JavaOne

Sun's 2004 Worldwide Java Developer Conference

java.sun.com/javaone/sf

The Design and Implementation of a Transactional Data Manager

Berkeley DB Java™ Edition

Charles Lamb

Architect
Sleepycat Software

http://www.sleepycat.com

SLEEPYCAT SOFTWARE

Makers of Berkeley DB

Java

Sun microsystems

Presentation Goal

Provide insight into the technical trade-offs involved in implementing a robust, high-performance, transactional database *in the Java™ programming language*

Agenda

Motivation and Overview

Code Examples

Log-Based, No-Overwrite Storage System

Highly Concurrent Tree Updates

Integration with Memory Management

Performance

Summary and Conclusions

Agenda

Motivation and Overview

Code Examples

Log-Based, No-Overwrite Storage System

Highly Concurrent Tree Updates

Integration with Memory Management

Performance

Summary and Conclusions

Motivation and Overview

Berkeley DB Java™ Edition

- A small-footprint, high-speed, transactional database for:
  - Backend storage or caching for web/app server
  - Embedded transactional database
  - Lightweight persistence for the Java programming language
  - Traditional transaction processing

Motivation and Overview

Standard DBMS features

- ACID properties
- B+Tree access method
- Record locking
  - Read-modify-write locks
  - Dirty reads
- Recovery
- Large database support
  - Hundreds of gigabytes of data
  - Tens of millions of records

Motivation and Overview

Unique features (discussed here)

- Log-based, no-overwrite storage system
- Highly concurrent tree updates
- Graceful interaction with JVM™ memory system

Motivation and Overview

Unique features (not discussed here)

- Schema independent
- No ad-hoc queries
- In-memory speeds, no IPC from client to server
- Lights-out operation
- N:M transaction:thread model
- Open source
- No native/Java Native Interface (JNI) code

Motivation and Overview

Design assumptions

- Log-based system improves performance
- Single-process, multi-threaded access
- Both steady and “bursty” workloads
- Read-mostly workloads
  - Support high-concurrency writes too
- Typically “in-memory” applications
  - But degrade gracefully if can’t fit in-memory

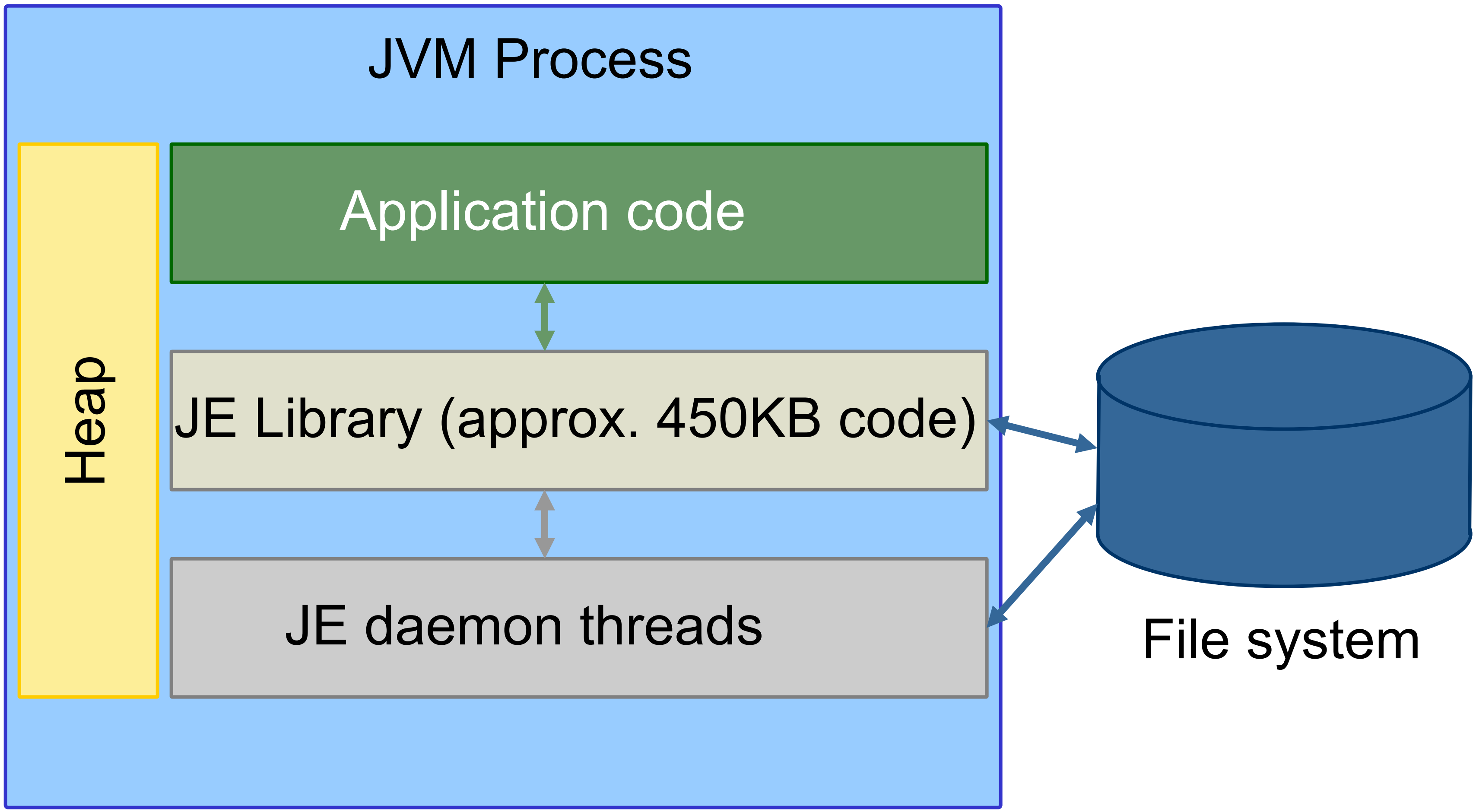
Motivation and Overview

Terminology

- Environment (think Relational DB ‘Database’)
  - JE Database(s) contained within environment
  - May be transactional
  - Sequentially numbered log files
  - Several daemon threads
- Database (think Relational DB ‘Table’)
  - Map: key/data pairs
  - May be transactional

Motivation and Overview

JE in the application JVM process



Motivation and Overview

APIs

- Persistent Collections API (built on base API)
  - Full implementation of Java Collections Framework technology
    - No new API to learn
  - Transactions, Iterators, exact and range queries, joins (set intersection)
  - Transparent use of secondary indices, foreign keys
  - Bindings allow different marshalling options
    - Compact form of Java API serialization
    - **`DataInput`/`DataOutput`** style marshalling
    - Strings and primitive wrapper classes

Motivation and Overview

APIs

- Base API
  - More specific operations than Persistent Collections
  - get/put of **`byte[]`** key/data pairs
  - Transactions, Cursors, exact and range queries, joins (set intersection)
  - Secondary indices, foreign keys
  - Multi-threaded transactions
  - Explicit configuration for performance related parameters

Agenda

Motivation and Overview

Code Examples

Log-Based, No-Overwrite Storage System

Highly Concurrent Tree Updates

Integration with Memory Management

Performance

Summary and Conclusions

Collections Example

Create a transaction

```
import com.sleepycat.collections.TransactionRunner;
import com.sleepycat.collections.TransactionWorker;

public class MyWorker implements TransactionWorker {
...
    private Environment env = new Environment(…);
...
    static public void main(String argv[]) {
        MyWorker worker = new MyWorker();
        TransactionRunner runner =
            new TransactionRunner(env);
        runner.run(worker);
    }
```

Collections Example

Add an entry to a map, read entries back

```
import com.sleepycat.collections.StoredIterator;

  public void doWork() {  // in a transaction
     // Add an entry
     map.put(“myKey”, “myData”);
     ...
     // Read entries back
     Iterator iter = map.entrySet().iterator();
     while (iter.hasNext()) {
         Map.Entry entry = (Map.Entry) iter.next();
         String keyStr = entry.getKey().toString();
         System.out.println(keyStr + “ “ +
                                  entry.getValue());
     }
     StoredIterator.close(iter);
  }
```

Base API Example

Add an entry to a database

```
import com.sleepycat.je.DatabaseEntry;
...
   // Add an entry
   Transaction txn =
       env.beginTransaction(null, null);
   DatabaseEntry key =
       new DatabaseEntry(“myKey”.getBytes());
   DatabaseEntry data =
       new DatabaseEntry(“myData”.getBytes());
   db.put(txn, key, data);
   txn.commit();
```

Base API Example

Read entries back

```
import com.sleepycat.je.Cursor;
...
   Transaction txn =
       env.beginTransaction(null, null);
   Cursor cursor = db.openCursor(txn, null);
   DatabaseEntry key = new DatabaseEntry();
   DatabaseEntry data = new DatabaseEntry();
   while (cursor.getNext
              (key, data, LockMode.DEFAULT) ==
          OperationStatus.SUCCESS) {
       System.out.println(
           new String(key.getData()) + “ “ +
           new String(data.getData()));
   }
   cursor.close();
   txn.commit();
```

Agenda

Motivation and Overview

Code Examples

Log-Based, No-Overwrite Storage System

Highly Concurrent Tree Updates

Integration with Memory Management

Performance

Summary and Conclusions

Log-Based, No-Overwrite Storage System

Overview

- Based on work by
  - Ousterhout, Rosenblum (1991)[1]
  - Seltzer, Bostic, McKusick, Staelin (1993)[2]
- Data is written only once (even updates)
  - Disk head generally stays on same track
  - Inserts and updates are fast
- Data can't be arbitrarily clustered
  - Out-of-cache reads are generally slower
  - Assumption: working set fits in memory
- Log and data (the "material DB") are the same
- Backup and restore are simplified

Log-Based, No-Overwrite Storage System

Java technology NIO usage

- Storage system uses NIO **`ByteBuffer`**
  - Makes it easy to set up a buffer pool
  - Caution: not thread safe
  - Caution: watch out for Direct Memory Buffers in 1.4.2

Log-Based Storage System

Cleaner (Persistent Garbage Collector)

- Every N MB's, log switches to a new file
- Periodic consolidation is required
  - Background daemon thread
  - User invoked through API call
- Cleaner scans old log files
  - For each entry in the file, determine if it is in the tree
    - Migrate live data to current log file
    - Discard (by ignoring) obsolete data
  - If all records processed successfully, delete the file

Log-Based Storage System

Cleaner log file selection

- Two cleaner functions
  - Data migration (easy)
  - Log file selection (harder)
    - Maintain utilization info about “live” data per log file
    - Stored in a database in the environment
      - No additional files or file formats
      - Recoverable

Log-Based Storage System

Cleaner challenges in Java technology-based applications

- Problem: How much IO/CPU is being used?
- Solution: Application invokes cleaner during quiet times
  - Future: Application can set cleaner thread priority
  - Future: Allow cleaner to run in a thread pool

Log-Based Storage System

Backup

- Full
  - Copy all log files
- Incremental
  - Copy all log files that are new or modified since last backup
- No “holes” in the log means it is always consistent
- Hot and cold backup are the same
  - No locks required during hot backup

Log-Based Storage System

Recovery

- Recovery from system failure
  - Open the database, JE performs recovery
- Recovery from media failure
  - Restore log files from backup
  - Open the database, JE performs recovery
- Interval between checkpoints bounds recovery time

Agenda

Motivation and Overview

Code Examples

Log-Based, No-Overwrite Storage System

Highly Concurrent Tree Updates

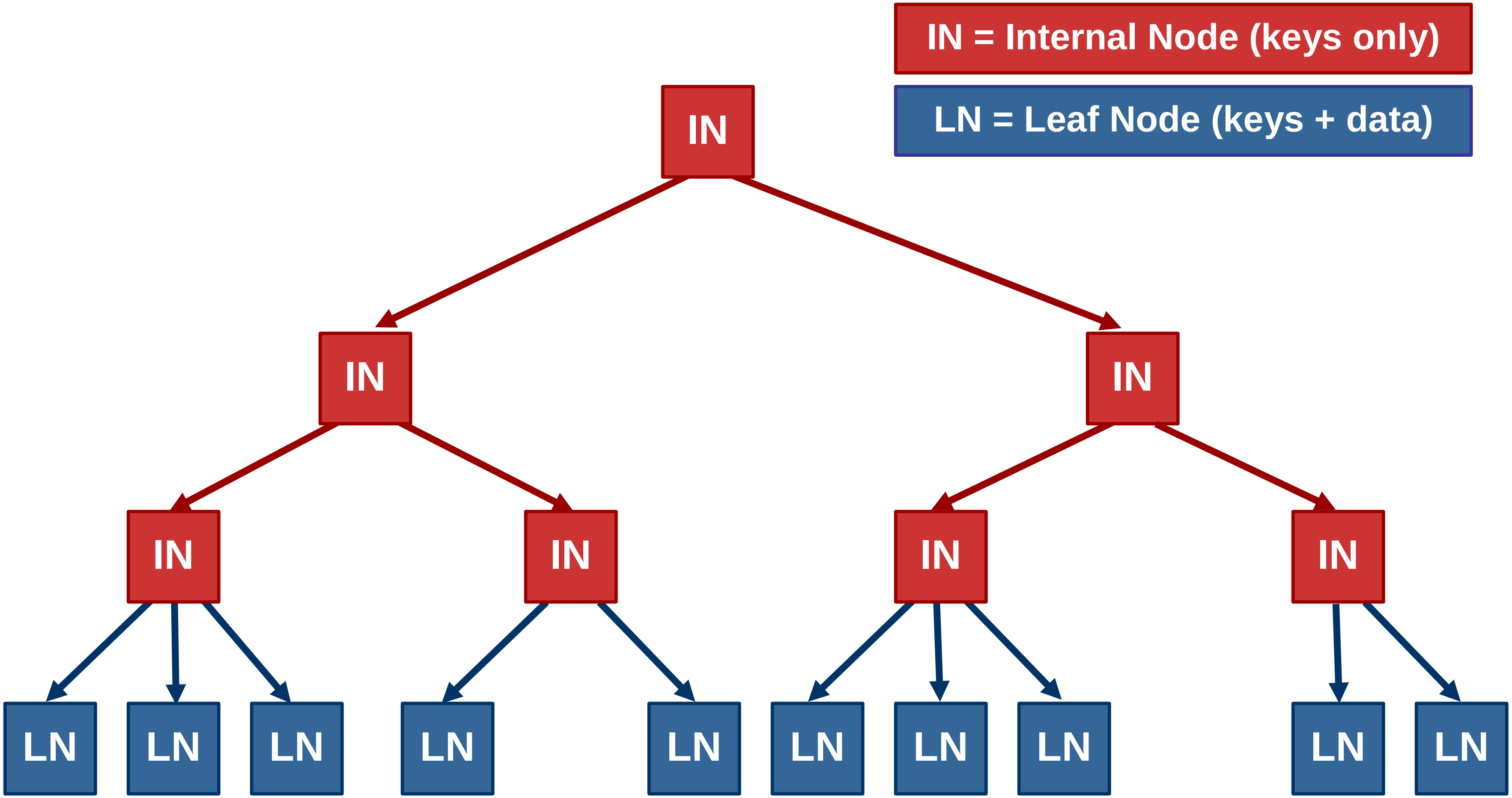
Integration with Memory Management

Performance

Summary and Conclusions

Highly Concurrent Tree Updates

Tree structure



Highly Concurrent Tree Updates

Latches

- Lightweight mutexes on internal data structures
- Exclusive and shared/exclusive varieties
- Never held across user API calls (short-lived)
- Deadlock avoidance, not deadlock detection
  - Multiple latch acquisitions require strict ordering
- **`synchronized`** keyword isn't sufficient
  - Fairness (first come, first served) required
  - Shared/exclusive necessary

Highly Concurrent Tree Updates

Logical locks

- Heavyweight mutexes for locking records
  - Available with or without transactions
- Held across API calls
- Read (shared), write (exclusive) varieties
- Standard two-phase locking semantics
  - Held until transaction end or cursor advance
- Deadlocks are detectable
- Implemented using latches
- First come, first served behavior

Highly Concurrent Tree Updates

Tree latching and locking

- Record-locking
- LNs (records) are locked, not latched
- INs are latched for short periods, not locked
- Latch couple INs during tree descent
- Never latch up the tree
  - Latching up can cause latch deadlocks
  - No parent pointers in the tree removes temptation

Highly Concurrent Tree Updates

Impact on transactions and recovery

- INs do not require rollback
- INs may contain keys that are aborted or not yet committed
  - Gets/puts block on LN lock during concurrent access
- LNs are transactional
  - Aborts cause reversion to last committed LN
  - LNs are the final authority on key/data pairs

Highly Concurrent Tree Updates

Compressor

- Tree splits
  - Opportunistic
  - Not undone in an abort
- Compressor
  - Tree rebalancing operations
  - Tree cleanup from delete operations
    - Remove deleted LNs and empty subtrees
  - Daemon thread or API call

Agenda

Motivation and Overview

Code Examples

Log-Based, No-Overwrite Storage System

Highly Concurrent Tree Updates

Integration with Memory Management

Performance

Summary and Conclusions

Integration With Memory Management

The JE memory cache

- JE shares JVM process with application
  - Lives within user-specified memory budget
- JE maintains it’s own memory cache
  - Approximates LRU behavior
  - JE cache chooses eviction target
  - Can’t use weak/soft references

Integration With Memory Management

Challenges

- Determining actual object space usage
  - J2SE™ 1.5 platform will help to calculate better object overhead sizes (in the future)
- Maintaining total JE cache space usage

But…

- There's no internal fragmentation
  - Everything is allocated as objects
  - No fixed size buffers/pages

Integration With Memory Management

Evictor

- Flushes memory cache
- Daemon or API call
- To evict an object
  - Select a victim (IN or LN)
  - Write to log if dirty
  - Null the reference to it
  - Let the GC do the rest
- One specialized concurrent data structure

Integration With Memory Management

Objects vs. pages

- Positives:
  - Objects have better granularity than pages
    - Locking
    - I/O
    - Memory management
  - Java technology is optimized for lots of small objects
- Negative:
  - Variable object sizes harder to manage

Agenda

Motivation and Overview

Code Examples

Log-Based, No-Overwrite Storage System

Highly Concurrent Tree Updates

Integration with Memory Management

Performance

Summary and Conclusions

Performance

Benchmark configuration

- Commodity hardware ($1600)
  - Dual CPU Pentium Xeon, 2.4GHz, hyper-threading
  - 1024 MB memory
  - Windows XP
  - 7200 RPM IDE disk
- J2SE 1.4.2 platform
- 300 byte records
  - 6 bytes key + 294 bytes data

Performance

Update benchmark methodology

- 20,000 x 300 byte records
- Baseline throughput (writes per second)
  - Straight java.nio writes with and without fsyncs*

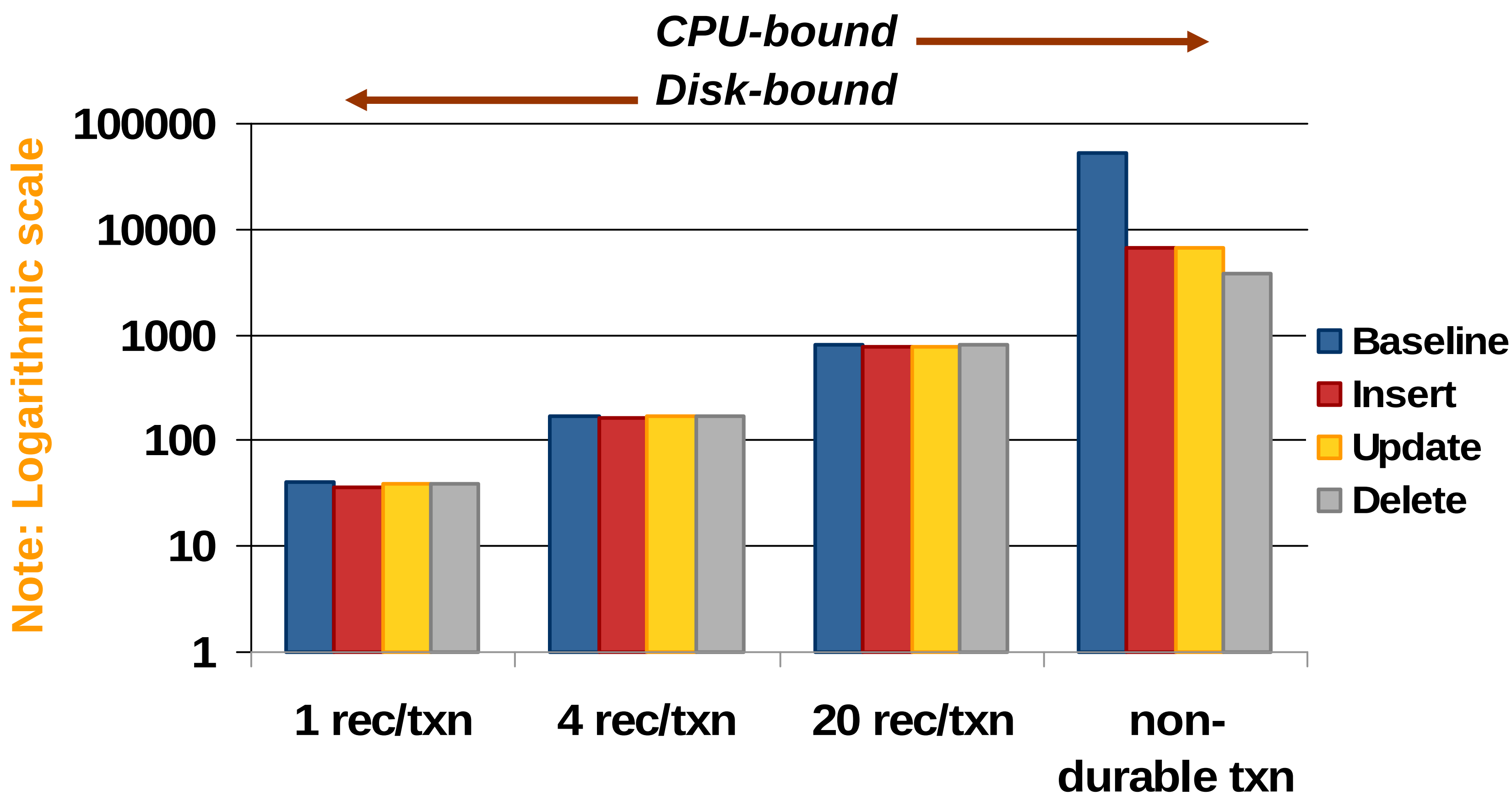
Compared to…

- JE insert, update, delete throughput
  - Vary number of records per transaction
  - With and without fsyncs* at transaction commit

* “without fsyncs” implies non-durable transactions

Performance

Modifications per second



Source: Sleepycat internal performance measurement tests

Performance

Read benchmark methodology

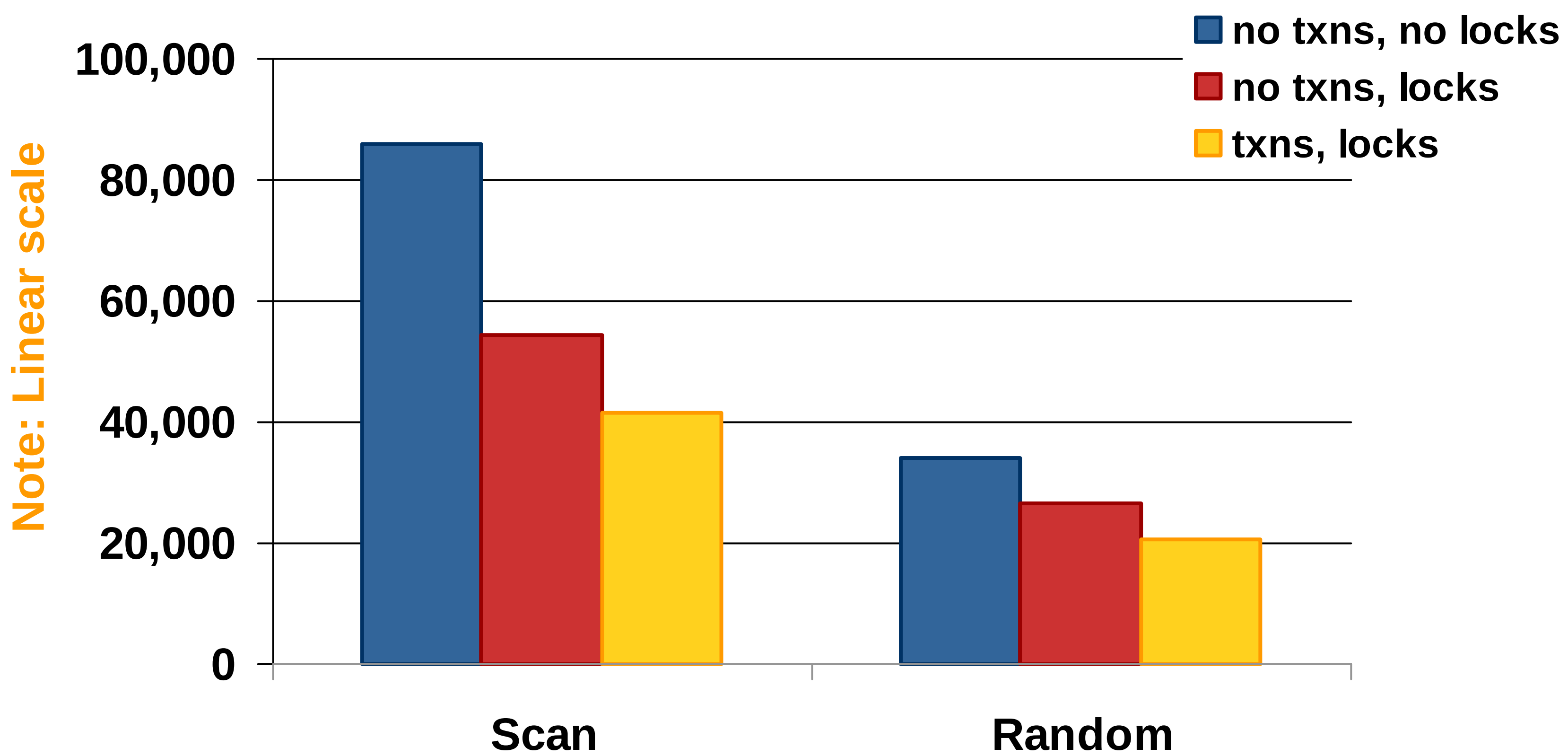
- 200,000 x 300 byte records
- Sequential data scans of entire database
  - With and without locks/transactions
  - When transactions are enabled, benchmark performs *one complete scan per transaction*

Compared to…

- Random reads
  - With and without locks/transactions
  - When transactions are enabled, benchmark performs *one read per transaction*

Performance

Reads per second–Warm cache



Source: Sleepycat internal performance measurement tests

Agenda

Motivation and Overview

Code Examples

Log-Based, No-Overwrite Storage System

Highly Concurrent Tree Updates

Integration with Memory Management

Performance

Summary and Conclusions

Summary

- Use of a log-based file system can improve write performance
- Maintain synergy with the garbage collector when managing memory in Java technology-based applications
- Create special highly-concurrent structures only as needed
- Optionally allow the application to manually schedule daemon functions

Conclusion

- High performance transaction processing is possible in the Java programming language if appropriate techniques are used

For More Information

- http://www.sleepycat.com to download .jar, source, Getting Started Guide, and docs
- References
  - [1] “The Design and Implementation of a Log-Structured File System”, Proceedings of the Thirteenth ACM Symposium on Operating System Principles, October 13–16, 1991.
  - [2] “An Implementation of a Log-Structured File System for UNIX®”, Proceedings 1993 Winter USENIX, San Diego.

Q&A

JavaOne™

Sun's 2004 Worldwide Java Developer Conference™

java.sun.com/javaone/sf

The Design and Implementation of a Transactional Data Manager

Berkeley DB Java™ Edition

Charles Lamb

Architect
Sleepycat Software

http://www.sleepycat.com

SLEEPYCAT SOFTWARE

Makers of Berkeley DB

Java™

Sun microsystems