The Role of In-Memory Database Systems in Routing Table Management for IP Routers

Abstract: Core Internet bandwidth grows at triple the rate of CPU power, but the promise of high-value applications can only be realized by managing much more data traffic at the network's edge. This requires rapid evolution of the fundamental edge infrastructure device, the IP router. To keep pace, routing table management (RTM) software within routers must respond quickly to changing protocol and provisioning requirements, but as demands increase, proprietary routing table implementations encounter limitations in scalability, extensibility, and ease of maintenance.

This paper examines the use of in-memory database systems (IMDS) within RTM software to overcome these barriers. In addition to greater development flexibility, IMDS technology provides built-in data integrity and fault tolerance, while meeting data lookup and update demands needed for RTM software to support IP routing functions. (Performance examples are provided for Linux and Windows 2000.) This solution improves infrastructure vendors' ability to produce new generations of routers faster and at less cost, improving their competitive position.

McObject LLC
33309 1st Way South
Suite A-208
Federal Way, WA 98003

Phone: 425-888-8505
E-mail: info@mcobject.com
www.mcobject.com

Introduction - General Routing Concepts

IP Routing is the main process used by Internet hosts and routers to deliver packets. The Internet is based on a hop-by-hop routing model. Each host or router that handles a packet examines the destination address in the IP header, computes the *next hop* that will bring the packet one step closer to its destination, and delivers the packet to the next hop, where the process is repeated.

To make this work, routers use *routing tables* to compute the next hop for a packet. Routing table lookup processes match destination addresses with next hop addresses. The content of the routing tables is determined both by *routing protocols* responsible for automatic (dynamic) table updates, and by the router's static network management function, for manual updates.

Architecturally, routers are usually divided into three components: the *control plane* that executes routing protocols, stacks, etc., the *management plane* that provides various provisioning and management functions, and the *data plane (forwarding*) that performs wire speed packet processing. (In fact, routers' control and management planes are often discussed as a single unit. Because this paper focuses on dynamic updating and other automated control capabilities, the term "control plane" is used broadly, even when possibly overlapping with management plane functions.)

Modern Internet router architectures perform the forwarding table lookup using either a central CPU or several CPUs positioned at the incoming interfaces. General-purpose CPUs work well for forwarding, at most, hundreds of thousands of packets per second. The port speeds of high-end routers range from hundreds of megabits (Mbps) all the way up to hundreds of gigabits per second. These high speeds imply the use of optical physical interfaces and very wide parallel electrical busses operating at high speeds. For instance, a 10Gbps line would need to be transformed into a bus operating at 100MHz that would be 64 bits wide. Even a 622Mbps interface may transmit up to 1.5 million packets per second. A router with 128 such interfaces would need to cope with 200 million forwarding decisions per second. Therefore, high-end routers use dedicated hardware to perform forwarding.

The control plane has evolved as the site of the router's more complex and data-hungry software functions, which run on a general purpose CPU and operating system. Key control plane software components, discussed below, include routing protocols; the routing table; and the routing table management (RTM) software that installs, modifies and deletes routes (often from multiple protocols) in the routing table, and synchronizes information between the routing and forwarding tables. In the fast-growing Internet environment, routing table management must provide fast *conversions*—that is, it must quickly reflect network topology changes in the router hardware.

Figure 1 shows the role of control, management and forwarding planes in router architecture.

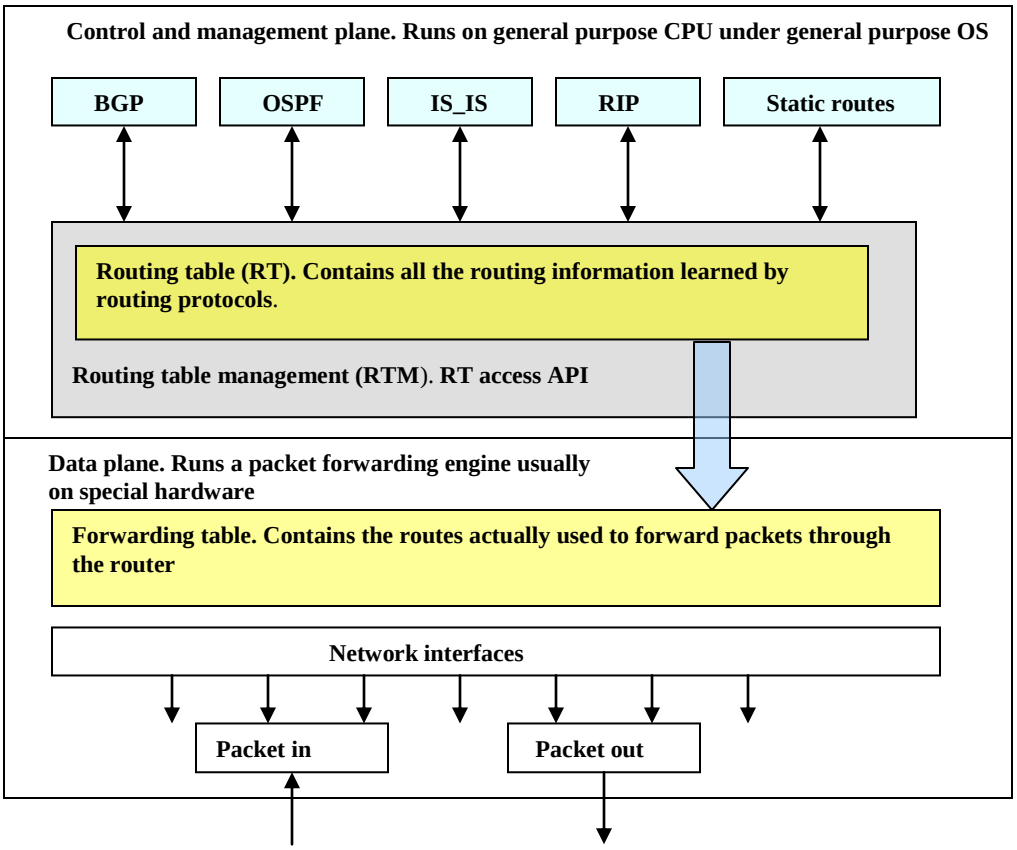

Figure 1

Protocols

Routing protocols determine optimal routing paths through a network, usually based on paths with the least delay and highest reliability. Typically, the best paths are those with the least number of hops, but traffic volume can influence the choice of paths, as well. Most routing protocols facilitate communication between routers and operate in a distributed manner, learning the presence and responsiveness of immediate neighbors through polling and listening routines, and through sharing routing table information with neighbors. Through this process, each router maintains a complete picture of a network in terms of connectivity and quality of specific links. Routers' ability to adapt to a changing network was demonstrated in dramatic fashion on September 11. Within an hour, Internet traffic flows returned to normal as traffic was re-routed around the damaged portions. Popular routing protocols include:

Routing Information Protocol (RIP). This is a distance vector routing protocol, choosing paths based on the distance (number of hops) to the destination. RIP maintains optimal paths by sending out update messages if the network topology changes. For example, if a router finds a faulty link, it will update its routing table, and then send a copy of the modified table to its neighbors. These neighbors update their tables and forward information to others, and so on. Within a short period, all routers will have the new information.

Open Shortest Path First (OSPF). The Internet Engineering Task Force (IETF) developed OSPF, which is often preferred over RIP for larger autonomous system networks. OSPF is a link-state routing protocol (sometimes referred to as distributed-database protocol, but not to be confused with the OLTP DBMS definition of distributed databases.) OSPF routers maintain a

map of the inter-network that is updated after any change to the topology. This map, called the link-state database, is synchronized between all the OSPF routers and is used to compute the routes in the routing table. Neighboring OSPF routers form an adjacency, which is a logical relationship between routers to synchronize the link-state database.

Topology changes are efficiently flooded across the entire inter-network to ensure that the link-state database on each router is synchronized and accurate. Changes to this database trigger a recalculation of the routing table.

As the size of this database increases, memory requirements and route computation times increase. To address this scaling problem, OSPF divides the inter-network into areas (collections of contiguous networks) connected to one another through a backbone area. Each router keeps a link-state database only for areas that are directly connected. Area border routers (ABRs) connect the backbone area to other areas.

Intermediate System to Intermediate System (IS-IS). This link-state protocol offers services similar to OSPF. However, IS-IS was developed by the International Organization for Standardization (ISO) as a part of the Open System Interconnection (OSI) network architecture. IS-IS periodically floods the network with link state information, allowing each router to maintain a complete picture of the network's topology.

Board Gateway Protocol (BGP). This is a protocol for exchanging routing information between gateway hosts (each with its own router) in a network of autonomous systems.

Routing Table Management Software

Routing Table Management (RTM) software provides a means of coordinating information obtained by the routing protocols and statically configured routes, and communicating this data between the control and data planes. In the current generation of routers, the RTM software maintains the routing table (discussed below) as an integrated structure containing persistent data required by control plane processes. The RTM software's functions allow protocols to install new routes (often from multiple protocols) in the routing table, provides means for efficient population of the forwarding data, and supports functionality to communicate new routes back to protocols.

Routing Table

The routing table is a central repository for storing one or more of a protocol instance's preferred routes to a given destination. Different protocol modules use the routing table in different ways. For instance, OSPF maintains its link state database separately and instantiates routes into the routing database when it performs a link state computation. From OSPF, there can be only one route to a given destination. RIP and BGP add routes received in updates from peers, so the routing table may contain as many BGP routes to a destination as there are peers.

While "routing table" suggests a single, straightforward data set, the term actually encompasses various data structures that are of interest to protocol implementations and are maintained by RTM software. For instance, the RTM usually maintains a list of changes to the routing table. This list is communicated to the forwarding plane and to all protocol instances every time a change occurs; protocols, in turn, pass the changes back to the network. Another example is structures that support equal-cost multi-path (ECMP) algorithms, which facilitate routing packets along multiple paths of equal cost. Overall, routing table algorithms are very data hungry – the routing table is accessed frequently with data requests ranging from simple to complex. To add to

the complexity, the routing table can be updated by various protocols simultaneously, requiring the RTM software to synchronize access and maintain data integrity.

To meet these demands, RTM software has typically maintained routing tables as in-memory data structures with multi-threaded access methods that satisfy protocol requirements and forwarding table synchronization policies. However, these proprietary routing table solutions are emerging as one of the principal challenges in developing next generation routing equipment. As device functionality increases, routers encounter significant data management issues in the areas of scalability, reliability, addressing complex data relationships, and ease of programming and maintenance.

Database Technology on the Control Plane?

Historically, developers of traditional (non-embedded) applications have addressed mounting data complexity with database management systems (DBMSs) which provide formalized methods for maintaining data integrity, constructing complex data relationships, and providing access to information quickly and efficiently. By replacing self-developed code with proven database APIs, DBMS technology reduces coding, debugging and maintenance requirements, decreasing the developer's burden.

Commercial database technology has been viewed as unsuitable for real-time processes such as Internet routing. Conventional DBMSs incorporate disk I/O, a mechanical process that is tremendously expensive in terms of performance. Even when deployed in memory—such as on a RAM-disk— conventional databases lag due to caching procedures, and to logging functions that provide for data recovery but are unnecessary for routing tables that are constantly replenished via the routing protocols.

The emergence of in-memory database systems (IMDSs) makes possible the use of true DBMS technology on the router control plane. Designed from the start for memory-only deployment, IMDS technology eliminates disk I/O, caching, logging and other performance overhead of conventional databases. Compared to "traditional" (disk-based) database systems, IMDSs save significant overhead in the following areas:

- There is no connection overhead – data management libraries are typically tightly linked with the application code
- Eliminates extra layers - designed from scratch with the assumption that data is in memory. This design streamlines data management tremendously, removing the layers of overhead typical of disk-based DBMSs
- Search algorithms are highly optimized for memory access
- Search translation – The IMDS points directly to the memory location of data elements. In contrast, conventional DBMSs usually point to a block number and an offset. The database needs to locate the block, load it into memory and find the appropriate memory location in the memory buffer
- IMDSs eliminate the need for buffer management. Conventional DBMSs assume that new data from disk will replace data in memory buffers, and therefore constantly write memory buffers to disk
- IMDS technology provides direct access to data. In a disk-based DBMS an application never gets access to a data element in the memory buffer. Instead, the data is copied elsewhere to memory, adding more overhead yet

Benefit: Simplified Development and Maintenance

In-memory databases are fast, but speed is just one benefit IMDSs bring to the control plane. Existing proprietary routing table implementations have typically been fast enough—but have fallen down in the areas of flexibility and maintainability. For example, new features that require new data to be managed, new access methods, or new relationships between data elements, often require substantial changes to the proprietary routing table data management, which lengthen implementation and debugging cycles, and increase risk.

With a database engine it is much easier to write and maintain the data management portion of the RTM code. Like proprietary routing tables, IMDSs provide an in-memory data repository enabling multi-threaded data access—but with the added benefits of proven data organization, efficient access methods and support for data integrity. Data layout, search algorithms, transaction implementations, error handling, and multi-threaded data access are handled by the database engine code and are of no concern for the RTM developer. In fact, a database management system is a tool that is specifically designed to help maintain data description changes - it is *easy* to change data structures, such as changing a field's data type, and to change access methods, such as adding a route enumeration request.

Carrier-class IP routers often demand high availability for both internal configuration and internal state data. The router architecture is engineered to meet "five-nines" availability requirements (99.999% up-time, which equals 5 minutes down per year) and often provides hardware and software fault-tolerance to support this requirement. The storage subsystem used for building the routing table management is a critical component in meeting this requirement. A DBMS's contribution to fault tolerance can include distributed transactions and two-phase commit, as well as a transaction manager and a distributed inter-process communication (IPC) mechanism that allows coordination between multiple data nodes.

The following examples show how in-memory database technology can enhance control plane development in next-generation routing technology. The database used in the examples is McObject's eXtremeDB, a small-footprint IMDS designed to meet the needs of telecommunications equipment, set-top boxes, and other resource-constrained intelligent devices.

Let's consider a trivial example. The routing table below shows the valid forwarding paths from a given source. Paths are based on static routes, learned via routing protocols, interface addresses, etc.

Address	Mask	Next Hop	Interface	Protocol	Preference	Age	Metric
0.0.0.0	0.0.0.0	100.220.0.57	1	RIP	1	22	2
100.0.0.0	255.0.0.0	0.0.0.0	1	Static	1	0	1
212.0.1.0	255.255.255.0	100.49.0.1	1	OSPF	1	8	1
…							
…							

Address refers to the packet destination IP address to which this route applies. This address is combined with the subnet mask to determine the destination route. 0.0.0.0 indicates the default gateway.

Mask is the subnet mask for the destination IP address in the Address field. The Mask 0.0.0.0 indicates the default gateway.

Next Hop is the IP address of the next system, for remote routes, in the path to the destination. 0.0.0.0 indicates a local route, in which there is no next hop.

Interface refers to the network interface through which traffic moves on this route.

Protocol represents the source of this routing table entry:

- RIP = Learned via Routing Information Protocol.
- OSPF = Learned via Open Shortest Path First protocol.
- Static = Configured static route.
- Default = The default gateway.

Preference is an arbitrary value that is used to rank routes received from different protocols or interfaces. The routing protocol process generally determines the active route by selecting the route with the lowest preference value.

Age represents a timer, which is counting down, and if that timer reaches zero, the route is removed.

Metric specifies the cost of using that particular route. A higher number in the metric column indicates a higher cost for using that route.

Database management systems describe real-world data using a Data Definition Language (DDL), which defines the names and attributes of the data items and data aggregates included in the database, and the relationships that exist and must be maintained between occurrences of those elements. A description of a database based on the DDL is called a *database schema*. The following eXtremeDB schema fragment describes the routing table:

```
class rt
{
 string     destination;
 string     mask;
 string     next_hop;
 string     interface_name;
 uint1      interface;
 uint1      age;
 uint1      metric;
 uint1      preference;

 tree       <destination, mask> route_idx;

};
```

In this example, a *tree index* is maintained for all routes in the database based on the destination/mask pair. Given this design, the RTM software will be able to perform various

routing table lookups or updates. Additional tree-based or hash-based indexes could be declared to satisfy virtually any search requirements. The database could maintain other indexes to support various enumeration capabilities. For example, when a new route is inserted into the table by a protocol, an index could be used to update the forwarding table with the best route based on the destination/mask and the metrics.

The above example is merely used for demonstration purposes. In fact, modern routing tables support multiple protocols, requiring more complex data structures and application queries. In actual implementations, database structures will be driven by protocol specifications, whose revisions, packet formats etc., are defined by Requests For Comments (RFCs) as well as desired features of the router. For example, the OSPF version 2 protocol is documented in RFC 2328, which also specifies the organization of the routing table data, routing table updates and lookups. As specifications evolve to address new requirements, use of a formal DDL greatly simplifies the description and maintainability of the application's data layout. The alternative is to provide management from within the application's source code, using a programming language like C.

Benefit: Isolation of Complex Data Management Functions

Methods provided by the database API enable further *isolation* of protocol software from complex data management functions. Such an "isolation layer" hides the database implementation details and the database access "rules" for transaction processing, cursor navigation, error handling and other functions. For example, in order to update a database, an application must start a transaction, make an update, then commit or roll back the transaction. An isolation layer is based on the API exported by the database runtime and combines all this processing into a simple macro or inline function exposed to protocols.

An isolation layer can also hide some interim data structures from the protocols. For example, when a protocol adds a new route into the database, that route has to be communicated to the other registered protocols and, in some cases, to the forwarding table. To do this, the RTM maintains a "change list" of the most recent routing table changes. The database isolation layer can maintain such a list in the form of a database cursor. A cursor is an entity that establishes a position within a sequence of objects, which may be defined by a tree index. It is possible to maintain a "virtual" change list and use the cursor API to navigate through the database, read the updated data and "notify" registered clients of the change.

In short, in addition to eliminating the need to implement complex data management software, an isolation layer can further simplify the implementation of the remaining RTM functions by providing a compact and straightforward API to routing table data structures.

A simple isolation layer API might take the form of:

Registration API	
rt_attach()	Opens a database session
rt_register()	Registers a database client notification callback
Routing table update API	
rt_add_route()	Adds a new route to the database
rt_update_route()	Updates an existing route

rt_delete_route()	Removes a route from the database
Enumeration API	
rt_enum_routes()	Enumerates all the routes over the entire database
rt_enum_dest()	Enumerates all the destinations in the database
Search API	
rt_find_route()	Searches the routing table for a route that exactly matches the specified route. The route to search for is indicated by a network address, subnet mask, and other route-matching criteria
rt_find_destination()	Searches the routing table for a destination that exactly matches the specified network address and subnet mask
rt_find_best_route()	Searches for the best route based on a route-matching criteria

Figure 2 shows the place for a database engine in the overall RTM architecture.

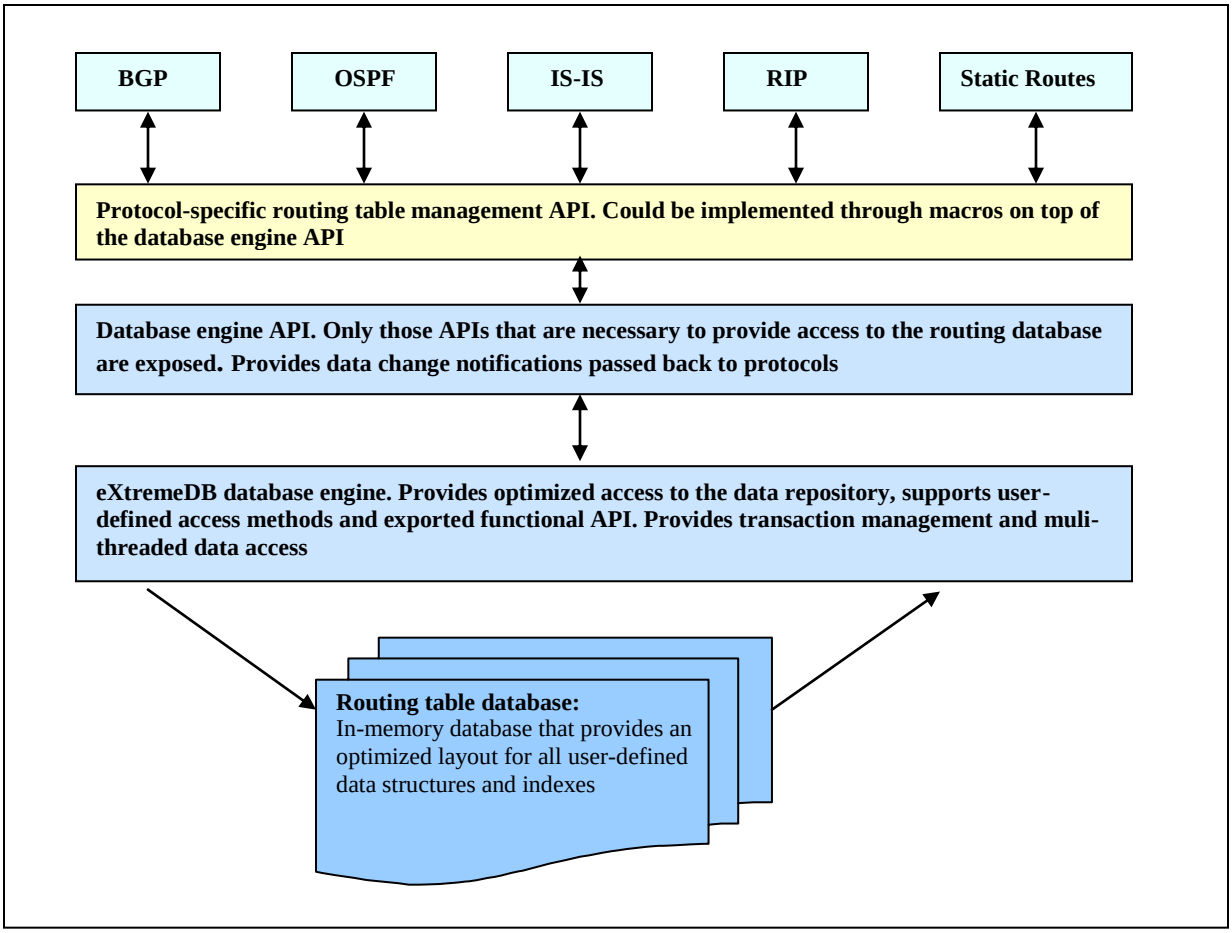

Figure 2

Benefit: Support for Multiple Routing Tables

A database management system greatly simplifies the use of multiple routing tables—a requirement of many emerging routers. For example, a router might maintain three different tables: one for unicast routes, one for multicast routes, and a third for Multi-Protocol Label Switching (MPLS). Today's multi-protocol routers also use multiple routing tables to separate particular groups of routes and add greater flexibility in manipulating routing information. For example, a router might maintain a default unicast routing table, another table for unicast routes used for multicast reverse path forwarding (RPF) lookup, a MPLS routing table for path information, a MPLS routing table for label-switched path (LSP) next hops, and so on. Further, router software might allow configuring a unicast routing table for a particular routing instance.

A DBMS to support data storage and data access methods for multiple routing tables provides a standard and robust way of mapping between them. A database engine naturally supports various ways of referencing related data through indexes, object identifiers, etc., and exposes a standard data access API that allows quick programmatic methods to easily find related objects. Scenarios in which this would prove useful include importing interface routes into more than one routing table, and applying different routing policies when exporting the same route to different peers.

The Performance Issue

Can a database deliver the required performance? Most routers have used proprietary routing table management due largely to concerns over this issue. Modern routers can hold up to several million entries in their routing table and should be able to perform at least hundreds of thousands of lookups per second. The database management systems intended for disk-based data repositories are algorithmically designed to minimize disk I/O; this is achieved at significant cost in CPU cycles dedicated to disk avoidance. In order to achieve the necessary performance, a memory-based database management system should be used, because the ultimate "performance" goal of the memory-based DBMS is to minimize CPU cycles since disk I/O is eliminated by definition.

The sample program below measures the performance of an in-memory database – eXtremeDB – for routing table inserts and lookups. In actual usage, the routing table software would perform more inserts than lookups. However, the goal of this application is not to provide an absolutely true-to-life example, but rather to prove the point that a database management engine can provide sufficient performance.

The program randomly generates IP addresses, each masked with three masks – 255.255.0.0, 255.255.255.0 or 255.255.255.255. We then insert these routes into the routing table, creating a unique tree-based index based on the {mask/ip address} pair. The database engine filters out duplicate pairs of {mask/ip address} as the new routes are installed into the routing table.

After the table is populated, the program searches for a route for a random packet. The algorithm is a combination of a linear search and a tree search, masking the address with each of the three possible masks and then using the index to locate the route. Therefore each address is searched up to three times to find the best match. If more than one route exists, the one with the longest matching prefix is chosen.

The following table reports the performance of insert operations and lookups using the eXtremeDB 1.2 in-memory database on Windows 2000 with a Pentium 4 1.4 GHz CPU and

256Mbytes of RAM, and Red Hat Linux 7.1 with a Celeron 1.1 GHz CPU and 512Mbytes of RAM. All times are in microseconds (there are one million microseconds per second).

	Win2K w/Tree Index	Linux w/Tree Index
Route insert	**5.75**	**7.7**
Total search time for a single route	**6.5**	**7.3**
Per read (total search time divided by the average number of reads per transaction)	**3.25**	**3.6**

The example program used the following eXtremeDB database schema:

```
/*******************************************************
 *                                                     *
 * Copyright(c) 2001 McObject, LLC. All Right Reserved.   *
 *                                                     *
 *******************************************************/
#define uint4 unsigned<4>
#define uint2 unsigned<2>
#define uint1 unsigned<1>

declare database Rt1 [ 1000000 ];


/* This class contains all possible masks */
class Mask
{
  uint4 mask;
  uint1 nbits;

  unique tree< nbits, mask > all;
};

class Route
{
  uint4 dest;
  uint4 mask;
  uint4 gateway;
  uint4 interf;
  uint2 preference;
  uint2 metric;

  unique tree < mask, dest > byMaskDest;
};
```

Figure 3

Two classes are declared for the database. Class Mask is used to hold network masks sorted by the number of significant bits in the mask (the longest mask is the last one). The code fragment in Figure 4 calculates the number of significant bits in the mask. Class Route represents the routing table itself – *dest* is a destination network address, *mask* is the mask for the network, *gateway, interf, metric* and *preference* fields (while being filled in) are not otherwise used in the example. The class is sorted by the pair <mask, *dest*>.

```
static uint1 calcNbits(uint4 mask)
{
  static int n8[16] = {0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4};
  int rc = 0, i;
  for(i =0; i <8; i++)  {
    int halfbt = ( mask >> (i<<2) ) & 15;
    rc += n8[halfbt];
  }
  return (uint1)rc;
}
```

Figure 4

```
static MCO_RET addRoute(mco_db_h db,
  uint4 dest, uint4 mask, uint4 gatew, uint4 interf, uint2 pref,
  uint2 weight)
{
  MCO_RET     rc;
  mco_trans_h t;
  Mask        hmask;
  Route       rt;
  uint1       nbit = calcNbits(mask);

  /* open write transaction. All database updates must be done in
     the context of a "write" transaction */
  rc =
   mco_trans_start(db,MCO_READ_WRITE,MCO_TRANS_FOREGROUND,&t);
  if(rc) return rc;
  /* add a new mask if necessary */
  rc =  Mask_all_find( t, nbit, mask, & hmask );
  if(rc == MCO_S_NOTFOUND){
    /* allocate a new mask object */
    rc = Mask_new(t, & hmask);
    if(rc) goto End;

    /* add values for the mask and nbits */
    Mask_mask_put(& hmask, mask);
    Mask_nbits_put(& hmask, nbit);
  }  else  {
    if(rc) goto End;
  }
  /* add a new route */
  rc = Route_new(t, &rt);
  if(rc) goto End;

  Route_dest_put(&rt, dest);
  Route_mask_put(&rt, mask);
  Route_gateway_put(&rt, gatew);
  Route_interf_put(&rt, interf);
  Route_interf_put(&rt, pref);
  Route_metric_put(&rt, weight);

End:
  rc = mco_trans_commit(t);
  return rc;
}
```

Figure 5

The code fragment in Figure 5 inserts (installs) new routes into the table. In this code fragment, a write transaction is opened, a new Route object is allocated (Route_new()) and written (Route_XXX_put()) and the transaction is committed. Note that there is no need for application to do anything to maintain indexes – they are built on the fly by the database engine. If any of the database operations within the transaction brackets (between mco_trans_start() and mco_trans_commit()) fail, then mco_trans_commit() will handle that situation gracefully and internally roll back the entire transaction, leaving the database in a consistent state.

The last code fragment, Figure 6, searches for the best route. A read-only transaction is started. A database cursor is built based on the "mask" index (Mask_all_index_cursor()). The cursor is traversed from the last object backwards, since the index is ordered so that the longest mask is the last. For each mask, the "find" (Route_byMaskDest_find()) method is used to match the destination address.

```
/* This function searches for a best match route in the database */
static int searchRoute
 (mco_db_h db, uint4 ip, uint4 * res_gatew, uint4 * res_interf)
{
  mco_cursor_t csr; // will hold a database cursor
  Mask hmask;       // will hold a handle to the Mask object
  uint4 mask;       // will hold mask value read from the database
  mco_trans_h t;    // transaction handle
  int ok = 0;       // status

/* start a read-only transaction; it is necessary to perform
     search-by-cursor operation */
  MCO_RET rc =
   mco_trans_start(db,MCO_READ_ONLY,MCO_TRANS_FOREGROUND,&t);

 /* make sure the transaction is started successfully */
  if(rc) return rc;
  /* Create a database cursor based on Mask_all tree index */
  rc = Mask_all_index_cursor(t, &csr);
  if(rc) goto End;

/* using a cursor, navigate through all possible masks */
  for(rc = mco_cursor_last(t, &csr); rc == MCO_S_OK;
      rc = mco_cursor_prev(t, &csr) ) {
    Route rt;
    /* obtain a handle to the Mask object from cursor */
    Mask_from_cursor(t, &csr, & hmask);
    /* using the handle, read mask value */
    Mask_mask_get( & hmask, & mask );

    rc = Route_byMaskDest_find(t, mask, mask & ip, & rt);
    if(rc == MCO_S_OK) {
      ok = 1; break;
    }
  }
 End:
  /* commit our transaction */
  rc = mco_trans_commit(t);
  return ok;
}
```

Figure 6

"No hassle" performance improvement of the test

There are ways to further improve the performance of the sample. Consider a trivial one – instead of building a unique tree index for the destination/mask pair, declare a hash index. The following schema reflects the change:

```
/*********************************************************
 *                                                       *
 * Copyright(c) 2001 McObject,LLC. All Right Reserved.   *
 *                                                       *
 *********************************************************/

#define uint4 unsigned<4>
#define uint2 unsigned<2>
#define uint1 unsigned<1>

declare database Rt1 [ 1000000 ];

class Mask
{
  uint4 mask;
  uint1 nbits;

  unique tree< nbits, mask > all;
};

class Route
{
  uint4 dest;
  uint4 mask;
  uint4 gateway;
  uint4 interf;
  uint2 preference;
  uint2 metric;

  /* this is new: we are declaring a hash index instead of a tree-
     based index*/
  hash < mask, dest > byMaskDest[2000000];
};
```

Figure 7

As shown in the following table, this change dramatically increases performance. The "price" for the performance increase is an extra 116K of memory used for the hash table. *But no changes are required in the sample code!*

	Win2K w/Hash Index	Linux w/Hash Index
Route insert	**3**	**3.75**
Total search time for a single route	**4**	**4.5**
Per read (total search time divided by the average number of reads per transaction)	**2**	**2.25**

Conclusion

Dramatic Internet traffic growth rates have ISPs worried. While CPU speeds may double every 18 months (Moore's law), Internet bandwidth grows three times as rapidly (Gilder's law). More traffic means today's routers need a huge performance boost, which providers are addressing via more powerful CPUs and network adapters, and increasingly sophisticated parallel architectures.

However, speed isn't the only problem. Within hardware routing technologies that realize wire speed transport capabilities, next-generation carrier-class routers need routing software that can scale as operators expand their networks and develop relationships with revenue-sharing partners. Each routing protocol should accommodate maximum expansion to support new services, subscribers and providers. Routing hardware must support a full suite of unicast routing protocols, including carrier-class implementations capable of restoring connectivity from any source to any destination very quickly, in the event of any link or router failure. These demands are outpacing the conception of routing tables as a proprietary outgrowth of routing table management software.

Until recently, commercial database technology was unsuitable for near real-time IP routing. However, the emergence of in-memory databases and other advances in database technology allow the application of DBMS technology to non-traditional areas, including routing table management. For IP router developers, proven database technology provides the benefits of DBMS features, including optimized access methods and data layout, standard and simplified navigation methods, built-in concurrency and data integrity mechanisms, and improved flexibility and fault-tolerance, while delivering the necessary performance. Adoption of this new breed of DBMS as a control and management plane software component simplifies IP router development, while addressing inter-network growth and ensuring high availability and reliability.