Extending a Persistent Object Framework to Enhance Enterprise Application Server Performance

John Grundy[1], Steve Newby[2], Thomas Whitmore[2] and Peter Grundeman[2]

[1]Department of Computer Science, University of Auckland
Private Bag 92019, Auckland, New Zealand

`john-g@cs.auckland.ac.nz`

[2]XSol Ltd
Parnell, Auckland, New Zealand

`{steve, thomas, peter}@xsol.com`

Abstract

High-volume transaction processing speed is critical for adequate performance in many enterprise application servers. We describe our experiences using an object-oriented persistency framework to achieve greatly enhanced server response by the transparent use of main-memory database technology. We took an application server whose data persistency is abstracted via a persistent object framework and replaced a version of the framework using a relational database for persistency with one that uses a memory database. No changes to any of the application server components were necessary to achieve this and we achieved between 10-20 times transaction processing performance improvement. We briefly discuss some extensions to our memory database and mapping framework necessary for large-scale enterprise systems support and for data-oriented systems integration. We hope our experiences will be useful for others, both in terms of techniques for abstracting object persistency mechanisms and in approaches to application server performance enhancement.

Keywords: persistent object frameworks, main-memory databases, transaction processing performance

1 Introduction

Most E-commerce systems use a standard multi-tier architecture: multiple clients (web-based for customers; typically desktop applications for staff; and other organisation's servers for Business-to-Business data exchange) connect to middle-tier application server(s), which in turn connect to database server(s) (Aleksy et al 1999, Bass et al 1998, Vogal, 1998). Despite the widespread use of multiple components and threads in the middle and database tiers, performance of the application server is usually the key bottleneck in such a system (Bass et al 1998, Vogal 1998). This in turn is usually due to bottlenecks accessing and modifying data in the database (Liu and Gorton 2000).

Several solutions exist for this problem, including the use of higher-performance database servers, local area networks and database server hosts (Liu and Gorton 2000); use of object and result set caching by the database or application server (GemStone Systems Inc 2000, Gorton et al 2000); use of main-memory or "real-time" databases (Kim and Bae 2000, Lu et al 1999); and further division of application server components and database tables across multiple hosts (Bass et al 1998, Vogal 1998). Each of these approaches has advantages and disadvantages, many requiring considerable design and implementation changes to the application server components or reorganisation of the database-managed information.

We describe our experiences in achieving a large performance enhancement for an E-commerce application server using a typical multiple-client, single-server, single-database 3-tier architecture. The application server originally used a relational database to manage enterprise information, accessed through a object persistency framework i.e. application server objects were made persistent or obtained from the database via this persistency framework. Our approach involved replacing the RDBMS persistency store implementation with a main-memory database (MMDB) implementation, without modifying any of the persistency framework interfaces the application server relied upon. The RDBMS was retained as the MMDB "mirror" (i.e. non-volatile physical storage) for simplicity. A very large performance improvement was achieved for the application server transaction throughput without having to resort to major architectural modification, expensive OO database or cache purchase, expensive hardware purchase, nor even complete replacement of the RDBMS server.

We motivate this work with an overview of our E-commerce system architecture and compare and contrast various possible solutions to system performance enhancement. We describe the software architecture of our solution, the design of our object persistency layer and our main-memory database. We discuss the integration of the application server with the new persistency mechanism and discuss a variety of performance measures taken, concluding by reviewing the advantages and disadvantages of our solution.
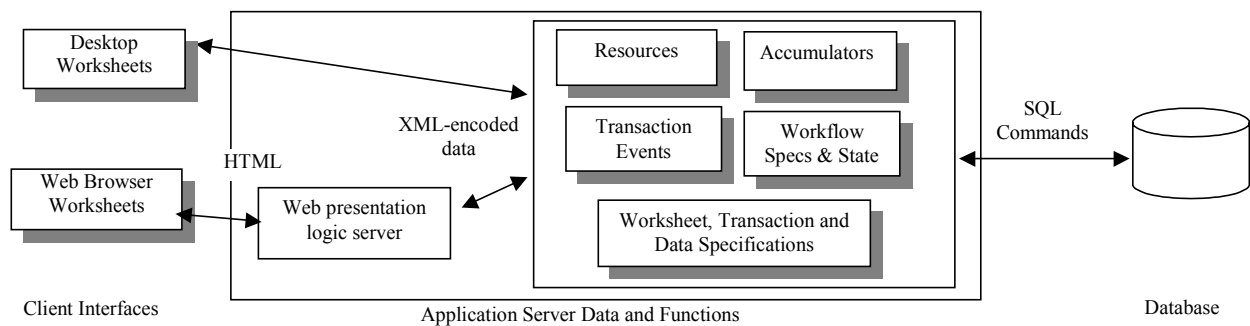
Figure 1. The XSol™ application architecture

2 Motivation

XSol Ltd have been developing a multi-tier enterprise system development product based around a conceptual framework, Enterprise Systems Logic™ (ESL) (Blackham et al 2001). This system provides business analysts facilities to define enterprise applications using a combination of workflows, transaction stages, and spread-sheet style forms and reports ("worksheets"). User interfaces include specification tools (business structure, workflow, transaction stage, worksheet design and reporting) and end user interfaces (workflow visualisation, to-do lists, worksheets and report output). An application server includes workflow and to-do list state, "reources" (data used by business transactions, like Customers, Products and Supplers), "transaction events" (business transaction state) and "accumulators" (data aggregation, like MonthlyProductSales and TotalCustomerOrders), along with various meta-data (i.e. workflow, worksheet, transaction and data specifications). A database stores application server state and meta-data.

The application server includes interfaces for XSol clients, exchanging XML-encoded data and meta-data with the front-end interfaces. These tools may be desktop clients or web-based clients. An interface to the database provides persistency management for the application server. The application server, database and web presentation server can run on the same host or on different hosts. Figure 1 illustrates this basic multi-tier architecture.

The enterprise specification tools and the workflow enactment engine are both light-weight, with a low number of clients and low amount of application server and database loading. The main task of the application server is to manage business transactions. When business transactions are run, these typically require resources to be looked up (find customer, product, suppler etc), a transaction event to be generated and stored, and accumulators (i.e. data aggregation used to control business processing logic and for reporting) to be updated. All of these tasks require application server data to be loaded from and stored to the database. For example, when ordering a product, a "ProductOrder" transaction might be created (and stored in the database), required customer and product sold data looked up in the database, and several accumulators updated (TotalProductSold for the product; AmountBoughtPerMonth for the customer, etc). We found that the performance of the original XSol application server was unsatisfactory due to the overhead of the high number of data selections and updates during business transaction processing.

There are a number of approaches that could be taken to improve our application server's performance. One common way is to modify its design and implementation to make use of a multi-tier architecture where the application tier is split into multiple, concurrent processes to exploit multiple processors or hosts (Bass et al 1998, Sessions 1998, Vogal 1998). Unfortunately profiling of the currently single process application server has shown most of the time spent processing transactions is in the RDBMS, so while a multi-tier solution would provide some benefit, as demonstrated in others' work (Liu and Gorton 2000, Sessions 1998), this is likely to be quite limited at a cost of major re-engineering. As the application server uses persistent objects, management performance may be enhanced by the use of an OODBMS or persistent object store (GemStone Systems Inc 2000, Excelon Corp 2001, Secant Corp 2000), or remote objects via CORBA, COM or Enterprise Java Beans (Liu and Gorton 2000, Sessions 1998, Vogal 1998). OODBMSs require transactions to be committed using similar storage mechanisms to a RDBMS, however, and DCOM and CORBA-based remote object systems often use an RDBMS themselves. Object caching (GemStone Systems Inc 2000, Wu 1999) and main-memory databases (Kim and Bae 2000, Lu et al 1999, TimesTen Inc. 2000) offer potential large performance gains. Caching holds object state in memory, enhancing object reads, but typically writes it through to persistent storage on writes, which become the bottleneck. Main-memory databases (MMDBs) utilise large main-memories to hold the entire database, with a non-volatile secondary storage copy of the database (typically disk). MMDBs provide in-memory complex multi-field indexing, which caches often do not, and typically write update events to a high-speed transaction log rather than doing disk-based indexed file updates like RDBMSs and OODBMSs. The added attraction of a MMDB is that changing an application server to make use of it is very simple, provided the object persistency framework in the application server adequately abstracts developers from the persistency system.
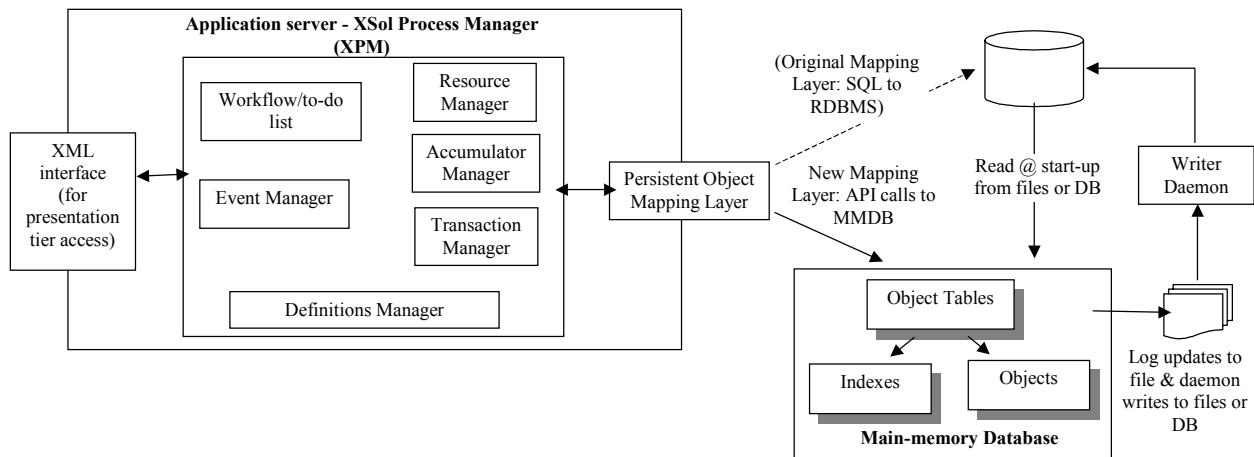
Figure 2. Basic architecture of our application server and data management tiers

3 Application Server Architecture

In order to improve the performance of our application server, and make it more amenable to supporting data-oriented integration with other enterprise systems, we developed a simple main-memory database (MMDB) to provide its data management capabilities. The architecture of our application server is illustrated in Figure 2.

The application server, or XSol Process Manager (XPM), is comprised of an XML interface for clients (effectively the presentation tier), data definition (meta-data) management, a workflow engine, exception manager, resource manager, transaction manager and accumulator manager (the middle tiers), and a persistent object mapping framework and database (the data management tier).

We developed a new version of our persistent object mapping framework, preserving the original object interfaces all of the XPM manager components used, which utilises a simple main-memory database to provide all XPM data. The main-memory database comprises an API, "tables" of persistent objects, and multi-field indexes on objects. On start-up, the MMDB loads its object tables from files (or a database) and thereafter all queries are handled by traversing the in-memory MMDB indexes. Updates to MMDB objects are logged to a file in transaction groupings (to preserve them in case of MMDB failure). A "writer daemon" reads log entries and applies to updates to the MMDB's disk-based mirror: either an ISAM-style indexed file structure or a database. We currently use an RDBMS as physical storage mirror for our MMDB, allowing us to use the exact same RDBMS tables the original mapping layer used, and allowing us to swap between RDBMS and MMDB as the application server's data management support. We built our own MMDB rather than use a commercial one as we only required basic data management functionality: our XPM application server is the only client; we could use our RDBMS as the main-memory database mirror and only simple transaction and crash-recovery support were necessary.

The design of the mapping framework, main memory database and some XPM persistent objects, is outlined in

Figure 3. Classes stereotyped <<XPM>> are unchanged in both versions of our XPM. Classes prefixed <<MMIF>> implement the same interface as the RDBMS interfacing persistent object classes, but provide transparent access to the main-memory database. MMDB-prefixed classes implement the MMDB. All XPM objects that need to be made persistent are specialisations of PersistentObject. All <<XPM>> and <<MMIF>> classes in this diagram have corresponding interfaces e.g. IPersistentObject, IPersistentBroker, ISelectPerformer etc which are accessed by the XPM and are the same for both versions of the framework. This allowed us to modify the implementation classes to provide our MMDB data persistency without having to change any code in the XPM itself. When starting up the XPM, either a MMDBPersistentBroker or RDBMSPersistentBroker object is created, both implementing the IPersistentBroker interface, to give the XPM either RDBMS or MMDB data persistency. If using the same RDBMS as the main-memory database mirror, these can even be swapped over while the XPM is running (which we found very useful for testing and performance analysis).

Each different type of PersistentObject has a ClassMap meta-data that specifies its corresponding DB table (or MMDB object table) the PersistentObject is mapped to. XPM objects can be mapped 1:1 to DB objects, split over several DB objects, or several XPM objects can be mapped to a single DB object. For example, Resource XPM objects are actually comprised of a Resource object and each resource field is represented by a Field object. This object aggregate is mapped onto a single DB table for efficient storage e.g. a "Product" resource object and its field objects are mapped to a single "DBProduct" object in the MMDB or a single row in the RDBMS.

The persistency framework provides a set of classes used to formulate queries over persistent objects and a set to perform updates. PersistentCriteria and its SelectCriteria aggregates are created by the XPM to formulate multi-field and multi-object queries. The original SelectPerformer in our RDBMS version of the mapping framework implementation translated these into SQL to run on the database, and translated the SQL result set into PersistentObject creations.
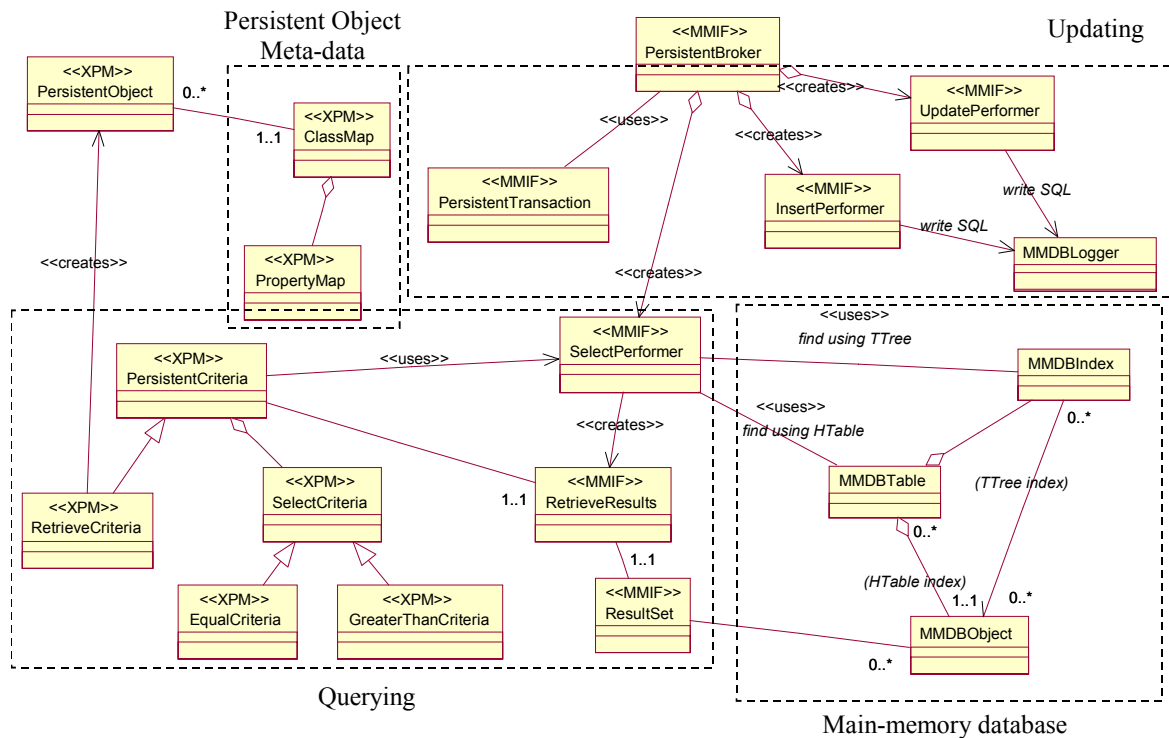
Figure 3. High-level OOD for the architecture

Our new mapping framework implementation of SelectPerformer translates these queries into API calls on the MMDBIndex tables, and translates lists of returned MMDBObjects into appropriate PersistentObject instances, using the appropriate ClassMap transformations. The PersistentTransaction, InsertPerformer, UpdatePerformer and DeletePerformer implementations in the RDBMS implementation of the framework generate SQL INSERT, UPDATE, DELETE and begin/commit transactions.

The MMDB implementation of these classes run MMDBTable API calls to add, update and delete MMDBObjects. The MMDBTable logs these updates to its transaction log, which is processed asynchronously by the writer daemon. A MMDB transaction stores MMDBObject updates as event objects which can be reversed to rollback the transaction. The transaction log is not written to until MMDB transaction commit, for efficiency.

4   Persistent Object Mapping Layer Design

In this section we briefly describe some of the key mapping layer interface functions, and discuss implementation differences between the RDBMS and MMDB versions of the framework that we developed. The PersistentObject is the root class for all XPM objects which need to be made persistent. It provides Save() and Retrieve() functions used to persistent an object or load it using its unique Object ID (Oid). Getter and setter functions provide reflective access to a persistent object's properties by name. A ClassMap provides a specification for mapping persistent objects and their properties to and from corresponding database table(s).

A PersistencyBroker provides a façade allowing transactions to be created, object state to be saved, objects to be deleted and objects to be retrieved based on selection criteria. Inserts, updates and deletes are delegated to "Performers" which use the ClassMap and given PersistentObject type to formulate appropriate SQL update statements to be run on the database. The PropertyMaps associated with a PersistenctObject's ClassMap provide names of object properties and Getter methods are called to extract the property values for INSERT and UPDATE commands. Transaction objects are used to record a set of persistent objects whose state has changed and then to action their Save() functions within a database transaction, rolling back if an update fails.

Objects are retrieved using their unique Oid, constraints on one or more of their property values, or by constraints on the properties of related persistent objects, linked to the PersistentObject by their Oid values. The XPM constructs a RetrieveCriteria object, adds required property constraints, joins and order by criteria, and then asks the PersistencyBroker to perform the query. A SelectPerformer translates the query into a SQL SELECT statement run against the database. After the query is run, new PersistentObject instances are created for each object located, the field values in the result set are copied into the PersistentObject's properties using the Setter methods and PropertyMap property names. The XPM obtains a list of the newly created PersistentObject objects that matched the query via GetObjects(). The PersistencyBroker provides some additional functionality, for example to create and drop tables, obtain database meta-data and so on.
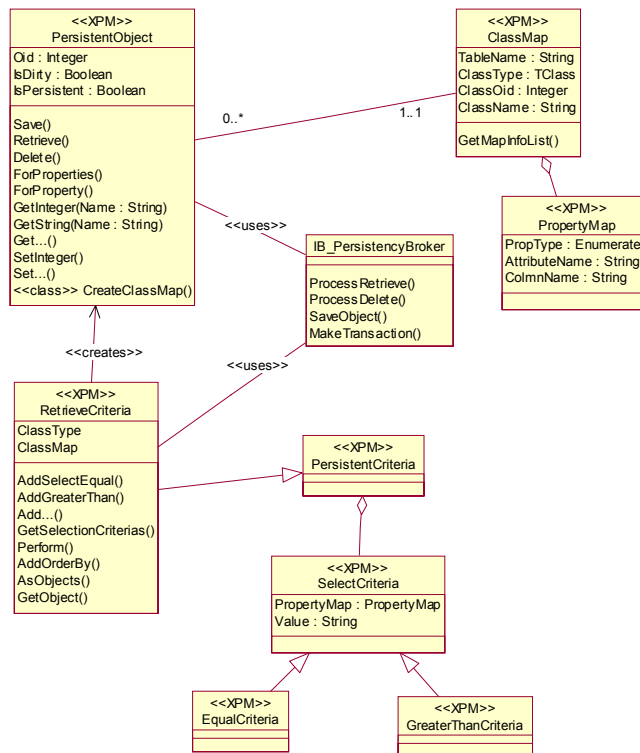
Figure 4. The persistent object mapping framework design

The XPM application server, persistent object layer and main-memory database are implemented in Delphi. The RTTI reflection mechanism is used to dynamically determine and call the get and set functions for PersistentObject subclass properties. This allows any XPM class to be made persistent by inheriting from PersistentObject and declaring its object attributes to be made persistent as properties, specifying set and get functions for each property. By performing all persistent operations via this persistent object framework, the XPM application server is isolated from the actual persistency mechanism used. By exchanging our original RDBMS persistency broker implementation, leaving unchanged the mapping layer interfaces the XPM uses, we re-implemented the mapping layer to use a main-memory database. The XPM runs as before but with its persistency needs provided by memory-resident data.

5   MMDB Design and Implementation

In this section we briefly discuss the design and implementation of our main-memory database and the re-implemented mapping framework classes that provide the XPM transparent persistency via this MMDB. Figure 5 shows some of the classes and their functions used in the mapping framework implementation and MMDB. The main-memory database comprises a set of MMDBTable objects, one for each group of same-typed MMDBObjects. An MMDBTable corresponds to an in-memory version of a RDBMS or OODBMS table. It indexes its MMDBObjects via a hashtable, keyed by the MMDBObject unique Object ID (Oid). An MMDBObject holds PersistentObject property values in a "packed" form to minimise memory usage. Each table has a single hashtable index on the Oid value of each MMDBObject it manages. Each MMDBTable also has zero or more MMDBIndex objects, which implement secondary T-Tree based ordered indexes. We use a T-tree rather than a B-Tree to provide multi-field, ordered indexes, as the T-tree gives better in-memory performance than a B-Tree (more commonly used for RDBMS indexes) (Lu et al 1999).

The UpdatePerformer and InsertPerformer classes originally constructed SQL update commands to achieve object creation and mofication, used to implement a PersistentObject.Save(). We modified these to instead look-up, create and modify MMDBObjects using the MMDBTable hash index. They also log the new MMDBTable state using a MMDBLogger. Currently they log SQL commands which are asynchronously run on the RDBMS mirror of the main-memory database (though we plan to log more compact change events and use indexed files for the MMDB mirror in the future).

Queries are translated by a SelectPerformer into findObjects(low_value, high_value) function calls on a MMDBIndex. Only simple optimisation of queries is currently used by our MMDB, but these have proved sufficient for our XPM's query needs to date. The index used returns a list of MMDBObjects which are further filtered by any criteria not used in the index. Joins are made to other MMDBTables using Oid references and filtering applied to joined MMDBObject property values.
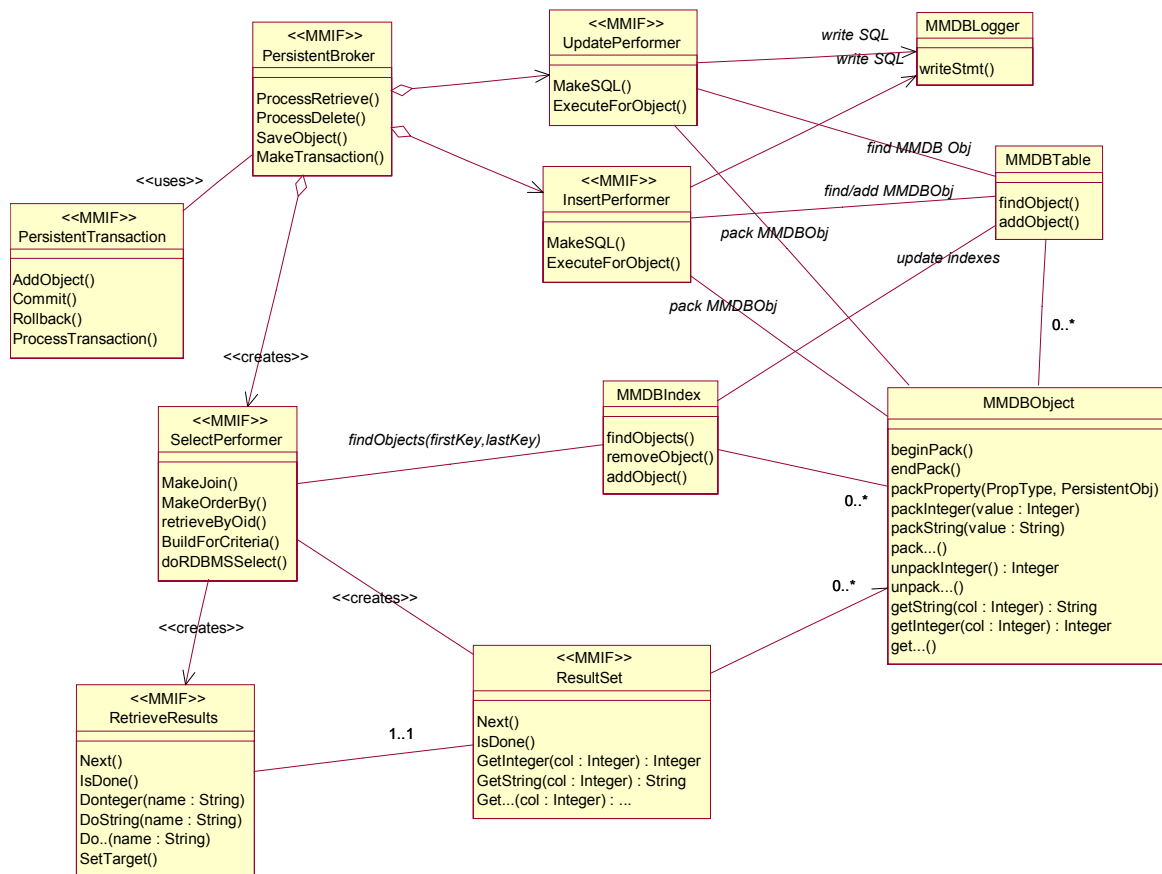
<<MMIF>>
PersistentBroker

ProcessRetrieve()
ProcessDelete()
SaveObject()
MakeTransaction()

<<MMIF>>
UpdatePerformer

MakeSQL()
ExecuteForObject()

MMDBLogger

writeStmt()

write SQL
write SQL

<<uses>>

<<MMIF>>
PersistentTransaction

AddObject()
Commit()
Rollback()
ProcessTransaction()

<<MMIF>>
InsertPerformer

MakeSQL()
ExecuteForObject()

find MMDB Obj

find/add MMDBObj

MMDBTable

findObject()
addObject()

pack MMDBObj

update indexes

pack MMDBObj

<<creates>>

<<MMIF>>
SelectPerformer

MakeJoin()
MakeOrderBy()
retrieveByOid()
BuildForCriteria()
doRDBMSSelect()

findObjects(firstKey,lastKey)

MMDBIndex

findObjects()
removeObject()
addObject()

MMDBObject

beginPack()
endPack()
packProperty(PropType, PersistentObj)
packInteger(value : Integer)
packString(value : String)
pack...()
unpackInteger() : Integer
unpack...()
getString(col : Integer) : String
getInteger(col : Integer) : Integer
get...()

0..*

0..*

0..*

<<creates>>

<<creates>>

<<MMIF>>
RetrieveResults

Next()
IsDone()
Donteger(name : String)
DoString(name : String)
Do..(name : String)
SetTarget()

1..1

<<MMIF>>
ResultSet

Next()
IsDone()
GetInteger(col : Integer) : Integer
GetString(col : Integer) : String
Get...(col : Integer) : ...

Figure 5. Main-memory database design

6   Performance

The database version of our mapping layer framework can be deployed with the XPM in two basic configurations: local RDBMS and remote RDBMS. The main-memory database version can be deployed in three configurations: MMDB and XPM in the same process; XPM and MMDB in different processes but running on the same host; and remote MMDB.

In addition, the MMDB's writer daemon and RDBMS mirror can be deployed on the same host as the MMDB or on different hosts. We briefly compare the performance of our XPM application server with the RDBMS and MMDB in some of these configurations. A range of XPM data processing transactions are used, including simple persistent object save and load, resource look-up (one row selected to multiple persistent objects), resource adjustment (multiple persistent objects to a single row update), and accumulator updates (combinations of several data lookups, computations and one row updates).

Here we compare the performance of the XPM with a local Interbase 5.5 RDBMS optimised for a single client and given a large memory buffer, to our optimised MMDB code running as part of the XPM process (local Delphi calls only) and running on the same host as the XPM but different process, communicating via a CORBA interface. These are the most likely initial configurations of our application server and databases. Queries and updates were done over tables with around 5,000 rows

and 1,000-10,000 transactions in each category were run. Table 1 summarises the kinds of transactions performed and the average transactions per second (tps) achieved over several runs. The first three transactions involve no XPM processing, just mapping layer, RDBMS and MMDB processing. The fourth and fifth are one DB row loaded to and from multiple persistent objects, involving limited XPM processing and substantial mapping framework processing. The last transaction is XPM processing-intensive, as well as requiring several DB look-ups and an update.

Profiling of the Delphi code implementing the XPM, mapping framework and the MMDB shows most of the time spent in transactions involving the RDBMS is in RDBMS API calls (around 85-90%). For the MMDB integrated into the XPM application server process, a major portion of the time is spent in the mapping framework translating between XPM persistent objects and MMDB objects (around 50-75% of the time), even when logging object updates to secondary storage. This also accounts for the low tps of the 3rd transaction with the MMDB – almost all of the time is spent copying MMDB objects into persistent XPM objects. With the CORBA-accessed MMDB, the communication overhead consumes much of the time. We have run the XPM with the RDBMS and CORBA-accessed MMDB on different hosts with a 100 megabit LAN connection, but performance greatly degrades due to networking overhead (the TPS measures for both fall by around 60-70%).

Transactions Performed	RDBMS on same host as XPM	XPM and MMDB in same Process	CORBA-accessed MMDB on same host as XPM
Load 1 object using OID	168 tps	16,393 tps	3,846 tps
Save 1 object	181 tps	6,250 tps	2,125 tps
1 Select returning 30 objects	84 tps	588 tps	467 tps
Load resource (1 row/MMDB object to 10 XPM persistent objects)	145 tps	2,053 tps	1,401 tps
Save 2 resources (20 XPM persistent objects to 2 rows/MMDB objects)	117 tps	1,470 tps	938 tps
Update 1 accumulator (3 XPM object loads and 1 save + XPM processing)	18 tps	264 tps	138 tps

Table 1. Performance measures

We also experimented with a shared memory link between the XPM and MMDB running on the same host in different processes. This used Win32 memory files and mutexes, and gives between 65-80% TPS of the XPM/MMDB in same process.

7   Discussion

The key advantage from the use of an MMDB is the transaction through-put performance boost the XPM application obtains. Querying speed-ups for query-intensive data range from 20-100 times speed-up. The more complex the criteria being used, the better the performance increase as the main-memory held indexes provide better traversal performance than the DBMS ones, even when the DBMS is completely buffered in main-memory. Update speed improvement is considerably lower, as the MMDB must log updates to physical storage in case of a crash, but these can be written much more quickly than database communication and updates performed. We originally planned to dispense with a RDBMS and use indexed files to mirror the MMDB, but found no great advantage to doing this. The RDBMS mirror used can be a simple database as no great performance demands are placed on it. The MMDB can even be switched off while the XPM is in use and switch to the RDBMS-based persistency broker, which we found useful for testing and performance monitoring. There were no changes made to the application server code in the XPM in order to incorporate our MMDB-based persistency mechanism. This is in contrast to a cache added to the XPM to buffer frequently-used persistent objects. It proved to be very limited as it could only support very simple queries over persistent objects, others requiring RDBMS SELECT calls, and writes were still sent directly to the RDBMS. An additional advantage of the MMDB approaches is that we are also able to incrementally enhance the main-memory database and make various performance optimisations to this without impacting at all on the XPM code.

We encountered some deficiencies in our approach and in our MMDB that are important to consider before adopting the solution to performance problems outlined above. A single process MMDB is limited to 2 GB maximum size due to 32-bit addressing limitations on most common hardware platforms. This can be overcome by using 64-bit memory addresses or by segmenting the MMDB into multiple processes. The first requires expensive hardware not (yet) commonly available and the second a more complex MMDB architecture and performance loss due to inter-process communication (though this can be partially overcome by simple mapping layer caching and/or use of shared memory-based communication, if all processes run on the same host machine). The MMDB transaction support and concurrency control are simplistic due to our XPM application server being its single client. Proper multi-threaded, concurrent client support is necessary, along with transaction isolation if it were to be used by more complex application servers. Definition of indexing, meta-data support and database maintenance could all be enhanced. Inter-process communication between the application server and MMDB could utilise shared memory or sockets rather than CORBA to further enhance performance. The mapping layer framework we used has some deficiencies that should be rectified. For example, proper key support for persistent objects, complex query specification and explicit NULL-value support were areas we found our design lacking when enhancing the mapping layer to support our MMDB. Currently any external systems data required by our application server must be accessed and updated via data import or distributed transaction mechanisms, negating any local performance gains.

A variety of extensions to our MMDB and mapping layer are in progress. These include supporting large, segmented datasets in the MMDB, multi-threaded client transaction isolation and concurrency control, and better crash-recovery and start-up speed of the MMDB. These changes again have no affect on the application server code. Conveniently our application server's Enterprise System Logic™-based object model and processing is very conducive to splitting the MMDB data across separate, multi-threaded, potentially multi-hosted processes. This means multiple, concurrent XPM components handling different functionality, such as

resource, transaction and accumulator processing, need only interact most of the time with one MMDB process. Our mapping layer may even cache some MMDB data locally in the XPM process client for further performance improvement. Our mapping layer is having more complex querying support and extensions to meta-data definitions added, along with various performance optimisations to reduce the overhead of creating persistent objects and reading their property values. We are also using our MMDB to facilitate data-based enterprise systems integration, by replicating external system data in the MMDB for use by the XPM and exchanging data update transactions with external systems (Blackham et al 2001).

8   Summary

Enhancing enterprise system application server performance can be a challenging task. By careful design and use of persistent object mapping layer abstractions the application server's data management can be isolated from its processing functions, allowing various back-end optimisations to be carried out. We greatly enhanced the performance of our application server using a main-memory database by storing all data and indexes in main-memory and writing optimised transaction logs. No code changes to the XPM were necessary to achieve great performance gains. A simple main-memory database is not difficult to build, but more complex database functionality such as serialised transaction isolation, concurrent multiple-client access and large data sets may mean use of a 3rd party commercial MMDB is an appropriate choice. We have had good experiences in using a persistent object framework to isolate application server and data management interaction. Careful design of the mapping layer should be undertaken at the outset, however, particular attention to meta-data specification and management, NULL value representation and complex query formulation.

9   Acknowledgments

10   References

1. ALEKSY M, SCHADER M, TAPPER C. (1999): Interoperability and interchangeability of middleware components in a three-tier CORBA-environment-state of the art. *Proc. Third International Enterprise Distributed Object Computing,* 204-213, IEEE CS Press.

2. BASS, L., CLEMENTS, P. AND KAZMAN, R. (1998): *Software Architecture in Practice*, Addison-Wesley.

3. BLACKHAM, J., GRUNDEMAN, P. GRUNDY, J., HOSKING, J., MUGRIDGE, W. (2001): Supporting Pervasive Business via Virtual Database Aggregation, *Proc. Evolve'2001: Pervasive Business,* Sydney, Australia, May 7-8 2001, DSTC.

4. ENGELHARDT, A. AND WARGITSCH, C. (1998): Scaling workflow applications with component and Internet technology: organizational and architectural concepts. *Proc. Thirty-First Hawaii International Conference on System Sciences*, Big Island of Hawaii, USA, January 6 - 9, IEEE CS Press.

5. EXCELON CORP. (2001): *ObjectStore white Paper*, www.excelon.com.

6. GEMSTONE SYSTEMS INC. (2000): *GemStone/J Application Server Persistent Caching*, www.gemstone.com.

7. GORTON, I., BREBNER, P., RAN, S., CHEN, S., LIU, A., PALMER, D. (2000): *Enterprise Middleware Evaluation Reports*, CMIS, CSIRO, See: http://www.cmis.csiro.au/adsat/publications.htm.

8. KIM, G.B. BAE, H.-Y.. (2000): Design and implementation of a query processor for real-time main memory database systems. *Journal of Korean Information Sciences Society* **6** (2): 113-119.

9. LIU, A., GORTON, I. (2000): Evaluating Enterprise Java Bean Technology, *Proc. Software – Methods and Tools*, Wollongong, Australia, IEEE CS Press.

10. LU, H., NG, Y. Y., TIAN, Z. (1999): T-tree or B-tree: main memory database index structure revisited. *Proc. 11th Australasian Database Conference*. 65-73. IEEE CS Press.

11. SECANT CORP, (2000): *Secant Extreme Persistent Object Service White Paper*, www.secant.com.

12. SESSIONS, R. (1998): *COM and DCOM: Microsoft's vision for distributed objects*, John Wiley & Sons.

13. TIMESTEN INC, (2000): *TimesTen Main Memory Database White Paper*, www.timesten.com.

14. VOGAL, A. (1998): CORBA and Enterprise Java Beans-based Electronic Commerce, *International Workshop on Component-based Electronic Commerce*, UC Berkeley, 25th July, Fisher Center for Management & Information Technology.

15. WU, E.. (1999): A CORBA-based architecture for integrating distributed and heterogeneous databases. *Proc. Fifth IEEE International Conference on Engineering of Complex Computer Systems*, 143-152. IEEE CS Press.