Berkeley DB Java Edition Architecture

An Oracle White Paper
September 2006

Berkeley DB Java Edition Architecture

EXECUTIVE OVERVIEW

Berkeley DB Java Edition is a 100% native Java, high-performance transactional database. It was designed to be functionally compatible with the ubiquitous, open source Berkeley DB and architecturally compatible with Java applications.

This paper presents the design and implementation of Berkeley DB Java Edition (JE), focusing on how it provides fast, reliable storage services when embedded in applications that run in a JVM. JE uses a log-structured disk representation for its database objects, avoiding unnatural marshalling of objects onto pages. The combination of object-level caching and log-structured storage permits fine-grain latching and record-level locking for highly concurrent applications. The log-structured design provides unsurpassed write performance and competitive read performance for applications whose data fits in memory.

1. INTRODUCTION

Berkeley DB Java Edition (JE) finds its functional roots in the Berkeley DB (DB) embedded database library. DB was designed to provide persistent, application-specific data storage in a fast, scalable, and easily administered package. It provides the traditional atomicity, consistency, isolation, and durability (ACID) guarantees of a relational database management system (RDBMS), without the need for separate installation of a database server or the services of a database administrator. In addition, Berkeley DB executes directly in the application's address space, allowing for single binary installation. This is an attractive proposition for manufacturers of embedded devices, such as set top boxes or mobile phones, as well as for developers of large heavily concurrent application servers, such as messaging and directory servers.

Java Edition opens up many additional possibilities. First, it brings the functionality of DB to environments that require 100% Java. Second, it creates the possibilities that storage can be provided directly in application servers, which may not want to rely on an external DBMS. Application servers frequently need robust data storage, where relying on the existence of a RDBMS is burdensome. Additionally, applications written for application server environments may want the benefit of a small-footprint, embedded data manager that does not incur the overhead of a JDBC interface and associated process switch. Even for those applications where an RDBMS using JDBC is appropriate, we anticipate JE will be used, as Berkeley DB is often used, as a fast front-end cache for the RDBMS.

Whether used in conjunction with an RDBMS or used directly, JE supports a variety of Java standards. The Java Transaction API (JTA) specifies an interface between Transaction Managers, Java Applications, and Resource Managers. JE implements the `XAResource` interface that is required for it to participate in a distributed XA transaction as a Resource Manager, using two-phase commit to satisfy the distributed transaction and recovery requirements.

The J2EE Connector Architecture (JCA) specifies a standard set of interfaces that can be used by a J2EE application server to communicate with backend data management systems. JE provides a Resource Adapter library to allow deployment as a backend transaction-processing module.

Finally, JE can be integrated into a Java Management Extension (JMX) environment. JMX allows a JE-based application to be managed and monitored via a variety of interfaces (e.g., SNMP). For instance, using JMX, a system administrator can view runtime performance statistics using a network management console.

In Section 2, we present the overall architecture of JE, introduce some Berkeley DB and JE specific terminology, and lay the groundwork for the in-depth design discussion that follows. Section 3 presents the implementation in detail, focusing on the aspects of JE's design that make it particularly well suited for the JVM environment. Section 4 compares JE to other Java implementations and more conventional data management techniques. Section 5 discusses how our customers use JE. Section 6 presents some performance results, and we conclude in Section 7.

2. ARCHITECTURAL OVERVIEW

As JE is functionally compatible with Berkeley DB, we have retained the key concepts and definitions from the initial Berkeley DB product. We begin by introducing these concepts and the terms Berkeley DB uses for them.

2.1 Terminology

A *transaction* is a group of operations that adhere to the ACID properties of Atomicity, Consistency, Isolation, and Durability [14]. Atomicity implies that a collection of operations appears atomically in the database; either all the operations appear or none of them do. Consistency means the database always presents a consistent view. (For example, consider a database of cats and owners. If Joy owns Elsa, we would never see a database with owner Joy without cat Elsa or cat Elsa without owner Joy.) Isolation describes the property that any transaction operates under the illusion that it is the only transaction operating in the database at a particular time. That is, there is no way to tell if there are multiple transactions active concurrently. Finally, durability implies that modifications made by a transaction are guaranteed to be persistent, even in the presence of system or application failure.

There are three primary interfaces to JE. The first uses Berkeley DB JE to implement a Java collection. The application uses standard collection methods to create, retrieve, and modify transactionally persistent data. Each Java collection object is associated with a database, described below.

The second interface is a new Direct Persistence Layer, introduced in JE 3.0. The Direct Persistence Layer is a type safe and convenient API for accessing persistent objects. A developer specifies primary and (optionally) secondary keys using Java annotations. The library transparently and seamlessly handles schema evolution as class definitions change.

The final interface uses the DB data abstraction of *key/data pairs*. Keys are byte arrays wrapped in a `DatabaseEntry` object. In JE, keys are true *primary indexes*, which means that `DatabaseEntry` objects are arranged in key-order. By default, JE sorts keys lexicographically, however, an application can optionally support their own sort and comparison functions. Data objects are similarly opaque structures from the point of view of the storage system. Applications may treat key/data pairs as objects, using a compact form of Java serialization, or as tuples of primitive values bound to objects. Applications may also create their own object-data bindings or work directly with byte arrays.

A *database* is collection of key/data pairs sharing sort and comparison. You can create multiple databases that use the same sort and comparison functions, of course, but within one database, there may be only one sort function and one comparison function. Through the JE API, applications create handles to referenced databases, and all accesses to the database are performed as methods off the database handle (or from objects created off the database handle; see *cursors* below). In conventional relational database parlance, a Berkeley DB database corresponds to a table. Databases are represented by a Java Database class; Database handles persist across transactions and are typically opened once when the application starts up and are left open for the lifetime of the application.

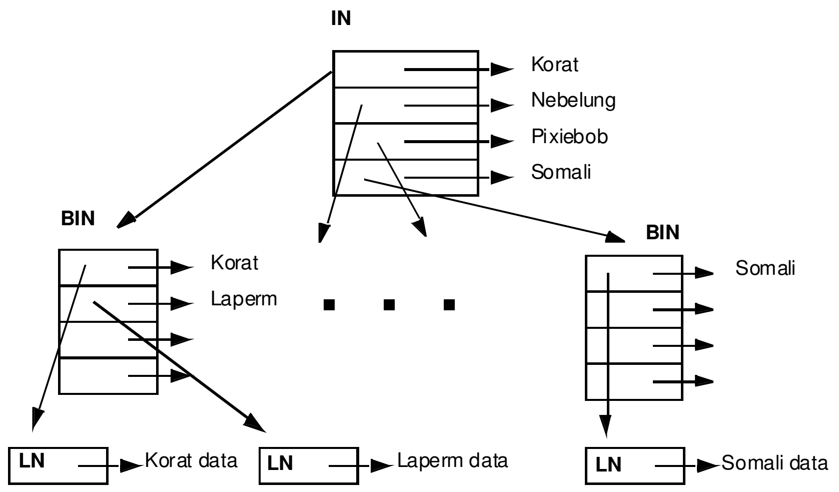
Databases support access to individual key/data pairs through keyed lookup. For iteration, the Database class provides an `openCursor` method, which returns a *Cursor* handle. Logically, cursors maintain a position in the Database. Cursors are local to a particular transaction. That is, cursors are created in the context of a particular transaction, and the cursor must be closed before the transaction commits or aborts. Cursors allow iteration over an entire database (either forwards or backwards) as well as traversal through sets of duplicates and within ranges of the items in the database.

A collection of databases comprises a *database environment*. In RDBMS parlance, an environment is typically called a database. For example, when implementing a personnel management system, the application might reference a collection of tables (databases in JE) for Employees, Managers, and Departments, and these would all be grouped together into a JE database environment. Typically, database environments correspond to directories or directory hierarchies in a file system.

The log files that support the database and the properties files that describe the environment's configuration live in the database environment's directory.

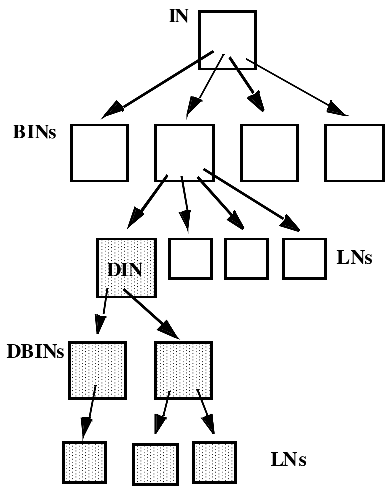
2.2 Mapping Databases and Environments to Data Structures

Databases in JE are implemented as B^{+}trees [8], where each leaf in the tree represents an individual key/data pair and is implemented as a Java object called a leaf node (LN). The data portion of the pair is stored as a reference in the LN while the key is logically part of the LN, but physically referenced by the node that points to the LN. These nodes that reference other nodes are called internal nodes (IN) and are implemented as arrays of key/leaf pairs. The Figure titled "Database structure" shows this structure. The lowest internal nodes are designated as bottom INs (BINs) rather than INs. BINs are a subclass of IN providing additional support for cursors. All the nodes within an Environment's tree structure have a unique node ID.



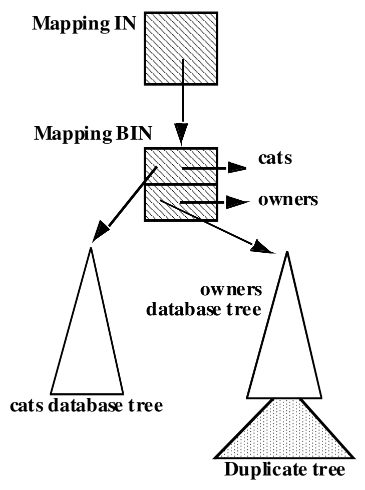
Database structure. Databases are implemented as B+trees with fixed size internal nodes (INs). BINs are subclasses of INs that support iteration over the elements in the tree.

If a key value has duplicate data values, then the BIN references a duplicate internal node (DIN), which is the root of a tree that stores all the duplicate data items. Duplicate trees are structured like database trees, with INs replaced by DINs and BINs replaced with DBINs. Thus, a database is potentially a tree of trees. This structure is shown in the Figure titled "Database structure with duplicates".



Database structure with duplicates. Duplicates are stored in a tree similar to the database tree that is rooted at a location in the tree typically occupied by an LN.

Finally, the collection of databases in a single environment is stored in yet another tree called the mapping tree. The mapping tree is simply a B$^+$tree whose keys are the various database names and whose "data objects" are the root nodes of the corresponding database tree. The result is a hierarchy of trees, with a maximum depth of three, as shown in the Figure titled "Environment data structures". While this hierarchical structure is intuitive and prevents having multiple index structures for the various logical entities being stored, it does introduce additional complexity in recovery processing.



Environment data structures. The mapping tree has a structure similar to the database and duplicate trees, but implements the name to database mapping.

Having mastered the basic terminology and structure of JE, we are now ready to explore implementation details in more depth.

3. THE IMPLEMENTATION

The implementation revolves around the log-structured design of JE. Rather than marshalling and unmarshalling Java objects to and from disk-backed pages, JE serializes objects and writes them sequentially to a log, optionally using the Java NIO API. All the necessary index structures sit atop this sequential log, and there is no other representation of the data.

We found that the `java.io.Serializable` interface was too heavyweight for our purposes, so the use of the term "serialize" throughout this paper references the generic action of marshalling objects, although our implementation does not use the `java.io.Serializable` interface to implement this functionality.

This design offers several advantages. First, JE does not have to pay the performance penalty of marshalling and unmarshalling variable length objects onto fixed-length pages. Second, JE can perform record-level locking without the significant recovery complexity usually associated with traditional RDBMS record-locking solutions. Third, there is only a single representation of the data, instead of the separate database and log files in conventional logging systems. Fourth, all writes happen sequentially, so unless the disk is used to handle read misses, the disk head never moves, providing near maximum disk bandwidth utilization.

The log-structured design is not without its disadvantages. The primary disadvantage is that a cache-miss for JE is more costly than a cache-miss in a traditional database engine. For example, a cursor traversing the database, which misses in the cache, will be forced to issue an I/O for any items not found in the cache, whereas a traditional on-disk structure will amortize a single I/O over some number of items as a page containing several items is read into memory from the disk.

While log-structured file systems have been discussed in great length in the file system community [24, 27, 28], there has been little attention paid them in the database community. Section 4.2 discusses this topic in more depth.

The rest of this section is organized as follows: First, we discuss synchronization in JE, which combines transactional and non-transactional access. Next, we present the details of the log-structured storage system. Then we present the logging and recovery implementation. We conclude the implementation discussion with descriptions of the various maintenance threads that manage the log-structured store.

3.1 Concurrency

JE uses record-level transactional locks on the LNs that store key/data pairs, providing highly concurrent database access. Internal nodes are maintained non-transactionally in order to preserve high-concurrency access, minimize the data written to the log, and simplify abort (as modifications do not have to be undone when transactions abort). Instead, modifications to internal nodes (INs) are synchronized via short-term latches protecting only against physical manipulation. Since Java monitors, using the synchronized keyword, cannot be used to perform latch coupling (atomically acquiring a new latch and dropping a held latch) and do not prevent starvation in the case that multiple threads are waiting for the same object, JE provides its own latching mechanism using Java monitors as the basis.

Each object that requires a latch (for example, an IN) contains a reference to a JE latch object. JE uses Java monitors to synchronize access to the latch object. When a thread holds a latch that is requested by another thread, the incoming thread waits on a synchronizer object.

Latch granting is strictly first in, first out. When a thread releases a latch it examines the queue and notifies the first entry in that queue. Since the releasing thread only calls notify on a single thread, we guarantee that threads are awakened in the order they appear on the queue, reducing the possibility of starvation.

Latch deadlocks must be avoided, because we perform no latch deadlock detection or resolution. Acquiring locks in a well-defined order is the most common deadlock-avoidance technique [30]. While descending the tree, this well-defined order is obvious: always acquire latches down the tree. For other data structures, such as the transaction table, we define a lock order and adhere to it.

Using short-term latches rather than transactional locks on internal nodes allows for improved concurrency but does introduce complexity in the implementation, especially in recovery.

JE uses a conventional lock manager to implement transactional locks. The lock table is an object that contains a hash table-mapping node IDs to lock objects.

Each lock contains a list of waiting and granted locks, so it contains all the information necessary to determine if a newly requested lock conflicts with those already granted or requested. If the calling transaction already has a lock of the correct mode for the object, we simply reuse the existing lock.

The lock manager implements a few simple policies. First, if a transaction owns a write lock on an object, then a request for a read lock on the object is always granted. Second, if a transaction owns a read lock on an object, then a requested write lock is placed added at the front of the waiters list. This helps prevent deadlocks if only one transaction jumps to the front of the wait list, but will ultimately lead to deadlocks that would have arisen anyway if more than one transaction were trying to upgrade from a read lock to a write lock. A read locked object will block incoming read lock requests when there is already a waiting write lock request, even though incoming readers do not conflict with current reader(s).

JE also supports dirty reads [14] providing weaker forms of isolation than purely ACID transactions. Dirty reading, or "read uncommitted," means that a transaction can read data that has been modified by a transaction that has not yet committed. In practice, this means that a transaction could read data that is never committed to the database, because the transaction that wrote it eventually gets aborted. Nonetheless, some applications welcome the performance improvement in exchange for this weaker form of consistency. Because dirty readers do not block other lockers and are never blocked by other locks, dirty reading is implemented via BIN latching. That is, while a node is physically being modified by another thread, the referencing BIN is latched. Therefore, no other threads can access the node while it is in the midst of a modification. We release the BIN latch after the modification is complete. This precisely provides dirty read semantics, so dirty reads are "free" in JE.

Access to the lock table could be coordinated either through a single hash-table mutex or per-hash-bucket mutexes. Berkeley DB's experience in implementing per-hash bucket locks showed they did not necessarily improve performance because the increased concurrency was offset by increased mutex activity, so JE uses a single lock table mutex. If this mutex becomes a point of contention, developers can configure JE to use multiple lock tables to reduce the contention.

Currently, JE supports configurable timeout-based deadlock detection. The Berkeley DB implementation provides a richer set of deadlock detection functionality, and we expect JE will eventually match that functionality. However, because JE uses latches and not transactional locks on internal nodes, deadlocks in JE can always be attributed to data access; internal nodes never contribute to deadlocks. Thus, we expect deadlocks to occur less often in JE than in more conventional systems.

By default, JE provides Repeatable Read isolation, since this is the natural consequence of record locking. If phantom elimination is important, an application can configure JE for serializable isolation [14]. Doing so introduces a small performance penalty, because JE must perform "next key" locking, effectively acquiring range locks during read and insert operation.

3.2 Storage System

JE implements a database and B$^+$tree abstraction via a log-structured storage manager. It stores database key/ data pairs as byte-arrays referenced from the leaves of the B$^+$tree. Log records make these elements persistent via log records that are serializations of the objects in the database. In order to make this concrete, we'll step through the insertion of a key/data pair into a database. Assume the database is open and there is sufficient room in the BIN to add a new entry. Insertion proceeds as follows:

1. Perform a tree search to locate the appropriate position for the new key/data pair.
2. The search procedure is a standard B$^+$tree traversal and is not discussed here.
3. Latch the BIN.
4. Create a new LN for the key/data pair.
5. Write the LN to the log. This write takes place in-memory and returns a log sequence number (LSN) that uniquely identifies the destination on-disk location of this element.
6. Modify the BIN to reference the new key/data pair. The BIN contains both an in-memory reference to the LN as well as the persistent LSN.
7. Mark the BIN dirty.
8. Unlatch the BIN

The tree structure is maintained via two sets of pointers: in-memory references and persistent LSNs. If the tree's nodes are in memory, JE uses the in-memory pointers. However, should the in-memory copy of the LN be evicted from memory in order to limit JE's memory consumption, the LSN is both necessary and sufficient to retrieve the LN from disk.

BINs are fixed-size arrays of entries, since they contain only references to keys and LNs. We use a binary search to locate a particular entry in the sorted BINs. The number of entries in a BIN is a configuration parameter. Smaller BINs lead to more efficient searching at the expense of a deeper tree; larger BINs lead to shallower trees, but more costly per-BIN searching. When a BIN becomes full, we must split the BIN, creating a new BIN to be inserted into the parent IN. Conventional split processing is inherently deadlock prone, because it involves making modifications above the current position in the tree. JE performs opportunistic IN splitting to avoid such deadlocks. During inserts, if the search down the tree detects a node that is full, then it opportunistically splits the

IN/BIN. Once the split has finished, the transaction continues its search down the tree (triggering additional split/grows as necessary). Even if the split is not necessary to accomplish the insert, or the transaction for which the insert is being performed aborts, the split remains in the tree and is not undone. The rationale is that even if the current operation does not trigger the split, the node is sufficiently full that a future transaction will make the split necessary. This opportunistic splitting makes the split processing in JE significantly simpler than the corresponding operations in Berkeley DB.

Every modification to an LN triggers an update to a BIN, because modification of an LN changes its log sequence number. In turn, BIN modifications trigger IN modifications, etc. In order to avoid repeatedly writing internal nodes (and slowing transactional throughput and wasting log space), INs(BINs) are simply marked dirty upon update. Simply delaying the writing of BINs is not sufficient, because we would still need to write BINs at checkpoints. Instead, we usually write *BIN Deltas*, smaller log records that identify the specific BIN entries that have changed. A background thread, described in Section 3.4, periodically scans the list of dirty internal nodes and writes either deltas or entire node into the log. If the system crashes prior to a checkpoint, the LN records in the log are sufficient to recover modifications to internal nodes.

In theory, if one were willing to replay the log from the beginning of time, one would never need to log the INs at all. However, recovery from the very beginning of the log would impose too heavy a penalty on startup, and it does not work if INs must be evicted from memory. The *checkpoint thread* serves a purpose identical to that in a conventional data management system: it writes dirtied objects to disk, trading I/O operations for improved recovery time.

An LN modification creates a new copy of the LN, which is written to the log, and updates the in-memory LSN "pointer" in the BIN. This no-overwrite storage system, characteristic of log-structured file systems, means that the previous version of a node, or its *before-image* in database parlance [14], is still accessible in the log. This leads to a simple transactional implementation: The first time a transaction modifies an LN its LSN is preserved. If the transaction aborts, restoring the original LSN of the LN effectively undoes the transaction's modifications. This same technique applies to modifications to duplicates and modifications to the mapping tree, meaning that JE easily and naturally provides transactional table creates, deletes, and renames.

Deletions are more complicated. Since BINs are not transactionally locked, deleted key values in the BINs must remain until the transaction commits. The rationale for this is twofold. First, if the key were removed, subsequent inserts to that BIN by other transactions might leave the BIN full. If the deleting transaction then aborts, we must restore the key to the BIN, but there would be no space, and we would be forced to perform a split during abort. Second, if we removed a unique key and a subsequent transaction inserted a new version of that unique key, then the abort might be impossible, because it would introduce a duplicate key (which can be disallowed). To solve both of these problems, we use a *lazy BIN compression*, leaving deleted keys in their BINs and nulling out the corresponding data entry for the key.

Leaving keys in BINs after a delete is a good short-term strategy, greatly simplifying abortion of a delete, but it is a poor long-term strategy. Each BIN entry costs a latch get/release during cursor traversal and limits the fanout of the BIN. The simplest way to resolve this problem would be to discard the deleted keys when the deleting transaction commits. Unfortunately, this would place BIN cleanup in the transaction commit path. As transaction commits are the normal case, it is desirable to optimize commit processing, and so rather than removing BIN entries upon commit, we remove deleted entries from BINs when we are about to write those BINs to the log.

3.3 Logging and Recovery

As mentioned in Section 3.2, an LN modification (e.g., create, update, or delete) produces a serialized representation of that LN in the log. The log consists of a collection of files in the file system. Each log file contains an initial record including a version number, creation timestamp, file sequence number, and the offset of the last record in the previous file to facilitate backward chaining.

Other than the header, a log contains a sequence of log records, referenced by LSN. The LSN identifies the specific log file and the offset within that log file. Each log record also has a header containing the log record type, a version, a pointer (offset) to the previous log entry, the record size, and a checksum.

Most database systems guarantee recoverability using write-ahead-logging [14], and JE uses write-ahead-logging, influenced by the log-structured design. The persistent pointer to a node is its LSN. However, an LSN does not exist until a record is written to the log. Therefore, in order for IN A to reference LN B, IN A must contain LN B's LSN. Thus, any version of A that appears in the log and references B must appear after B, because otherwise A could not know the LSN of B.

Transaction commit processing in JE is similar to that in conventional systems. To provide maximal concurrency, we first release all read locks. Then JE writes a commit record into the log and flushes the log up to and including the commit record. Then JE releases the transaction's write locks and returns to the caller.

Transaction abort is slightly more complicated. Abort processing must restore all data to its pre-transaction state, however, it need not restore the physical tree to its pre-transaction state. That is, any operations on the internal nodes of the tree can be left as they are, since they are not part of the transaction's modifications.

Each node modified by a transaction stores the pre-transaction LSN of the node. "Undoing" a transaction requires restoring these saved LSNs to the tree. Therefore, abort processing consists of the following steps:

1. Write a transaction abort record to the log.
2. Walk the list of locks for the transaction and for each write lock, retrieve the abort LSN for each node.
    a. For each node identified in 2
3. Search the tree for the BIN referencing the node and undo the operation on the BIN (i.e., remove a newly created entry; restore an original value for an update; replace a newly removed entry).
4. Release locks
5. Return to the application

The log must contain enough information to allow the system to recover from failure, in addition to supporting transaction commit and abort processing. Our recovery algorithm uses a combination of logical and physical recovery, because we use a combination of locking and latching during normal operation. This is unusual in a database engine, and certainly the most complex and challenging aspect of the JE implementation.

Recall that the JE environment is structured as a tree of trees with a potential depth of three trees (mapping tree, database tree, duplicate tree). While the elegance of this architecture provides many benefits, such as code reuse and uniformity, the penalty for this architecture occurs in recovery. Because we do logical recovery, we need to recover each level of this hierarchical tree in order, first the mapping tree, then the database tree, and then duplicate trees. For example, in order to recover a database, we need to be able to open it. In order to open a database, we need to be able to find its name in the mapping tree. Therefore, before being able to recover any database, we must fully recover the mapping tree.

Our recovery algorithm currently requires ten passes over portions of the log. (For comparison Berkeley DB requires four passes over selected portions of its log during recovery: one to find the end of the log, one to open files, one to roll backward undoing aborted transactions, and one to roll forward to redo committed transaction). In both JE and Berkeley DB, the portion of the log that must be repeatedly processed is determined by the checkpoint interval; to a first approximation, the part of the log that is repeatedly traversed is that part written since the last checkpoint.

If JE does significantly more processing during recovery than Berkeley DB, it will be more sensitive to the checkpoint interval than either Berkeley DB or other conventional database engines. The number of passes is a result of our initial focus on correctness over performance; we focused first on making it correct, leaving performance optimizations to later releases. By the time a database engine is running recovery, something bad has already happened, and the primary focus must be on not making the situation worse. We have identified a number of ways to combine some of these passes and reduce the total to a more manageable number, but this has not yet become a compelling issue. Finally, the typical association of large amounts of memory on machines where recovery may have to process large numbers of log records make this less of a concern than it might have been a decade ago.

Recovery has four phases, each of which requires one or more passes over the log.

The first phase establishes the limits on allocated node ids and transactions.

1. Read all INs to find the largest allocated node ID (so that we can begin allocating new IDs) and the largest used transaction ID (before we need to perform any transactional operations).

The next three passes recover the mapping tree, which maps database IDs to actual databases:

2. Read all the BIN Delta records for the mapping tree.
3. Undo all aborted LNs in the map (i.e., roll backward). Keep track of all committed transaction IDs.
4. Redo all committed LNs in the map (i.e., roll forward).

The next two passes reconstruct the physical structure of all the database trees:

5. Read all the INs and link them back into the tree.
6. Read all the BIN Deltas and apply those to the INs.

Next, we reconstruct duplicate data trees, similarly to how we reconstructed the database trees. The reason this must be implemented as a separate set of passes is because we cannot necessarily traverse enough of the tree to access the duplicate trees until pass numbers 5 and 6 are complete.

7. Read all the DINs and DBINs and link them back into the tree.
8. Read all the DBIN Deltas and apply those to the INs.

And finally, we execute the conventional roll back and roll forward phases:

9. Roll backward undoing aborted transactions.
10. Roll forward reapplying the committed transactions.

3.4 Checkpoint Thread

The goal of checkpointing is to bound the time necessary for recovery. Checkpoint accomplishes this by writing dirty INs to the log. Our initial design for checkpoint took the naive approach that we would write each dirty IN to the log at every checkpoint. Unfortunately, simple calculations reveal that in a steady-state random workload, we would end up writing the entire tree on every checkpoint. This was obviously unacceptable.

The solution for bounding the size of checkpoints is twofold: we use incremental logging for IN updates and we flush INs in layers corresponding to their depth in the tree. Rather than logging entire INs whenever an IN is dirty at a checkpoint, we log a delta record. A delta record contains a reference to the last intact copy of an IN and a collection of changes that, when applied to the intact IN, produce the current state of the IN. For example, consider the following list of IN modifications.

- BIN 10 logged in full version (LSN = 10)
- LNa added to BIN10
- LNb added to BIN10
- LNa deleted from BIN10

Now, assume that we begin a checkpoint (A). The log record for BIN 10 contains:

- a reference to LSN 10 (intact BIN)
- a delta for delete of LNa
- a delta for the add of LNb.

Now, assume that we add another LN (c) to BIN 10 and begin a new checkpoint. The log record for BIN 10 would be everything in the previous log record plus a delta for the addition of LNc.

When the delta list for an IN becomes sufficiently large, we simply write a new copy of the IN (the threshold at which we make this transition is a configurable parameter).

Each time we write a new copy of an IN, we also dirty another IN (its parent). Therefore, newly dirtied INs are not processed in the current checkpoint, thus bounding the size and time required for any single checkpoint.

The second key design issue for checkpointing is logging INs in tree order, from those nodes deepest in the tree towards the root. This provides both an optimization as well as correctness. Let's say that we were to allow internal nodes to be written in any order—for expository purposes, let's say that they are written in their node ID order. Consider a tree where IN-8 references IN-2 which references IN-5. During a checkpoint IN-2 is initially clean, so it is not written, but IN-5 is dirty, so it is written. The act of writing IN-5 dirties IN-2, but IN-2 will not be written during this checkpoint, because we've already passed its turn. However, imagine now that IN-8 is also dirty, so it gets written. We've now written to disk a "new" version of IN-8 that does not reference a modified IN-2. As a result, the modification to IN-2 will be lost during recovery if the system now crashed.

If we force INs to be written in tree order from the lowest/deepest levels towards the root, the scenario described above cannot happen and our tree remains recoverable. In addition, we limit the amount of data written during a checkpoint, improving performance as well.

3.5 Node Eviction

Like conventional data managers, JE must limit its memory consumption to an application-specified amount. Like Berkeley DB, JE allows the application to configure this limit. The node eviction algorithm enforces this limit.

Both BINs and INs are candidates for eviction, and both BINs and INs can be evicted regardless of their participation in active transactions. INs are not evicted if they have resident children.

We treat BINs and LNs differently, because we do not explicitly track LNs for eviction. When we select a BIN for eviction, we first check whether it has any resident LNs. If it does, we evict them and leave the BIN in memory; if it does not have resident children, then we evict the BIN. This gives preference to evicting LNs over BINS without having to track LNs explicitly.

JE maintains a list of in-memory INs and uses this list for memory eviction. However, a few nodes are excluded from eviction. We exclude all INs belonging to the mapping tree, since the mapping tree should be small, and since any open file handles will hold a reference to objects in the mapping tree. We exclude all INs that have children present in-memory. This is because children may be dirty and will need access to the IN when it is time to write them to disk. Finally, we exclude BINs that have open cursors referencing them, since we will almost certainly access them in the near future.

To select a particular IN for eviction we borrow from the virtual memory page replacement literature to approximate LRU [30]. Borrowing from the Berkeley DB memory management design, we maintain a 64-bit monotonically increasing generation count (G) for each IN, BIN and DIN in the system. This is a system-wide generation counter. On every application-initiated access to an IN, BIN, or DIN we assign the current value of the generation number to the node and increment the generation number. So, the G of a particular node indicates how recently it was accessed. If we always evicted the node with the minimum G, we would implement LRU; evicting the node with the maximum G implements MRU. In order to avoid creating a hot spot, we never latch the generation count, so we must assume that nodes in the system may have the same generation number.

We invoke eviction on every database operation when memory usage is above a (configurable) threshold. The eviction algorithm selects and evicts nodes until a (configurable) desired amount of memory has been freed. An invocation of the algorithm examines nodes in batches of N (configurable) evitable nodes and selects the node with the lowest G value for eviction.

To select a node, the evictor walks the list of INs in an arbitrary order different than both the creation order and key proximity. When it reaches the end of the list, the evictor cycles back to the beginning. When eviction completes (that is, it has freed a sufficient amount of memory), it saves the current position in the IN list and begins its next run at that point.

Because of the arbitrary order of the IN list, this algorithm is essentially a clock algorithm approximating LRU without having to sort or maintain the nodes in G-value order. The larger the batch considered during eviction (N), the closer the approximation to LRU.

3.6 Cleaning

No discussion of a log-structured system would be complete without a discussion of the cleaner. As has been demonstrated [24], selecting the appropriate segments (or JE log files) for cleaning has a significant effect on cleaner performance. JE maintains a utilization profile for each log file so the cleaner can select the eligible log file with the lowest utilization. A log file is eligible for cleaning if its removal does not interfere with recovery. Thus, any log file is eligible if it does not contain the last checkpoint and has not been written since the last checkpoint.

JE cleaning consists of reading a log file, appending records that are still "live" back into the log, and then reclaiming the space freed up by the now useless log file. A record is alive if the node it references is still present in the database. If log files are copied to archival media before being removed, the collection of archived log files provides the basis for catastrophic recovery.

There are two types of information in the utilization profile: summary and detail. Summary information for each file indicates approximately how much of the file is obsolete and how much it will cost to clean the file. It contains the total number of nodes, the number of obsolete nodes, and the average size of the nodes. The summary information is cached in memory and is used to select the next log file to be cleaned.

The detailed information for each log file consists of a list of the byte offsets in the file for all entries that are known to be obsolete. Without detailed summary information, identifying obsolete entries requires traversing the live database tree to determine if the entry in the log file exists in the tree. The detail information is used to avoid this potentially costly check during cleaning.

The utilization profile is stored as an internal hidden database. Each record in that database contains both the summary and some of the detail information for a particular log file. A log file can have multiple profile records with each one containing the complete summary information and the detail information accumulated since the last record for the log file was written. By storing the detail information incrementally and in a packed form, the utilization profile information is less than 2% of the total disk space used in the environment.

JE tracks utilization profile information during live database operations. In most cases, this is accomplished without additional latching by tracking utilization while a latch is already held. The challenge in keeping utilization information accurate is keeping it transactionally consistent, both during regular operation and during recovery.

While the detail information makes cleaning obsolete entries in a log file efficient, we must still migrate the active entries to the end of the log. This requires updating the live tree so parent nodes reflect the new location (LSN) of their migrated children. The parents must be flushed to disk before the cleaned log file is deleted. In many applications, especially those whose data sets do not fit in JE's memory cache, the migration of active nodes and the accompanying BIN update comprise the bulk of the cleaning overhead.

JE needs to accomplish two goals with respect to cleaning. The first goal is to keep the cost of node migration as low as possible. The second goal is to distribute the work of migration among the threads of an application such that the cleaner keeps up with the application's activity. This second goal is particularly challenging for applications whose working sets exceed the size of the JE cache. We use *lazy migration* to address both of these goals.

Lazy migration defers both the migration of live nodes and the flushing of parent nodes for as long as possible—until the next time the particular node is flushed due to a checkpoint or eviction. Because multiple migrated LNs might have the same parent BIN that needs to be flushed, deferring the flush significantly reduces the number of times that BINs are written to the log. Additionally, by offloading some cleaning activity into the checkpointer and evictor, the cleaner is better able to keep up with write-intensive applications. Application threads perform eviction before each database operation, and since cleaner migration is performed as part of this eviction, the application is throttled appropriately.

While marking active LNs for lazy migration, the cleaner maintains a look-ahead cache. When the cleaner latches a BIN to mark an LN for migration, it also checks the look-ahead cache for other LNs belonging to that BIN. Since adjacent LNs commonly migrate together, this look-ahead policy reduces the number of tree lookups.

In general, the cleaner tries to achieve a target disk space utilization (50% by default) for the database as a whole. However, even with lazy migration, the cleaner can fall behind in some extreme cases. In these cases, the cleaner uses an additional technique, called *proactive migration*.

If the cleaner is not meeting its target utilization, it maintains a list of files that will be cleaned in order to meet the target. The evictor and checkpointer will then proactively migrate LN entries in those files, working ahead of the cleaner, reducing the amount of work performed as part of cleaning. This too has the effect of throttling the application until equilibrium is reached.

No matter how efficient cleaning is, it is possible to create an application where a single cleaner thread cannot keep up with the application threads performing I/O, because the cleaner thread competes for I/O bandwidth to read the log files being cleaned. This effect is most pronounced in applications with high write rates and non-durable transactions. JE supports multiple cleaner threads to accommodate such applications. Each thread cleans a separate log file. Alternately, increasing the size of the cleaner's read buffer also helps the cleaner keep in in the presence of write-hungry applications.

We expect that further improvements can be made in both cleaner efficiency and balancing cleaning and application workloads. Based on historical experience with log-structured file systems, it is likely that we will be tuning the cleaner for as long as the JE software is maintained.

4. RELATED WORK

There are three types of prior work related to this paper. First, there is the enormous literature on transactional systems. Second, there is the rich research history in log-structured file systems. Lastly, there are a number of alternative pure Java database implementations. We discuss each of these areas in the next three sections.

4.1 Transactional Systems

The transaction concept grew out of the database community and data processing needs of the early 70's [13]. With a few notable exceptions, transactions were entirely a service of the database management system until fairly recently. The notable exception is the Quicksilver system [16], which used transactions throughout a distributed system to provide consistent state management. In the late 80's and early 90's there were a number of investigations of the feasibility of providing transaction support in file systems [19, 25, 26], but none of these approaches seemed to have broad impact. Instead, journaling file systems, which borrow the database logging concept to provide meta-data integrity, have become common [6, 11, 13]. More recently, the Reiser4 file system [23] fully embodies transactional support. Each of the file system's system calls is implemented as a transaction, and the intention is to export transaction begin, commit, and abort functionality to user-level.

To the best of our knowledge, Berkeley DB was the first system to provide full transactional recovery in a library-package. This packaging allowed applications to embed services typically found in DBMSs within an application, so that applications could be deployed without requiring database administration. JE addresses this same market in the context of a JVM or J2EE environment.

4.2 Log-Structured File Systems

Although the first relational database (System R [1]) used a no-overwrite storage system, maintaining shadow copies of data until commit time, Ousterhout and Rosenblum's log-structured file system (LFS) [24] is the intellectual ancestor of the JE design. JE implements all the ideas of a log-structured file system, but provides a database abstraction rather than a file-system abstraction atop this segmented log. While a conventional LFS uses fixed size segments for its log, JE uses files to represent segments. Since JE sits on top of a file system, JE is able to store more meta-data to assist in optimizing the cleaning process.

There have been a number of studies criticizing LFS for excessive cleaner overhead [27, 28], and this is a concern for JE as well. However, the fact that we are targeting a Java environment allows us to make a set of assumptions that simplify the problem.

First, we expect a large class of applications to have data that is entirely memory resident. In these applications, cleaning's I/O overhead, which is the typical Achilles' heel of log-structured file systems, is not a big issue. The only performance impact results from the cleaner competing with the application for CPU time. Increasing processing power and today's multi-threaded processors make this a smaller problem than it has been historically.

Second, memories are an order of magnitude larger today than a decade ago. Larger memories allow for cleaning more data simultaneously, and aggregating data during cleaning has been shown to lessen the burden of cleaning [28].

Third, disks are an order of magnitude larger today than a decade ago. Larger disks allow cleaning to be postponed until down periods when the database environment is not as active [4].

Fourth, the complexity of a log-structured file system implementation is found in disk space accountability: log-structured solutions have a difficult time accounting for disk space usage, and any specific write into the file system can consume more blocks than are freed up by the write, leading to implementation complexity and possible starvation. The JE implementation does not have to worry about either of these issues, because it sits on top of a conventional file system.

4.3 Java Databases

There are a number of pure Java database products on the market today, although none is directly comparable to JE.

There are a number of Java SQL products such as McKoi [20], HyperSonic SQL [17], Axion [3], and Derby [10], all of which rely on a JDBC interface. Although JDBC permits both embedded and client-server use, none of these products core emphasis is on embedded use, so there is greater competition for resources within the JVM using these products than there is with JE. Additionally, the reliance on SQL means that programmers must organize and access data in a relational model within the context of an object oriented language, while JE provides a natural collections-oriented interface, a Direct Persistence Layer, and its native key/data pair interface.

In the realm of more object-oriented data management products, db4o [9] is a newcomer in this space. It provides "simple object data access" (SODA) as well as query-by-example (QbE). However, db4o assumes that object graph storage is sufficient for all applications providing far less data management flexibility than is available with JE.

Finally, there are a few simple Java persistent storage managers such as JDBM [18] and Solinger SDBM [29]. They provide simple dbm/ndbm-like [2] access to persistent data in Java, but do not provide multi-threaded access, nor do they provide transactional guarantees, but are instead focused on relatively simple data management.

5. APPLICATIONS USING JE

The key distinction between JE and the other systems discussed in the previous section is its flexibility. If an application is not wedded to a SQL data management interface, then JE can be molded to address practically every data management need. Indeed, this is precisely what we observe in our customers' applications. In this section, we provide three examples of how customers are using JE.

5.1 Internet Archive's Heritrix Web Crawler

The Internet Archive (archive.org) is an internet library that currently holds over 50 billion URLs (and rising). Heritrix is the Internet Archive's open source Web crawler. Its emphasis is on its pluggable, extensible architecture that facilitates customization and external contribution. Historically, the size of Heritrix's Web crawl was bounded by a large in-memory Java collection. In adopting JE, the goal was to retain the Java collection abstraction while handling a data set that could exceed memory capacity without suffering a significant performance penalty.

Heritrix now uses JE to maintain a queue of all the URIs to crawl. JE is the backend for a Java Map that caches these URIs and can grow without bound. This application uses both the native API as well as the collections API, but does not use transaction support.

The Internet Archive, using the Heritrix Web crawler, archives the majority of the Internet for posterity. Old versions of most Web sites are available for all time. This requires a massive amount of storage. They are investigating a petabyte-sized storage system called the 'Petabox' (http://www.archive.org/web/petabox.php) of their own design to manage the volume of data required for this task. Their experience with JE has led them to begin working toward a JE-based backing store for the Wayback Machine. When finished, this will quickly become one of the largest databases in the world.

5.2 Amazing Media

Amazing Media provides a Web-based classified listings application. The application architecture is service-based and services use JE as the repository for persistent data. The design goal of the application is to keep services as simple as possible. One way of achieving this is to maintain an object-oriented view of the data in the service, but provide transactional support, performance, and reliability.

The application provides familiar Java serialization via a custom implementation with a high-performance, efficient transactional data storage. It uses dynamic analysis of the "getters" and "setters" of the objects to identify which parts of the object need to be persisted. The key characteristic of JE that made it suitable for this application is its agnosticism with respect to the data it stores. This application uses JE's transactions and the native API, including secondary indexes.

5.3 TIBCO's Business Events

BusinessEvents is a rule- and complex event-processing system used to correlate events and execute rules based upon those correlations. It must provide highly reliable, high throughput persistence of the system state, capable of meeting TIBCO's target event/second rates. In this application, JE is used to store system state every 20-30 seconds, performing checkpointing and the ability to recover after a failure. BusinessEvents uses the native API and transactions.

6. PERFORMANCE

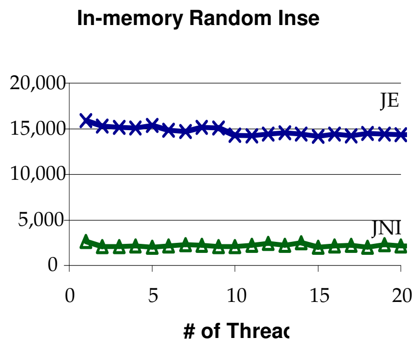
We began this paper by citing some of the advantages of JE and its log-structured architecture. Up to this point, we've focused largely on JE's architecture and its functional flexibility. In this section, we illustrate its performance characteristics. We compare JE 2.1.30 to its JNI counterpart implemented atop Berkeley DB's C library using a pre-release version of 4.5 (referred to as JNI for the rest of this paper). Such a comparison is not perfect as the C library has been in widespread commercial use for nearly a decade and has been optimized, while JE has been in commercial use for less than two years. JE has undergone much less extensive performance tuning. Nonetheless, it is the best comparison available and highlights the areas where JE's architecture delivers outstanding performance.

6.1 Evaluation Platform

All the test results reported here were run on a Dell Inspiron 8600 running Windows XP. The machine has a 1.8 GHz Pentium M with a 2 MB L2 cache and 1 GB of main memory. The default disk is a 7200 RPM 60 GB ATA-100 Hitachi HT S726060M9AT00 with an 8 MB buffer.

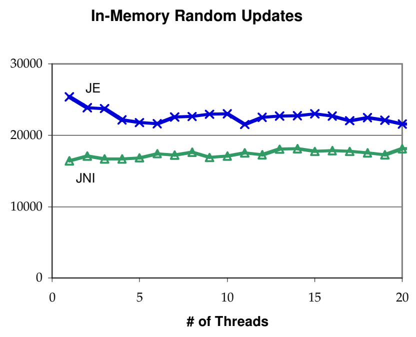
6.2 Insert Performance

Our first test demonstrates how JE's architecture delivers outstanding write performance. The insert benchmark began with the database empty. We then (transactionally) added 200,000 key/data pairs where the keys were 6-byte alphanumeric strings and the data items were 294 bytes. The cache was sized so that the entire database fit in the cache for both JE and JNI, but checkpoints were enabled. Since the database fits entirely in memory, the only I/O activity is due to log writes and checkpoints. The systems are both configured to commit asynchronously, so log records are written to disk only when the in-memory buffer fills or at checkpoint. The results in the graph titled "In-memory Random Insertion" show that the benchmark performance is determined completely by the system's ability to write data to disk at checkpoint. JE's log-structured architecture delivers near-sequential disk write performance, because data need only be written to the log. In contrast, JNI checkpoints must update database pages in-place, producing random I/O performance.



6.3 Update Performance

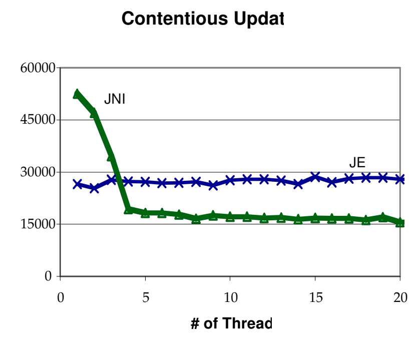
JE's log-structured storage provides similar performance benefits when we update existing data items. We begin with the database created in the Insert benchmark and then select 200,000 records uniformly at random and update them. The graph titled "Random Updates" demonstrates JE's write-optimized design, delivering near sequential disk performance instead of JNI's random disk performance.



6.4 Concurrency

Our next benchmark explores the improved concurrency possible due to JE's record level locking. This benchmark is similar to the one in Section 6.3, except that rather than select the records uniformly at random, we skew the distribution selecting less than 1% of the records for update. This sets the stage to explore the behavior of the system under contention. Our expectation is that the record-level locking of JE will produce better scalability than JNI's page-level locking.

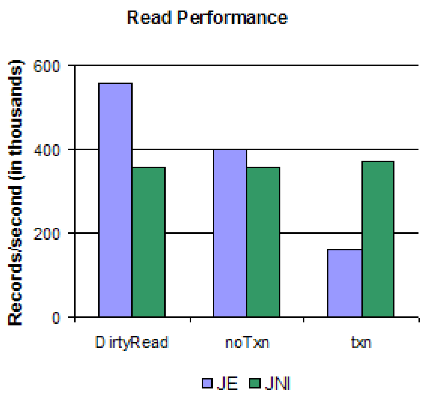
The graph titled "Contentious Updates" shows that JE performs as we expect. Its performance does not degrade under a highly concurrent workload. With 200,000 items in the database, even with 20 threads selecting from among the 100 "hot" records, the probability that two threads conflict is low. In contrast, JNI's page locking induces significant hotspots, and throughput drops.



It is worth noting that in the single-thread case, JNI outperforms JE significantly. The highly skewed access pattern in this benchmark means that JNI dirties only a few pages (under 10); on checkpoint, it has little data to write to disk. Thus, in the absence of contention, JNI performance is limited by its ability to flush data at checkpoint, and there is so little data, the resulting performance is significantly higher than we saw in earlier tests. In contrast, JE's no-overwrite storage means that log records are generated for each update even if the item being updated was recently updated. This highlights the trade-offs inherent in conventional and no-overwrite storage systems.

6.5 In-memory read Performance

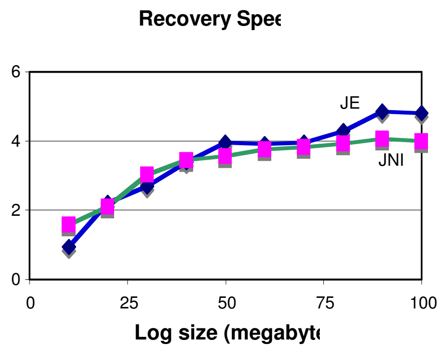
Having demonstrated the outstanding JE write performance, we next turn to read performance. In this test we populate the database as before (200,000 key/data pairs with 6-byte keys and 294-byte data items). This time, once the database has been populated, we iterate over the entire data using a cursor. We present three variations of this test: Degree-3 serializable reads (txn), Degree-1 dirty reads (DirtyRead), and non-transactional reads (noTxn). The graph titled "Read Performance" shows the results. JNI locking is similar under all configurations and we observe little difference between them. However, JE takes advantage of weaker semantics delivering significant performance improvement. In the DirtyRead configuration, JE bypasses the lock manager entirely, using the high-speed latches to control concurrent access, producing the excellent JE-DirtyRead performance. In the non-transactional case, even though JE must obtain more locks than JNI, its performance is comparable and even slightly better than JNI's. JE's transactional performance falls significantly below the non-transactional performance due to the difference in the transactional and non-transactional locking implementations. The



transactional implementation is optimized for collections of many locks and this benchmark pays the overhead for these optimizations, even though there is only a single data item being locked.

6.6 Recovery Performance

Our last test measures JE's recovery time to alleviate concerns over its more complicated recovery mechanisms. In this test, we produce a large log file by running update transactions similar to those used in Section 6.3. Then we truncate that log to a variety of sizes ranging from 10 to 100 MB and measure how long it takes to recover the log file. As shown in the graph titled "Recovery Speed," JE and JNI exhibit comparable recovery times, demonstrating that JE's more complicated recovery does not incur significant overhead, since recovery time is dominated by the time to read the log and associated data items.



7. CONCLUSIONS

We have presented the design and implementation of the Berkeley DB Java Edition, a native Java transactional data manager. JE's log-structured storage system delivers outstanding write performance without jeopardizing read performance.

8. AVAILABILITY

Additional information about Berkeley DB Java Edition and the full product including source code, documentation, sample code and test code is available for download from:
http://www.oracle.com/technology/products/berkeley-db/je/index.html.

9. REFERENCES

1. Astrahan, M., et al, "System R: A Relational Approach to Database Management," *ACM Transactions on Database Systems*, p. 97, June 1976.
2. AT&T, DBM(3X), *Unix Programmer's Manual*, Seventh Edition, Volume 1, January, 1979.
3. Axion, http://axion.tigris.org, project home page.
4. Blackwell, T., Harris, J., Seltzer., M. "Heuristic Cleaning Algorithms in Log-Structured File Systems," *Proceedings of the 1995 USENIX Technical Conference*, pp. 277–288, New Orleans, LA, Jan. 1995.
5. NDBM(3), 4.3BSD *Unix Programmer's Manual Reference Guide*, University of California, Berkeley, 1986.
6. Chutani, S., Anderson, O., Kazer, M., Leverett, B., Mason, W.A., Sidebotham, R. "The Episode File System," *Proceedings of the 1992 Winter USENIX Technical Conference,* pp. 43–60. San Francisco, CA, Jan. 1992.
7. Cloudscape, http://www-3.ibm.com/software/data/ cloudscape. Product Overview.
8. Comer, D., "The Ubiquitous B-tree," *ACM Computing Surveys* Volume 11, number 2, June 1979.
9. db4objects: www.db4o.com.
10. Derby, http://incubator.apache.org/derby.
11. Elkhardt, K., Bayer, R. "A Database Cache for High Performance and Fast Restart in Database Systems," *ACM Transactions on Database Systems,* 9(4), pp. 503– 525. Dec. 1984.
12. Firstsql, http://www.firstsql.com, Home page.
13. Gray, J., "The transaction concept: Virtues and limitations," Proceedings of the Seventh International Conference on Very Large Databases, pp. 144–154, 1981.
14. Gray, J., Reuter, A. *Transaction Processing: Concepts and Techniques.* San Mateo, CA: Morgan Kaufmann, 1993.
15. Hagmann, R. "Reimplementing the Cedar File System Using Logging and Group Commit," *Proceedings of the 11th SOSP,* pp. 155–162. Austin, TX, Nov. 1987.
16. Haskin, R., Malachi, Y., Sawdon, W., Chan, G. "Recovery Management in QuickSilver," *ACM Transactions on Computer Systems,* 6(1), pp. 82–108. Feb. 1988.
17. Hypersonic SQL, http://hsqldb.sourceforge.net, software distribution page.
18. JDBM, http://jdbm.sourceforge.net, software distribution page.
19. Kumar, A., Stonebraker, M., "Performance Evaluation of an Operating System Transaction Manager," *Proceedings of the 13th International Conference on Very Large Data Bases*," Brighton England, pp. 473-481, 1987.
20. McKoi, http://mckoi.com/, McKoi project home page.
21. Olson, M., Bostic, K., Seltzer, M., "Berkeley DB," *Proceedings of the 1999 Freenix Conference, Monterey*, CA, June 1999.
22. Pointbase, http://www.pointbase.com, Pointbase home page.
23. ReiserFS4, http://www.namesys.com/v4/v4.html, Reiser4 design and rationale.
24. Rosenblum, M., Ousterhout, J. "The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer Systems*, 10(1), pp. 26–52. Feb. 1992.

25. Seltzer, M., Stonebraker, M., "Transaction Support in Read Optimized and Write Optimized File Systems," Proceedings of the 16th International Conference on Very Large Databases, Brisbane, Australia, 174–185, 1990.
26. Seltzer, M., "Transaction Support in a Log-Structured File System," *Proceedings of the Ninth International Conference on Data Engineering*, Vienna, Austria, pp. 503–501, 1993.
27. Seltzer, M., Bostic, K., McKusick, M.K., Staelin, C. "An Implementation of a Log-Structured File System for UNIX," *Proceedings of the 1993 USENIX Winter Technical Conference*, pp. 307–326. San Diego, CA, Jan. 1993.
28. Seltzer, M., Smith, K., Balakrishnan, H., Chang, J., McMains, S., Padmanabhan, V. "File System Logging versus Clustering: A Performance Comparison," *Proceedings of the 1995 USENIX Technical Conference*, pp. 249–264. New Orleans, LA, Jan. 1995.
29. Solinger SDBM, http://sourceforge.net/projects/solinger, software distribution page.
30. Tanenbaum, Modern Operating Systems, second edition, pp. 182–183, 214–220, Prentice-Hall, Englewood, NJ, 2001.

Berkeley DB Java Edition Architecture
September 2006

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com