TRANSACTIONAL MEMORY MANAGER

Claim For Priority/Cross Reference to Related Applications

This non-provisional application claims the benefit of U.S. Provisional Patent Application Serial No. 60/330,842, filed November 1, 2001, which is incorporated by reference in its entirety, and U.S. Provisional Patent Application Serial No. 60/365,169, filed March 19, 2002, which is incorporated by reference in its entirety.

Technical Field

This disclosure relates to computer systems. More specifically, this disclosure relates to a method and system for providing high-speed database searching with concurrent updating for large database systems.

Background of the Invention

As the Internet continues its meteoric growth, scaling domain name service (DNS) resolution for root and generic top level domain (gTLD) servers at reasonable price points is becoming increasingly difficult. The A root server (i.e., a.root-server.net) maintains and distributes the Internet namespace root zone file to the 12 secondary root servers geographically distributed around the world (i.e., b.root-server.net, c.root-server.net, etc.), while the corresponding gTLD servers (i.e., a.gtld-servers.net, b.gtld-servers.net, etc.) are similarly distributed and support the top level domains (e.g., *.com, *.net, *.org, etc.). The ever-increasing volume of data coupled with the unrelenting growth in query rates is forcing a complete rethinking of the hardware and software infrastructure needed for root and gTLD DNS service over the next several years. The typical single server installation of the standard "bind" software distribution is already insufficient for the demands of the A root and will soon be unable to meet even gTLD needs. With the convergence of the public switched telephone network (PSTN) and the Internet, there are opportunities for a general purpose, high performance search mechanism to provide features normally associated with Service Control Points (SCPs) on the PSTN's SS7 signaling network as new, advanced services are offered that span the PSTN and the Internet, including Advanced Intelligent Network (AIN), Voice Over Internet Protocol (VoIP) services, geolocation services, etc.

Brief Description of the Drawings

FIG. 1 is a system block diagram, according to an embodiment of the present invention.

FIG. 2 is a detailed block diagram that illustrates a message data structure, according to an embodiment of the present invention.

FIG. 3 is a detailed block diagram that illustrates a message latency data structure architecture, according to an embodiment of the present invention.

FIG. 4 is a detailed block diagram that illustrates a general database architecture, according to an embodiment of the present invention.

FIG. 5 is a detailed block diagram that illustrates a general database architecture, according to an embodiment of the present invention.

FIG. 6 is a detailed block diagram that illustrates a general database architecture, according to an embodiment of the present invention.

FIG. 7 is a detailed block diagram that illustrates a general database architecture, according to an embodiment of the present invention.

FIG. 8 is a detailed block diagram that illustrates a general database architecture, according to an embodiment of the present invention.

FIG. 9 is a top level flow diagram that illustrates a method for searching and concurrently updating a database, according to an embodiment of the present invention.

FIG. 10 is a top level flow diagram that illustrates a method for searching and concurrently updating a database, according to an embodiment of the present invention.

Detailed Description

Embodiments of the present invention provide a method and system for high-speed database searching with concurrent updating for large database systems. Specifically, a plurality of search queries may be received over a network, the database may be searched, and a plurality of search replies may be sent over the network. While searching the database, new information may be received over the network, a plurality of new database elements may be created based on the new

information and a dirty bit may be set within each new database element. A pointer to each new database element may be written to the database using a single uninterruptible operation and the dirty bit within each new database element may be cleared.

FIG. 1 is a block diagram that illustrates a system according to an embodiment of the present invention. Generally, system 100 may host a large, memory-resident database, receive search requests and provide search responses over a network. For example, system 100 may be a symmetric, multiprocessing (SMP) computer, such as, for example, an IBM RS/6000® M80 or S80 manufactured by International Business Machines Corporation of Armonk, New York, a Sun Enterprise™ 10000 manufactured by Sun Microsystems, Inc. of Santa Clara, California, etc. System 100 may also be a multi-processor personal computer, such as, for example, a Compaq ProLiant™ ML530 (including two Intel Pentium® III 866 MHz processors) manufactured by Hewlett-Packard Company of Palo Alto, California. System 100 may also include a multiprocessing operating system, such as, for example, IBM AIX® 4, Sun Solaris™ 8 Operating Environment, Red Hat Linux® 6.2, etc. System 100 may receive periodic updates over network 124, which may be concurrently incorporated into the database.

In an embodiment, system 100 may include at least one processor 102-1 coupled to bus 101. Processor 102-1 may include an internal memory cache (e.g., an L1 cache, not shown for clarity). A secondary memory cache 103-1 (e.g., an L2 cache, L2/L3 caches, etc.) may reside between processor 102-1 and bus 101. In a preferred embodiment, system 100 may include a plurality of processors 102-1 ... 102-P coupled to bus 101. A plurality of secondary memory caches 103-1 ... 103-P may also reside between plurality of processors 102-1 ... 102-P and bus 101 (e.g., a look-through architecture), or, alternatively, at least one secondary memory cache 103-1 may be coupled to bus 101 (e.g., a look-aside architecture). System 100 may include memory 104, such as, for example, random access memory (RAM), etc., coupled to bus 101, for storing information and instructions to be executed by plurality of processors 102-1 ... 102-P.

Memory 104 may store a large database, for example, for translating Internet domain names into Internet addresses, for translating names or phone numbers into network addresses, for providing and updating subscriber profile data, for providing and updating user presence data, etc. Advantageously, both the size of the database and the number of translations per second may be very large. For example, memory 104 may include at least 64 GB of RAM and may host a 500M

(i.e., $500x10^6$) record domain name database, a 500M record subscriber database, a 450M record telephone number portability database, etc.

On an exemplary 64-bit system architecture, such as, for example, a system including at least one 64-bit big-endian processor 102-1 coupled to at least a 64-bit bus 101 and a 64-bit memory 104, an 8-byte pointer value may be written to a memory address on an 8-byte boundary (i.e., a memory address divisible by eight, or, e.g., 8N) using a single, uninterruptible operation. Generally, the presence of secondary memory cache 103-1 may simply delay the 8-byte pointer write to memory 104. For example, in one embodiment, secondary memory cache 103-1 may be a look-through cache operating in write-through mode, so that a single, 8-byte store instruction may move eight bytes of data from processor 102-1 to memory 104, without interruption, and in as few as two system clock cycles. In another embodiment, secondary memory cache 103-1may be a look-through cache operating in write-back mode, so that the 8-byte pointer may first be written to secondary memory cache 103-1, which may then write the 8-byte pointer to memory 104 at a later time, such as, for example, when the cache line in which the 8-byte pointer is stored is written to memory 104 (i.e., e.g., when the particular cache line, or the entire secondary memory cache, is "flushed").

Ultimately, from the perspective of processor 102-1, once the data are latched onto the output pins of processor 102-1, all eight bytes of data are written to memory 104 in one contiguous, uninterrupted transfer, which may be delayed by the effects of a secondary memory cache 103-1, if present. From the perspective of processors 102-2 ... 102-P, once the data are latched onto the output pins of processor 102-1, all eight bytes of data are written to memory 104 in one contiguous, uninterrupted transfer, which is enforced by the cache coherency protocol across secondary memory caches 103-1 ... 103-P, which may delay the write to memory 104 if present

However, if an 8-byte pointer value is written to a misaligned location in memory 104, such as a memory address that crosses an 8-byte boundary, all eight bytes of data can not be transferred from processor 102-1 using a single, 8-byte store instruction. Instead, processor 102-1 may issue two separate and distinct store instructions. For example, if the memory address begins four bytes before an 8-byte boundary (e.g., 8N – 4), the first store instruction transfers the four most significant bytes to memory 104 (e.g., 8N – 4), while the second store instruction transfers the four least significant bytes to memory 104 (e.g., 8N). Importantly, between these two separate store instructions, processor 102-1 may be interrupted,

or, processor 102-1 may loose control of bus 101 to another system component (e.g., processor 102-P, etc.). Consequently, the pointer value residing in memory 104 will be invalid until processor 102-1 can complete the second store instruction. If another component begins a single, uninterruptible memory read to this memory location, an invalid value will be returned as a presumably valid one.

Similarly, a new 4-byte pointer value may be written to a memory address divisible by four (e.g., 4N) using a single, uninterruptible operation. Note that in the example discussed above, a 4-byte pointer value may be written to the 8N – 4 memory location using a single store instruction. Of course, if a 4-byte pointer value is written to a location that crosses a 4-byte boundary, e.g., 4N – 2, all four bytes of data can not be transferred from processor 102-1 using a single store instruction, and the pointer value residing in memory 104 may be invalid for some period of time.

System 100 may also include a read only memory (ROM) 106, or other static storage device, coupled to bus 101 for storing static information and instructions for processor 102-1. A storage device 108, such as a magnetic or optical disk, may be coupled to bus 101 for storing information and instructions. System 100 may also include display 110 (e.g., an LCD monitor) and input device 112 (e.g., keyboard, mouse, trackball, etc.), coupled to bus 101. System 100 may include a plurality of network interfaces 114-1 ... 114-O, which may send and receive electrical, electromagnetic or optical signals that carry digital data streams representing various types of information. In an embodiment, network interface 114-1 may be coupled to bus 101 and local area network (LAN) 122, while network interface 114-O may coupled to bus 101 and wide area network (WAN) 124. Plurality of network interfaces 114-1 ... 114-O may support various network protocols, including, for example, Gigabit Ethernet (e.g., IEEE Standard 802.3-2002, published 2002), Fiber Channel (e.g., ANSI Standard X.3230-1994, published 1994), etc. Plurality of network computers 120-1 ... 120-N may be coupled to LAN 122 and WAN 124. In one embodiment, LAN 122 and WAN 124 may be physically distinct networks, while in another embodiment, LAN 122 and WAN 124 may be via a network gateway or router (not shown for clarity). Alternatively, LAN 122 and WAN 124 may be the same network.

As noted above, system 100 may provide DNS resolution services. In a DNS resolution embodiment, DNS resolution services may generally be divided between network transport and data look-up functions. For example, system 100 may be a back-end look-up engine (LUE) optimized for data look-up on large data sets, while

plurality of network computers 120-1 ... 120-N may be a plurality of front-end protocol engines (PEs) optimized for network processing and transport. The LUE may be a powerful multiprocessor server that stores the entire DNS record set in memory 104 to facilitate high-speed, high-throughput searching and updating. In an alternative embodiment, DNS resolution services may be provided by a series of powerful multiprocessor servers, or LUEs, each storing a subset of the entire DNS record set in memory to facilitate high-speed, high-throughput searching and updating

Conversely, the plurality of PEs may be generic, low profile, PC-based machines, running an efficient multitasking operating system (e.g., Red Hat Linux® 6.2), that minimize the network processing transport load on the LUE in order to maximize the available resources for DNS resolution. The PEs may handle the nuances of wire-line DNS protocol, respond to invalid DNS queries and multiplex valid DNS queries to the LUE over LAN 122. The number of PEs for a single LUE may be determined, for example, by the number of DNS queries to be processed per second and the performance characteristics of the particular system. Other metrics may also be used to determine the appropriate mapping ratios and behaviors.

Generally, other large-volume, query-based embodiments may be supported, including, for example, telephone number resolution, SS7 signaling processing, geolocation determination, telephone number-to-subscriber mapping, subscriber location and presence determination, etc.

In an embodiment, a central on-line transaction processing (OLTP) server 140-1 may be coupled to WAN 124 and receive additions, modifications and deletions (i.e., update traffic) to database 142-1 from various sources. OLTP server 140-1 may send updates to system 100, which includes a local copy of database 142-1, over WAN 124. OLTP server 140-1 may be optimized for processing update traffic in various formats and protocols, including, for example, HyperText Transmission Protocol (HTTP), Registry Registrar Protocol (RRP), Extensible Provisioning Protocol (EPP), Service Management System/800 Mechanized Generic Interface (MGI), and other on-line provisioning protocols. A constellation of read-only LUEs may be deployed in a hub and spoke architecture to provide high-speed search capability conjoined with high-volume, incremental updates from OLTP server 140-1.

In an alternative embodiment, data may be distributed over multiple OLTP servers 140-1...140-S, each of which may be coupled to WAN 124. OLTP servers 140-1...140-S may receive additions, modifications, and deletions (i.e., update traffic) to their respective databases 142-1...142-S (not shown for clarity) from various sources. OLTP servers 140-1...140-S may send updates to system 100, which may include copies of databases 142-1...142-S, other dynamically-created data, etc., over WAN 124. For example, in a geolocation embodiment, OLTP servers 140-1...140-S may receive update traffic from groups of remote sensors. In another alternative embodiment, plurality of network computers 120-1 ... 120-N may also receive additions, modifications, and deletions (i.e., update traffic) from various sources over WAN 124 or LAN 122. In this embodiment, plurality of network computers 120-1 ... 120-N may send updates, as well as queries, to system 100.

In the DNS resolution embodiment, each PE (e.g., each of the plurality of network computers 120-1 ... 120-N) may combine, or multiplex, several DNS query messages, received over a wide area network (e.g., WAN 124), into a single Request SuperPacket and send the Request SuperPacket to the LUE (e.g., system 100) over a local area network (e.g., LAN 122). The LUE may combine, or multiplex, several DNS query message replies into a single Response SuperPacket and send the Response SuperPacket to the appropriate PE over the local area network. Generally, the maximum size of a Request or Response SuperPacket may be limited by the maximum transmission unit (MTU) of the physical network layer (e.g., Gigabit Ethernet). For example, typical DNS query and reply message sizes of less than 100 bytes and 200 bytes, respectively, allow for over 30 queries to be multiplexed into a single Request SuperPacket, as well as over 15 replies to be multiplexed into a single Response SuperPacket. However, a smaller number of queries (e.g., 20 queries) may be included in a single Request SuperPacket in order to avoid MTU overflow on the response (e.g., 10 replies). For larger MTU sizes, the number of multiplexed queries and replies may be increased accordingly.

Each multitasking PE may include an inbound thread and an outbound thread to manage DNS queries and replies, respectively. For example, the inbound thread may un-marshal the DNS query components from the incoming DNS query packets received over a wide area network and multiplex several milliseconds of queries into a single Request SuperPacket. The inbound thread may then send the Request SuperPacket to the LUE over a local area network. Conversely, the outbound thread may receive the Response SuperPacket from the LUE, de-

multiplex the replies contained therein, and marshal the various fields into a valid DNS reply, which may then be transmitted over the wide area network. Generally, as noted above, other large-volume, query-based embodiments may be supported.

In an embodiment, the Request SuperPacket may also include state information associated with each DNS query, such as, for example, the source address, the protocol type, etc. The LUE may include the state information, and associated DNS replies, within the Response SuperPacket. Each PE may then construct and return valid DNS reply messages using the information transmitted from the LUE. Consequently, each PE may advantageously operate as a stateless machine, i.e., valid DNS replies may be formed from the information contained in the Response SuperPacket. Generally, the LUE may return the Response SuperPacket to the PE from which the incoming SuperPacket originated; however, other variations may obviously be possible.

In an alternative embodiment, each PE may maintain the state information associated with each DNS query and include a reference, or handle, to the state information within the Request SuperPacket. The LUE may include the state information references, and associated DNS replies, within the Response SuperPacket. Each PE may then construct and return valid DNS reply messages using the state information references transmitted from the LUE, as well as the state information maintained thereon. In this embodiment, the LUE may return the Response SuperPacket to the PE from which the incoming SuperPacket originated.

FIG. 2 is a detailed block diagram that illustrates a message data structure, according to an embodiment of the present invention. Generally, message 200 may include header 210, having a plurality of sequence number 211-1 ... 211-S and a plurality of message counts 212-1 ... 212-S, and data payload 215.

In the DNS resolution embodiment, message 200 may be used for Request SuperPackets and Response SuperPackets. For example, Request SuperPacket 220 may include header 230, having a plurality of sequence number 231-1 ... 231-S and a plurality of message counts 232-1 ... 232-S, and data payload 235 having multiple DNS queries 236-1 ... 236-Q, accumulated by a PE over a predetermined period of time, such as, for example, several milliseconds. In one embodiment, each DNS query 236-1 ... 236-Q may include state information, while in an alternative embodiment, each DNS query 236-1 ... 236-Q may include a handle to state information.

Similarly, Response SuperPacket 240 may include header 250, having a plurality of sequence number 251-1 ... 251-S and a plurality of message counts 252-1 ... 252-S, and data payload 255 having multiple DNS replies 256-1 ... 256-R approximately corresponding to the multiple DNS queries contained within Request SuperPacket 220. In one embodiment, each DNS reply 256-1 ... 256-R may include state information associated with the corresponding DNS query, while in an alternative embodiment, each DNS reply 256-1 ... 256-R may include a handle to state information associated with the corresponding DNS query. Occasionally, the total size of the corresponding DNS replies may exceed the size of data payload 255 of the Response SuperPacket 240. This overflow may be limited, for example, to a single reply, i.e., the reply associated with the last query contained within Request SuperPacket 220. Rather than sending an additional Response SuperPacket 240 containing only the single reply, the overflow reply may be preferably included in the next Response SuperPacket 240 corresponding to the next Request SuperPacket. Advantageously, header 250 may include appropriate information to determine the extent of the overflow condition. Under peak processing conditions, more than one reply may overflow into the next Response SuperPacket.

For example, in Response SuperPacket 240, header 250 may include at least two sequence numbers 251-1 and 251-2 and at least two message counts 252-1 and 252-2, grouped as two pairs of complementary fields. While there may be "S" number of sequence number and message count pairs, typically, S is a small number, such as, e.g., 2, 3, 4, etc. Thus, header 250 may include sequence number 251-1 paired with message count 252-1, sequence number 251-2 paired with message count 252-2, etc. Generally, message count 252-1 may reflect the number of replies contained within data payload 255 that are associated with sequence number 251-1. In an embodiment, sequence number 251-1 may be a two-byte field, while message count 252-1 may be a one-byte field.

In a more specific example, data payload 235 of Request SuperPacket 220 may include seven DNS queries (as depicted in FIG. 2). In one embodiment, sequence number 231-1 may be set to a unique value (e.g., 1024) and message count 232-1 may be set to seven, while sequence number 231-2 and message count 232-2 may be set to zero. In another embodiment, header 230 may contain only one sequence number and one message count, e.g., sequence number 231-1 and message count 232-1 set to 1024 and seven, respectively. Typically, Request

SuperPacket 220 may contain all of the queries associated with a particular sequence number.

Data payload 255 of Response SuperPacket 240 may include seven corresponding DNS replies (as depicted in FIG. 2). In this example, header 250 may include information similar to Request SuperPacket 220, i.e., sequence number 251-1 set to the same unique value (i.e., 1024), message count 252-1 set to seven, and both sequence number 252-2 and message count 252-2 set to zero. However, in another example, data payload 255 of Response SuperPacket 240 may include only five corresponding DNS replies, and message count 252-1 may be set to five instead. The remaining two responses associated with sequence number 1024 may be included within the next Response SuperPacket 240.

The next Request SuperPacket 240 may include a different sequence number (e.g., 1025) and at least one DNS query, so that the next Response SuperPacket 240 may include the two previous replies associated with the 1024 sequence number, as well as at least one reply associated with the 1025 sequence number. In this example, header 250 of the next Response SuperPacket 240 may include sequence number 251-1 set to 1024, message count 252-1 set to two, sequence number 251-2 set to 1025 and message count 252-2 set to one. Thus, Response SuperPacket 240 may include a total of three replies associated with three queries contained within two different Request SuperPackets.

FIG. 3 is a detailed block diagram that illustrates a message latency data structure architecture, according to an embodiment of the present invention. Message latency data structure 300 may include information generally associated with the transmission and reception of message 200. In the DNS resolution embodiment, message latency data structure 300 may include latency information about Request SuperPackets and Response SuperPackets; this latency information may be organized in a table format indexed according to sequence number value (e.g., index 301). For example, message latency data structure 300 may include a number of rows N equal to the total number of unique sequence numbers, as illustrated, generally, by table elements 310, 320 and 330. In an embodiment, SuperPacket header sequence numbers may be two bytes in length and define a range of unique sequence numbers from zero to 2^{16}-1 (i.e., 65,535). In this case, N may be equal to 65,536. Latency information may include Request Timestamp 302, Request Query Count 303, Response Timestamp 304, Response Reply Count 305, and Response Message Count 306. In an alternative embodiment, latency information may also include an Initial Response Timestamp (not shown).

In an example, table element 320 illustrates latency information for a Request SuperPacket 220 having a single sequence number 231-1 equal to 1024. Request Timestamp 302 may indicate when this particular Request SuperPacket was sent to the LUE. Request Query Count 303 may indicate how many queries were contained within this particular Request SuperPacket. Response Timestamp 304 may indicate when a Response SuperPacket having a sequence number equal to 1024 was received at the PE (e.g., network computer 120-N) and may be updated if more than one Response SuperPacket is received at the PE. Response Reply Count 305 may indicate the total number of replies contained within all of the received Response SuperPackets associated with this sequence number (i.e., 1024). Response Message Count 306 may indicate how many Response SuperPackets having this sequence number (i.e., 1024) arrived at the PE. Replies to the queries contained within this particular Request SuperPacket may be split over several Response SuperPackets, in which case, Response Timestamp 304, Response Reply Count 305, and Response Message Count 306 may be updated as each of the additional Response SuperPackets are received. In an alternative embodiment, the Initial Response Timestamp may indicate when the first Response SuperPacket containing replies for this sequence number (i.e., 1024) was received at the PE. In this embodiment, Response Timestamp 304 may be updated when additional (i.e., second and subsequent) Response SuperPackets are received.

Various important latency metrics may be determined from the latency information contained within message latency data structure 300. For example, simple cross-checking between Request Query Count 303 and Response Reply Count 305 for a given index 301 (i.e., sequence number) may indicate a number of missing replies. This difference may indicate the number of queries inexplicably dropped by the LUE. Comparing Request Timestamp 302 and Response Timestamp 304 may indicate how well the particular PE/LUE combination may be performing under the current message load. The difference between the current Request SuperPacket sequence number and the current Response SuperPacket sequence number may be associated with the response performance of the LUE; e.g., the larger the difference, the slower the performance. The Response Message Count 306 may indicate how many Response SuperPackets are being used for each Request SuperPacket, and may be important in DNS resolution traffic analysis. As the latency of the queries and replies travelling between the PEs and LUE increases, the PEs may reduce the number of DNS query packets processed by the system.

Generally, the LUE may perform a multi-threaded look-up on the incoming, multiplexed Request SuperPackets, and may combine the replies into outgoing, multiplexed Response SuperPackets. For example, the LUE may spawn one search thread, or process, for each active PE and route all the incoming Request SuperPackets from that PE to that search thread. The LUE may spawn a manager thread, or process, to control the association of PEs to search threads, as well as an update thread, or process, to update the database located in memory 104. Each search thread may extract the search queries from the incoming Request SuperPacket, execute the various searches, construct an outgoing Response SuperPacket containing the search replies and send the SuperPacket to the appropriate PE. The update thread may receive updates to the database, from OLTP 140-1, and incorporate the new data into the database. In an alternative embodiment, plurality of network computers 120-1 ... 120-N may send updates to system 100. These updates may be included, for example, within the incoming Request SuperPacket message stream.

Accordingly, by virtue of the SuperPacket protocol, the LUE may spend less than 15% of its processor capacity on network processing, thereby dramatically increasing search query throughput. In an embodiment, an IBM® 8-way M80 may sustain search rates of 180k to 220k queries per second (qps), while an IBM® 24-way S80 may sustain 400k to 500k qps. Doubling the search rates, i.e., to 500k and 1M qps, respectively, simply requires twice as much hardware, i.e., e.g., two LUEs with their attendant PEs. In another embodiment, a dual Pentium® III 866 MHz multi-processor personal computer operating Red Hat Linux® 6.2 may sustain update rates on the order of 100K/sec. Of course, increases in hardware performance also increase search and update rates associated with embodiments of the present invention, and as manufacturers replace these multiprocessor computers with faster-performing machines, for example, the sustained search and update rates may increase commensurately. Generally, system 100 is not limited to a client or server architecture, and embodiments of the present invention are not limited to any specific combination of hardware and/or software.

FIG. 4 is a block diagram that illustrates a general database architecture according to an embodiment of the present invention. In this embodiment, database 400 may include at least one table or group of database records 401, and at least one corresponding search index 402 with pointers (indices, direct byte-offsets, etc.) to individual records within the group of database records 401. For example, pointer 405 may reference database record 410.

In one embodiment, database 400 may include at least one hash table 403 as a search index with pointers (indices, direct byte-offsets, etc.) into the table or group of database records 401. A hash function may map a search key to an integer value which may then be used as an index into hash table 403. Because more than one search key may map to a single integer value, hash buckets may be created using a singly-linked list of hash chain pointers. For example, each entry within hash table 403 may contain a pointer to the first element of a hash bucket, and each element of the hash bucket may contain a hash chain pointer to the next element, or database record, in the linked-list. Advantageously, a hash chain pointer may be required only for those elements, or database records, that reference a subsequent element in the hash bucket.

Hash table 403 may include an array of 8-byte pointers to individual database records 401. For example, hash pointer 404 within hash table 403 may reference database record 420 as the first element within a hash bucket. Database record 420 may contain a hash chain pointer 424 which may reference the next element, or database record, in the hash bucket. Database record 420 may also include a data length 421, and associated fixed or variable-length data 422. In an embodiment, a null character 423, indicating the termination of data 422, may be included. Additionally, database record 420 may include a data pointer 425 which may reference another database record, either within the group of database records 401 or within a different table or group of database records (not shown), in which additional data may be located.

System 100 may use various, well-known algorithms to search this data structure architecture for a given search term or key. Generally, database 400 may be searched by multiple search processes, or threads, executing on at least one of the plurality of processors 102-1 ... 102-P. However, modifications to database 400 may not be integrally performed by an update thread (or threads) unless the search thread(s) are prevented from accessing database 400 for the period of time necessary to add, modify, or delete information within database 400. For example, in order to modify database record 430 within database 400, the group of database records 401 may be locked by an update thread to prevent the search threads from accessing database 400 while the update thread is modifying the information within database record 430. There are many well-known mechanisms for locking database 400 to prevent search access, including the use of spin-locks, semaphores, mutexes, etc. Additionally, various off-the-shelf commercial databases provide specific commands to lock all or parts of database 400, e.g., the

lock table command in the Oracle 8 Database, manufactured by Oracle Corporation of Redwood Shores, California, etc.

FIG. 5 is a block diagram that illustrates a general database architecture according to another embodiment of the present invention. In this embodiment, database 500 may include a highly-optimized, read-only, master snapshot file 510 and a growing, look-aside file 520. Master snapshot file 510 may include at least one table or group of database records 511, and at least one corresponding search index 512 with pointers (indices, direct byte-offsets, etc.) to individual records within the group of database records 511. Alternatively, master snapshot file 510 may include at least one hash table 513 as a search index with pointers (indices, direct byte-offsets, etc.) into the table or group of database records 511. Similarly, look-aside file 520 may include at least two tables or groups of database records, including database addition records 521 and database deletion records 531. Corresponding search indices 522 and 532 may be provided, with pointers (indices, direct byte-offsets, etc.) to individual records within the database addition records 521 and database deletion records 531. Alternatively, look-aside file 520 may include hash tables 523 and 533 as search indices, with pointers (indices, direct byte-offsets, etc.) into database addition records 521 and database deletion records 531, respectively.

System 100 may use various, well-known algorithms to search this data structure architecture for a given search term or key. In a typical example, look-aside file 520 may include all the recent changes to the data, and may be searched before read-only master snapshot file 510. If the search key is found in look-aside file 520, the response is returned without accessing snapshot file 510, but if the key is not found, then snapshot file 510 may be searched. However, when look-aside file 520 no longer fits in memory 104 with snapshot file 510, search query rates drop dramatically, by a factor of 10 to 50, or more, for example. Consequently, to avoid or minimize any drop in search query rates, snapshot file 510 may be periodically updated, or recreated, by incorporating all of the additions, deletions and modifications contained within look-aside file 520

Data within snapshot file 510 are not physically altered but logically added, modified or deleted. For example, data within snapshot file 510 may be deleted, or logically "forgotten," by creating a corresponding delete record within database deletion records 531 and writing a pointer to the delete record to the appropriate location in hash table 533. Data within snapshot file 510 may be logically modified by copying a data record from snapshot file 510 to a new data record within

database addition records 521, modifying the data within the new entry, and then writing a pointer to the new entry to the appropriate hash table (e.g., hash table 522) or chain pointer within database addition records 521. Similarly, data within snapshot file 510 may be logically added to snapshot file 510 by creating a new data record within database addition records 521 and then writing a pointer to the new entry to the appropriate hash table (e.g., hash table 522) or chain pointer within database addition records 521.

In the DNS resolution embodiment, for example, snapshot file 510 may include domain name data and name server data, organized as separate data tables, or blocks, with separate search indices (e.g., 511-1, 511-2, 512-1, 512-2, 513-1, 513-2, etc., not shown for clarity). Similarly, look-aside file 520 may include additions and modifications to both the domain name data and the name server data, as well as deletions to both the domain name data and the name server data (e.g., 521-1, 521-2, 522-1, 522-2, 523-1, 523-2, 531-1, 531-2, 532-1, 532-2, 533-1, 533-2, etc., not shown for clarity).

FIG. 6 is a detailed block diagram that illustrates a general database architecture, according to an embodiment of the present invention. Generally, database 600 may be organized into a single, searchable representation of the data. Data set updates may be continuously incorporated into database 600, and deletes or modifications may be physically performed on the relevant database records to free space within memory 104, for example, for subsequent additions or modifications. The single, searchable representation scales extremely well to large data set sizes and high search and update rates, and obviates the need to periodically recreate, propagate and reload snapshot files among multiple search engine computers.

In a DNS resolution embodiment, for example, database 600 may include domain name data 610 and name server data 620. Domain name data 610 and name server data 620 may include search indices with pointers (indices, direct byte-offsets, etc.) into blocks of variable length records. As discussed above, a hash function may map a search key to an integer value which may then be used as an index into a hash table. Similarly, hash buckets may be created for each hash table index using a singly-linked list of hash chain pointers. Domain name data 610 may include, for example, a hash table 612 as a search index and a block of variable-length domain name records 611. Hash table 612 may include an array of 8-byte pointers to individual domain name records 611, such as, for example, pointer 613 referencing domain name record 620. Variable-length domain name

record 620 may include, for example, a next record offset 621, a name length 622, a normalized name 623, a chain pointer 624 (i.e., e.g., pointing to the next record in the hash chain), a number of name servers 625, and a name server pointer 626. The size of both chain pointer 624 and name server pointer 626 may be optimized to reflect the required block size for each particular type of data, e.g., eight bytes for chain pointer 624 and four bytes for name server pointer 626.

Name server data 630 may include, for example, a hash table 632 as a search index and a block of variable-length name server records 631. Hash table 632 may include an array of 4-byte pointers to individual name server records 631, such as, for example, pointer 633 referencing name server record 640. Variable-length name server record 640 may include, for example, a next record offset 641, a name length 642, a normalized name 643, a chain pointer 644 (i.e., e.g., pointing to the next record in the hash chain), a number of name server network addresses 645, a name server address length 646, and a name server network address 647, which may be, for example, an Internet Protocol (IP) network address. Generally, name server network addresses may be stored in ASCII (American Standard Code for Information Interchange, e.g., ISO-14962-1997, ANSI-X3.4-1997, etc.) or binary format; in this example, name server network address length 646 indicates that name server network address 647 is stored in binary format (i.e., four bytes). The size of chain pointer 644 may also be optimized to reflect the required name server data block size, e.g., four bytes.

Generally, both search indices, such as hash tables, and variable-length data records may be structured so that 8-byte pointers are located on 8-byte boundaries in memory. For example, hash table 612 may contain a contiguous array of 8-byte pointers to domain name records 611, and may be stored at a memory address divisible by eight (i.e., an 8-byte boundary, or 8N). Similarly, both search indices, such as hash tables and variable-length data records may be structured so that 4-byte pointers are located on 4-byte boundaries in memory. For example, hash table 632 may contain a contiguous array of 4-byte pointers to name server records 631, and may be stored at a memory address divisible by four (i.e., a 4-byte boundary, or 4N). Consequently, modifications to database 600 may conclude by updating a pointer to an aligned address in memory using a single uninterruptible operation, including, for example writing a new pointer to the search index, such as a hash table or writing a new hash chain pointer to a variable-length data record.

FIG. 7 is a detailed block diagram that illustrates a general database architecture, according to an embodiment of the present invention. Generally,

database 700 may also be organized into a single, searchable representation of the data. Data set updates may be continuously incorporated into database 700, and deletes or modifications may be physically performed on the relevant database records to free space within memory 104, for example, for subsequent additions or modifications. The single, searchable representation scales extremely well to large data set sizes and high search and update rates, and obviates the need to periodically recreate, propagate and reload snapshot files among multiple search engine computers.

Many different physical data structure organizations are possible. An exemplary organization may use an alternative search index to hash tables for ordered, sequential access to the data records, such as the ternary search tree (trie), or TST, which combines the features of binary search trees and digital search tries. In a text-based applications such as, for example, whois, domain name resolution using DNS Secure Extensions (Internet Engineering Taskforce Request for Comments: 2535), etc. TSTs advantageously minimize the number of comparison operations required to be performed, particularly in the case of a search miss, and may yield search performance metrics exceeding search engine implementations with hashing. Additionally, TSTs may also provide advanced text search features, such as, e.g., wildcard searches, which may be useful in text search applications, such as, for example, whois, domain name resolution, Internet content search, etc..

In an embodiment, a TST may contain a sequence of nodes linked together in a hierarchical relationship. A root node may be located at the top of the tree, related child nodes and links may form branches, and leaf nodes may terminate the end of each branch. Each leaf node may be associated with a particular search key, and each node on the path to the leaf node may contain a single, sequential element of the key. Each node in the tree contains a comparison character, or split value, and three pointers to other successive, or "child," nodes in the tree. These pointers reference child nodes whose split values are less than, equal to, or greater than the node's split value. Searching the TST for a particular key, therefore, involves traversing the tree from the root node to a final leaf node, sequentially comparing each element, or character position, of the key with the split values of the nodes along the path. Additionally, a leaf node may also contain a pointer to a key record, which may, in turn, contain at least one pointer to a terminal data record containing the record data associated with the key (e.g., an IP address). Alternatively, the key record may contain the record data in its entirety. Record data may be stored in binary format, ASCII text format, etc.

In an embodiment, database 700 may be organized as a TST, including a plurality of fixed-length search nodes 701, a plurality of variable-length key data records 702 and a plurality of variable-length terminal data records 703. Search nodes 701 may include various types of information as described above, including, for example, a comparison character (or value) and position, branch node pointers and a key pointer. The size of the node pointers may generally be determined by the number of nodes, while the size of the key pointers may generally be determined by the size of the variable-length key data set. Key data records 702 may contain key information and terminal data information, including, for example, pointers to terminal data records or embedded record data, while terminal data records 703 may contain record data.

In an embodiment, each fixed-length search node may be 24 bytes in length. Search node 710, for example, may contain an eight-bit comparison character (or byte value) 711, a 12-bit character (or byte) position 712, and a 12-bit node type/status (not shown for clarity); these data may be encoded within the first four bytes of the node. The comparison character 711 may be encoded within the first byte of the node as depicted in FIG. 7, or, alternatively, character position 712 may be encoded within the first 12 bits of the node in order to optimize access to character position 712 using a simple shift operation. The next 12 bytes of each search node may contain three 32-bit pointers, i.e., pointer 713, pointer 714 and pointer 715, representing "less than," "equal to," and "greater than" branch node pointers, respectively. These pointers may contain a counter, or node index, rather than a byte-offset or memory address. For fixed-length search nodes, the byte-offset may be calculated from the counter, or index value, and the fixed-length, e.g., counter*length. The final four bytes may contain a 40-bit key pointer 716, which may be a null value indicating that a corresponding key data record does not exist (shown) or a pointer to an existing corresponding key data record (not shown), as well as other data, including, for example, a 12-bit key length and a 12-bit pointer type/status field. Key pointer 716 may contain a byte offset to the appropriate key data record, while the key length may be used to optimize search and insertion when eliminating one-way branching within the TST. The pointer type/status field may contain information used in validity checking and allocation data used in memory management.

In an embodiment, key data record 750 may include, for example, a variable-length key 753 and at least one terminal data pointer. As depicted in FIG. 7, key data record 750 includes two terminal data pointers: terminal data pointer 757 and

terminal data pointer 758. Key data record 750 may be prefixed with a 12-bit key length 751 and a 12-bit terminal pointer count/status 752, and may include padding (not shown for clarity) to align the terminal data pointer 757 and terminal data pointer 758 on an 8-byte boundary in memory 104. Terminal data pointer 757 and terminal data pointer 758 may each contain various data, such as, for example, terminal data type, length, status or data useful in binary record searches. Terminal data pointer 757 and terminal data pointer 758 may be sorted by terminal data type for quicker retrieval of specific resource records (e.g., terminal data record 760 and terminal data record 770). In another embodiment, key data record 740 may include embedded terminal data 746 rather than, or in addition to, terminal data record pointers. For example, key data record 740 may include a key length 741, a terminal pointer count 742, a variable-length key 743, the number of embedded record elements 744, followed by a record element length 745 (in bytes, for example) and embedded record data 746 (e.g., a string, a byte sequence, etc.) for each of the number of embedded record elements 744.

In an embodiment, terminal data record 760, for example, may include a 12-bit length 761, a 4-bit status, and a variable-length string 762 (e.g., an IP address). Alternatively, variable length string 762 may be a byte sequence. Terminal data record 760 may include padding to align each terminal data record to an 8-byte boundary in memory 104. Alternatively, terminal data record 760 may include padding to a 4-byte boundary, or, terminal data record 760 may not include any padding. Memory management algorithms may determine, generally, whether terminal data records 760 are padded to 8-byte, 4-byte, or 0-byte boundaries. Similarly, terminal data record 770 may include a 12-bit length 771, a 4-bit status, and a variable-length string 772 (e.g., an IP address).

Generally, both search indices, such as TSTs, and data records may be structured so that 8-byte pointers are located on 8-byte boundaries in memory. For example, key pointer 726 may contain an 8-byte (or less) pointer to key data record 740, and may be stored at a memory address divisible by eight (i.e., an 8-byte boundary, or 8N). Similarly, both search indices, such as TSTs, and data records may be structured so that 4-byte pointers are located on 4-byte boundaries in memory. For example, node branch pointer 724 may contain a 4-byte (or less) pointer to node 730, and may be stored at a memory address divisible by four (i.e., a 4-byte boundary, or 4N). Consequently, modifications to database 700 may conclude by updating a pointer to an aligned address in memory using a single

uninterruptible operation, including, for example writing a new pointer to the search index, such as a TST node, or writing a new pointer to a data record.

FIG. 8 is a detailed block diagram that illustrates a general database architecture, according to an embodiment of the present invention. As above, database 800 may also be organized into a single, searchable representation of the data. Data set updates may be continuously incorporated into database 800, and deletes or modifications may be physically performed on the relevant database records to free space within memory 104, for example, for subsequent additions or modifications. The single, searchable representation scales extremely well to large data set sizes and high search and update rates, and obviates the need to periodically recreate, propagate and reload snapshot files among multiple search engine computers.

Other search index structures are possible for accessing record data, In an embodiment, database 800 may use an alternative ordered search index, organized as an ordered access key tree (i.e., "OAK tree"). Database 800 may include, for example, a plurality of variable-length search nodes 801, a plurality of variable-length key records 802 and a plurality of variable-length terminal data records 803. Search nodes 801 may include various types of information as described above, such as, for example, search keys, pointers to other search nodes, pointers to key records, etc. In an embodiment, plurality of search nodes 801 may include vertical and horizontal nodes containing fragments of search keys (e.g., strings), as well as pointers to other search nodes or key records. Vertical nodes may include, for example, at least one search key, or character, pointers to horizontal nodes within the plurality of search nodes 801, pointers to key records within the plurality of key records 802, etc. Horizontal nodes may include, for example, at least two search keys, or characters, pointers to vertical nodes within the plurality of search nodes 801, pointers to horizontal nodes within the plurality of search nodes 801, pointers to key records within the plurality of key records 802, etc. Generally, vertical nodes may include a sequence of keys (e.g., characters) representing a search key fragment (e.g., string), while horizontal nodes may include various keys (e.g., characters) that may exist at a particular position within the search key fragment (e.g., string).

In an embodiment, plurality of search nodes 801 may include vertical node 810, vertical node 820 and horizontal node 830. Vertical node 810 may include, for example, a 2-bit node type 811 (e.g., "10"), a 38-bit address 812, an 8-bit length 813 (e.g., "8"), an 8-bit first character 814 (e.g., "I") and an 8-bit second character

815 (e.g., "null"). In this example, address 812 may point to the next node in the search tree, i.e., vertical node 820. In an embodiment, 38-bit address 812 may include a 1-bit terminal/nodal indicator and a 37-bit offset address to reference one of the 8-byte words within a 1 Tbyte (~10^{12} byte) address space of memory 104. Accordingly, vertical node 810 may be eight bytes (64 bits) in length, and, advantageously, may be located on an 8-byte word boundary within memory 104. Generally, each vertical node within plurality of search nodes 801 may be located on an 8-byte word boundary within memory 104.

A vertical node may include a multi-character, search key fragment (e.g., string). Generally, search keys without associated key data records may be collapsed into a single vertical node to effectively reduce the number of vertical nodes required within plurality of search nodes 801. In an embodiment, vertical node 810 may include eight bits for each additional character, above two characters, within the search key fragment, such as, for example, 8-bit characters 816-1, 816-2 ... 816-N (shown in phantom outline). Advantageously, vertical node 810 may be padded to a 64-bit boundary within memory 104 in accordance with the number of additional characters located within the string fragment. For example, if nine characters are to be included within vertical node 810, then characters one and two may be assigned to first character 814 and second character 815, respectively, and 56 bits of additional character information, corresponding to characters three through nine, may be appended to vertical node 810. An additional eight bits of padding may be included to align the additional character information on an 8-byte word boundary.

Similarly, vertical node 820 may include, for example, a 2-bit node type 821 (e.g., "10"), a 38-bit address 822, an 8-bit length 823 (e.g., "8"), an 8-bit first character 824 (e.g., "a") and an 8-bit second character 825 (e.g., "null"). In this example, address 822 may point to the next node in the search tree, i.e., horizontal node 830. Accordingly, vertical node 820 may be eight bytes in length, and, advantageously, may be located on an 8-byte word boundary within memory 104. Of course, additional information may also be included within vertical node 820 if required, as described above with reference to vertical node 810.

Horizontal node 830 may include, for example, a 2-bit node type 831 (e.g., "01"), a 38-bit first address 832, an 8-bit address count 833 (e.g., 2), an 8-bit first character 834 (e.g., "•"), an 8-bit last character 835 (e.g., "w"), a variable-length bitmap 836 and a 38-bit second address 837. In this example, first character 834 may include a single character, "•" representing the search key fragment "la" defined by vertical nodes 810 and 820, while last character 831 may include a

single character "w," representing the search key fragment "law" defined by vertical nodes 810 and 820, and the last character 835 of horizontal node 830. First address 832 may point to key data record 840, associated with the search key fragment "la," while second address 837 may point to key data record 850 associated with the search key fragment "law."

Bitmap 836 may advantageously indicate which keys (e.g., characters) are referenced by horizontal node 830. A "1" within a bit position in bitmap 836 indicates that the key, or character, is referenced by horizontal node 830, while a "0" within a bit position in bitmap 836 may indicate that the key, or character, is not referenced by horizontal node 830. Generally, the length of bitmap 836 may depend upon the number of sequential keys, or characters, between first character 834 and last character 835, inclusive of these boundary characters. For example, if first character 834 is "a" and last character 835 is "z," then bitmap 836 may be 26 bits in length, where each bit corresponds to one of the characters between, and including, "a" through "z." In this example, additional 38-bit addresses would be appended to the end of horizontal node 830, corresponding to each of the characters represented within bitmap 836. Each of these 38-bit addresses, as well as bitmap 836, may be padded to align each quantity on an 8-byte word boundary within memory 104. In an embodiment, the eight-bit ASCII character set may be used as the search key space so that bitmap 836 may be as long as 256 bits (i.e., 2^8 bits or 32 bytes). In the example depicted in FIG. 8, due to the special reference character "•" and address count 833 of "2," bitmap 836 may be two bits in length and may include a "1" in each bit position corresponding to last character 835.

In an embodiment, and as discussed with reference to key data record 750 (FIG. 7), key data record 850 may include, for example, a variable-length key 853 and at least one terminal data pointer. As depicted in FIG. 8, key data record 850 includes two terminal data pointers, terminal data pointer 857 and terminal data pointer 858. Key data record 850 may be prefixed with a 12-bit key length 851 and a 12-bit terminal pointer count/status 852, and may include padding (not shown for clarity) to align the terminal data pointer 857 and terminal data pointer 858 on an 8-byte boundary in memory 104. Terminal data pointer 857 and terminal data pointer 858 may each contain a 10-bit terminal data type and other data, such as, for example, length, status or data useful in binary record searches. Terminal data pointer 857 and terminal data pointer 858 may be sorted by terminal data type for quicker retrieval of specific resource records (e.g., terminal data record 860 and terminal data record 870).

In another embodiment, and as discussed with reference to key data record 740 (FIG. 7), key data record 840 may include embedded terminal data 846 rather than a terminal data record pointer. For example, key data record 840 may include a key length 841, a terminal pointer count 842, a variable-length key 843, the number of embedded record elements 844, followed by a record element length 845 (in bytes, for example) and embedded record data 846 (e.g., a string, a byte sequence, etc.) for each of the number of embedded record elements 844.

In another embodiment, and as discussed with reference to terminal data record 760 (FIG. 7), terminal data record 860, for example, may include a 12-bit length 861, a 4-bit status, and a variable-length string 862 (e.g., an IP address). Alternatively, variable length string 862 may be a byte sequence. Terminal data record 860 may include padding (not shown for clarity) to align each terminal data record to an 8-byte boundary in memory 104. Alternatively, terminal data record 860 may include padding (not shown for clarity) to a 4-byte boundary, or, terminal data record 860 may not include any padding. Memory management algorithms may determine, generally, whether terminal data records 760 are padded to 8-byte, 4-byte, or 0-byte boundaries. Similarly, terminal data record 870 may include a 12-bit length 871, a 4-bit status, and a variable-length string 872 (e.g., an IP address).

Generally, both search indices, such as OAK trees, and data records may be structured so that 8-byte pointers are located on 8-byte boundaries in memory. For example, vertical node 810 may contain an 8-byte (or less) pointer to vertical node 820, and may be stored at a memory address divisible by eight (i.e., an 8-byte boundary, or 8N). Similarly, both search indices, such as OAK trees, and data records may be structured so that 4-byte pointers are located on 4-byte boundaries in memory. Consequently, modifications to database 800 may conclude by updating a pointer to an aligned address in memory using a single uninterruptible operation, including, for example writing a new pointer to the search index, such as an OAK trees node, or writing a new pointer to a data record.

The various embodiments discussed above with reference to FIG. 8 present many advantages. For example, an OAK tree data structure is extremely space efficient and 8-bit clean. Regular expression searches may be used to search vertical nodes containing multi-character string fragments, since the 8-bit first character (e.g., first character 814), the 8-bit second character (e.g., second character 8-15) and any additional 8-bit characters (e.g., additional characters 816-1 ... 816-N) may be contiguously located within the vertical node (e.g., vertical node

810). Search misses may be discovered quickly, and, no more than N nodes may need to be traversed to search for an N-character length search string.

FIG. 9 is a top level flow diagram that illustrates a method for searching and concurrently updating a database without the use of operating system or database table locks, according to embodiments of the present invention.

An update thread and a plurality of search threads may be created (900). In an embodiment, system 100 may spawn a single update thread to incorporate updates to the local database received, for example, from OLTP server 140-1 over WAN 124. In other embodiments, system 100 may receive updates from OLTP servers 140-1 ... 140-S over WAN 124, and from plurality of network computers 120-1 ... 120-N over WAN 124 or LAN 122. System 100 may also spawn a search thread in response to each session request received from the plurality of network computers 120-1 ... 120-N. For example, a manger thread may poll one or more control ports, associated with one or more network interfaces 114-1 ... 114-O, for session requests transmitted from the plurality of network computers 120-1 ... 120-N. Once a session request from a particular network computer 120-1 ... 120-N is received, the manage thread may spawn a search thread and associate the search thread with that particular network computer (e.g., PE).

In an alternative embodiment, system 100 may spawn a number of search threads without polling for session requests from the plurality of network computers 120-1 ... 120-N. In this embodiment, the search threads may not be associated with particular network computers and may be distributed evenly among the plurality of processors 102-1 ... 102-P. Alternatively, the search threads may execute on a subset of the plurality of processors 102-1 ... 102-P. The number of search threads may not necessarily match the number of network computers (e.g., N).

A plurality of search queries may be received (910) over the network. In an embodiment, plurality of network computers 120-1 ... 120-N may send the plurality of search queries to system 100 over LAN 122, or, alternatively, WAN 124. The plurality of search queries may contain, for example, a search term or key, as well as state information that may be associated with each query (e.g., query source address, protocol type, etc.). State information may be explicitly maintained by system 100, or, alternatively, a state information handle may be provided. In a preferred embodiment, each of the plurality of network computers 120-1 ... 120-N may multiplex a predetermined number of search queries into a single network

packet for transmission to system 100 (e.g., a Request SuperPacket 220 as depicted in FIG. 2).

In an alternative embodiment, a plurality of search queries and the new information may be received (910, 960) concurrently over the network. For example, plurality of network computers 120-1 ... 120-N may send the plurality of search queries and the new information to system 100 over LAN 122, or, alternatively, WAN 124. The plurality of search queries may contain, for example, a search term or key, as well as state information that may be associated with each query (e.g., query source address, protocol type, etc.). The new information may include, for example, additions, modifications or deletions to database, and may be grouped together as a transaction with an associated identifier. For example, in an embodiment, each of the plurality of network computers 120-1 ... 120-N may multiplex a predetermined number of search queries and new information into a single network packet for transmission to system 100, such as, for example, a single Request SuperPacket 220 (new information not depicted for clarity). For those queries that depend upon new information within the transaction, the state information associated with those queries may include the transaction identifier, and, typically, may be maintained by system 100. When the update thread applies the transaction to the database (i.e., e.g., an ongoing transaction), search queries that depend upon the transaction will pend until the update thread successfully completes and commits the transaction.

Each search query may be assigned (920) to one of the search threads for processing. In an embodiment, each search thread may be associated with one of the plurality of network computers 120-1 ... 120-N and all of the search queries received from that particular network computer may be assigned (920) to the search thread. In other words, one search thread may process all of the search queries arriving from a single network computer (e.g., a single PE). In an embodiment, each search thread may extract individual search queries from a single, multiplexed network packet (e.g., Request SuperPacket 220 as depicted in FIG. 2), or, alternatively, the extraction may be performed by a different process or thread.

In another embodiment, the search queries received from each of the plurality of network computers 120-1 ... 120-N may be assigned (920) to different search threads. In this embodiment, the multi-thread assignment may be based on an optimal distribution function which may incorporate various system parameters including, for example, processor loading. Of course, the assignment of search

queries to search threads may change over time, based upon various system parameters, including processor availability, system component performance, etc. Various mechanisms may be used to convey search queries to assigned search threads within system 100, such as, for example, shared memory, inter-process messages, tokens, semaphores, etc.

Each search thread may search (930) the database based on the assigned search queries. In an embodiment, each search thread may extract individual search queries from a single, multiplexed network packet (e.g., Request SuperPacket 220 as depicted in FIG. 2), or, alternatively, the extraction may be performed by a different process or thread. Clearly, searching the database may depend upon the underlying structure of the database. In an embodiment, searching the database may depend upon the modifications contained within a particular transaction for those search queries dependent upon the transaction.

Referring to the database embodiment illustrated in FIG. 4, database 400 may be searched (930) for the search key. The data record (e.g., database record 420) corresponding to the search key may then be determined. Referring to the database embodiment illustrated in FIG. 5, look-aside file 520 may first be searched (930) for the search key, and, if a match is not determined, then snapshot file 510 may be searched (930). The data record corresponding to the search key may then be determined.

Referring to the database embodiment illustrated in FIG. 6, domain name data 610 may first be searched (930) for the search key, and then the resource data within name server data 630, corresponding to the search key, may then be determined. For example, for the "la.com" search key, a match may be determined with domain name record 620 in domain name data 610. The appropriate information may be extracted, including, for example, name server pointer 626. Then, the appropriate name server record 640 may be indexed using name server pointer 626, and name server network address 647 may be extracted.

Referring to the database embodiment illustrated in FIG. 7, the TST may be searched (930) for the search key, from which the resource data may be determined. For example, for the "law.com" search key, search nodes 701 may be searched (930), and a match determined with node 730. Key pointer 736 may be extracted, from which the key data record 750 may be determined. The number of terminal data pointers 752 may then be identified and each terminal data pointer may be extracted. For example, terminal data pointer 757 may reference terminal

data record 760 and terminal data pointer 758 may reference and terminal data record 770. The variable-length resource data, e.g., name server network address 762 and name server network address 772, may then be extracted from each terminal data record using the length 761 and 771, respectively..

Referring to the database embodiment illustrated in FIG. 8, the OAK tree may be searched (930) for the search key, from which the resource data may be determined. For example, for the "law.com" search key, search nodes 801 may be searched (930), and a match determined with node 830. Second address 837 may be extracted, from which the key data record 850 may be determined. The number of terminal data pointers 852 may then be identified and each terminal data pointer may be extracted. For example, terminal data pointer 857 may reference terminal data record 860 and terminal data pointer 858 may reference and terminal data record 870. The variable-length resource data, e.g., name server network address 862 and name server network address 872, may then be extracted from each terminal data record using the length 861 and 871, respectively.

Each search thread may create (940) a plurality of search replies corresponding to the assigned search queries. If a match is not found for a particular search key, the reply may include an appropriate indication, such as, for example the null character. Referring to FIGS. 6–8, for example, a search key might be "law.com" and the corresponding resource data might be "180.1.1.1". More than one name server network address may be associated with a search key, in which case, more than one name server network address may be determined.

The replies may be sent (950) over the network. In an embodiment, each search thread may multiplex the appropriate replies into a single network packet (e.g., Response SuperPacket 240) corresponding to the single network packet containing the original queries (e.g., Request SuperPacket 220). Alternatively, a different process or thread may multiplex the appropriate replies into the single network packet. The response network packet may then be sent (950) to the appropriate network computer within the plurality of network computers 120-1 ... 120-N via LAN 122, or alternatively, WAN 124. In one embodiment, the response packets may be sent to the same network computer from which the request packets originated, while in another embodiment, the response packets may be sent to a different network computer.

The update thread may receive (960) new information over the network. In an embodiment, new information may be sent, for example, from the OLTP server 140-

1 to system 100 over WAN 124. In other embodiments, system 100 may receive updates from OLTP servers 140-1 ... 140-S over WAN 124, and from plurality of network computers 120-1 ... 120-N over WAN 124 or LAN 122. As discussed above, in an embodiment, plurality of network computers 120-1 ... 120-N may send the plurality of search queries and the new information to system 100 over LAN 122, or, alternatively, WAN 124. Consequently, in this embodiment, the plurality of search queries and the new information may be received (910, 960) concurrently over the network.

In the DNS resolution embodiment, for example, the new information may include new domain name data, new name server data, a new name server for an existing domain name, etc. Alternatively, the new information may indicate that a domain name record, name server record, etc., may be deleted from the database. Generally, any information contained within the database may be added, modified or deleted, as appropriate. In an embodiment, several modifications to the database may be grouped together as a transaction and applied to the database as a consistent modification set.

For example, a transaction may include various combinations of database record additions, modifications or deletions. Because search access to the database is not restricted, an indicator field, (e.g., "dirty bit") may be provided within each database record to notify the search threads that, when the dirty bit is set for a particular database record, database modifications associated with a transaction are in progress and a subsequent query-retry of that particular database record is required. Once the transaction has been applied and the modifications are complete, the dirty bits may be cleared for all the new database elements effected by the transaction. In some sense, the new information may be considered to be "committed." Thus, the database may be transformed from one valid state to another valid state without restricting search access to the database.

Advantageously, no operating system or database table locks are required to prevent search queries from accessing the database during these update periods. A slight performance penalty is incurred, because a search query may need to be repeated if the dirty bit is determined to be set for any particular database record. The dirty bit may be located within the most significant word of the database record, so that the bit may be inspected as soon as this word is transferred from memory 104 to processor 102-1, for example. Additional memory transfers associated with the remaining portion of the database record may thus be avoided if the dirty bit is determined to be set. The query-retry period may be on the order of nano-seconds

for the exemplary system embodiments discussed with reference to FIG. 1. Typically, the dirty bit may be cleared before the query-retry accesses the particular database record again.

Alternately, or when a dirty bit is set for during ongoing transaction, the point-in-time consistent query result may be reconstructed from the contents of the redo log, or log manager, for example, as is common practice in transactional databases systems. For search queries that may encounter a dirty bit due to a single in-progress modification that is not part of an ongoing transaction, repeating the query may usually incur a lesser performance penalty than reconstructing the query result from the log manager. Where the dirty bit is due to an ongoing transaction with an extended set of modifications received over an extended period of time, reconstructing the query result from the log manager may be preferred, so that the query result may not be unduly delayed.

While the number of database record modifications within a single transaction is generally unlimited, typically, a transaction includes sufficient information to maintain the atomicity, consistency, isolation and durability of the database. Many different transactions may be envisioned for each database embodiment depicted within FIGS. 4 and 6–8. Referring to FIG. 4, for example, a transaction may include modifying database records 410 and 420, modifying database record 420 and adding a new database record (e.g., database record 430), modifying database record 420 and deleting a database record (e.g., database record 410), etc. Referring to FIG. 6, for example, a transaction may include modifying domain name record 620 and name server record 640, deleting domain name record 620 and adding domain name record 615, etc. Referring to FIG. 7, for example, a transaction may include modifying key data record 750 and terminal data record 760 and deleting terminal data record 770, adding key data record 780 and deleting key data record 740, etc. Similarly, referring to FIG. 8, for example, a transaction may include modifying key data record 850 and terminal data record 860 and deleting terminal data record 870, adding key data record 880 and deleting key data record 840, etc.

The update thread may create (970) a plurality of new elements based on the new information. Typically, modifications to the information contained within an existing element of the database may be incorporated by creating a new element based on the existing element and then modifying the new element to include the new information. During this process, the new element may not be visible to the search threads or processes currently executing on system 100 until a pointer to

the new element has been written to the database. Generally, additions to the database may be accomplished in a similar fashion, without necessarily using information contained within an existing element. In one embodiment, the deletion of an existing element from the database may be accomplished by adding a new, explicit "delete" element to the database. In another embodiment, the deletion of an existing element from the database may be accomplished by overwriting a pointer to the existing element with an appropriate indicator (e.g., a null pointer, etc.). In this embodiment, the update thread does not create a new element in the database containing new information

Referring to FIG. 4, for example, memory space for a new data record (e.g., data record 430) may be allocated from a memory pool associated with database records 401. New information may be copied to data 432 of data record 430, and other information may be calculated and added to data record 430, such as, for example, chain pointer 434, data pointer 435, etc. A dirty bit 408 may also be included within new data record 430. Referring to the database embodiments depicted in FIGS. 6–8, for example, the new information may include new domain names and/or domain name servers to be added to the database.

Referring to FIG. 6, for example, memory space for a new domain name record 615 may be allocated from a memory pool associated with the domain name records 611, or, alternatively, from a general memory pool associated with domain name data 610. The new domain name may be normalized and copied to the new domain name record 615, a pointer to an existing name server (e.g., name server record 655) may be determined and copied to the new domain name record 615. A dirty bit 618 may be included within new domain name record 615. Other information may be calculated and added to new domain name record 615, such as, for example, a number of name servers, a chain pointer, etc. In more complicated examples, the new information may include a new search key with corresponding resource data.

Referring to FIG. 7, in a more complicated example, a new search node 705, as well as a new key data record 780, may be created. In this example, the new search node 705 may include a comparison character ("m"), in the first position, that is greater than the comparison character ("l"), in the first position, of existing search node 710. Consequently, search node 705 may be inserted in the TST at the same "level" (i.e., 1st character position) as search node 710. Before search node 705 is committed to the database, the 4-byte "greater than" pointer 715 of search node 710 may contain a "null" pointer. Search node 705 may also include a 4-byte key

pointer 706 which may contain a 40-bit pointer to the new key data record 780. Key data record 780 may include a key length 781 (e.g., "5") and type 782 (e.g., indicating embedded resource data), a variable length key 783 (e.g., "m.com"), a number of embedded resources 784 (e.g., "1"), a resource length 785 (e.g., "9"), a variable-length resource string 786 or byte sequence (e.g., "180.1.1.1") and dirty bit 707. Memory space may be allocated for search node 705 from a memory pool associated with TST nodes 701, while memory space may be allocated for the key data record 770 from a memory pool associated with plurality of key data records 702.

Referring to FIG. 8, for example, a new search node 890, as well as a new key data record 880, may be created. In this example, the new search node 890 may be a horizontal node including, for example, a two-bit node type 891 (e.g., "01"), a 38-bit first address 892, an eight-bit address count 893 (e.g., 2), an eight-bit first character 894 (e.g., "l"), an eight-bit last character 895 (e.g., "m"), a variable-length bitmap 896 and a 38-bit second address 897. First address 892 may point to vertical node 820, the next vertical node in the "l <...>" search string path, while second address 897 may point to key data record 880 associated with the search key fragment "m." Key data record 880 may include a key length 881 (e.g., "5") and type 882 (e.g., indicating embedded resource data), a variable length key 883 (e.g., "m.com"), a number of embedded resources 884 (e.g., "1"), a resource length 885 (e.g., "9"), a variable-length resource string 886 or byte sequence (e.g., "180.1.1.1") and dirty bit 807. Memory space may be allocated for search node 890 from a memory pool associated with plurality of search nodes 801, while memory space may be allocated for key data record 880 from a memory pool associated with plurality of key data records 802.

The new information may also include several modifications to existing records within the database. Referring to FIG. 4, the new information may include modifications to data record 410. In this example, new data record 420 may be created and the information from data record 410 copied thereto. As above, memory space for data record 420 may be allocated from a memory pool associated with database records 401. The modifications may then be applied to data 422. Data records 410 and 420 may also include dirty bits 406 and 407, respectively.

Referring to FIG. 6, the new information may include modifications to name server record 640, such as, for example, a new IP address (e.g., "180.2.1.2"). In this example, new name server record 660 may be created and the information

from old name server record 640 copied thereto. As above, memory space for name server record 660 may be allocated from a memory pool associated with the name server records 631, or, alternatively, from a general memory pool associated with name server data 630. The new name server IP address may then be copied to the appropriate field within name server record 660, i.e., e.g., name server IP address 667. A dirty bit 668 may be included within new name server record 660. Similar modifications to the various elements within the database embodiments described with reference to FIGS. 7 and 8 are also contemplated.

The new information may also include the deletion of at least one existing element within the database. In one embodiment, no new element may be created, but the dirty bit of the element to be deleted may be set by the update thread. In another embodiment, a new, explicit "delete" element may be created, with the dirty bit set, indicating that the former element has been removed from the database. Referring to FIG. 4, for example, the new information may include the deletion of data record 410, which may include dirty bit 407. Referring to FIG. 6, for example, the new information may include the deletion of domain name record 670, which may include dirty bit 678. Similar deletions to the various elements within the database embodiments described with reference to FIGS. 7 and 8 are also contemplated.

The update thread may set (975) a dirty bit within each of the plurality of new elements. As noted above, the dirty bit may notify the search threads that the particular database record is associated with a current transaction, and that a subsequent query-retry of the database should be performed. Thus, each of the database records effected by a transaction may be identified. Referring to FIGS. 4 and 6–8, for example, the update thread may set a dirty bit within each of the database records affected by the transaction. Dirty bit 408 may be set to "1" for new data record 430 and dirty bits 407 and 406 may be set to "1" for modified data records 410 and 420, respectively. Dirty bit 618 may be set to "1" for new domain name record 615 and dirty bits 606 and 668 may be set to "1" for modified name server records 640 and 660, respectively. Dirty bits 707 and 807 may be set to "1" for new key data records 780 and 880, respectively.

For clarity, the top level flow diagram illustrated in FIG. 9 is extended to FIG. 10 though flow diagram connection symbol "A." Referring to FIG. 10, for database records to be deleted, the update thread may also set (1075) a dirty bit within the appropriate database records. For example, dirty bit 407 may be set to "1" for deleted data record 410 and dirty bit 678 may be set to "1" for deleted domain name

record 670. Data record 420 and 430, domain name record 615, name server record 660 and key data records 780 and 880 may be considered to be "new" elements within the database, while modified data record 410, modified name server record 640, deleted data record 410 and deleted domain name record 670 may be considered to be "old" elements within the database. In these examples, data record 410 is used as both a "modified" data record and as a "deleted" data record.

The update thread may write (980) a pointer to the database using a single uninterruptible operation. Generally, a new element may be committed to the database, (i.e., become instantaneously visible to the search threads, or processes), by writing a pointer to the new element to the appropriate location within the database. As discussed above, this appropriate location may be aligned in memory, so that the single operation includes a single store instruction of an appropriate length. Even though the new elements may be visible to the search threads after the pointer write, the "set" dirty bit notifies the search threads that each new database element may be part of a current transaction, and that a subsequent query-retry, or reconstruction from the redo log, may be necessary. For database embodiments containing multiple indices, it may be possible for one index to contain pointers to "old" elements while another index to contain pointers to "new" elements. Consequently, in the DNS resolution embodiment, for example, two domain name records with the same domain name, or primary key, may exist within the search space simultaneously, but only during a transaction involving that record for a unique index.

Referring to FIG. 4, an 8-byte pointer corresponding to new data record 430 may be written to hash table 403. Referring to FIG. 6, an 8-byte pointer corresponding to new domain name record 615 may be written to hash table 612. Importantly, these hash table entries may be aligned on 8-byte boundaries in memory 104 to ensure that a single, 8-byte store instruction is used to update this value. Referring to FIG. 7, a 4-byte pointer corresponding to the new search node 705 may be written to the 4-byte "greater-than" node pointer 715 within search node 710. Importantly, the node pointer 715 may be aligned on a 4-byte boundary in memory 104 to ensure that a single, 4-byte store instruction may be used to update this value. Referring to FIG. 8, plurality of search nodes 801 may also include a top-of-tree address 899, which may be aligned on an 8-byte word boundary in memory 104 and may reference the first node within plurality of search nodes 801 (i.e., e.g., vertical node 810). An 8-byte pointer corresponding to the new search

node 890 may be written to the top-of-tree address 899 using a single store instruction. In each of these embodiments, just prior to the store instruction, the new data are not visible to the search threads, while just after the store instruction, the new data are visible to the search threads. Thus, with a single, uninterruptible operation, the new data may be committed to the database without the use of operating system or database table locks.

Referring to FIG. 10, for database records to be deleted from the database, in an embodiment, a pointer, or pointers, to the existing record may be written (1080) with a null pointer using a single uninterruptible operation. The null pointer may de-reference the existing record and indicate that the existing record has been deleted from the database. Referring to FIG. 4, for example, data record 410 may be deleted from database 400 by overwriting the appropriate entry within hash table 403 with an
8-byte null pointer. Referring to FIG. 6, for example, domain name record 670 may be deleted from database 600 by overwriting the appropriate entry within hash table 612 with an 8-byte null pointer. In an alternative embodiment, an 8-byte pointer to a new, "explicit" delete record, corresponding to a "deleted" domain name record 670, may be written to hash table 613. In this embodiment, modifications, additions and deletions to the database may be accomplished similarly.

The update thread may clear (985) the dirty bit within each of the plurality of new elements. In an embodiment, the dirty bit may be cleared from each new element by setting the dirty bit to "0." For example, and as discussed with reference to FIGS. 4 and 6–8, dirty bit 406 and 408 may be set to "0" for data records 420 and 430, respectively. Dirty bit 618 may be set to "0" for domain name record 615, dirty bits 606 and 668 may be set to "0" for name server records 640 and 660, respectively. Dirty bits 707 and 807 may be set to "0" for key data records 780 and 880, respectively. In an embodiment, the dirty bit may be set to "0" for each of the new elements in any order. After the dirty bits within each of the new elements have been cleared (985), the "old," or existing, database elements are no longer active, i.e., referenced within the database. In an embodiment, the dirty bits within these elements may then be cleared by setting the dirty bit to "0," while in an alternative embodiment, the dirty bits may not be cleared at all.

In an embodiment, the update thread may physically delete (990) existing database elements that have been modified after the dirty bits are cleared (985) from each of the new elements. Advantageously, the physical deletion of these modified elements from memory 104 may be delayed to preserve consistency of in-

progress searches. For example, after an existing element has been modified and the corresponding new element committed to the database, the physical deletion of the existing element from memory 104 may be delayed so that existing search threads that have a result, acquired just before the new element was committed to the database, may continue to use the previous state of the data. The update thread may physically delete (990) the existing element after all the search threads that began before the existing element was modified have finished.

Similarly, after an existing element has been deleted from the database, the physical deletion of the existing element from memory 104 may be delayed so that existing search threads that have a result, acquired just before the existing element was deleted from the database, may continue to use the previous state of the data. Referring to FIG. 10, the update thread may physically delete (1090) the existing element after all the search threads that began before the existing element was deleted have finished.

Potential complications may arise from the interaction of methods associated with embodiments of the present invention and various architectural characteristics of system 100. For example, the processor on which the update thread is executing (e.g., processor 102-1, 102-2, etc.) may include hardware to support out-of-order instruction execution. In another example, system 100 may include an optimizing compiler which may produce a sequence of instructions, associated with embodiments of the present invention, that have been optimally rearranged to exploit the parallelism of the processor's internal architecture (e.g., processor 102-1, 102-2, etc.). Many other complications may readily be admitted by one skilled in the art. Data hazards arising from out-of-order instruction execution may be eliminated, for example, by creating dependencies between the creation (970) of the new element and the pointer write (980) to the database.

In one embodiment, these dependencies may be established by inserting additional arithmetic operations, such as, for example, an exclusive OR (XOR) instruction, into the sequence of instructions executed by processor 102-1 to force the execution of the instructions associated with the creation (970) of the new element to issue, or complete, before the execution of the pointer write (980) to the database. For example, the contents of the location in memory 104 corresponding to the new element, and containing the dirty bit, may be XOR'ed with the contents of the location in memory 104 corresponding to the pointer to the new element. Subsequently, the address of the new element may be written (980) to memory 104

to commit the new element to the database. Numerous methods to overcome these complications may be readily discernable to one skilled in the art.

Several embodiments of the present invention are specifically illustrated and described herein. However, it will be appreciated that modifications and variations of the present invention are covered by the above teachings and within the purview of the appended claims without departing from the spirit and intended scope of the invention.

What is claimed is:

1. A multi-threaded network database system, comprising:

at least one processor coupled to a network; and

a memory coupled to the processor, the memory including a database and instructions adapted to be executed by the processor to:

create an update thread and a plurality of search threads;

assign each of a plurality of search queries, received over the network, to one of the plurality of search threads;

for each search thread:

search the database according to the assigned search queries,

create a plurality of search replies corresponding to the assigned search queries, and

send the plurality of search replies over the network; and

for the update thread:

create a plurality of new elements according to new information received over the network,

set a dirty bit within each of the plurality of new elements,

without restricting access to the database for the plurality of search threads, write a pointer to each of the plurality of new elements to the database using a single uninterruptible operation, and

clear the dirty bit within each of the plurality of new elements.

2. The system of claim 1, wherein the instructions further include:

for the update thread:

set a dirty bit within at least one existing element to be deleted from the database, and

without restricting access to the database for the plurality of search threads, de-reference the existing element to be deleted using a single uninterruptible operation.

3. The system of claim 1, wherein the instructions further include:

for the update thread:

set a dirty bit within at least one existing element to be modified in the database before the pointer is written to corresponding new element, and

clear the dirty bit within the existing element after the pointer is written to the corresponding new element.

4. The system of claim 1, wherein the single uninterruptible operation is a store instruction.

5. The system of claim 4, wherein the store instruction writes four bytes to a memory address located on a four byte boundary.

6. The system of claim 4, wherein the store instruction writes eight bytes to a memory address located on an eight byte boundary.

7. The system of claim 4, wherein the processor has a word size of at least n-bytes, the memory has a width of at least n-bytes and the store instruction writes n-bytes to a memory address located on an n-byte boundary.

8. The system of claim 1, wherein the plurality of search queries are received within a single network packet.

9. The system of claim 1, wherein the plurality of search replies are sent within a single network packet.

10. The system of claim 1, wherein said restricting access includes database locking.

11. The system of claim 1, wherein said restricting access includes spin locking.

12. The system of claim 11, wherein said spin locking includes the use of at least one semaphore.

13. The system of claim 12, wherein said semaphore is a mutex semaphore.

14. The system of claim 1, further comprising a plurality of processors and a symmetric multi-processing operating system.

15. The system of claim 14, wherein the plurality of search threads perform at least 100,000 searches per second.

16. The system of claim 15, wherein the update thread performs at least 10,000 updates per second.

17. The system of claim 16, wherein the update thread performs between 50,000 and 130,000 updates per second.

18. The system of claim 1, wherein the pointer to the new element is written to a search index.

19. The system of claim 18, wherein the search index is a TST.

20. The system of claim 1, wherein the pointer to the new element is written to a data record within the database.

21. A method for searching and concurrently updating a database, comprising:

creating an update thread and a plurality of search threads;

assigning each of a plurality of search queries, received over the network, to one of the plurality of search threads;

for each search thread:

searching the database according to the assigned search queries,

creating a plurality of search replies corresponding to the assigned search queries, and

sending the plurality of search replies over the network; and

for the update thread:

creating a plurality of new elements according to new information received over the network,

setting a dirty bit within each of the plurality of new elements,

without restricting access to the database for the plurality of search threads, writing a pointer to each of the plurality of new elements to the database using a single uninterruptible operation, and

clearing the dirty bit within each of the plurality of new elements.

22. The method of claim 21, wherein the instructions further include:

for the update thread:

setting a dirty bit within at least one existing element to be deleted from the database, and

without restricting access to the database for the plurality of search threads, de-referencing the existing element to be deleted using a single uninterruptible operation.

23. The method of claim 21, further comprising:

for the update thread:

setting a dirty bit within at least one existing element to be modified in the database before the pointer is written to corresponding new element, and

clearing the dirty bit within the existing element after the pointer is written to the corresponding new element.

24. The method of claim 21, wherein the single uninterruptible operation is a store instruction.

25. The method of claim 23, wherein the store instruction writes four bytes to a memory address located on a four byte boundary.

26. The method of claim 23, wherein the store instruction writes eight bytes to a memory address located on an eight byte boundary.

27. The method of claim 21, wherein the plurality of search queries are received within a single network packet.

28. The method of claim 21, wherein the plurality of search replies are sent within a single network packet.

29. The method of claim 21, wherein said restricting access includes database locking.

30. The method of claim 21, wherein said restricting access includes spin locking.

31. The method of claim 30, wherein said spin locking includes the use of at least one semaphore.

32. The method of claim 31, wherein said semaphore is a mutex semaphore.

33. The method of claim 21, wherein the plurality of search threads perform at least 100,000 searches per second.

34. The method of claim 21, wherein the update thread performs at least 10,000 updates per second.

35. The method of claim 34, wherein the update thread performs between 50,000 and 130,000 updates per second.

36. The method of claim 21, wherein the pointer to the new element is written to a search index.

37. The method of claim 21, wherein the pointer to the new element is written to a data record within the database.

38. A computer readable medium including instructions adapted to be executed by at least one processor to implement a method for searching and concurrently updating a database, the method comprising:

creating an update thread and a plurality of search threads;

assigning each of a plurality of search queries, received over the network, to one of the plurality of search threads;

for each search thread:

searching a database according to the assigned search queries,

creating a plurality of search replies corresponding to the assigned search queries, and

sending the plurality of search replies over the network; and

for the update thread:

creating a plurality of new elements according to new information received over the network,

setting a dirty bit within each of the plurality of new elements,

without restricting access to the database for the plurality of search threads, writing a pointer to each of the plurality of new elements to the database using a single uninterruptible operation, and

clearing the dirty bit within each of the plurality of new elements.

39. The computer readable medium of claim 38, wherein the method further includes:

for the update thread:

setting a dirty bit within at least one element to be deleted from the database, and

without restricting access to the database for the plurality of search threads, de-referencing the element to be deleted using a single uninterruptible operation.

40. The computer readable medium of claim 38, wherein the method further includes:

for the update thread:

setting a dirty bit within at least one existing element to be modified in the database before the pointer is written to corresponding new element, and

clearing the dirty bit within the existing element after the pointer is written to the corresponding new element.

41. The computer readable medium of claim 38, wherein the single uninterruptible operation is a store instruction.

42. The computer readable medium of claim 41, wherein the store instruction writes four bytes to a memory address located on a four byte boundary.

43. The computer readable medium of claim 41, wherein the store instruction writes eight bytes to a memory address located on an eight byte boundary.

44. The computer readable medium of claim 38, wherein the plurality of search queries are received within a single network packet.

45. The computer readable medium of claim 38, wherein the plurality of search replies are sent within a single network packet.

46. The computer readable medium of claim 38, wherein said restricting access includes database locking.

47. The computer readable medium of claim 38, wherein said restricting access includes spin locking.

48. The computer readable medium of claim 47, wherein said spin locking includes the use of at least one semaphore.

49. The computer readable medium of claim 48, wherein said semaphore is a mutex semaphore.

50. The computer readable medium of claim 38, wherein the pointer to the new element is written to a search index.

51. The computer readable medium of claim 38, wherein the pointer to the new element is written to a data record within the database.

52. The system of claim 8, wherein the new information is received within the single network packet.

53. The method of claim 27, wherein the new information is received within the single network packet.

54. The computer readable medium of claim 44, wherein the new information is received within the single network packet.