Embedded Databases

The unglamorous database option that works

By Anton Okmianski

Anton is a senior software engineer and a technical leader at Cisco Systems. He can be contacted at aokmians@ureach.com.

Broadband device provisioning involves storing device-configuration information, generating configuration files, and then delivering the files to devices on demand via the Internet. In a large deployment, a single-system installation might need to handle 5 million or more devices, with each device requesting a configuration every time it reboots. Bursts resulting from area electricity outages introduce additional performance requirements.

To meet the high-performance requirements associated with provisioning broadband devices, we at Cisco Systems (http://www.cisco.com/) designed our Broadband Provisioning Registrar (BPR) to have a distributed architecture. At the heart of the system was a central database to store device-configuration data. However, we were wary of using standard relational databases, which are bloated with features we didn't need, and performance and management overhead we didn't want. In the end, we settled on Berkeley DB, a lightweight embedded database from Sleepycat Software (http://www.sleepycat.com/).

When considering Berkeley DB, our first challenge was to design a database layer that mapped our relatively complex data schema into the Berkeley DB's simple key/data pairs. The next challenge was to deliver performance of at least 150 provisioning transactions per second on relatively modest hardware—2x750-MHz Sun Fire 280R with 10,000 RPM disks in a RAID5 setup. In this article, I'll explain how we tackled these challenges.

Berkeley DB

Berkeley DB is an open-source embedded database library that provides scalable, transaction-protected data management services to applications. It provides a variety of storage/access methods including dynamic hash tables, B+trees, persistent queues, and numbered records. In short, it is a toolkit for writing customized databases.

Berkeley DB isn't a typical open-source software project. While its source code is freely available and it is free for redistribution with other open-source projects, Sleepycat controls the development of Berkeley DB and provides commercial support for it. Since our project was not open source, we paid for a license and support. That said, the key advantage for us was the royalty-free redistribution scheme that let us deploy the database on as many distributed components of our system as necessary.

Unlike standard databases that function as standalone servers, embedded databases such as Berkeley DB are software libraries that developers can embed into their applications. The database then functions in the application's process. The application itself can be a server and can use the embedded database library to implement custom database logic.

Berkeley DB has a performance advantage over general-purpose databases because it does not have interprocess communication (IPC) overhead with application servers. Nor does Berkeley DB provide a generic complex query language like SQL. Instead, a developer can customize the database for specific access patterns.

In our application, the provisioning API was the only way to communicate with our central server. Therefore, we knew in advance the different query types that the database needed to support. This let us optimize data storage and database layer logic for these specific queries.

Technical Challenge

Again, a challenge we faced involved determining how to map a complex and potentially dynamically adjustable data schema into Berkeley DB's key/data pairs. Our logical data schema consisted of over 30 different entity types with numerous relationships between them. Since we realized that our schema would evolve rapidly during development and over subsequent releases, we needed a design that would ensure significant schema flexibility. To make our solution customizable by end users, we wanted to let them store custom attributes of potentially complex types in our data objects. On top of this, we had to be able to handle at least 150 provisioning transactions per second and scale to 15 million records/objects in a single database with an average object size of approximately 1 KB. A typical transaction would involve read and write database access in order to create a device record and associate it with a given service level. Since our database size exceeded available memory on target hardware, we had to be able to support efficient read and write access to disk.

The Solution: Physical Layout

Our solution was to implement a design that was a hybrid of object-oriented and network-model database-design approaches. Since our central server was written in Java, we used the Berkeley DB Java API. Our database layer exposed data as Java objects with collections of named attributes and relationships (Figure 1). Relationships provided a mechanism to lookup related objects, while indexed attributes provided an ability to lookup objects by their attribute values.
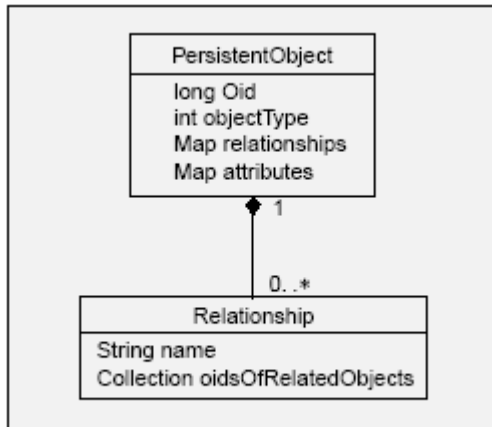
Figure 1: Database layer.

Internally, objects were identified through unique object IDs (OIDs), much like in OODBMS implementations. This isolated an object's identity from its attributes and resulted in simpler and more efficient update logic when attribute values changed.

In our design, we used only Berkeley DB's B+tree storage/access methods. One such B+tree index, called "MainIndex," contained a collection of all objects in serialized form keyed by OID. More precisely, we used OIDs as the Berkeley DB keys and serialized BLOBs as the data items. We used our optimized Java serialization to convert between run-time objects and BLOBs.

Additional B+tree indexes were used for maintaining indexed attributes. Such indexes provided a mapping from attribute values to object OIDs, essentially serving as secondary indexes for object lookup. Berkeley DB supports any key type as long as you provide a custom key comparison function. For example, in our BPR product, we were able to support foreign language strings for indexed attributes by providing a custom-comparison function that compares Unicode strings. Although Berkeley DB treats keys and data as byte strings, this is not a limitation. In fact, it forces the application to take control over serialization of data, which can be good for versioning and optimizations.

The combination of the MainIndex and attribute indexes permitted object lookup by either OID or indexed attribute value. Once the object data was retrieved and the object was deserialized, its attributes and relationships were exposed to the application. Our attributes were simply collections of name-value pairs, where values could be any serializable object. This provided flexibility as object attribute structures could change easily. In fact, the database layer explicitly knew only about attributes, which were designated as indexed. When indexed attribute values changed, our database layer code updated the attribute indexes automatically. In Berkeley DB 4.0, Sleepycat provided a secondary index feature, which could also be used for lookup of records based on secondary keys.

Finally, every object had a set of relationships to other objects. Each relationship had a name—*Owner* to *Modem*, for example—and contained a set of OIDs of related objects. In this example, given an *Owner* object, the database layer could determine a set of OIDs of related *Modem* objects, then retrieve each *Modem* object by OID from MainIndex.

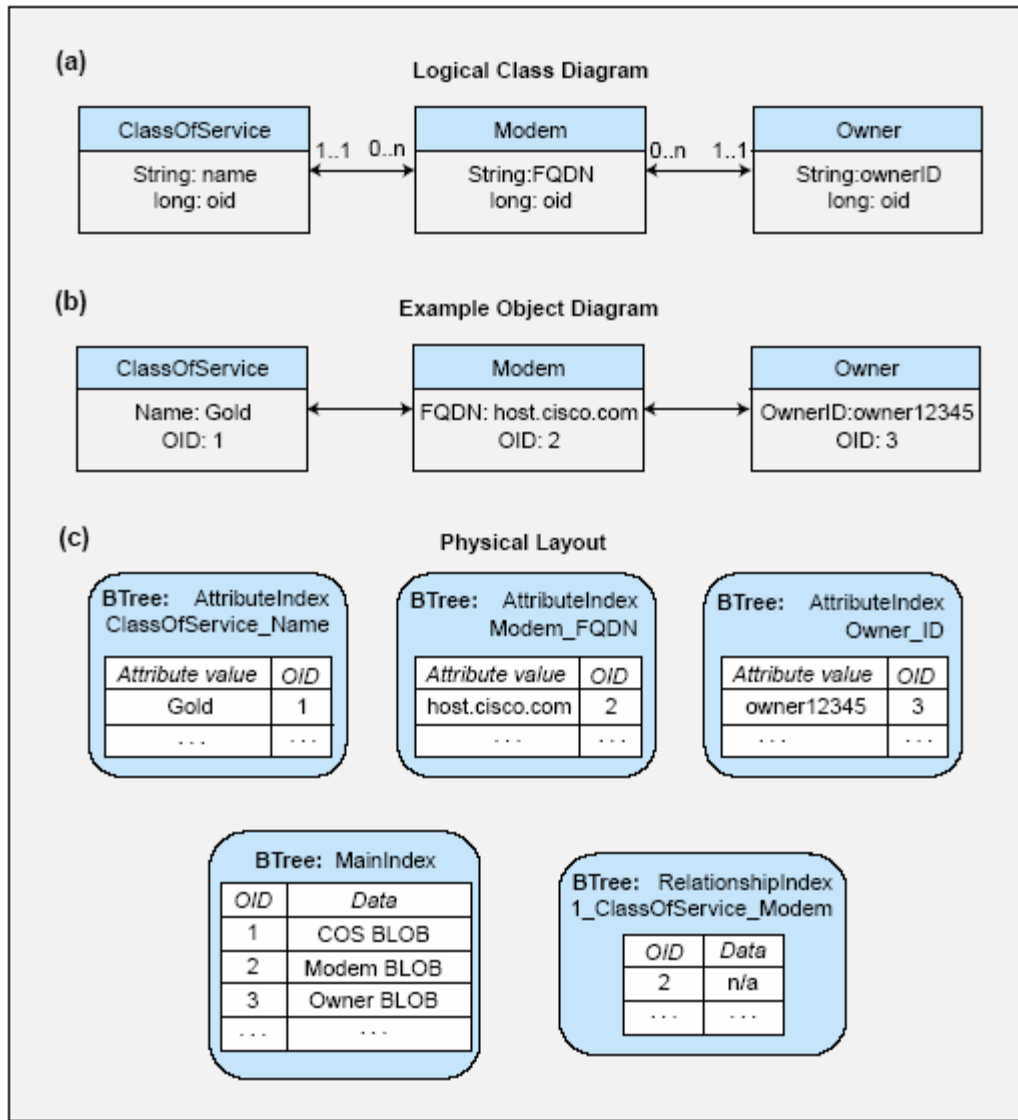Small cardinality relationships were serialized directly with the *PersistentObject*, while large cardinality relationships were represented as separate Berkeley DB B+tree indexes for scalability. In the latter case, the relationship simply defined the name of the index that contained the list of OIDs for the relationship. We also used some large cardinality relationship indexes to store additional data about related objects, such

as foreign keys. This let us make common queries without having to deserialize all objects on the other side of the relationship. For example, a *ClassOfService* to *Modem* relationship could contain five *mln Modem* OIDs. In order to scale to that size, we used B+tree indexes to store this list of OIDs for this relationship. Additionally, since we often needed to make a query for a list of *Modem* MAC addresses belonging to a given *ClassOfService*, we stored *Modem* MAC addresses along with *Modem* OIDs in the relationship index. This enabled fast sequential access to a list of MAC addresses of *Modem*s related to a given *ClassOfService*.

In Figure 2, an example of how a graph of three related objects gets persisted in B+tree indexes is shown. Figure 2(a) shows an entity relationship diagram between three sample objects. The relationship from *ClassOfService* to

Modem is high cardinality, meaning that many *Modem*s can be related to the same *ClassOfService*. The remaining relationships are of low cardinality. In Figure 2(b), we show one instance of each class. The *name*, *FQDN,* and *OwnerID* parameters are stored in *Map attributes* of each object as depicted in Figure 1.

Figure 2(c) shows how the objects are stored in B+tree indexes in Berkeley DB. All three objects get placed into *MainIndex*, where the key is the OID and the data is the serialized BLOB of each respective object. Each object has an entry in a corresponding attribute index for its indexed attributes—*name, FQDN,* and *OwnerID,* respectively. Finally, because the *ClassOfService* to *Modem* relationship is high cardinality, there is also a separate relationship index that stores the list of OIDs for *Modem*s related to this *ClassOfService* instance.

(a) Logical Class Diagram

ClassOfService		Modem		Owner
String: name long: oid	1..1 0..n	String:FQDN long: oid	0..n 1..1	String:ownerID long: oid

(b) Example Object Diagram

ClassOfService	Modem	Owner
Name: Gold OID: 1	FQDN: host.cisco.com OID: 2	OwnerID:owner12345 OID: 3

(c) Physical Layout

BTree: AttributeIndex ClassOfService_Name

Attribute value	OID
Gold	1
. . .	. . .

BTree: AttributeIndex Modem_FQDN

Attribute value	OID
host.cisco.com	2
. . .	. . .

BTree: AttributeIndex Owner_ID

Attribute value	OID
owner12345	3
. . .	. . .

BTree: MainIndex

OID	Data
1	COS BLOB
2	Modem BLOB
3	Owner BLOB
. . .	. . .

BTree: RelationshipIndex 1_ClassOfService_Modem

OID	Data
2	n/a
. . .	. . .

Figure 2: Physical layout.

Lock Management

One issue with Berkeley DB that we had to work around was the lack of record-level locking. Berkeley DB does locking at a page-level granularity. To gain better concurrency, we implemented a record/object-level locking system on top of Berkeley DB. To do this, we disabled Berkeley DB's locking subsystem and single threaded all accesses to Berkeley DB.

Updates presented a challenge. Because we ran with Berkeley DB locking disabled, we had to guarantee that there was only one Sleepycat update transaction at any given time. At the same time, we did not want to limit users of our server to only one concurrent update transaction. Our solution was to allow concurrent user-level update transactions, but batch all updates until commit, at which point we wrote all changes to the database in a single-threaded fashion. User-level transactions were implemented as a thin layer on top of

Sleepycat transactions. We recorded all updates done within each user-level transaction in an in-memory change-set. During the commit of a user-level transaction, we started a new Sleepycat transaction and applied all changes by performing Berkeley DB writes. We synchronized our code in such a way that only one user-level transaction could commit at a time. This ensured that we never had more than one Berkeley DB transaction updating the database. In our implementation, we maintained one change-set for each B+tree index, and used the change-set to track changes for each user-level transaction in progress. All database access to any index had to go through the change-set to make sure that transactional changes were reflected correctly in the results of B+tree index access. For example, if a user-level transaction deleted a given object but did not yet commit, subsequent attempts to lookup the same object in the database in the same transaction would return nothing. This was assured because the change-set would have had a record indicating that this object was deleted by this in-progress transaction.

Single-threaded access may have translated into somewhat slower performance, but we did not see any significant performance degradation in our application. This could be attributed to the fact that each atomic Berkeley DB operation was relatively fast. We used only B+tree *get*, *set*, and *get-next* operations, and there were no complex SQL queries. We were still able to meet and exceed our performance requirement of 150 transactions per second.

Managing the locking policy ourselves also gave us some benefits. While we usually used a record/object-level locking granularity, in some hotspots, even that was too coarse a grain. Since we controlled the locking granularity, we were able to use finer grained locks, something we might not have been able to do in a conventional object-oriented system. For example, with some objects, it was acceptable if one transaction were to change the attributes, while another changed the relationships of the object. In another example, for some large indexes, we allowed concurrent traversal and modification of a given relationship by many transactions without locking either the entire relationship or the object. In these cases, transaction isolation was sufficient. One such scenario was a *ClassOfService* to *Modem* relationship. We wanted many concurrent transactions to be able to associate *Modem*s to *ClassOfService*. This meant that the relationship locking needed to be isolated from the object locking. Custom lock management let us improve performance and gave us the flexibility to address concurrency hotspots as our schema and access pattern evolved.

Addressing Performance Bottlenecks

The main bottlenecks we had to address in our application were disk reads, disk writes, and data transformation between storage and runtime format (serialization).

Berkeley DB has an in-memory cache, where it keeps the most frequently used pages. However, with a purely random access pattern and a large database, the cache can only do so much. Consequently, disk access time (primarily seek time) can be a performance-limiting factor.

Our design took advantage of locality of reference (see *The Logical Design of Operating Systems*, by A. Shaw, Prentice

Hall, 1974) to improve cache efficiency. Essentially, some objects that were often used together were also created around the same time. For example, a single user operation could trigger creation of a group of related objects, such as *Modem* and *Owner*. With Berkeley DB's B+tree implementation, there is a high probability that sequentially created records with sequential keys will be stored on the same B+tree node and, therefore, on the same disk block. Our objects were stored in MainIndex, where we used OIDs as keys. Our OIDs were simply sequential numbers that were unique across multiple invocations of a Java Virtual Machine (JVM). Thus, OIDs for two objects, created at around the same time, approximated in value. This meant that there was a good likelihood that these two objects would get placed on the same disk page. If these objects were then used together, the page would have to read from disk only once.

Another way to improve read performance was to improve cache efficiency. Cache efficiency depends on the compactness of data and B+tree page utilization. Given a certain set of keys, the utilization would be best if they're added in a sorted order. This is a function of how B+tree pages get split as the tree grows.

The largest index in our implementation was MainIndex because it stored serialized object data. Since the keys in this index were OIDs and our OIDs were sequential, all we had to do to achieve optimal utilization was guarantee that OIDs were issued in the same order in which objects were committed to the database. To achieve this, each new object was assigned a temporary OID first. The temporary OID was then replaced with a permanent one during commit.

Disk-write performance was another bottleneck we had to address. Disk-writes are often a bottleneck in transactional systems because, at transaction commit, the transaction log must be flushed to disk to guarantee durability. On the other hand, buffering writes let the disk schedule them in a manner most efficient for the movement of the head on disk. The challenge was to be able to buffer writes belonging to more than one transaction, while still maintaining transaction durability guarantees, which called for data to be stored in persistent storage before the commit operation returned.

Berkeley DB has a useful feature that helps buffer transactional writes by supporting asynchronous flushing of transactions. You can commit the transaction without immediately flushing the data to the log by using the TXN_NOSYNC flag. You can then flush all the buffers separately with a *log_flush()* function. This feature let us schedule multiple transaction commits per single disk flush. A transaction class wrapper ensured that we did not return from commit until all the buffers were flushed, which guaranteed that the data made it to disk before we considered the transaction complete. This feature significantly improved performance of our update transactions.

Berkeley DB 4.0 introduced the "group commit" feature, which provided the same functionality without any special actions on the part of the application.

The final bottleneck we had to address involved serialization overhead. Data serialization is the process of transforming in-memory object representation to binary form. In our case, we needed to serialize a Java *PersistentObject* ([Figure 1](#)) with all of

its relationships and attributes, which could have been potentially complex types.

Java has built-in functionality for object serialization, but it produces noncompact binaries and lacks performance because it is a general-purpose implementation and stores a large class descriptor for each class. Because we serialized each *PersistentObject* separately, this design would have been detrimental (it would have stored a class descriptor for every class that was used in serialization in each serialization stream and, thus, for every *PersistentObject*).

We addressed this by implementing custom serialization with compact predefined object class IDs for all well-known classes. This created a more compact storage footprint. Smaller footprint, in turn, resulted in better serialization performance, faster writes/reads to/from disk, and better cache efficiency because more data could be kept in cache.

Preventing Database Corruption

In mission-critical systems, database corruption must be avoided at all costs. To this end, we evaluated the possibility of database corruption at several levels. Berkeley DB has some built-in anticorruption features, such as checksums for log records. However, the biggest risk of corruption with an in-process database stems from the possibility that your application can accidentally corrupt the memory of the embedded database. These corrupted pages can then be written to disk.

Fortunately, the possibility of such errors was virtually eliminated because we wrote our application in Java and used a Sleepycat-provided JNI bridge to the Berkeley DB library. Since the JVM heap is separate from the native heap made available to Berkeley DB by the JVM, the Java application should not be able to corrupt Berkeley DB by accidentally stepping on its memory.

We also tried to prevent logical data inconsistency (another form of database corruption) on a higher level. We implemented automatic relationship cardinality verification at commit time to guarantee that the logical layer schema constraints will not be violated.

Conclusion

Embedded databases with seemingly small feature sets can be powerful tools. Berkeley DB proved to be a great choice for our needs and may serve other projects well. With merely 15,000 lines of Java code, we built a flexible database layer and achieved performance of several hundred read/write transactions per second.