blocks the execution of the application thread until another thread calls the function pthread_cond_signal(), which releases a single application thread that is blocked on the argument condition variable, or the function pthread_condition_broadcast(), which releases all application threads blocked on the argument condition variable. Both pthread_cond_signal() and pthread_cond_broadcast() are atomic operations and do not affect the functionality of subsequent calls to pthread_cond_wait() using the same argument variable.

[29] Since a thread should not be blocked on a group that has already completed, the blocked flag variable is used to bypass the call to pthread_cond_wait() during the time that an NGO has completed but has not yet been destroyed. Once the number of completed requests equals the number of requests made and the NGO status is 'inactive', the blocked flag is set to 'unblocked' and a call is made to pthread_cond_broadcast() using the condition variable of the NGO to free up any threads that are blocked on the NGO.

[30] Once the notification manager 110 has created the NGO 114, the notification manager inserts the NGO 114 into the global list 112. The notification manager also copies a handle to the NGO 114 and places the handle in the thread specific list 116 corresponding to the application thread 90. The handle entry can then only be removed by the application thread 90 calling to close the NGO 114. No other thread can add requests to or close the NGO.

[31] Referring to FIG. 6, a block diagram of messaging when requesting a task according to one embodiment of the invention is shown. The application thread

90 sends a request for a task to the thread sever manager 92. The request for a task includes an address of a client-defined procedure and an address of any parameters required by the client-defined procedure. The thread server manager 92 forwards the request to the request manager 100. In response to the request, the request manager 110 creates a new request object 102. Upon creation, the request object 102 calls the notification manager 110, which increments the request count of the open NGOs 114 identified by the handles 118 within the thread specific list 116 corresponding to the application thread 90. The notification manager 110 returns a copy of the list of handles to open NGOs in the thread specific list 116 of the application thread 90 to the newly created request object 102.

[32] In addition to the list of handles to open NGOs, the request object 102 also includes the address of the client-defined routine that is to be executed by a worker thread, and the address of arguments used by the routine. The request manager 100 then inserts the request object 102 into a request manager queue 104 corresponding to the worker thread group associated with the application thread, and increments the semaphore of the request manager queue.

[33] Referring to FIG. 7, a block diagram of messaging when a task completes according to one embodiment of the invention is shown. When a task completes, the request manager 100 deletes the request object 102 which contained the task (or more precisely, contained the address of the routine that was to be executed) terminates. The destructor function of the request object 102 notifies the notification manager 110 that the task request has been completed and provides the notification manager with the list of handles of open NGOs. The notification manager 110 then increments the completed request count in each of the NGOs having handles in the list provided by the request object 102. Since the list of handles contained in the request object 102 is a copy of the list of handles that existed at the time the request object was created, any NGOs created after the request object was created will not have their completed request count incremented.

137988

[34] If after incrementing the completed request count of an NGO the incremented request count equals the request count, the NGO may be deleted by the notification manager. The NGO is deleted if no threads are blocked on the NGO, the NGO has a status of 'inactive', and the number of completed requests equals the number of requests. Once these conditions are met, the NGO will be deleted by the thread that caused the conditions to be met.

[35] In the preferred embodiment, the control process 14, the at least one API 18, and the triggered functions of the database and are in the form of software within a processor, but may more generally be in the form of any combination of software or hardware, including hardware within an integrated circuit. The processor need not be a single device, but rather the instructions could be located in more than one device.

[36] The invention has been described using a particular task synchronization scheme. The invention can be use other synchronization schemes while still realizing the benefit of reduced memory usage and faster fetch times, but building of the cache will be slower.

[37] The synchronization scheme of the invention has been described as being used within the control process during building of the cache. The synchronization scheme can also be used by worker threads during maintenance of the cache, or even by multi-thread processes within the requesting processes. In fact, any multi-thread process can benefit from the faster execution time provided by the synchronization scheme described herein.

[38] The embodiments presented are exemplary only and persons skilled in the art would appreciate that variations to the embodiments described above may be made without departing from the spirit of the invention. Methods that are logically equivalent or similar to the method described above with reference

to FIG. 3 may be used to implement the methods of the invention. The scope of the invention is solely defined by the appended claims.

137988

I/WE CLAIM:

1. A method of providing a plurality of processes with access to data stored in a database, comprising:

copying the data from the database to a cache in shared memory, the shared memory being accessible by each process;

maintaining synchronicity between the database and the cache; and

providing each process with an Application Program Interface (API) containing instructions for accessing the data within the cache.

2. A system for carrying out the method of claim 1.

3. A computer-readable medium storing processor instructions for carrying out the method of claim 1.

4. A cache for storing data from a database storing a plurality of tables, the cache having a data structure comprising:

for each table, at least one linked list of data segments storing data from the table;

for each table, a control segment storing an address of the first data segment in each linked list; and

a master segment storing an address for each control segment.

5. A method of providing job synchronization comprising, for each job:

dividing the job into a plurality of tasks;

grouping the tasks into at least one task group, each task group being a single operation for synchronization purposes and comprising at least one of the tasks;

executing each of the tasks on a worker thread belonging to a specific thread server pool; and

notifying all waiting client applications of completion of the job only upon completion of the last task of the job.

6. A server for carrying out the method of claim 5.

7. A computer-readable medium storing processor instructions for carrying out the method of claim 5.

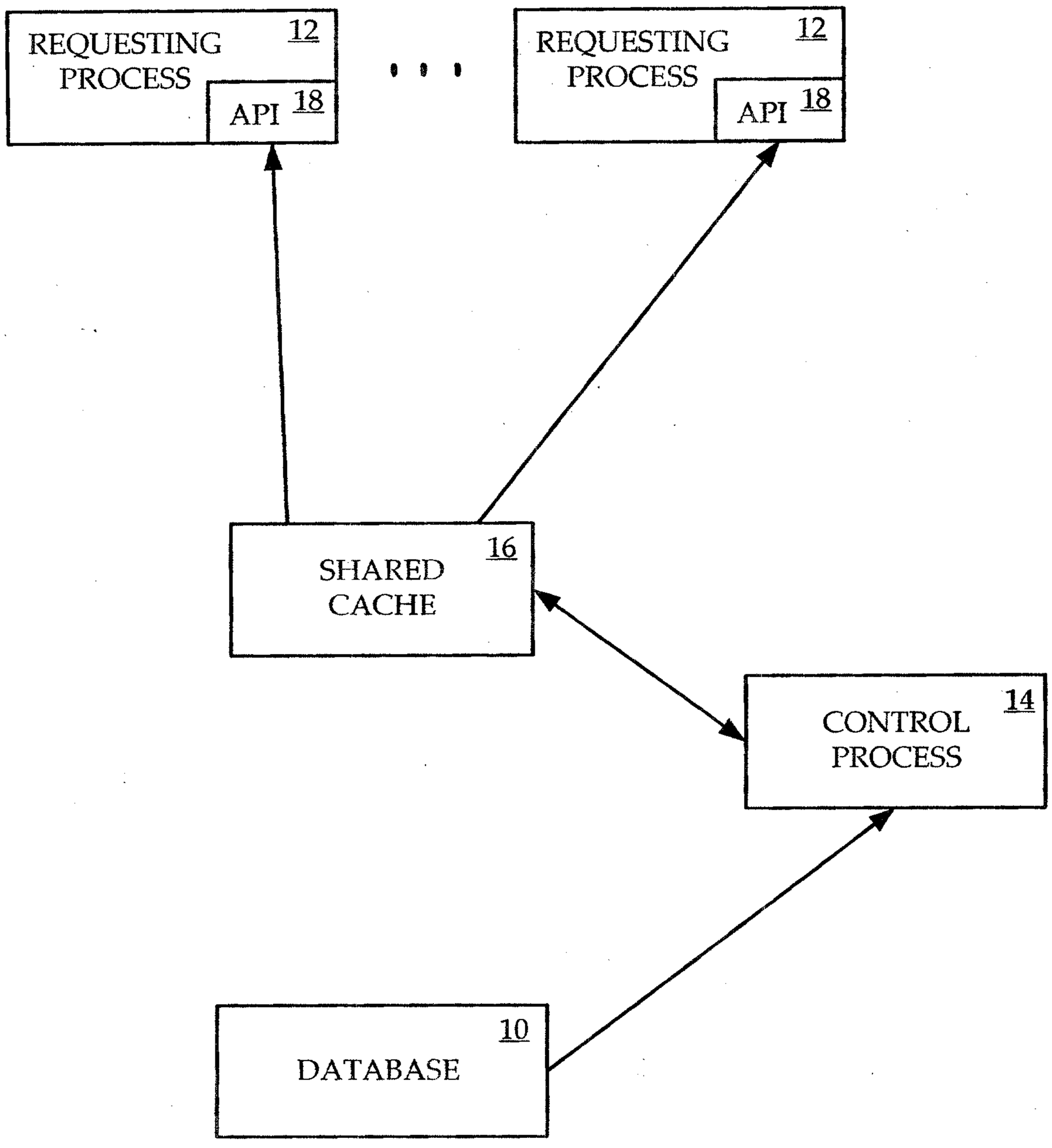

REQUESTING PROCESS
12
API
18
REQUESTING PROCESS
12
API
18
SHARED CACHE
16
CONTROL PROCESS
14
DATABASE
10

FIG. 1

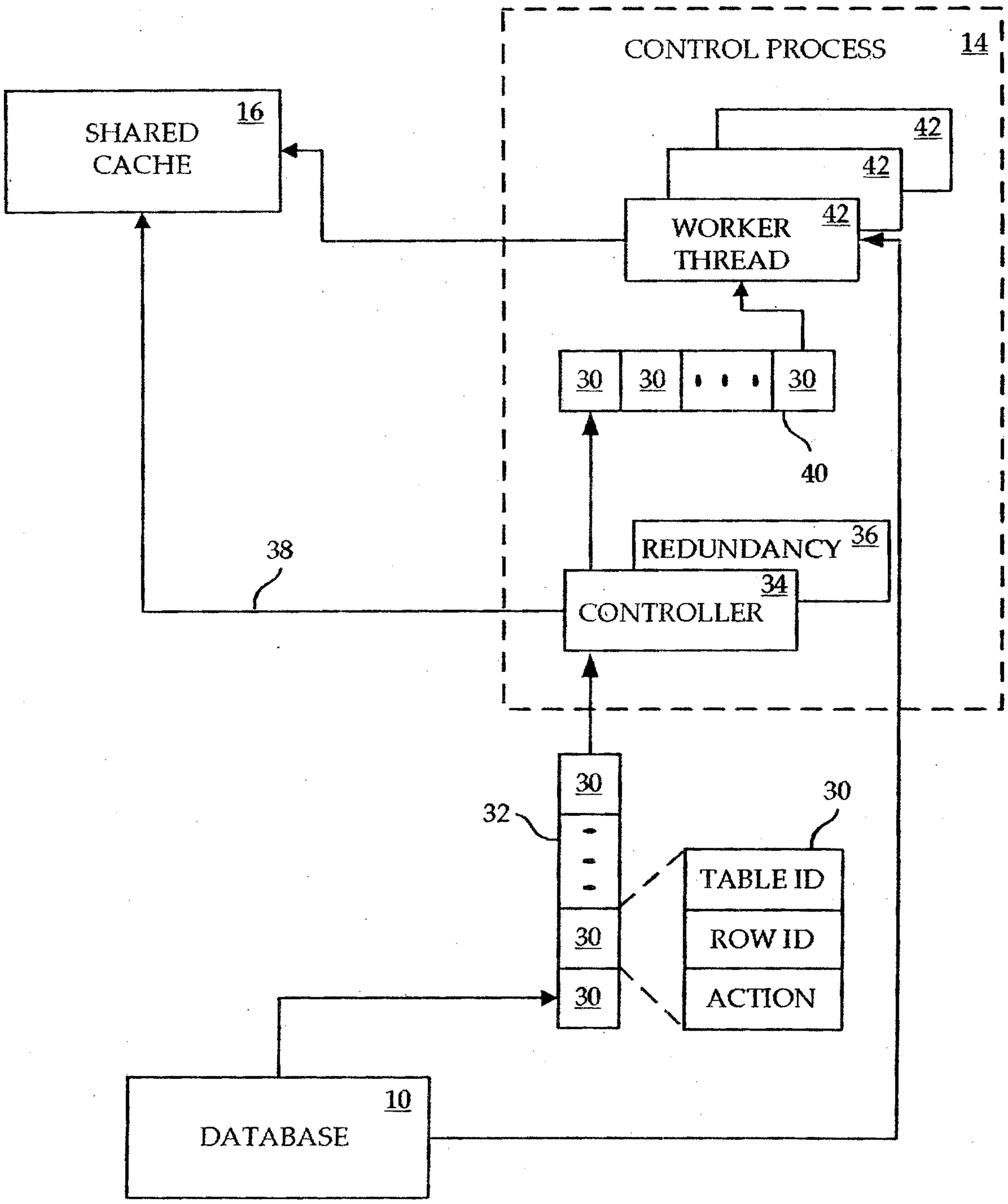

CONTROL PROCESS
14
SHARED CACHE
16
42
42
42
WORKER THREAD
30
30
30
40
REDUNDANCY
36
38
34
CONTROLLER
30
32
30
TABLE ID
ROW ID
ACTION
30
30
10
DATABASE

FIG. 2

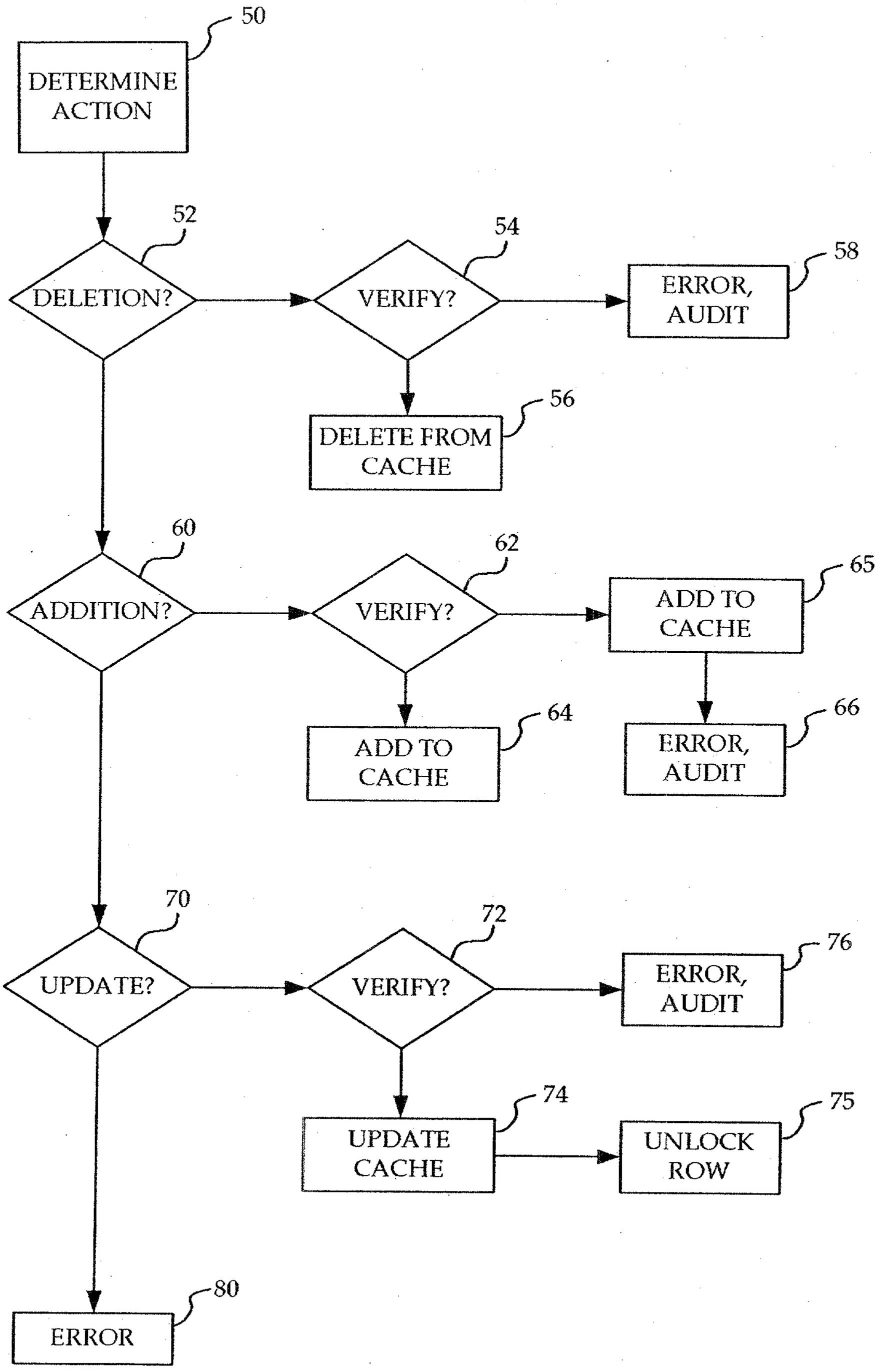

DETERMINE ACTION
50
DELETION?
52
VERIFY?
54
ERROR, AUDIT
58
DELETE FROM CACHE
56
ADDITION?
60
VERIFY?
62
ADD TO CACHE
65
ADD TO CACHE
64
ERROR, AUDIT
66
UPDATE?
70
VERIFY?
72
ERROR, AUDIT
76
UPDATE CACHE
74
UNLOCK ROW
75
ERROR
80

FIG. 3

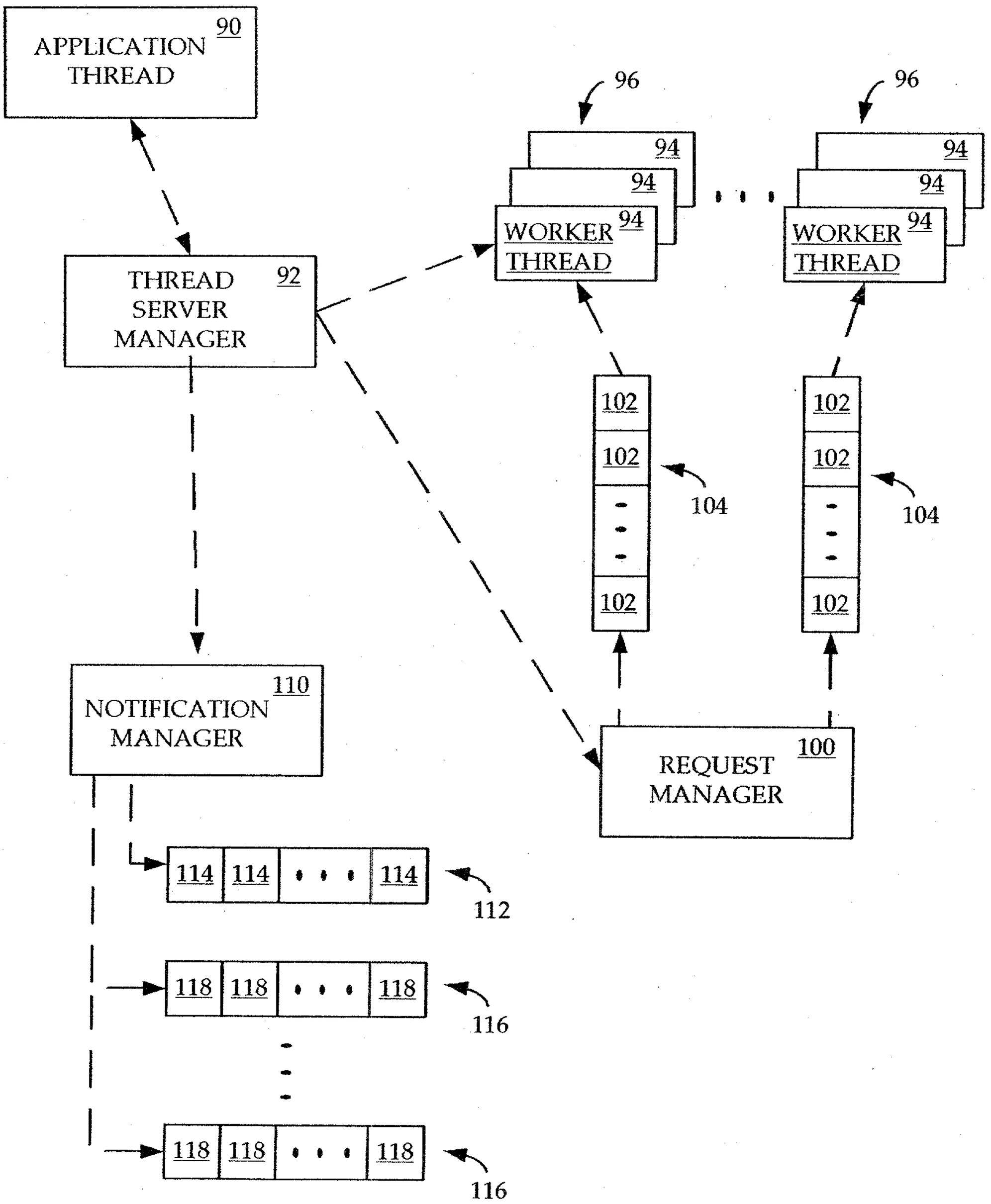

APPLICATION THREAD
90
THREAD SERVER MANAGER
92
WORKER THREAD
94
96
102
104
NOTIFICATION MANAGER
110
REQUEST MANAGER
100
114
112
118
116

FIG. 4

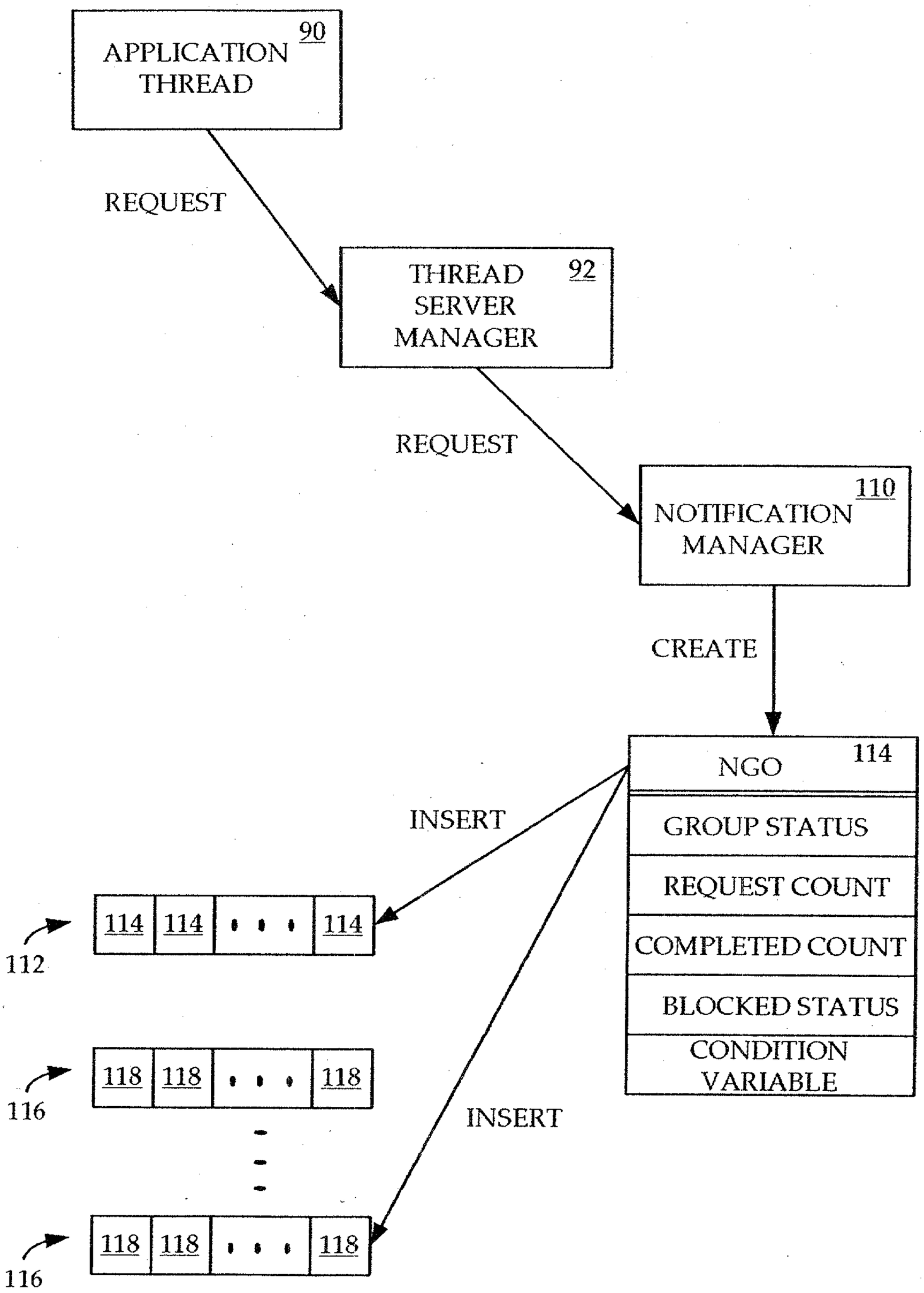

APPLICATION THREAD
90
REQUEST
THREAD SERVER MANAGER
92
REQUEST
NOTIFICATION MANAGER
110
CREATE
NGO
114
GROUP STATUS
REQUEST COUNT
COMPLETED COUNT
BLOCKED STATUS
CONDITION VARIABLE
INSERT
INSERT
114
114
114
112
118
118
118
116
118
118
118
116

FIG. 5

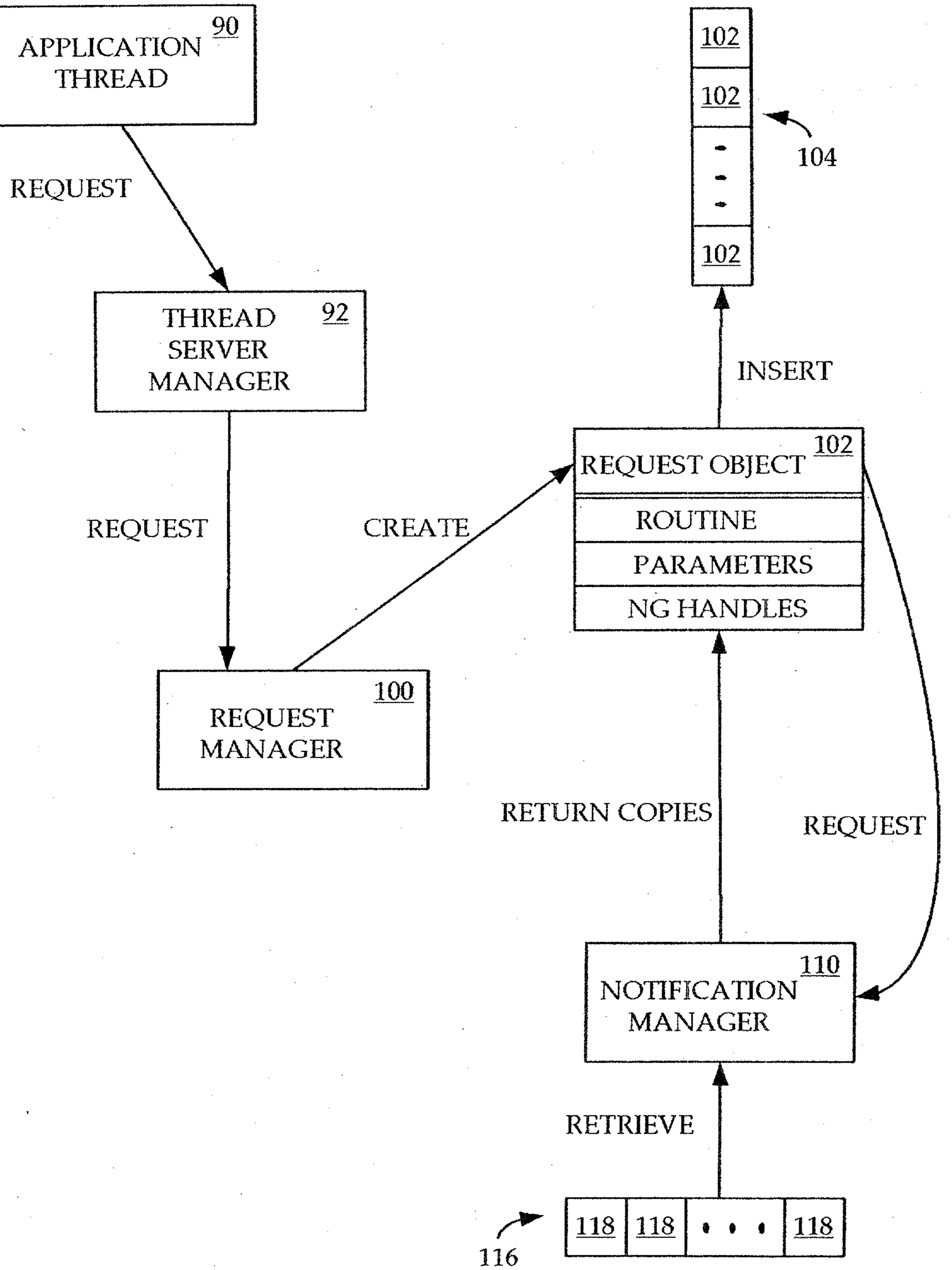

APPLICATION THREAD
90
REQUEST
THREAD SERVER MANAGER
92
REQUEST
REQUEST MANAGER
100
CREATE
REQUEST OBJECT
102
ROUTINE
PARAMETERS
NG HANDLES
INSERT
102
102
102
104
RETURN COPIES
REQUEST
NOTIFICATION MANAGER
110
RETRIEVE
118
118
118
116

FIG. 6

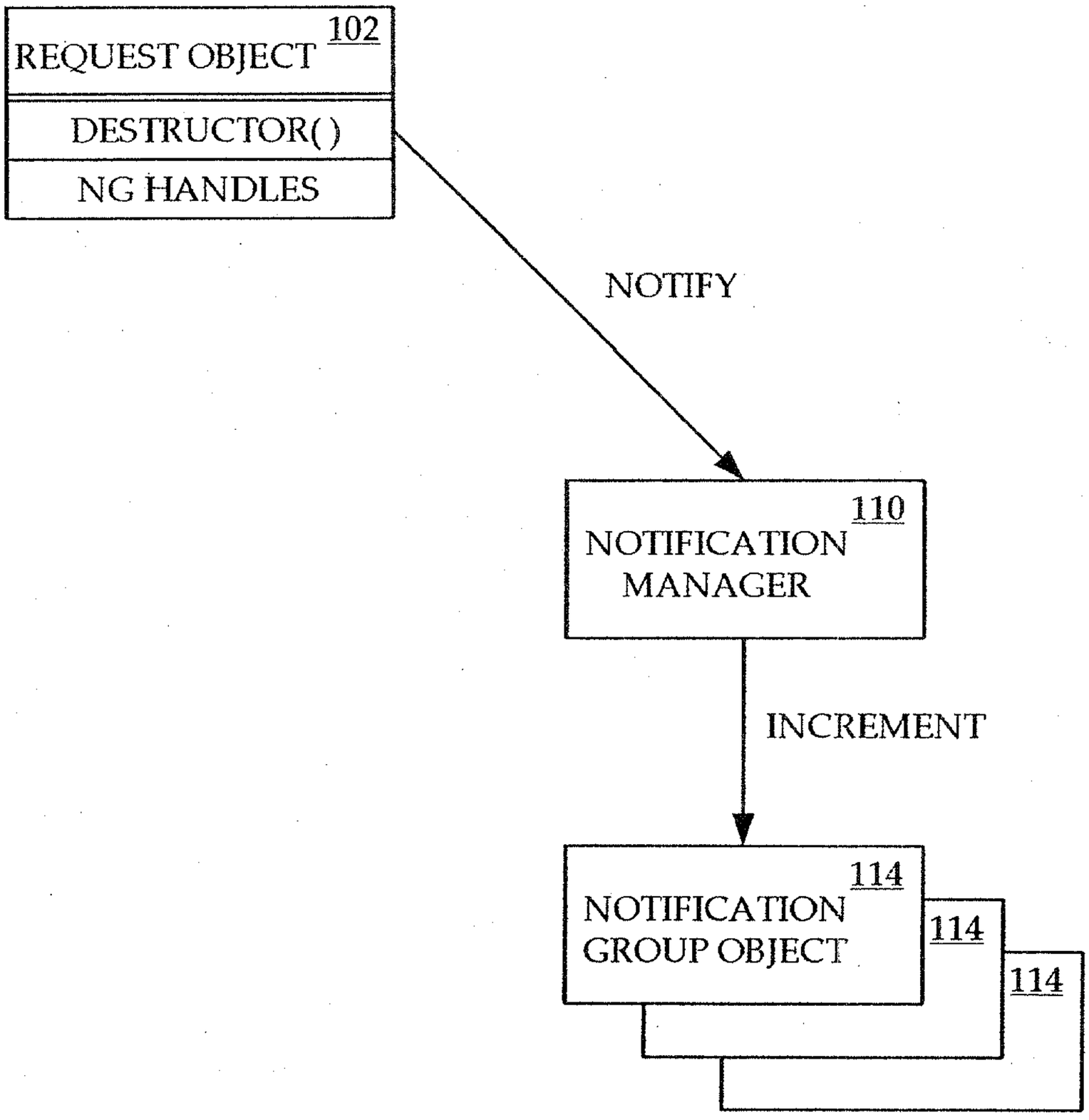
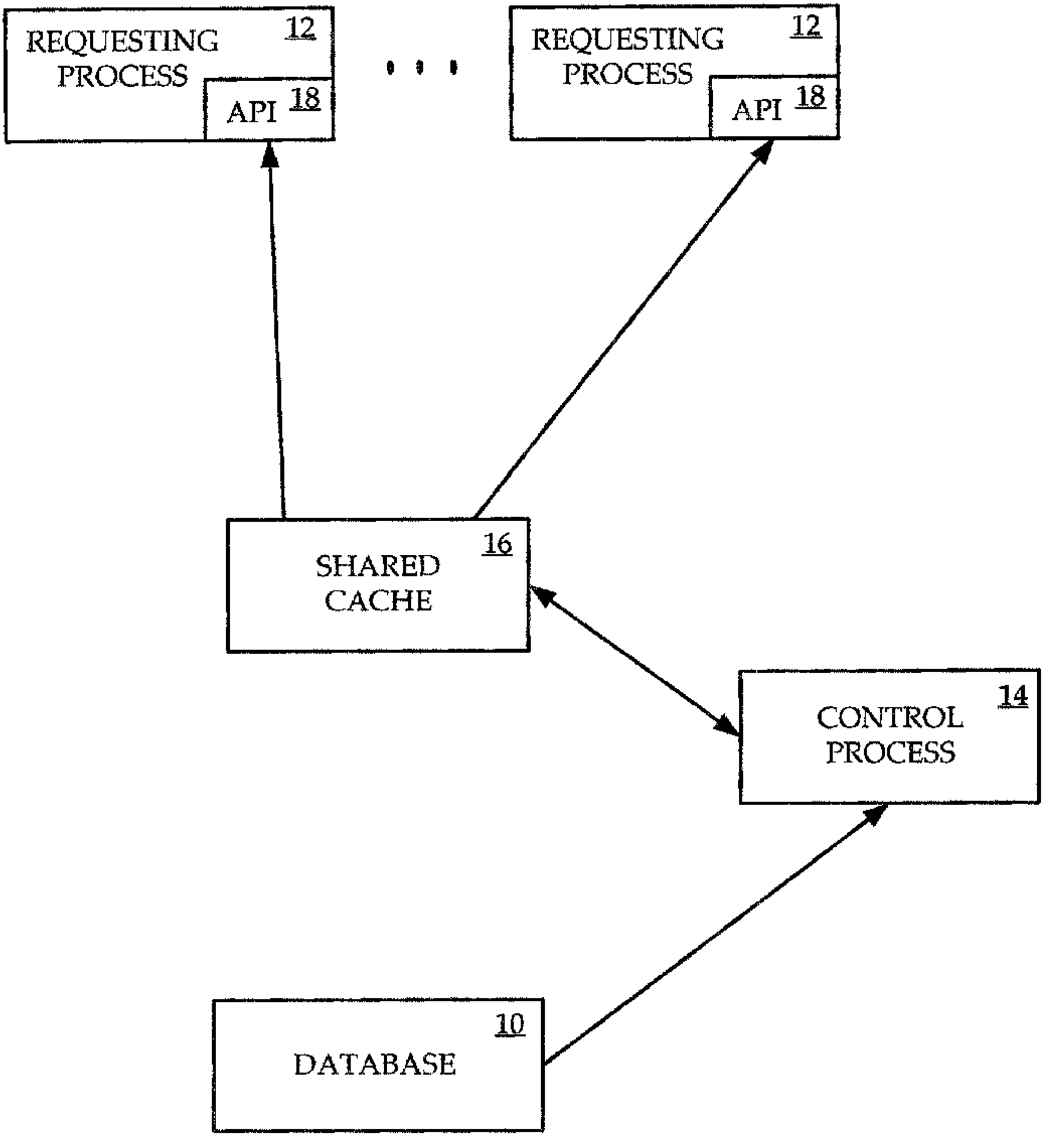

REQUEST OBJECT
102
DESTRUCTOR( )
NG HANDLES
NOTIFY
NOTIFICATION MANAGER
110
INCREMENT
NOTIFICATION GROUP OBJECT
114
114
114

FIG. 7


REQUESTING PROCESS
12
API
18
REQUESTING PROCESS
12
API
18
SHARED CACHE
16
CONTROL PROCESS
14
DATABASE
10