TDT4740 AUTUMN PROJECT 2006

APACHE DERBY SMP SCALABILITY

PER OTTAR RIBE PAHR AND ANDERS MORKEN
SUPERVISOR: SVEIN-OLAF HVASSHOVD

Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Computer and Information Science

Abstract

We investigate the scalability properties of Apache Derby, an open source Java database system, on multiprocessor systems. We use dynamic tracing to locate points of resource contention between threads that limit Derby's scalability.

To limit the complexity of analysis while still yielding interesting insights, we concentrate our efforts on a simple single-tuple read transaction workload.

The observed contention points are then analyzed, with tracing and manual auditing, to better understand why they cause contention and how the implementation can be improved to relieve the problem.

Based on these findings, possible improvements to the Derby lock and cache subsystems are suggested.

Preface

This report is the result of our autumn project for the course TDT4740 *Database Technology and Distributed Systems, Specialization* in the autumn semester of 2006. We have studied the open source database Apache Derby to understand how it scales on multiprocessor systems and identify performance bottlenecks and possible improvements. The project has been intended as a prestudy for a possible Master's thesis, but we hope that our work is also of interest to the Derby community.

We would like to thank our supervisor Svein-Olaf Hvasshovd for the help and input we have received during this project. We also thank those at Sun Microsystems in Trondheim that helped formulate the task and for discussion of our early findings: Olav Sandstaa, Øystein Grøvlen and Knut Anders Hatlen.

Our task for this project is:

> *Derby: SMP scalability*
>
> *Investigate the scalability of Derby when increasing the number of CPUs on a multi-cpu machine, e.g., when scaling from one to 16 CPUs. Workload: read-only (SELECT) on a main-memory database. Identify the extra cost of running on multiple CPUs, identify the bottlenecks for linear throughput scalability and suggest areas for improvements.*

December 18, 2006

Per Ottar Ribe Pahr

Anders Morken

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

This chapter serves as an introduction to the project and this report. We will describe the background for the project, the task we were given, our goals and the structure of the report.

1.1 The project

This is our project report for what is commonly called the "autumn project" for the course TDT4740 *Database Technology and Distributed Systems, Specialization* at IDI, NTNU. The course is part of the ninth semester of the Master of Science degree in Computer Science. 22.5 credit points[1] are given for the course, of which 15 points are for the project and 7.5 points for two theoretical subjects the students are required to take in addition to the project.

We have chosen to specialize in database systems, which is part of the "Data and Information Management" direction. In addition to the project we have picked the subjects TDT27 *Reliability and continuous availability in database systems* and TDT33 *Transaction processing* for our specialization course.

1.2 The task

In testing by Sun Microsystems Derby's scalability on SMP[2] systems has been evaluated and found to be somewhat limited.

For a very simple read-only workload throughput increased by only a factor of 3.5 when the number of CPUs was increased from 1 to 8. In cooperation with NTNU a task was formulated in order to look into the reasons for these limitations and to see if the performance of Derby on SMP systems could be improved.

[1] ECTS credit points. An academic year of full-time study equals 60 credit points.

[2] Symmetric multiprocessing systems - a single system with multiple processors and shared memory

Our task for this project was formulated as:

> *Derby: SMP scalability*
>
> *Investigate the scalability of Derby when increasing the number of CPUs on a multi-cpu machine, e.g., when scaling from one to 16 CPUs. Workload: read-only (SELECT) on a main-memory database. Identify the extra cost of running on multiple CPUs, identify the bottlenecks for linear throughput scalability and suggest areas for improvements.*

We found this task interesting, and chose it after some discussion with the supervisor responsible for the task, Svein-Olaf Hvasshovd. In discussions with Sun Microsystems we decided to look at only a simple single-tuple read only workload. We have not done any benchmarks on as many as 16 CPUs, but the scalability issues are obvious on the 4-way and 8-way systems we had available for testing.

1.3 Purpose

The goal of the project is to study the scalability of Derby on multiprocessor systems. We want to understand how Derby scales when running on a multiprocessor system and *why*. This will help us identify bottlenecks that impede performance. Based on this knowledge we can suggest possible changes to Derby to improve performance on multiprocessor systems.

We might also, if we have enough time, do test implementations of such changes to see if performance can be improved. This is also a possible task for a Master's thesis. In this regard the project can be seen as a prestudy for further work on a possible thesis.

1.4 Report structure

In this report we will in Chapter 2 introduce the reader to the "state of the art", i.e. to database systems in general and to Apache Derby. Even though we describe some of the theory behind database systems, the reader is assumed to have some knowledge of the field and of computer science in general.

Chapter 3 describes the workload for our experiments, the test systems and the tools we have used.

In Chapter 4 we present the results of our work and the benchmarks and tests we have performed.

Chapter 5 gives our analysis of the findings.

Finally, Chapter 6 provides our conclusion and suggestions for improvements to make Derby scale better on multiprocessor systems.

Chapter 2

State of the art

This chapter first provides a brief introduction to some of the fundamental concepts and problems in the design and implementation of database systems. We then take a closer look at the architecture of Apache Derby and present some prior work on performance analysis of Derby that we are aware of.

2.1 Database fundamentals

A relational DBMS provides persistent storage of data organized as relations. A relation contains data in the form of tuples (rows). The DBMS also provides an interface for queries to the database, usually through some variant of SQL.

Consistency constraints can be placed on relations, e.g. to avoid duplicate entries or disallow `NULL` values for certain fields. Simple database operations (queries) are assumed to be atomic, i.e. they are performed successfully or they have no effect at all. A good introduction to database systems is given in [22].

The application programmer often requires the atomicity property for a sequence of operations[1]. This introduces the *transaction* concept. A transaction is a sequence of one or more database operations grouped together as one logical operation. The sequence as a whole must either be performed successfully or have no effect at all.

To summarize, we say that a transaction processing system must satisfy the ACID properties, which are:

- **Atomicity:** A transaction either completes successfully or has no effect on the database.
- **Consistency:** A transaction cannot leave the database in an inconsistent state. If the database is consistent when a transaction begins, consistency is preserved when the transaction completes. Four degrees of consistency are defined in [25], these will be explained in Section 2.1.4.

[1] A classic example is the transfer of money between two bank accounts. You want to make sure that if you debit one account you also credit the other.

- **Isolation:** A transaction must appear isolated from other database operations. No intermediate results can be visible to other transactions, even if several transactions are executed concurrently.
- **Durability:** Once a transaction has completed successfully, i.e., committed, its results are persistent. This is ensured by logging and recovery algorithms.

2.1.1 Storage and indexes

A DBMS must have a storage manager that provides persistent storage, usually on disk. It can either access storage devices directly, or through the file system abstraction layer provided by the operating system it is running on. Often a table, or even an entire database, is stored within one logical file.

Records within a file can be stored unordered, called a *heap*, or sequentially based on a key value. Access to records can be classified as sequential, i.e. reading all or a consecutive range of records, or random based on key/field value.

For a naive implementation, a random lookup on field value has to read the whole[2] file to see if any matching rows exist.

To avoid this an *index* on the field can be built for the table. An index contains tuples on the form *(key, address)* where *key* is the value for the indexed field and *address* is the location (physical or logical) where the record containing this value can be found.

An index is typically much smaller than the data table, since only the key and corresponding address are stored, while the table rows can be arbitrarily large. Thus an index lookup to find the key, and then a direct access to the correct location in the file containing the table, is almost always faster than a sequential search.

When tables grow to a large number of rows the index also grows. This can be countered by building an index to the index – a hierarchical index structure.

Index structures can be grouped into two main types, order preserving and randomizing. Different hashing methods randomize the order of data. Hierarchical tree structures preserve order. The most popular tree is the B-tree and its variants [12].

A B-tree is a balanced tree structure, i.e. all leaf nodes are at the same level. Each node can contain an arbitrary, but fixed, number of keys. If a node can contain $2d$ keys, it can contain $2d + 1$ pointers to children, and we say that it is of *order d*. Nodes in a B-tree, except the root, are required to be at least $\frac{1}{2}$ full.

The B^+-tree is a B-tree where all data pointers are stored in the leaf nodes. The leaf nodes are also linked in increasing key order to provide sequential range lookups. Most DBMS provide B-tree indexing, and often the B^+ variant is used.

2.1.2 Buffer management

Usually a DBMS provides its own buffer management system in stead of, or on top of, the file system cache provided by the operating system [51]. The OS cache will

[2] If the file is ordered sequentially on the interesting field we only have to read until we find a larger value.

often use a simple buffer management algorithm, usually an LRU approximation like CLOCK [9].

These algorithms usually work well for general applications, but the access patterns of a DBMS are different. For instance it is more important to keep the upper levels of a hierarchical index in the buffer than it is to keep random data pages.

If the LRU algorithm is used and a large sequential read of data pages occurs, e.g. to produce some aggregate for the table, the frequently used root node of the index may be thrown out in favor of data pages that will not be needed again. In this case LRU exhibits worst case behavior.

The OS buffer manager will also prefetch blocks in sequential order so the next block is available when needed. This comes at a performance penalty if the blocks are not needed, which is often the case in database applications.

The DBMS on the other hand usually knows which blocks it will fetch next, although the read might not be sequential. The solution is a "smart" buffer manager that can be informed of the current access pattern and what blocks to fetch next (if any).

Implementing a seperate buffer manager in the DBMS is apparently easier than getting OS vendors to change the way their buffer managers work. A description of some DBMS buffer management algorithms and a performance analysis can be found in [11].

2.1.3 Logging and recovery

To provide the durability guarantee for data a DBMS uses a log on stable storage to record actions that modify the database. In case of a system crash this log can be used to *recover* the database, i.e. restore it to a consistent state.

Requiring that all modifications are written to the log before they affect the database is called Write Ahead Logging (WAL). When an update has been written to the log it is not necessary to force the update of the database to disk, because it can be redone from the log record in case of failure. This is called a *no-force* policy and allows for greater effiency in I/O scheduling and buffer management.

Correspondingly we can allow "dirty" values from uncommitted transactions to be written to the database, because the log can be used to undo the operations after a crash. This also allows for greater I/O scheduling flexibility and is called a *steal* policy.

Checkpoints can be used to reduce the amount of work that has to be done in case of recovery. To improve durability and performance the log should be kept on a seperate disk (or disk array) from the database.

ARIES [37] (Algorithm for Recovery and Isolation Exploiting Semantics) is a logging and recovery algorithm based on WAL and a steal, no-force policy. There are several variants of the ARIES algorithm, and they are used in many database systems.

Recovery in ARIES is done in three phases: *analysis*, *redo* and *undo*. The analysis phase scans the log from the last checkpoint to the end of the log and determines where the redo phase phase must start. Then all operations of transactions that were active at the time of the crash, but not yet reflected on disk, are redone. This is called *repeating history*.

Finally the undo phase rolls back the operations of uncommitted transactions. Operations are undone in reverse chronological order. *Log Sequence Numbers* (LSN) are used to determine if a log record is reflected on disk. Each log record is identified by a unique and monotonically increasing LSN.

When updating a disk page the LSN of the corresponding log record is written to the page header. This enables the algorithm to determine if a log record is reflected on disk. ARIES supports fine granularity locking, and latches are used for for page level consistency.

2.1.4 Concurrency control

To allow multiple clients and improve performance a DBMS supports concurrent execution of multiple transactions. Often each transaction is executed as a seperate process or thread. A discussion of different execution models can be found in [26].

To isolate running transactions from each other a DBMS must provide some form of concurrency control. Concurrency control mechanisms are either locking, *pessimistic*, or non-locking, *optimistic*. Two optimistic methods are described in [33].

A performance comparison of optimistic and locking concurrency control under different assumptions about resource availability is performed in [1]. The conclusion is that locking methods perform better on systems with high resource utilization. This is because optimistic methods waste resources by restarting transactions. On a system with low utilization, or under the assumption of infinite resources, optimistic methods win. These assumptions are obviously not always realistic.

Commercial database management systems are designed for performance, and they must provide high throughput under load. Cost effectiveness has also been an important factor for evaluation of databases. Thus, most DBMS use dynamic locking, described by Gray in [25]. The paper also defines four *degrees of consistency* for a transaction T:

- **Degree 0:** T does not overwrite dirty (uncommitted) data of other transactions.
- **Degree 1:** Degree 0 and T does not commit any writes before EOT (End Of Transaction).
- **Degree 2:** Degree 1 and T does not read dirty data of other transactions.
- **Degree 3:** Degree 2 and other transactions do not dirty any data read by T before T completes.

In a system using locking for concurrency control there is a tradeoff between the simplicity of the lock protocol and the degree of concurrency provided. A disctinction is made between read (shared) and write (exclusive) locks. This allows several concurrent read-only transactions to access the same data, while exclusive access is needed for writing.

The granularity (i.e. size) of lockable objects is also of concern. If the lockable unit is the whole database only one transaction is allowed to write at any time, and this

	NL	IS	IX	S	SIX	X
NL	✓	✓	✓	✓	✓	✓
IS	✓	✓	✓	✓	✓	-
IX	✓	✓	✓	-	-	-
S	✓	✓	-	✓	-	-
SIX	✓	✓	-	-	-	-
X	✓	-	-	-	-	-

Table 2.1: Lock mode compatibilities as given in [25].

will also block read-only transactions. Thus a locking protocol defines a hierarchy of lockable units of different granularities.

For example, a simple lock protocol might have the whole database, a table and a single record as the lockable units. In a real system this gets more complicated as files and disk blocks need to be lockable, and it might also be desirable to lock parts of a file or value ranges in a table.

The hierarchy of lockable units might be represented as a directed acyclic graph (DAG). The locking protocol must ensure that if a lock is obtained for a node in the graph then access to all descendants of that node is obtained with the same lock mode. E.g. if an exclusive lock is obtained on a table the transaction implicitly has locked all the rows of the table in exclusive mode.

Intention locks are introduced to indicate that a transaction intends to lock a unit at a lower level in the hierarchy. In order to obtain a lock on a node all parents of that node must be locked in the corresponding intention mode. This allows the locking protocol to more efficiently (i.e. at a higher level in the hierarchy) determine if a lock request can be granted.

Gray's paper on granularity of locks [25] defines six lock modes: NL (no lock), IS (intention share), IX (intention exclusive), S (share), SIX (share intention exclusive) and X (exclusive). The compatibilities of these access modes are given in Table 2.1.

2.2 Apache Derby

Derby [2] is a relational database management system implemented in Java, and is intended to be able to run wherever a Java Virtual Machine (JVM) is available. It supports a relatively large subset of the SQL-99 and SQL-2003 standards [49] while still maintaining a relatively small code footprint of 2.1 megabytes for the core Derby JAR file. A few hundred additional kilobytes are required to add network client/server functionality.

2.2.1 History and background

Derby was born in 1996 when Cloudscape Inc. started to develop a database system in Java, and the first release was in 1997. It was then known as JBMS, and was later

renamed to Cloudscape. Cloudscape was acquired by Informix Software in 1999. IBM then acquired the database assets of Informix Software in 2001, and Cloudscape was renamed IBM Cloudscape.

In 2004, IBM donated the code to the Apache Software Foundation [5], and it became the open source project Apache Derby, a part of the Apache DB project. Derby is still marketed by IBM as Cloudscape [29]. Sun Microsystems is also involved in the development of Derby and releases it under the name Java DB [52].

2.2.2 Overview

Derby can be used in both an embedded and a client/server configuration. In embedded mode the database engine and client application run in the same JVM. A conceptual diagram of Derby in embedded mode is shown in Figure 2.1. In client/server mode the application and Derby client can run in a different JVM and access the Derby server across a TCP/IP network, as illustrated in Figure 2.2. The interface between the Derby engine and the client application is JDBC [32]. The Derby development team strives to maintain strict adherence to the SQL standards.

2.2.3 Derby internals and architecture

The Derby project strives to adhere to best practices and standards in many ways. This starts with the developer interfaces - the Derby developers are careful to adhere to the JDBC [32] and SQL2003 [31] standards in the developer-visible interfaces. However, this approach extends beyond interfaces and into the Derby engine itself.

Derby's client/server architecture and communications protocol is based on the Distributed Relational Database Architecture (DRDA) standard [13] published by the Open Group [41]. DRDA originates from IBM, and is also used for networking in IBMs DB2 RDBMS [36]. Considering the IBM/Cloudscape heritage of Derby, it comes as no surprise that DRDA is used.

Logging and recovery [18] is based on ARIES [37], with some minor modifications described in [18]. This implies that transactional concurrency control in Derby is based on locking, as opposed to approaches like multi-version concurrency control [8].

Derby has a relatively clean architecture, with well separated interfaces and implementations and a modular design. This will be discussed in the next section.

2.2.4 Overall architecture

Derby is built in a modular, service-oriented fashion. A module or set of modules provide a specific service, e.g. cache management or locking. Derby is also designed to have a reasonably clean separation between modules' interface definitions and implementations of those interfaces. In the source code, this is evident from the separation of the packages into `org.apache.derby.iapi.*`, which contain the interface definitions for Derby's internal API, and `org.apache.derby.impl.*`, which contain implementations of the modules defined in the internal API.

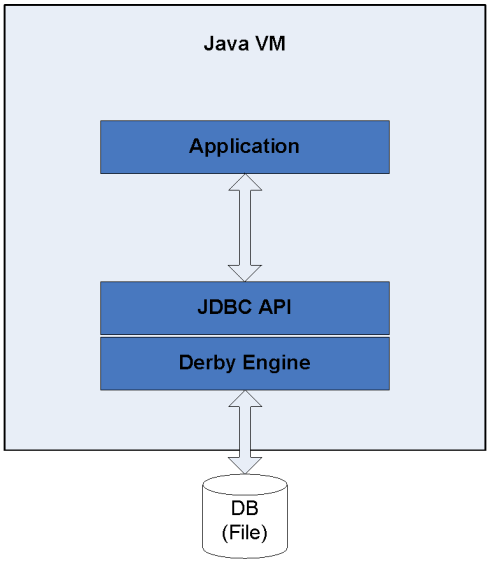

Figure 2.1: Derby Embedded

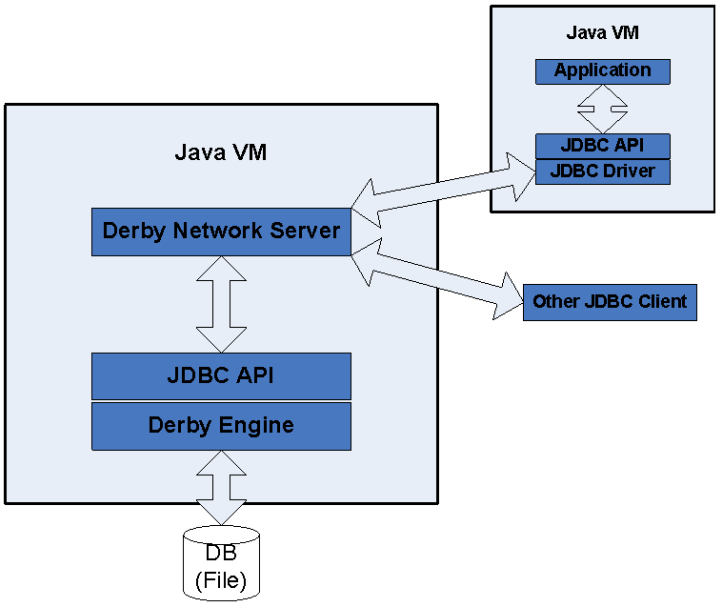

Figure 2.2: Derby Client/Server

When a part of Derby requires a service, e.g., it needs a buffer cache, it will request access to this service from the central Monitor subsystem. The Monitor subsystem will find and activate an implementation of the service interface to provide the functionality as requested. Figure 2.3 illustrates the overall architecture of Derby.

This enables Derby to have multiple independent implementations of an interface and choose the appropriate implementation when it is booted. This is used e.g. to enable

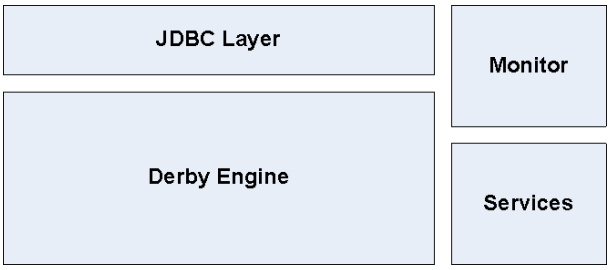

Figure 2.3: Derby Architecture

support for the appropriate version of JDBC depending on the runtime version of the JVM Derby is running in. For most services, however, Derby has only one implementation. Derby supports Java 1.3.x and higher versions.

2.2.5 Notable subsystems in Derby

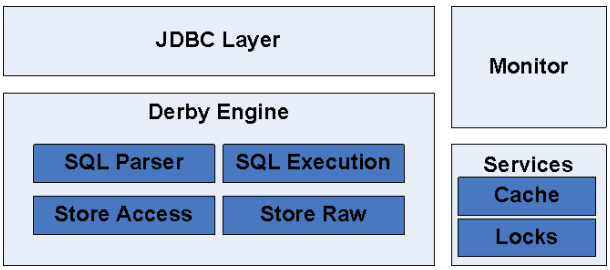

Figure 2.4: Derby Subsystems

Some of the subsystems in Derby are illustrated in Figure 2.4. As previously mentioned, Derby's application interface is JDBC, which is provided by the JDBC driver layer in `org.apache.derby.jdbc.*`. In the embedded case, i.e. no networking between client and server, this is a pretty thin layer, and most of the work on parsing and executing SQL is passed on to the the SQL Execution subsystem.

Derby executes SQL by parsing the incoming query, building an optimized query plan, and compiling that query plan into a Java class which is then dynamically loaded and called to execute the query. Most of this work is carried out by services described in `org.apache.derby.iapi.sql.*` and implemented in `org.apache.derby.impl.sql.*`.

The compiled execution plan class utilizes the services of the SQL execution component `org.apache.derby.impl.sql.execute`, which in turn uses other components of Derby to access data files and indexes, manage transactions, perform locking, cache data, et cetera.

2.2.6 Lock management

Of particular interest to us, as will become evident later in this report, is the lock management subsystem. It is specified by the interfaces in `org.apache.derby.iapi.services.locks` package, more specifically the `LockFactory` interface. It specifies an

interface for client to request and release locks on objects implementing the `Lockable` interface.

These locks are assigned to a compatibility space, which is a somewhat abstract concept roughly defined as the the space in which two equal locks on the same object are compatible. An example of such a compatility space is a transaction. Two different parts of Derby may request the same locks on behalf of the same transaction during the transaction's life time, and those locks will be granted. We will see later that this happens quite often.

Implementation

The current implementation of the `LockFactory` interface, `org.apache.derby.impl.services.locks.SinglePool` maintains, as its name implies, a pool of locks maintained in a single `LockSet`. `LockSet` is an extended `HashTable`, keyed by `Lockable`s.

`SinglePool` extends `HashTable` as well, to store `LockSpace` objects keyed by the generic objects used as compatibility spaces. Only one object each of `SinglePool` and `LockSet` is created. This requires all lock requests to go through common synchronization points – all lock requests are serialized. The impact of this will be investigated later.

Lock types

Data locks have a scope of either table, row or range (multiple rows). Escalation of multiple row locks to a table-level lock for performance reasons is done automatically, and the escalation threshold is a tunable parameter. Derby provides shared, update and exclusive lock types. Compatibility between lock types are shown in Table 2.2.

The actual implementation is a bit more complicated than this. Locking is handled differently for containers (a table or index structure) and rows. For containers we are doing hierarchical locking, so intention locks are used. The lock types as defined in `org.apache.derby.iapi.store.raw.ContainerLock` are CIS (Container Intent Shared), CIX (Container Intent Exclusive), CS (Container Shared), CU (Container Update) and CX (Container Exclusive). The compatibility of container locks are given in Table 2.3.

For row locking we need to consider different isolation levels. There are separate locks types for level 3 (serializable) and for level 2 and lower isolation levels. The lock types, as defined in `org.apache.derby.iapi.store.raw.RowLock`, are RS2 (Row Shared lock for level 2 and below), RS3 (Row Shared lock for level 3), RU2 (Row Update level 2 and below), RU3 (Row Update level 3), RIP (Row Insert Previous key), RI (Row Insert), RX2 (Row Exclusive level 2 and lower) and RX3 (Row Exclusive level 3). The resulting compatibility matrix is shown in Table 2.4.

For further information on the lock management system the reader is encouraged to take a look at the JavaDoc and comments in the Derby source code. The Derby JavaDoc is available online [3].

	Held		
Request	Shared	Update	Exclusive
Shared	✓	-	-
Update	✓	-	-
Exclusive	-	-	-

Table 2.2: Simple lock compatibility matrix

	Held				
Request	CIS	CIX	CS	CU	CX
CIS	✓	✓	✓	-	-
CIX	✓	✓	-	-	-
CS	✓	-	✓	-	-
CU	-	-	✓	-	-
CX	-	-	-	-	-

Table 2.3: Container lock compatibility matrix

	Held							
Request	RS2	RS3	RU2	RU3	RIP	RI	RX2	RX3
RS2	✓	✓	✓	✓	✓	-	-	-
RS3	✓	✓	✓	✓	-	-	-	-
RU2	✓	✓	-	-	✓	-	-	-
RU3	✓	✓	-	-	-	-	-	-
RIP	✓	-	✓	-	✓	✓	✓	-
RI	-	-	-	-	✓	-	-	-
RX2	-	-	-	-	✓	-	-	-
RX3	-	-	-	-	-	-	-	-

Table 2.4: Row lock compatibility matrix

2.2.7 Storage

The storage subsystem in Derby is composed of the access and raw layers. The access layer provides an interface for accessing conglomerates (tables or indexes) and rows. Locking is performed at the access level as described in Section 2.2.6.

Indexes in Derby are provided by a B-Tree [12] implementation. The B-Tree index is implemented in the `org.apache.derby.impl.store.access.btree` package. This is a standard B^+-Tree where all pointers are stored in the leaf nodes, as described in Section 2.1.1.

Concurrent access to the index is supported using the locking system and page latches. Deadlocks are prevented by always requesting latches in top–down and left–to–right order in the B-Tree, and holding at most two page latches at any time. When traversing the B-Tree and holding a latch on a parent and a child node, the parent node has to be released before a latch on a subchild can be acquired. For more information on the B-Tree implementation in Derby we refer to [16].

The raw storage layer provides page based storage in files to the access layer. Latches are used to ensure mutual exclusion at the raw page level. This is implemented in `org.apache.derby.impl.store.raw.data.BasePage`, which uses the locking services to handle latches. Latches are actually maintained in the same synchronized hash table as locks by `org.apache.derby.impl.services.locks.SinglePool`.

Contrary to traditional assumptions, obtaining or releasing a latch therefore incurs overhead comparable to that of obtaining or releasing a lock. The traditional assumption is that latches are about an order of magnitude cheaper than locks [37]. Latches in Derby are exclusive, there are no shared (read-only) latches.

2.2.8 Cache management

Caching in Derby is defined by the `org.apache.derby.iapi.services.cache.*` interfaces, and implemented by the `org.apache.derby.impl.services.cache.*` classes. The main class is the `Clock` class, which implements a cache with a replacement policy based on the "CLOCK" algorithm. A description of the CLOCK policy can be found in [9]. This implementation also maintains serialization of all accesses to the cache itself. Objects retrieved from the cache are independent from it, and their thread safety must be considered separately – accessing a cached object does not require locking the cache.

2.3 Derby performance

In this section we will present and comment on some of the benchmarks and performance analyses of Derby. We are aware of some work that has been done to analyze and improve the performance of Derby. Some general performance tips for Derby can be found in the Tuning Derby [55] guide.

2.3.1 Benchmarking by Sun Microsystems

Sun presented a performance evaluation [4] of Derby at ApacheCon [6] US 2005. Performance tips are provided both for system configuration and tuning of Derby parameters. The importance of using prepared statements and writing efficient SQL is also stressed.

The second part of the presentation provides benchmarks of Derby (Embedded and Client/Server) against two other popular open source DMBSs, MySQL [39] and PostgreSQL [47]. The benchmarks were run for different workloads (TPC-B [54] like and `SELECT`-only) for a small main-memory database and for a larger database on disk. Note that the database systems were not configured and tuned for performance, this should not be considered a proper benchmark.

In this test Derby, both embedded and client/server, outperforms MySQL and PostgreSQL for TPC-B-like load with the larger on-disk database. MySQL wins the main-memory database test, while PostgreSQL performs poorly for both of the update intensive benchmarks. For the single record `SELECT` test Derby is outperformed by both MySQL and PostgreSQL for the main-memory database, but only by PostgreSQL for the larger database. For the main-memory database there is a considerable difference in throughput between Derby embedded and client/server. This is attributed to high CPU usage in the network server and to network overhead. Derby sends twice the number of packets per transaction compared to MySQL and PostgreSQL for the `SELECT` load.

2.3.2 PolePosition benchmarks

PolePosition [45] is an open source database benchmark. Results are published for several open source DBMS and ORM[3] systems, including Derby. Results for commercial database systems are not available because licenses generally disallow the publication of benchmark results.

The benchmark is composed of several tests, called circuits, that represent different workloads. For example, the "Bahrain" circuit is composed of write, query, update and delete operations on flat objects, while the "Imola" circuit is a simple read benchmark that retrieves objects by ID. Results for the benchmarks are published at [46].

The validity of the PolePosition benchmarks has been questioned. The systems tested have varying transactional support and ACID compliance. Derby's full support for transactions, with default isolation level "Read Committed", has a performance impact. Looking at the source code for the JDBC benchmarks in PolePosition also reveals that some of the tests are not exclusively using prepared statements. Executing regular statements, instead of prepared statements, will impact the performance of Derby a lot.

[3]Object Relational Mapping - interfaces object oriented systems to relational data management

2.3.3 DERBY-1704

DERBY-1704 is an issue in the Derby project's issue tracker, containing a patch to the Derby lock system to eliminate some global synchronization points. This is achieved by partitioning each of the `SinglePool` and `LockSet` hash tables into 16 different hash tables based on the hash code of the `Lockable` object. The patch and benchmark results can be found in the Apache issue tracking system JIRA [15].

The benchmarks show no negative effects from the patch on a single CPU system. For multithreading (number of clients greater than one) on SMP systems there is significant improvement. It is apparent that the global synchronization in the locking system has an impact on the performance of Derby.

We will take a closer look at the DERBY-1704 patch with DTrace.

2.3.4 Berkeley DB Java Edition and Derby benchmarks by Oracle

Oracle recently published a paper [43] with benchmarks of Oracle Berkeley DB Java Edition and Apache Derby. Berkeley DB [42] Java Edition is a Java version of the Berkeley DB embedded database.

The original Berkeley DB was developed at U.C. Berkeley as part of the BSD project, then by Sleepycat Software. Sleepycat Software was acquired by Oracle in February 2006. Berkeley DB is released under a dual licensing scheme: It is available under an open source license (the Sleepycat Public License) for use in open source products, and under a commercial license which can be bought from Oracle. For more information about Berkeley DB and its history see [7].

Berkeley DB is a lightweight database. It provides only simple key/value record storage with a persistence system for Java objects layered on top. There is no support for SQL or other query languages, only programmatic queries, and it does not support a client/server architecture. It does, however, support ACID transactions. Because there is no overhead for an SQL layer and table structures it is a reasonable expectation that Berkeley DB Java Edition will be faster than Derby in most or all cases.

The paper presents benchmarks for object persistence, using the Direct Persistent Layer (DPL) for Berkeley DB Java Edition and Hibernate [27] for ORM with Derby. An ORM solution using the JDBC API of Derby can of course be expected to add even more overhead compared to Berkeley DB's internal DPL. Most of the benchmarks are also run with the disk write cache turned on, thus sacrificing durability.

It is indeed, as pointed out in the introduction to the paper, an "apples-to-oranges" comparison. The question of "Berkeley DB Java Edition vs. Apache Derby" is probably not about performance, but more likely about the feature set needed for the application. Berkeley DB Java Edition outperforms Derby, but has a minimal feature set. A critique of the paper by one of the Derby developers can be found at [14], including a response from one of the Berkeley DB Java Edition developers.

Chapter 3

Our challenge

Our challenge is to analyze the performance and scalability of Derby for a read-only workload. Our goal is to identify bottlenecks and problems that can cause the observed lack of scalability, and suggest possible changes that can improve the scalability of Derby. If time and resources permit, we may also test proof-of-concept implementations of such changes and evaluate their impact.

This chapter describes the test workload for our benchmarks, the test systems and the tools we have used for the project.

3.1 The workload

For our benchmarks we create and populate a database, and then stress it with a read-only workload.

A simple schema with ∼100 byte rows:

```
CREATE TABLE FOO (
a INTEGER NOT NULL,
b CHAR(96) NOT NULL,
PRIMARY KEY(a) )
```

We use a default data set of 100 000 rows, with values of `a` from 0 to 99999. The primary testing load is N threads doing `SELECT a FROM foo WHERE a = ?; COMMIT`, with values of `a` randomly chosen from the interval in the table, in a tight loop.

3.1.1 Workload implementation

The benchmark program, called *selectload*, was implemented in two parts. A main class, `Select`, is responsible for initializing the database (creating and populating it with data) and coordinating the Worker threads and overall test execution, and `Worker`, which extends `java.lang.Thread` and is, naturally, our worker thread.

The main class runs a benchmark by first initializing a database (if requested), then warming up the Derby system by first selecting the whole database to fill the cache and running a Worker thread for the specified number of "warmup" transactions, and then

finally running the requested number of real Worker threads executing the specified number of transactions. Timing information and statistics are collected and aggregated when all worker threads have finished.

The source code can be found on the enclosed CD-ROM.

3.1.2 Variations for lock tracing

To find out how locks are set for `UPDATE` and `DELETE` queries we used a slightly modified version of our *selectload* program. `SELECT a FROM foo WHERE a = ?` is simply substituted for `UPDATE FOO SET b = ? WHERE a = ?` and `DELETE FROM FOO WHERE a = ?` respectively.

3.2 Test systems

For our work on this project we have used different systems to experiment with Derby and run benchmarks.

3.2.1 Workstations

Our workstations, as provided by IDI, are 3.0GHz P4 machines with 1GB of main memory. We chose Ubuntu Linux as the operating system for our workstations and have been running and building Derby under Java versions 1.3, 1.4, 1.5 and 1.6 (beta) on Linux.

3.2.2 4-way Sun Enterprise

IDI has also provided us access to a Sun Enterprise 450, *lind.idi.ntnu.no*, for SMP benchmarks. This is a 4-way system with 64 bit UltraSPARC-II CPUs and 4GB of main memory running Solaris 10. The system has a faulty memory module, so performance is afflicted by frequent traps when memory errors are corrected by ECC. We assume this to have a similar effect on all benchmarks, so that the relative numbers are representative even though performance may not be on par with the full potential of the machine.

We have used Java version 1.5 and 1.6 (beta) for our tests on Solaris 10. Java 1.6 was needed for the DTrace tests requiring extended probes.

3.2.3 Sun Fire V490

We borrowed some time on a V490 from ITEA, the IT-department at NTNU. This is a 4-CPU system with dual core UltraSPARC IV CPUs, which means it can execute 8 simultaneous threads, and 16 GB main memory. This system was also running Solaris 10.

3.3 Environment and tools

We have done all our tests using Sun's Java implementation. Building Derby requires JDK 1.3 and 1.4, and optionally JDK 1.6 for JDBC 4.0 support, and some additional libraries and utilities for testing and SQL parser generation.

We have been running Derby under the 1.5 JVM, which is the latest stable release version, and under the 1.6 beta version when support for extended DTrace probes was required.

3.3.1 IntelliJ IDEA

IntelliJ IDEA [30] is a commercial Java IDE by JetBrains. It includes a Java editor with "smart" code completion and many other advanced features. We have used the debugger in IntelliJ to analyze the inner workings of Derby.

The debugger allows us to attach to a JVM running Derby, pause at different types of breakpoints, step through execution and explore the full stack trace and variable values at any point during execution. We used the time limited demo version of IntelliJ for this project.

3.3.2 NetBeans

NetBeans [40] is an open source Java IDE developed by Sun Microsystems. The NetBeans IDE is built on the NetBeans Platform, which is a framework for Java desktop application development. NetBeans provides an advanced profiler that has been useful for us in this project.

The profiler can be attached to a running JVM and allows profiling of CPU and memory usage, and also threads profiling which reveals time spent in different states during thread lifetime. The profiler is low overhead, only the instrumented subset of the application code is affected by profiling.

3.3.3 DTrace

DTrace[1] is a tracing framework built into the Solaris 10[2] operating system. It allows real time instrumentation of both the operating system kernel and user applications. This can be used for tracking down problems and analyzing performance.

DTrace allows the user (administrator) to write programs for instrumentation that are executed when *probes* are triggered. There are probes for almost everything that can happen in a Solaris system, e.g. operating system events and system calls.

Additional probes may also be provided, as done by the Java "hotspot" provider.

[1] DTrace is short for Dynamic Tracing

[2] DTrace is released as part of Solaris 10 under the open source CDDL license and is currently being ported to the FreeBSD operating system. It will also be included in Mac OS X 10.5 "Leopard", which is based on the FreeBSD kernel.

DTrace adds no overhead if no DTrace programs are running, and relatively little overhead when probes are triggered.

DTrace programs are written in a C-like language called D. Because DTrace programs run in kernel space this language is restricted so that stability and security is not compromised.

A DTrace program contains clauses that enable probes and statements to be performed when the probes *fire*. Identification of the process or thread ID that triggered a probe is possible, and this allows detailed statistics to be produced per process or thread.

Chapter 4

Results

In this chapter we present the results of our work. We have looked at the internals of the locking system, and at how Derby performs for the simple workload described in Section 3.1. Scalability properties will be observed by varying the number of concurrent threads as well as the number of CPUs available to Derby.

The simplest metric available to us, but still a very useful one, is simply the time required for our benchmark application to complete the requested number of single tuple read transactions. The throughput in terms of transactions per second is measured across all the worker threads in the benchmark. We will first look at throughput on 1, 2, 4 and 8 threads on an 8-way system measured with our *Selectload* benchmark.

We then present some results on thread execution and Java-level locks measured with our own DTrace scripts. The source code for these scripts is included in Appendix A for reference.

We have also used DTrace to study the effects of the DERBY-1704 patch on locking and monitor contention.

This is followed by a detailed examination of the lock system. We have used a debugger to obtain execution traces for single-row `SELECT`, `UPDATE` and `DELETE` transactions.

4.1 Throughput

Figure 4.1 shows a very basic measurement of how total throughput develops as we increase concurrency in Derby. The numbers from the raw measurements are shown in table 4.1. The measurements were done on the 8-core system, so hardware should not be a limiting factor[1]. The benchmarks were done with our *Selectload* program only, so there should be no extra overhead added by instrumentation except for the gathering of timing/statistics in *Selectload* which should have minimal, if any, impact on performance. Note that the numbers are not per-thread, they are the total for all threads.

[1] I.e. there is one core available for each thread.

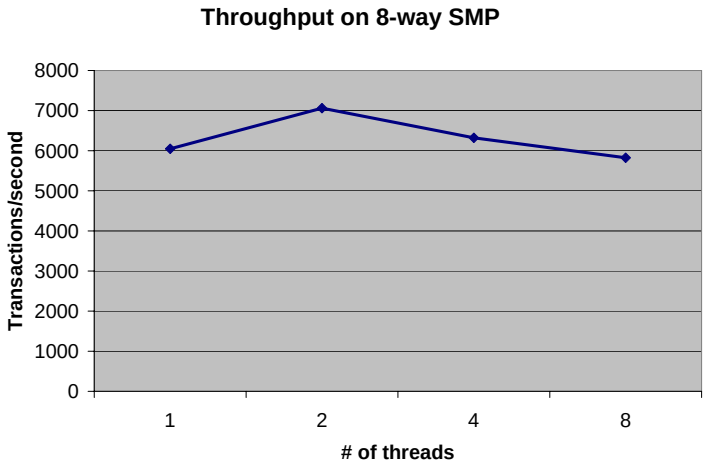

Figure 4.1: Throughput development as more threads are added

Workers	1	2	4	8
Txns/sec	6048	7061	6322	5824

Table 4.1: Throughput in transactions per second as more threads are added

4.2 Thread execution state

For detailed measurements, we have used the Dynamic Tracing Framework (DTrace) [20] available in Solaris 10 [48] from Sun Microsystems.

We have used DTrace to explore how the worker threads spend their lifetime by inserting probes into the OS scheduler and timing each thread as it goes on and off a CPU, and noting the reason for ejection from the CPU when it goes offline.

We only monitored the worker threads in our test application, we did not concern ourselves with any system threads in the JVM or any applications outside the JVM. The DTrace script used for these measurements can be found in Listing A.1.

As we can see in figure 4.2, the amount of time spent waiting for user-level locks[2] increases dramatically as concurrency increases. A certain increase in lock contention is natural as concurrency increases, and is to be excpected, but it should be as small as possible.

The time spent waiting for user-level locks when running with a single thread can probably be attributed to Java-level overhead such as stop-the-world garbage collection, described in [28], and coordination with threads other than the worker thread itself. We will not go into further detail on this, as we are primarily interested in wait-

[2]Locks in user-level code, as opposed to locks in the operating system kernel.

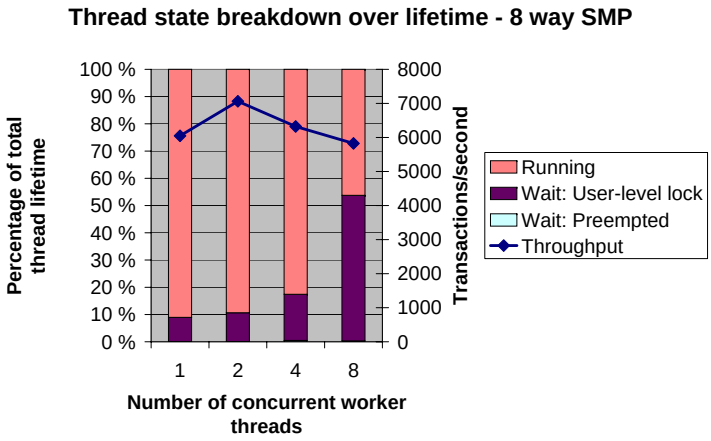

Figure 4.2: How threads spend their time in different concurrency levels

ing time development as concurrency increases.

4.3 Java-level lock investigations

Another useful tool provided by DTrace and JDK 1.6 is the ability to instrument Java-level locking events such as lock contention in monitors and looking at Object.wait() / Object.notify() events. These probes are enabled by the use of the `-XX:+ExtendedDTraceProbes` option for the JVM and are documented in [21].

We have used these probes to collect timing information for monitor contention and wait()/notify() events, and also to record the types of objects that caused the delays. A dtrace script capable of collecting this is included in listing A.1.

We performed timing measurements in four runs with 1, 2, 4 and 8 worker threads, where each thread performed 400 000, 200 000, 100 000 and 50 000 transactions, respectively for a total of 400 000 transactions for each measurement.

Table 4.2 lists the essential results of these measurements. We can see that both wait events and contention events increase dramatically in both frequency and time consumption as concurrency increases. Another interesting observation is that the amout of time spent on wait/notify events increases significantly compared to time spent on monitor contention as we go from 4 to 8 threads.

DTrace can also tell us which classes are involved in the wait/notify and contention events. We see in table 4.3 that only one class is involved in significant wait events, and that the total time consumed by calls to wait() is much larger than the time consumed by waiting for contended monitors.

Workers	1	2	4	8
Txns/sec	5834	6148	2157	2004
Object waits	0	4927	310384	617023
Object wait time (msec)	0	1080	63846	730705
Monitor contention events	0	81383	1745499	1919993
Monitor contention time (msec)	0	3156	187053	340602
Wait time / contention time	-	0,34	0,34	2,14

Table 4.2: Adding up total thread time spent waiting for object notifications and contended monitors.

Wait type	**Class**	**Events**	**Wait time (ms)**
wait/notify	o.a.derby.impl.services.locks.ActiveLock	617023	730705
contention	o.a.derby.impl.services.locks.LockSet	1901327	338690
contention	java.io.PrintStream	6	1645
contention	o.a.derby.impl.services.locks.SinglePool	14771	215
contention	o.a.derby.impl.services.cache.Clock	2274	40
contention	o.a.derby.impl.services.locks.ActiveLock	1609	9
contention	o.a.derby.impl.store.raw.data.StoredPage	4	0
contention	o.a.derby.impl.sql.GenericPreparedStatement	1	0
contention	o.a.derby.impl.store.raw.data.AllocationCache	1	0

Table 4.3: Time spent waiting for contended monitors and explicit wait()/notify() for various classes in the 8-thread run in table 4.2.

4.4 DTrace measurements for the DERBY-1704 patch

We have also used DTrace with our monitor contention script to investigate the DERBY-1704 [15] patch that splits the hash tables in the lock manager to remove the global syncrhonization points. The benchmarks attached in the issue tracker at [15] show a significant increase in throughput with this patch. We want to investigate how it compares with "plain vanilla" Derby when it comes to monitor contention.

The measurements were taken with our DTrace monitor contention program. The tests were run on a build of the latest 10.2 version of Derby from the source repository[3] and on a build of the same code with the patch applied. The tests were run on the 4 CPU Sun system. The test parameters used was a 100.000 row database, 8 worker threads and 10.000 transactions per thread.

We present summarized results here, the full results are included in the Appendix, Section B.2.

4.4.1 DTrace results for "vanilla" Derby 10.2

Here we presents results of our DTrace measurements of a "vanilla" Derby 10.2. Table 4.4 shows the top 5 blocking objects by block count.

[3]In mid September, the first official stable release of Derby 10.2 was 6 October 2006.

Class name	Events
org.apache.derby.impl.services.locks.LockSet	93754
java.util.Hashtable	22209
org.apache.derby.impl.services.cache.Clock	12681
org.apache.derby.impl.services.locks.SinglePool	2948
org.apache.derby.impl.services.cache.CachedItem	607

Table 4.4: "Vanilla" 10.2: Top blocking objects, by count

Class name	Wait time (ms)
org.apache.derby.impl.services.locks.LockSet	139154
java.util.Hashtable	27144
org.apache.derby.impl.services.cache.Clock	22726
org.apache.derby.impl.services.locks.SinglePool	7262
org.apache.derby.impl.store.raw.xact.TransactionTable	1879

Table 4.5: "Vanilla" 10.2: Top blocking objects, by total wait time

Table 4.5 shows the top 5 blocking objects by total wait time.

Table 4.6 shows the top 5 blocking methods by count. Note that `lockObject()` appears twice. This is not an error, there are different methods with the same name, but different method signatures.

Method name	Events
org.apache.derby.impl.services.locks.LockSet.lockOject()	40856
org.apache.derby.impl.services.locks.LockSet.unlock()	26401
java.util.HashTable.get()	20863
org.apache.derby.impl.services.locks.LockSet.lockOject()	8713
org.apache.derby.impl.services.cache.Clock.release()	4767

Table 4.6: "Vanilla" 10.2: Top blocking methods, by count

4.4.2 DTrace results for the DERBY-1704 patch

These are the results of our DTrace measurements for the same benchmark, but against Derby 10.2 with the preliminary patch for "DERBY-1704" applied. Table 4.7 shows the top 5 blocking objects by block count.

Table 4.8 shows the top 5 blocking objects by total wait time. Note the appearance of `java.io.PrintStream`, this is probably caused by contention when our benchmark program writes information to standard out.

Table 4.9 shows the top 5 blocking methods, by count. Note that `LockSet.lockOject()` and `LockSet.unlock()` is in the top 5 for wait time, but not for block count.

Class name	Events
java.util.HashTable	49441
org.apache.derby.impl.services.cache.Clock	22619
org.apache.derby.impl.store.raw.xact.TransactionTable	1075
org.apache.derby.impl.services.cache.CachedItem	1020
org.apache.derby.impl.store.raw.data.AllocationCache	521

Table 4.7: "1704" 10.2: Top blocking objects, by count

Class name	Wait time (ms)
java.util.HashTable	78386
org.apache.derby.impl.services.cache.Clock	46187
org.apache.derby.impl.store.raw.xact.TransactionTable	2759
java.io.PrintStream	2012
org.apache.derby.impl.services.cache.CachedItem	1363

Table 4.8: "1704" 10.2: Top blocking objects, by total wait time

Class name	Events
java.util.HashTable.get()	34236
org.apache.derby.impl.services.cache.Clock.release()	8525
org.apache.derby.impl.services.cache.Clock.find()	8184
org.apache.derby.impl.services.locks.LockSet.lockOject()	4390
org.apache.derby.impl.services.locks.LockSet.unlock()	3779

Table 4.9: "1704" 10.2: Top blocking methods, by count

4.5 How is the lock subsystem used?

After collecting some evidence that indicates the locking subsystem is a good candidate for further investigation, we proceeded to collect data on how the locking subsystem is used and what it is doing to cause these problems. For this job, tools such as print statements and a plain old debugger with breakpoints and the ability to inspect the call stack sufficed.

As explained earlier, locking in Derby is handled by the `org.apache.derby.impl.services.locks.SinglePool` class. This class implements the `org.apache.derby.iapi.services.locks.LockFactory` interface and extends `java.util.HashTable` to provide a single synchronization point for all lock requests and a synchronized lock table for the system. This is used for all locks and lock requests in Derby, including page latches.

We modified the `SinglePool` class to log all lock requests if a debug flag is set and ran tests for simple `SELECT`, `UPDATE` and `DELETE` transactions. Setting break points in the trace method and connecting the IntelliJ debugger to the JVM running Derby allowed us to explore the call stack and execution state on all entries into the lock subsystem.

Traces showing how single-row `SELECT`, `UPDATE` and `DELETE` transactions use the lock subsystem follow. An analysis of these traces will be presented in Chapter 5.

4.5.1 Lock actions in a single-row SELECT transaction

This is a trace of the calls into `SinglePool` for a transaction executing a single record `SELECT` on the primary key. Each trace line contains first the method calling the lock system, then the `LockFactory` method called, then the essential parameters to that method, such as the lock mode desired and the identity of the object to be locked.

A line by line explanation of the trace will be given. The trace starts when the prepared statement is executed, thus we are only seeing the calls resulting from the actual `SELECT` query.

The transaction isolation level used while performing this trace was "READ COMMITTED". For brevity we have omitted the full stack traces, we include only the method calling into the lock manager (and the position of the call in the source file), the method in the `SinglePool` being called, and some essential parameters to the call that help explain what is being done.

This is a simple lookup on the index primary key value only. Note that there is no access to the actual data container, we are only searching for the value in the B-Tree. For a `SELECT *` or `UPDATE` query this would be different – we would then do locking also for the data records and latching of the corresponding disk pages.

The SELECT lock trace

1. `org.apache.derby.impl.store.raw.xact.RowLocking2.lockContainer(RowLocking2↩.java:104): lockObject (no latch)(type 'IS')Container(0, 1024)`

   During index lookup an IS (Intention Shared) lock is set on the container (file)

holding the B-Tree structure. The `RowLocking2` class represents the isolation level (read committed) which is used in the transaction.

- Synchronization-wise, this visits the `LockSet` monitor once, then the `SinglePool` monitor once.

2. `org.apache.derby.impl.store.raw.xact.RowLocking2.lockContainer(RowLocking2↩ .java:119): isLockHeld (type: 'X')Container(0, 1024)`

   After setting an IS lock on the container a check is performed to see if the transaction already has an exclusive lock on the container – if so the IS lock can be dropped at once.

   - This visits the `SinglePool` monitor once.

3. `org.apache.derby.impl.store.raw.xact.RowLocking2.lockContainer(RowLocking2↩ .java:125): isLockHeld (type: 'S')Container(0, 1024)`

   A corresponding check to see if the transaction already holds a shared lock for the container.

   - This visits the `SinglePool` monitor once.

4. `org.apache.derby.impl.store.raw.data.BasePage.setExclusive(BasePage.java↩ :1781): latched. qual: (BaseContainerHandle:(Container(0, 1041)))ref:↩ Page(1,Container(0, 1041))`

   A latch is obtained for the page containing the root node of the B-Tree.

   - This visits the `LockSet` monitor once.

5. `org.apache.derby.impl.store.raw.data.BasePage.setExclusive(BasePage.java↩ :1781): latched. qual: (BaseContainerHandle:(Container(0, 1041)))ref:↩ Page(5,Container(0, 1041))`

   The next page to be examined in the B-Tree lookup is latched. This is a leaf node in the B-tree.

   - This visits the `LockSet` monitor once.

6. `org.apache.derby.impl.store.raw.data.BasePage.releaseExclusive(BasePage↩ .java:1903): unlatched Page(1,Container(0, 1041))`

   The latch on the B-tree root page is dropped as the latch on the next page to be examined has been obtained.

   - This visits the `LockSet` monitor once.

7. `org.apache.derby.impl.store.raw.xact.RowLocking2.lockRecordForRead(RowLocking2↩ .java:166): lockObject (no latch)(type 'S')Record id=1 Page(5,Container↩ (0, 1041))`

   A read lock is set on a single record in the B-tree, still during the B-Tree search.

   - This visits the `LockSet` monitor once, then the `SinglePool` monitor once.

8. `org.apache.derby.impl.store.raw.xact.RowLocking2.lockRecordForRead(RowLocking2↩ .java:166): lockObject (no latch)(type 'S')Record id=20 Page(26,Container↩ (0, 1024))`

Another record is locked for reading, but we note that this is a record in a different container. As the query should only return one row we presume that this is the record corresponding to the B-tree record locked above, only this is the record in the data container.

- This visits the `LockSet` monitor once, then the `SinglePool` monitor once.

9. `org.apache.derby.impl.store.raw.data.BasePage.releaseExclusive(BasePage↩ .java:1903): unlatched Page(5,Container(0, 1041))`

   The latch on the B-Tree leaf page is released.

   - This visits the `LockSet` monitor once.

10. `org.apache.derby.impl.store.raw.xact.RowLocking2.unlockRecordAfterRead↩ (RowLocking2.java:216): unlock (type 'S')Record id=1 Page(5,Container↩ (0, 1041))`

    The record in the B-tree is unlocked.

    - This visits the `SinglePool` monitor once, then the `LockSet` monitor once.

11. `org.apache.derby.impl.store.raw.xact.RowLocking2.unlockContainer(RowLocking2↩ .java:248): unlockGroup: BaseContainerHandle:(Container(0, 1024))`

    The `ResultSet` returned by the query to the application is being closed, and the shared lock on the container is dropped.

    - Visits the `LockSet` monitor once for every lock found in the group to be unlocked.

12. `org.apache.derby.impl.store.raw.xact.Xact.releaseAllLocks(Xact.java:1857)↩ : unlockGroup: 6548`

    The `ResultSet` is being closed, and as autocommit is enabled, this causes a commit. All locks are released during transaction commit.

    As the transaction has been committed, the following calls to the locking subsystem are not really a part of the traced transaction, but they are interesting nonetheless, so we continue our analysis:

    - Same synchronization as previous.

13. `org.apache.derby.impl.store.raw.xact.Xact.releaseAllLocks(Xact.java:1857)↩ : unlockGroup: 6548`

    Commiting and releasing all locks *again*. This is due to an explicit commit() call after the result set is closed with autocommit enabled. The fact that this is permitted is possibly a bug in Derby – doing an explicit commit() on a Connection object with autocommit enabled is certainly a bug in our benchmark application, at least.

    - Same synchronization as previous.

14. `org.apache.derby.impl.store.raw.xact.Xact.releaseAllLocks(Xact.java:1857)↩ : unlockGroup: 6548`

    The transaction object is still alive and releaseAllLocks is called another time when the connection is closed.

- Same synchronization as previous.

15. `org.apache.derby.impl.store.raw.xact.Xact.releaseAllLocks(Xact.java:1857)↩ : unlockGroup: 6548`

    And one final time, this is the result of an error arising when closing the connection. Abort is then performed on all transactions, even if this transaction has been committed.

    - Same synchronization as previous.

In this trace we can make several interesting observations, and these are discussed further in Section 5.5.1.

4.5.2 Lock actions in a single-row UPDATE transaction

A trace like the previous was performed for a single-row `UPDATE`. Relative to the previous trace we provide fewer comments, as many of the things that happen are very similar to what happened in the `SELECT` trace.

The different types of lock actions (`lockObject`, `isLockHeld`, etc) have the same synchronization properties in this trace as in the `SELECT` trace, so this information is omitted from the following traces.

While not all items are commented separately here, the next comment addresses all the items since the previous comment.

Section 5.5.2 discusses this trace further.

The UPDATE lock trace

1. `org.apache.derby.impl.store.raw.xact.RowLocking2.lockContainer(RowLocking2↩ .java:104): lockObject (no latch)(type 'IX')Container(0, 1024)`

    A `RowChanger` result set has been opened and the B-tree search is started by acquiring an IX lock on the data container.

2. `org.apache.derby.impl.store.raw.xact.RowLocking2.lockContainer(RowLocking2↩ .java:119): isLockHeld (type: 'X')Container(0, 1024)`

    And a check is performed to see if we already own an exclusive lock on the container.

3. `org.apache.derby.impl.store.raw.xact.RowLocking2.lockContainer(RowLocking2↩ .java:104): lockObject (no latch)(type 'IX')Container(0, 1024)`

4. `org.apache.derby.impl.store.raw.xact.RowLocking2.lockContainer(RowLocking2↩ .java:119): isLockHeld (type: 'X')Container(0, 1024)`

    The same two locking operations are performed again, this time caused by the opening of a `NormalizeResultSet`.

5. `org.apache.derby.impl.store.raw.data.BasePage.setExclusive(BasePage.java:1781): latched. qual: (BaseContainerHandle:(Container(0, 1041)))ref:Page(1,Container(0, 1041))`

   The root block in the B-tree is latched.

6. `org.apache.derby.impl.store.raw.data.BasePage.setExclusive(BasePage.java:1781): latched. qual: (BaseContainerHandle:(Container(0, 1041)))ref:Page(5,Container(0, 1041))`

   Then the next block in the B-tree is latched.

7. `org.apache.derby.impl.store.raw.data.BasePage.releaseExclusive(BasePage.java:1903): unlatched Page(1,Container(0, 1041))`

   The root block can then be unlatched.

8. `org.apache.derby.impl.store.raw.xact.RowLocking2.lockRecordForRead(RowLocking2.java:166): lockObject (no latch)(type 'S')Record id=1 Page(5,Container(0, 1041))`

   Getting a shared lock on the record in the B-tree.

9. `org.apache.derby.impl.store.raw.xact.RowLocking3.lockRecordForWrite(RowLocking3.java:278): lockObject (no latch)(type 'X')Record id=16 Page(27,Container(0, 1024))`

   Locking the row in the data container for writing. `RowLocking3` indicates isolation level 3 (serializable) for the transaction.

10. `org.apache.derby.impl.store.raw.data.BasePage.releaseExclusive(BasePage.java:1903): unlatched Page(5,Container(0, 1041))`

    The leaf node in the B-tree is unlatched.

11. `org.apache.derby.impl.store.raw.data.BasePage.setExclusive(BasePage.java:1781): latched. qual: (BaseContainerHandle:(Container(0, 1024)))ref:Page(27,Container(0, 1024))`

    And the data page is latched.

12. `org.apache.derby.impl.store.raw.xact.RowLocking3.lockRecordForWrite(RowLocking3.java:278): lockObject (no latch)(type 'X')Record id=16 Page(27,Container(0, 1024))`

    Another exclusive lock for the row to be updated.

13. `org.apache.derby.impl.store.raw.data.BasePage.releaseExclusive(BasePage.java:1903): unlatched Page(27,Container(0, 1024))`

    The data page is unlatched.

14. `org.apache.derby.impl.store.raw.data.BasePage.setExclusive(BasePage.java:1781): latched. qual: (BaseContainerHandle:(Container(0, 1024)))ref:Page(27,Container(0, 1024))`

    And latched again.

15. `org.apache.derby.impl.store.raw.xact.RowLocking3.lockRecordForWrite(RowLocking3.java:278): lockObject (no latch)(type 'X')Record id=16 Page(27,Container(0, 1024))`

    And again an exclusive lock is set for the data row.

16. `org.apache.derby.impl.store.raw.data.BasePage.releaseExclusive(BasePage.java:1903): unlatched Page(27,Container(0, 1024))`

    The data page is unlatched.

17. `org.apache.derby.impl.store.raw.xact.RowLocking2.unlockRecordAfterRead(RowLocking2.java:216): unlock (type 'S')Record id=1 Page(5,Container(0, 1041))`

    The shared lock on the B-tree row is released.

18. `org.apache.derby.impl.store.raw.xact.RowLocking2.unlockContainer(RowLocking2.java:248): unlockGroup: BaseContainerHandle:(Container(0, 1024))`

19. `org.apache.derby.impl.store.raw.xact.RowLocking2.unlockContainer(RowLocking2.java:248): unlockGroup: BaseContainerHandle:(Container(0, 1024))`

20. `org.apache.derby.impl.store.raw.xact.RowLocking2.unlockContainer(RowLocking2.java:248): unlockGroup: BaseContainerHandle:(Container(0, 1024))`

    All locks on the B-tree data container are released.

21. `org.apache.derby.impl.store.raw.xact.Xact.releaseAllLocks(Xact.java:1857): unlockGroup: 6751`

22. `org.apache.derby.impl.store.raw.xact.Xact.releaseAllLocks(Xact.java:1857): unlockGroup: 6751`

23. `org.apache.derby.impl.store.raw.xact.Xact.releaseAllLocks(Xact.java:1857): unlockGroup: 6751`

24. `org.apache.derby.impl.store.raw.xact.Xact.releaseAllLocks(Xact.java:1857): unlockGroup: 6751`

    All locks owned by the transaction are dropped, as for `SELECT` this is done 4 times.

4.5.3 Lock actions in a single-row DELETE transaction

This is a trace of the locking done by a single-row `DELETE`.

As in the previous trace, uncommented trace items are considered in the next comment, for brevity. We assume the reader has read the previous traces, and does not need all the details each time.

Section 5.5.3 discusses this trace further.

The DELETE lock trace

1. `org.apache.derby.impl.store.raw.xact.RowLocking2.lockContainer(RowLocking2↩ .java:104): lockObject w/o latch ContainerKey - IX, Container(0, 960)↩`

2. `org.apache.derby.impl.store.raw.xact.RowLocking2.lockContainer(RowLocking2↩ .java:119): isLockHeld ContainerKey - X, Container(0, 960)`

   An IX lock is set for the B-tree data container and we check if the transaction already owns an exclusive lock.

3. `org.apache.derby.impl.store.raw.xact.RowLocking2.lockContainer(RowLocking2↩ .java:104): lockObject w/o latch ContainerKey - IX, Container(0, 960)↩`

4. `org.apache.derby.impl.store.raw.xact.RowLocking2.lockContainer(RowLocking2↩ .java:119): isLockHeld ContainerKey - X, Container(0, 960)`

   The same is done again, because a different result set is opened.

5. `org.apache.derby.impl.store.raw.data.BasePage.setExclusive(BasePage.java↩ :1781): latched a StoredPage - page id Page(1,Container(0, 977))`

   The B-tree root node is latched.

6. `org.apache.derby.impl.store.raw.data.BasePage.setExclusive(BasePage.java↩ :1781): latched a StoredPage - page id Page(6,Container(0, 977))`

   And then the leaf node.

7. `org.apache.derby.impl.store.raw.data.BasePage.releaseExclusive(BasePage↩ .java:1903): unlatched StoredPage - page id Page(1,Container(0, 977))↩`

   And the root unlatched.

8. `org.apache.derby.impl.store.raw.xact.RowLocking2.lockRecordForRead(RowLocking2↩ .java:166): lockObject w/o latch RecordId - S, Record id=1 Page(6,Container↩ (0, 977))`

   A shared lock on the record in the B-tree is obtained.

9. `org.apache.derby.impl.store.raw.xact.RowLocking3.lockRecordForWrite(RowLocking3↩ .java:278): lockObject w/o latch RecordId - X, Record id=22 Page(32,Container↩ (0, 960))`

   An exclusive lock on the data record is obtained.

10. `org.apache.derby.impl.store.raw.data.BasePage.releaseExclusive(BasePage↩ .java:1903): unlatched StoredPage - page id Page(6,Container(0, 977))↩`

    The B-tree page is unlatched.

11. `org.apache.derby.impl.store.raw.data.BasePage.setExclusive(BasePage.java:1781): latched a StoredPage - page id Page(6,Container(0, 977))`

    And latched again.

12. `org.apache.derby.impl.store.raw.data.BasePage.releaseExclusive(BasePage.java:1903): unlatched StoredPage - page id Page(6,Container(0, 977))`

    And then unlatched...

13. `org.apache.derby.impl.store.raw.data.BasePage.setExclusive(BasePage.java:1781): latched a StoredPage - page id Page(32,Container(0, 960))`

    The data page is latched.

14. `org.apache.derby.impl.store.raw.xact.RowLocking3.lockRecordForWrite(RowLocking3.java:278): lockObject w/o latch RecordId - X, Record id=22 Page(32,Container(0,960))`

    And an exclusive lock obtained, again.

15. `org.apache.derby.impl.store.raw.data.BasePage.releaseExclusive(BasePage.java:1903): unlatched StoredPage - page id Page(32,Container(0, 960))`

    The data page is unlatched.

16. `org.apache.derby.impl.store.raw.xact.RowLocking2.unlockRecordAfterRead(RowLocking2.java:216): unlock RecordId - Record id=1 Page(6,Container(0, 977))`

    The shared lock on the B-tree record is released.

17. `org.apache.derby.impl.store.raw.xact.RowLocking2.unlockContainer(RowLocking2.java:248): unlockGroup: BaseContainerHandle:(Container(0, 960))`

18. `org.apache.derby.impl.store.raw.xact.RowLocking2.unlockContainer(RowLocking2.java:248): unlockGroup: BaseContainerHandle:(Container(0, 960))`

19. `org.apache.derby.impl.store.raw.xact.RowLocking2.unlockContainer(RowLocking2.java:248): unlockGroup: BaseContainerHandle:(Container(0, 960))`

    The data container is unlocked three times, as for `UPDATE`.

20. `org.apache.derby.impl.store.raw.xact.Xact.releaseAllLocks(Xact.java:1857): unlockGroup: 195`

21. `org.apache.derby.impl.store.raw.xact.Xact.releaseAllLocks(Xact.java:1857): unlockGroup: 195`

22. `org.apache.derby.impl.store.raw.xact.Xact.releaseAllLocks(Xact.java:1857): unlockGroup: 195`

23. `org.apache.derby.impl.store.raw.xact.Xact.releaseAllLocks(Xact.java:1857): unlockGroup: 195`

    And finally all locks are released four times.

Chapter 5

Analysis

This chapter provides our analysis of the findings we have presented in Chapter 4. This includes the basic throughput measurements, DTrace measurements of thread lifetime, the impact of Derby's usage of Java synchronization for thread safety, a comparison of locking in plain Derby and the modified lock system in the DERBY-1704 patch, and a detailed study of Derby's locking subsystem. We will also take a quick look at other possible Derby SMP scalability issues.

5.1 Throughput

We see that the speedup as we add processing threads is actually *negative* as we go from one to eight concurrent threads. In an ideal world, this read-only workload should scale nearly linearly as we exploit more processing power, as no exclusive database locks are required. Naturally, we can't expect perfect linear speedup, but for a read-only load with a low probability for database lock contention significant speedup should be possible.

5.2 Thread execution state

Table 5.1 contains a detailed breakdown of how threads spend their time in different states for various concurrency levels. The total number of transactions (1600000) executed is constant between tests. The first five lines are wait states, and the "running" state is the state where the threads are actually active on the CPU. The measured throughput is included for comparison. See figure 4.2 for a graphical representation of the data in this table.

We note that the time spent running code is reduced from over 90% to less than 50% when going from 1 to 8 concurrent worker threads. Still, with eight times as many threads and a reduction of per-thread active time of about 50%, there is still an overall increase in total time spent running.

This indicates that not only do wait times increase by a large factor, but processing overhead also increases significantly when contention increases.

Workers	1	2	4	8
rwlock	0	1	0	2
kernel-level lock	0	0	10	11
condition variable	0	0	3	39
preempted	4	75	4503	6749
user-level lock	23573	47735	164122	1151074
running	239918	402356	799788	996924
txns/sec	6048	7061	6322	6824

Table 5.1: Total thread lifetime breakdown for various concurrency levels

5.3 Java-level lock investigations

We have found that the amount of contention in Derby increases rapidly as more concurrent threads work in the same table. For low levels of concurrency, waits caused by monitor synchronization dominate. When we reach a concurrency level of 8, explicit wait/notify wastes twice as much time as monitor synchronization.

At this point, we have to distinguish between contention for Java objects' monitors and contention for database locks (both page latches and table/row-level locks) in Derby's lock subsystem. Java object monitors are used to provide thread-safe access to shared objects for multiple threads, while Derby's locks are used to provide consistency and isolation in a transactional context. Derby also uses its own locks for page latching.

As a generalized rule, object monitor contention is an artifact of Derby's implementation and choices made during development, while database lock contention is an artifact of the workload we test the database with and the fact that Derby manages concurrency control with locking[1].

Rewriting the parts of Derby affected by excessive monitor contention may be an effective way to alleviate such problems, but database lock contention cannot be fixed by changing Derby alone (except with a major rewrite to a different concurrency control scheme), as Derby performs that locking on behalf of the application's transactional requirements.

5.3.1 Wait/Notify time

We note that the only class with a significant amount of `Object.wait()`/`Object.notify()` time is the `ActiveLock` class of the locking subsystem. On further investigation with DTrace and the Derby source code [2] we have found that this is caused by threads waiting for a lock to be granted. The relevant method is `ActiveLock.waitForGrant(int timeout)` in the locks package.

To identify the locks causing the waits, we patched some logging into the `ActiveLock` class. A condensed summary of the result, in the form of a "top ten" list, is given in table 5.2.

[1] As opposed to other methods for concurrency control, e.g. multiversion concurrency control [44], [8] or optimistic methods.

We see that there are only latches among the highly contended locks. The very distinctive "stepping" and very significant difference in contention between row 1 and 2 and row 5 and 6 indicates that we are seeing the top of the B-tree here. We already know that Derby places the root of its B-tree in the first page of the container.

Presently, all latches in Derby are exclusive. Pages are latched and unlatched by page consumers, such as B-tree traversal code, using the `setExclusive()` and `releaseExclusive()` methods of the `BasePage` class.

In the B-tree concurrency control algorithms described in papers such as [38] and [35] a distinction is made between shared and exclusive latches. Obviously, rules similar to the regular lock compatibility matrixes apply to latches. Many threads may hold a shared latch on a page if they only intend to read it. Only one thread may hold an exclusive latch, and only a thread holding an exclusive latch on a page may write to it.

A very large percentage of the visits to the higher-level nodes in the B-tree are read-only, even for data modification statements such as `INSERT`, `UPDATE` and `DELETE` that usually will only need to modify the leaf nodes of the B-tree.

However, care must be taken when insert or delete statements cause page splits and merges - in such cases exclusive locks must be obtained on higher-level nodes, and care must be taken to avoid deadlocks. For example: Simply escalating a latch S→X on a split leaf's parent is a deadlock hazard when an X latch is held on the leaf page.

S/X-latches would come at a cost, though - the overhead of managing them is higher than simple mutual exclusion. Thus they may not be of interest for all cases, but investigating their impact on non-leaf B-tree pages would almost certainly be an interesting exercise. [50] discusses a number of approaches to B-tree concurrency control using more advanced lock modes and evaluates their performance, but it may not be directly applicable to the simpler approach in Derby.

# of waits	Lock
527935	Latch: Page(1,Container(0, 977))
4062	Latch: Page(129,Container(0, 977))
3479	Latch: Page(263,Container(0, 977))
3406	Latch: Page(133,Container(0, 977))
3382	Latch: Page(393,Container(0, 977))
22	Latch: Page(523,Container(0, 977))
19	Latch: Page(125,Container(0, 977))
15	Latch: Page(181,Container(0, 977))
14	Latch: Page(95,Container(0, 977))
14	Latch: Page(351,Container(0, 977))

Table 5.2: Top ten database locks causing waits in a 8-thread run on a table with 100 000 rows, with each thread performing 50 000 transactions.

5.3.2 Monitor contention

It is also apparent from the measurements in table 4.3 that monitor contention is a significant issue, and the lock subsystem seems a significant contributor to delays here

as well. We also note the presence of the `Clock` cache manager in the list.

Method signature	# of waits
LockSet.lockObject(Object, Lockable, Object, int, Latch)	3408
LockSet.unlock(Latch, int)	1714
SinglePool.latchObject(Object, Lockable Object, int)	459
SinglePool.unlock(Object, Object, Lockable, Object)	330
SinglePool.unlatch(Latch)	264
SinglePool.lockAnObject(Object, Object, Lockable, Object, int, Latch)	260
java.util.Hashtable.get(Object)	221
LockSet.unlock(Latch, int)	153
LockSpace.unlockGroup(LockSet, Object)	110
SinglePool.lockObject(Object, Object, Lockable, Object, int)	103

Table 5.3: Top ten methods waiting for contended monitors in a 8-thread run on a table with 100 000 rows, with each thread performing 1000 transactions.

Table 5.3 shows the methods "responsible" for causing monitor contention in a test run with 8 threads on a 4 CPU machine. It was one of the earliest tests we performed. Note that the the configuration was somewhat different from the tests performed above, but we still feel confident that the results are relevant and can be considered together with the above results.

The information in table 5.3 was obtained using DTrace and JVM probes, collecting a stack trace every time a thread was suspended to wait for a busy monitor. A less processed version of the output from DTrace is included in Listing B.1.

As the raw stack traces included some JVM internals they were somewhat hard to read, and it also caused a few apparent duplicate methods in the output after we trimmed the JVM internals. In the table above we've simply added the duplicates together. While the numbers may not be perfectly exact, they are very enlightening as to where the contention points are.

`LockSet` and `SinglePool` are the most important classes of the lock manager subsystem, and their prominence in table 5.3 support the suspicion that the lock manager is a bottleneck. The only non-Derby class in the table is `Hashtable`. `LockSet` extends `Hashtable`, so it could be that the lock manager is the culprit here. Inspection of the `Clock` class in the cache manager reveals that a `Hashtable` is a primary component there as well. We have already seen that there is significant contention in the cache manager, but table 5.3 does not pinpoint any further problems there.

Regardless, we feel confident that locking in Derby is a significant problem for scalability.

5.4 DTrace measurements for the DERBY-1704 patch

Comparing the results in Section 4.4.1 and Section 4.4.2 it is apparent that monitor contention in the lock management system is reduced with the "1704" patch. Splitting

the hash table into 16 different tables based on hash value will not completely remove contention, but if the distribution by hash value is fair it should help quite a lot.

A problem with these numbers is that the patched code uses the hash tables, i.e. `java.util.Hashtab` in the results, as the synchronization monitors, instead of `org.apache.derby.impl.services.lock` as the "vanilla" version of Derby does. This means that contention on the split hash tables is counted together with contention on the `Hashtable` in the cache manager. Contention is reduced, though, the numbers are not simply shifted around. This is supported by the increase in throughput provided by the patch.

The appearance of `lockObject()` and `unlock()` in the top blocking methods list reveal that there is, as expected, still some contention in the lock management system. As we have already noted, in Section 5.3.2, the cache management system seems to exhibit contention problems similar to the lock management system. `Hashtable` is a synchronized data structure, and the cache manager `Clock` class extends `Hashtable`. Access to the cache thus goes through a single synchronization point. This is quite similar to the global synchronization in `SinglePool`. As contention is reduced in the lock management system, the bottleneck moves to the cache manager.

5.5 How is the lock subsystem used?

This is our analysis of the lock traces presented in Section 4.5.

5.5.1 Lock actions in a single-row SELECT transaction

First off, there are a lot of visits to the locking subsystem for just one row. But it's not necessarily as bad as it seems, some of them are only necessary at the start and end of the query. The cost of these operations will be amortized over all the rows if a query matches multiple rows. And some locking *is* necessary, after all.

As discussed in section 2.2.6, there are only one object each of the `SinglePool` and `LockSet` classes, so the monitors on those objects become global synchronization points. These are the "giant locking monitors" referred to later.

The number of visits to the locking subsystem combined with the global mutual exclusion in the lock manager causes significant scalability issues. Even workloads that perform parallel work on different tables will be harmed by this. Even if they have no data in common they will be forced to contend for the lock table monitors.

The department of redundancy department [2]

How can we reduce the number of visits to the locking subsystem?

In the beginning of the trace, we observe that the transaction requests an intention shared lock on the container (table), and then checks to see if it already has an exclusive or shared lock on the container. These calls both have critical sections protected by the giant locking monitors.

[2] This is supposedly written on a door sign in [10]

If these three calls were merged into *one* call that both aquired a lock and checked for existing locks, and reported back on what it found, we would save two trips into the monitor. The tradeoff would be that we would have to hold the monitor for a longer time.

Latching and the giant locking monitors

We have found that latches and locks are treated much as equals in the Derby locking subsystem. Both types of concurrency control primitives are managed in the same data structures and incur similar overhead. The most significant cost scalability-wise is the fact that latching shares the giant `LockSet` monitor with most other calls to the locking subsystem.

This is contrary to the common assumption, such as in the ARIES series of papers [37] [35] [38], that locks are about an order of magnitude more expensive than latches. Conversely, latches should be cheap.

However, in Derby latches are not cheap. Table 5.4 lists some measurements taken of the principal lock manager methods implemented by `SinglePool`. The measurements were taken on the same `SELECT` workload we've used earlier, running in "Read Committed" mode. We only ran one thread, to eliminate waits due to contention. Our objective was to measure the fast path. We see that the average cost of aquiring and releasing a latch is only a few percent cheaper than that of aquiring and releasing a lock.

For comparison, we also profiled a simple single-threaded program that aquired and released a `java.util.concurrent.locks.ReentrantLock` (available in Java 1.5 and newer versions) from the Java runtime to see how fast a mutual exclusion lock could work, and found it to be at least an order of magnitude or so cheaper than the Derby latches. See Table 5.5. These measurements were taken on one of our Pentium 4 workstations.

Method	Time	Invocations	Time/invocation
lockObject()	8217 ms	120000	68 µs
unlockGroup()	4920 ms	80002	61 µs
latchObject()	6587 ms	120000	55 µs
unlatch()	6302 ms	120000	53 µs
isLockHeld()	3723 ms	80002	47 µs
unlock()	3229 ms	40000	47 µs

Table 5.4: Time spent in important lock manager methods

Method	Time	Invocations	Time/invocation
unlock()	160 ms	100000	1,6 µs
lock()	150 ms	100000	1,5 µs

Table 5.5: Aquiring and releasing a Java 1.5 ReentrantLock

Care is taken elsewhere in the Derby code, such as in the B-tree scanning code in `org.apache.derby.impl.store.access.btree.BTreeScan`, to avoid the possibility of deadlocks involving latches, so there should be ample opportunity to reduce the cost of latching. Latches could be managed in separate data structures that do not have to care about deadlock detection, and do not share any monitors with the full-scale locking subsystem.

5.5.2 Lock actions in a single-row UPDATE transaction

Section 4.5.2 gives a list of the lock actions during a single-row `UPDATE`. The first two actions performed are similar to the first steps performed in a `SELECT` transastion. An intention exclusive lock is acquired on the container, and a check for an already existing exclusive lock is made. By looking at stack traces in a debugger we established that these calls were made by a Derby-internal `RowChanger` result set being opened.

Then, on line three and four, we see the exact same calls to acquire a IX lock and check for an existing X lock. These calls are made made by a different Derby-internal resultset: `NormalizeResultSet`. We presume that Derby builds a "tree" of sorts of internal result sets to execute queries, and that both of the resultsets seen here take care of their own locking needs.

On lines five through seven we see latching in a regular search in the B-tree. Following that we observe a shared lock being obtained on a record in the B-tree container (Record id 1 on page 5 in container 1041), and then an exclusive lock being obtained on a record in the data container (record id 16 on page 27 in container 1024). We presume that these are both "the same record", one an index record and the other a full data record. At first glance, this locking seems somewhat redundant, but we'll take a closer look.

Closer inspection of stack traces and relevant source code reveals that both these locks are obtained by the B-tree search code - the shared lock in the B-tree is obtained to "protect the position in the B-tree", according to a source code comment, while the row lock in the data container is acquired prior to returning it as a result from the B-tree search/fetch operation.

After this the B-tree search seems to be finished, as the leaf node in the B-tree (page 5 in container 1041) is unlatched. We're now done with line 10 of the trace.

Now the processing of the data page starts. First the page is latched, an exclusive lock on the data row is set for the second time, and the data page is unlatched – on lines 11-13. These three steps are then duplicated on lines 14-16 – page latching, row locking and unlatching.

The first round of latch, lock, unlatch is caused by a `IndexRowToBaseRowResultSet` finishing the lookup of the base table row from the index row returned by the earlier B-tree scan. The result from this lookup is then passed on to the `UpdateResultSet` which then uses the previously created `RowChangerImpl` resultset to actually modify the row, and `RowChangerImpl` latches and acquires locks on its own.

Then we have reached the end of the transaction, and locks are being released. First the shared lock on a row in the B-tree container is released, then all locks on the B-tree is released (three times), and then all locks associated with the transaction are released

(four times). We note some redundancy here as well, and a strong similarity to the end of the `SELECT` transaction described previously.

5.5.3 Lock actions in a single-row DELETE transaction

We'll go through this trace faster than the update trace above, as we find a lot of the same behavior here. Some container IDs have changed as this trace was made on a different instantiation of the database, but they are hardly relevant for the investigation. They are only used to differentiate between the B-tree file and the data heap file.

Starting with the container locks, lines 1-4 acquire IX locks and check for X locks on the data container. This is done twice, just as in the update trace, and for the same reason. There is one internal result set for searching, and one for changing the data.

Lines 5-10 are very similar to the update trace as well. The B-tree is traversed, the desired row is located and locks are obtained on both the row in the B-tree and the row in the data file.

The latch actions in line 11 and 12 are the row being deleted from the B-Tree index, and 13, 14 and 15 is the row being deleted from the base table. We note that no X row lock was acquired on the row in the B-tree for the delete action there. The lock released in 16 is the B-tree scan being closed, and the transaction ends in a similar manner to the update transaction.

5.6 Overall findings in the lock subsystem

By its very nature as the central coordinator of transactional isolation and consistency the lock manager is involved at many stages of statement and transaction execution. In Derby's implementation locks and latches are managed by the same entity, and managed in the same shared data structures. This simplifies system-wide deadlock detection and debugging of deadlocks involving both locks and latches.

Obviously the correctness of the lock manager is vital to maintain the ACID guarantees of the database system. As such the current implementation makes sense. It was built with correctness and debugging support in mind, and support for high concurrency makes verifying correctness and debugging significantly harder.

However, the current implementation's use of shared data structures protected by mutexes will not scale up with multiple client threads. Actual performance has been observed to decline as concurrency increases beyond two concurrent threads, and the total increase in throughput between one and two client threads is very small.

Compounding the problem of coarse-grained mutexes in the lock manager is the use of the lock manager. We have seen in all of the three detailed traces that some locks are acquired more than once. Some of the visits to the lock subsystem could possibly be skipped with a slight modification of the interfaces.

If we attached a list of a transaction's locks to the transaction object itself, we would not have to lock the global lock table to verify the locks held by the transaction. This would save a number of visits to the giant lock monitors. Every call to `LockFactory.-`

`isLockHeld()` could use the local transaction lock table. The redundant lock calls, made by update and delete statements, could check the local transaction lock table before going all the way to the global lock table. This has some potential to reduce contention for access to the global lock table, especially for statements that update and delete data, as they use two internal result sets who both carry out the necessary locking.

While the scope of this project was to assess Derby's scalability as a whole, it has by now narrowed into a treatise on its locking subsystem. While we are fairly certain that the locking subsystem is a scalability problem, we also have indications that it isn't the only point of interest.

5.7 Potential scalability issues outside the lock manager

Table 4.3 details what classes are involved when objects spend time waiting for monitors or in `Object.wait()`. Among the locking related classes and the `PrintStream` contention (caused by the worker threads in our benchmark application who print a little to System.out when they start and stop), we also see the `org.apache.derby.impl.services.cache.Clock` class. As we've mentioned earlier, `Clock` is Derby's page cache subsystem, named for the replacement algorithm it implements.

5.7.1 The cache manager

Source code inspection of the `Clock` implementation reveals that it suffers from a problem similar to the `SinglePool` and `LockSet` implementations. There is one single synchronization point protecting the whole internal state of the cache manager. This state includes a `Hashtable` (the `Clock` class actually extends `Hashtable`) used to look up actual cache entries from their identities and an `ArrayList` used to maintain an ordering of pages for the CLOCK algorithm.

Whenever a page consumer acquires or releases a page, the cache manager is involved, requiring synchronization - and the CLOCK algorithm itself (the clock hand movement) requires exclusive access to the `Clock` object as well. The implementation already recognizes this problem and tries to walk the clock hand over only a part of the cache each time to avoid locking the cache for long periods of time.

Another cause for concern regarding the cache manager is that we can assume that the number of cache accesses is very roughly a function of the throughput of the entire system. The more data pages Derby is processing, the higher the load on the cache manager. Thus, it is not unreasonable to expect contention in the cache manager to become more of a problem as more problems are solved elsewhere and throughput is improved.

The contention and synchronization overhead is not an inherent property of CLOCK as a page replacement algorithm. A CLOCK variant is successfully used in the Linux kernel's virtual memory management subsystem [24], as well as other operating systems such as Solaris [34], who we assume have gone to some lengths to ensure scalability on multiple CPUs.

Other page cache replacement algorithms can be expensive in terms of synchronization as well. A Google Summer of Code [23] project by Gokul Soundarajan during the summer of 2006 investigated possible improvements in Derby's cache manager and appears to have concluded that many algorithms were too expensive synchronization-wise, possibly based on experiences by the PostgreSQL developers [17]. His findings are documented on the Derby project's Wiki at [19].

5.7.2 Other sources of contention

The DERBY-1704 [15] experimental patch to improve concurrency in the lock manager does seem to work for reducing contention, as seen in section 4.4.2. However, as contention in the lock manager is reduced, other sources of contention are revealed. Foremost among those is the cache manager, but we also see some other classes that may warrant further inspection. One of these is the `org.apache.derby.impl.store.raw.xact.TransactionTable`. This is a table used to keep track of all active transactions.

Chapter 6

Conclusion and future work

This chapter provides our conclusions based on the presented results and our analysis. We suggest possible areas for improvements to make Derby scale better on SMP systems. Finally we discuss further academic work based on this project, e.g. the opening for a Master's thesis based on this project.

6.1 Current scalability

We have found that Derby scales poorly on SMP systems. There is little or no speedup as more threads are added, even if the number of CPUs available is higher than the number of concurrent threads. 8 threads running on an 8 core system shows a *negative* speedup compared to a single thread workload on the same system.

The lock management system stands out as a primary bottleneck. This is due to the fact that all database locks and page latches in Derby are kept in a single, synchronized `Hashtable`. This global synchronization point results in contention when running multiple threads, even if the actual database locks are not in conflict. For example, concurrent operations on different databases will cause contention there.

The cache manager also seems to be a major bottleneck. Applying the DERBY-1704 patch to the lock system makes this more clear. The issue with the cache manager is similar to that of the lock manager, i.e. global synchronization on a single `Hashtable`. This is apparently also the case for the `TransactionTable` class, which keeps track of all active transaction. This might be a problem for a workload creating a lot of transactions, as our benchmark does.

6.2 Possible improvements

This leads us to the conclusion that the lock and cache management systems should be optimized to improve the scalability of Derby on SMP systems. We here present some ideas for improvements. We are aware that the Derby community is already working on some of these issues.

6.2.1 The lock management system

The most obvious target for improvement is the lock manager. The Derby community is also aware of this, and some work has already been done to look into the role of the lock manager in scalability issues.

These are suggestions that we think can improve the lock manager:

- **Split the lock tables into multiple segments.**

  The single `Hashtable` in the lock pool can be split into multiple tables based on the hash value of the objects to be locked. This will reduce contention as the global synchronization point is removed. A split into N tables will, if we assume a fair hash function, reduce the possibility of contention to $1/N$. Of course this will only help when the problem is Java synchronization in the lock manager, not if there is contention for the actual database locks. This has been done in the DERBY-1704 experimental patch, which shows a significant improvement in throughput.

- **Move latching out of the lock manager.**

  Latches in Derby use the lock manager, and incurs the same overhead as locks. An alternative is a light-weight latch implementation, as it is easier to ensure correct operation (i.e. deadlock avoidance) for latching that it is for database-level locks. This has been discussed by the Derby community, and there is work in progress to improve latching performance.

- **Give each transaction its own cache of held locks.**

  This could help reduce the number of calls into the lock manager to check if a transaction already holds a lock equal to or greater than the one needed. Since all these calls are synchronized they are a source of contention in the current lock manager implementation. Reducing the number of calls to the central lock manager to a minimum should improve performance.

- **Investigate high concurrency data structures and/or lock-free algorithms.**

  The use of high concurrency data structures and/or lock-free algorithms might be a possibility in the lock and cache managers. It is likely that such solutions require significant implementation effort, as well as require language functionality not present in Java versions prior to 1.5. The atomic compare-and-swap primitives that most lock-free algorithms are based on were introduced in Java 1.5 along with a number of data structure implementations based on them, such as a lock-free linked list and a high-concurrency hash table. This concurrency support does not come for free, however, they are somewhat more expensive than regular locked data structures in the non-contended cases.

6.2.2 The cache management system

The next target would probably be the cache manager, which is a big `Hashtable` and heavily synchronized. The cache manager could be changed to remove or reduce global synchronization.

Splitting the hash table could be a partial solution for the cache manager as well. One could also consider reimplementing `Clock` in a manner that requires less work to be done in critical sections. One possibility is a split between synchronization in the hash table (for lookups and additions/removal from the cache) and synchronization in the data structure used to evict pages, which is separate from the fast lookup hash table. This would permit other threads to look up items in the cache without waiting for a (possibly expensive) page eviction operation.

The page cache implementation in Linux [24] uses two linked lists, where the ends of the lists are the clock hands.

`Clock` stores elements in one big `ArrayList` and the "clock hand" is an index into the array. Eviction of a page thus requires $O(n)$ time as the array is updated. A CLOCK implementation using linked lists would only require constant time per item visited and added or removed to the cache, and could thus be quite cheap.

A rewrite to use linked lists could also pave the way for a possible lockless (or at least lock-reduced) version using Java 1.5 features.

6.2.3 Other improvements

`TransactionTable` may also be worth to look at. It is difficult to say how much of a problem this is, but it is possible that it will become the next bottleneck if the lock and cache managers are improved.

It is also quite possible that `Hashtable`s, or other synchronized data structures, are used elsewhere in Derby even if synchronization is not needed. To improve SMP scalability the use of synchronization in other parts of Derby should also be investigated.

6.3 Future work

During this project we have found that Derby has significant scaling problems on SMP systems. We have identified some points of interest where improvements can definitely be made. A more thorough analysis of the scalability of Derby could also be interesting.

We would like to continue our work on Derby in a Master's thesis next semester. Improving the locking system or the cache manager are possible tasks that we are interested in working on.

Bibliography

[1] Rakesh Agrawal, Michael J. Carey, and Miron Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Trans. Database Syst.*, 12(4):609–654, 1987.

[2] Apache Derby. `http://db.apache.org/derby/`. Accessed 22 September 2006.

[3] Apache Derby JavaDoc. `http://db.apache.org/derby/javadoc/`. Accessed 03 October 2006.

[4] Apache Derby Performance. `http://wiki.apache.org/apachecon-data/attachments/Us2005OnlineSessionSlides/attachments/ApacheCon05usDerbyPerformance.pdf`. Accessed 10 October 2006.

[5] Welcome! - the apache software foundation. `http://www.apache.org/`. Accessed 30 November 2006.

[6] ApacheCon Conferences. http://apachecon.com/. Accessed 10 October 2006.

[7] Berkeley DB - Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Berkeley_DB`. Accessed 11 December 2006.

[8] Philip A. Bernstein and Nathan Goodman. Multiversion concurrency control – theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, 1983.

[9] Richard W. Carr and John L. Hennessy. WSCLOCK – a simple and effective algorithm for virtual memory management. In *SOSP '81: Proceedings of the eighth ACM symposium on Operating systems principles*, pages 87–95, New York, NY, USA, 1981. ACM Press.

[10] G. Chapman, J. Cleese, T. Gilliam, E. Idle, T. Jones, and M. Palin. Monthy Python's Flying Circus: "The Ministry of Silly Walks", episode fourteen. First aired by the British Broadcasting Corporation on 15 September 1970.

[11] Hong-Tai Chou and David J. DeWitt. An evaluation of buffer management strategies for relational database systems. In Alain Pirotte and Yannis Vassiliou, editors, *VLDB'85, Proceedings of 11th International Conference on Very Large Data Bases, August 21-23, 1985, Stockholm, Sweden*, pages 127–141. Morgan Kaufmann, 1985.

[12] Douglas Comer. The Ubiquitous B-Tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[13] The Database Interoperability Consortium. *Distributed Relational Database Architecture (DRDA)*. The Open Group, January 2004.

[14] David Van Couvering's Blog: Oracle Benchmarks BDB vs Apache Derby. `http://weblogs.java.net/blog/davidvc/archive/2006/11/oracle_benchmar_1.html`. Accessed 1 December 2006.

[15] [#DERBY-1704] Allow more concurrency in the lock manager - ASF JIRA. `http://issues.apache.org/jira/browse/DERBY-1704`. Accessed 16 October 2006.

[16] org.apache.derby.impl.store.access.btree. `http://db.apache.org/derby/papers/btree_package.html`. Accessed 26 October 2006.

[17] Email discussion: "Google SoC: Derby Cache Manager". `http://thread.gmane.org/gmane.comp.apache.db.derby.devel/21263`. Accessed 10 December 2006.

[18] Derby Logging and Recovery. `http://db.apache.org/derby/papers/recovery.html`. Accessed 17 October 2006.

[19] DerbyLruCacheManager - Db-derby Wiki. `http://wiki.apache.org/db-derby/DerbyLruCacheManager`. Accessed 10 December 2006.

[20] Sun Microsystems – BigAdmin: DTrace. `http://www.sun.com/bigadmin/content/dtrace/`. Accessed 03 October 2006.

[21] DTrace Probes in HotSpot VM. `http://java.sun.com/javase/6/docs/technotes/guides/vm/dtrace.html`. Accessed 5 December 2006.

[22] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer D. Widom. *Database Systems: The Complete Book*. Prentice Hall, October 2001.

[23] Google Summer of Code program. `http://code.google.com/soc/`. Accessed 10 December 2006.

[24] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. University of Limerick, February 2004.

[25] Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, and Irving L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In *IFIP Working Conference on Modelling in Data Base Management Systems*, pages 365–394, 1976.

[26] Joseph M. Hellerstein and Michael Stonebraker. Anatomy of a database system. In *Readings in Database Systems, 4th Edition*. 2005.

[27] hibernate.org - hibernate. `http://hibernate.org/`. Accessed 1 December 2006.

[28] The Java HotSpot Performance Engine Architecture. `http://java.sun.com/products/hotspot/whitepaper.html`. Accessed 19 October 2006.

[29] IBM software - cloudscape - product overview: Cloudscape. `http://www.ibm.com/software/data/cloudscape/`. Accessed 30 November 2006.

[30] IntelliJ IDEA. `http://www.jetbrains.com/idea/`. Accessed 03 October 2006.

[31] International Organization for Standardization. *ISO/IEC 9075-(1-4,9-11,13,14):2003, The SQL:2003 Standard*, 12 2003.

[32] JDBC Documentation. `http://java.sun.com/j2se/1.5.0/docs/guide/jdbc/`. Accessed 03 October 2006.

[33] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[34] Jim Mauro and Richard McDougall. Prentice Hall, October 2000.

[35] C. Mohan. ARIES/KVL: a key-value locking method for concurrency control of multiaction transactions operating on b-tree indexes. In *Proceedings of the sixteenth international conference on Very large databases*, pages 392–405, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[36] C. Mohan. IBM's relational DBMS products: features and technologies. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 445–448, New York, NY, USA, 1993. ACM Press.

[37] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[38] C. Mohan and Frank Levine. ARIES/IM: an efficient and high concurrency index management method using write-ahead logging. In *SIGMOD '92: Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, pages 371–380, New York, NY, USA, 1992. ACM Press.

[39] MySQL AB :: MySQL: The World's Most Popular Open Source Database. `http://mysql.org/`. Accessed 10 October 2006.

[40] Netbeans. `http://www.netbeans.org`. Accessed 04 December 2006.

[41] The Open Group: Enterprise Architecture Standards, Certification and Services. `http://www.opengroup.org/`. Accessed 26 October 2006.

[42] Oracle berkeley DB. `http://www.oracle.com/technology/products/berkeley-db/index.html`. Accessed 1 December 2006.

[43] Oracle berkeley DB java edition vs. apache derby: A performance comparison. `http://www.oracle.com/technology/products/berkeley-db/pdf/je-derby-performance.pdf`. Accessed 1 December 2006.

[44] Christos H. Papadimitriou and Paris C. Kanellakis. On concurrency control by multiple versions. *ACM Trans. Database Syst.*, 9(1):89–99, 1984.

[45] PolePosition. `http://www.polepos.org/`. Accessed 11 October 2006.

[46] PolePosition Results. http://polepos.sourceforge.net/results/html/index.html. Accessed 12 October 2006.

[47] PostgreSQL: The world's most advanced open source database. http://www.postgresql.org/. Accessed 10 October 2006.

[48] Solaris Enterprise System. http://www.sun.com/software/solaris/. Accessed 03 October 2006.

[49] SQLvsDerbyFeatures - Db-derby Wiki. http://wiki.apache.org/db-derby/SQLvsDerbyFeatures. Accessed 03 October 2006.

[50] V. Srinivasan and Michael J. Carey. Performance of B+ Tree Concurrency Algorithms. *VLDB J.*, 2(4):361–406, 1993.

[51] Michael Stonebraker. Operating system support for database management. *Commun. ACM*, 24(7):412–418, 1981.

[52] Java DB. http://developers.sun.com/prodtech/javadb/. Accessed 30 November 2006.

[53] Sun Microsystems, Inc. *Solaris Dynamic Tracing Guide*, 1 2005.

[54] TPC-B. http://www.tpc.org/tpcb/default.asp. Accessed 11 October 2006.

[55] Tuning Derby. http://db.apache.org/derby/docs/dev/tuning/. Accessed 17 October 2006.

Abbreviations and Terms

Some abbreviations and terms used in this report.

Term	Definition
ACID	Atomicity, Consistency, Isolation, and Durability
API	Application Programming Interface
ARIES	Algorithm for Recovery and Isolation Exploiting Semantics
CDDL	Common Development and Distribution License
DPL	Direct Persistent Layer
DRDA	Distributed Relational Database Architecture
GC	Garbage Collection
IDE	Integrated Development Environment
JAR	Java Archive
JCE	Java Cryptography Extension
JDBC	Java Database Connectivity
JDK	Java Development Kit
JVM	Java Virtual Machine
LSN	Log Sequence Number
ORM	Object Relational Mapping
RDBMS	Relational Database Management System
SMP	Symmetric Multiprocessing
SQL	Structured Query Language
TPC	Transaction Processing Performance Council

Appendix A

Source code listings

Here we include some of our source code.

A.1 DTrace thread state script

Listing A.1 is a DTrace program that aggregates the total time spent in different execution states for the interesting Java threads in our benchmark application.

This script is based on examples in Sun's guide to the DTrace in [53], more specifically the chapter on the "sched" provider.

Listing A.1: DTrace thread state script

```
#!/usr/sbin/dtrace -Zs

self string thread_name;
self char* str_ptr;
self uint thread_id;
self uint java_thread_id;
self uint interesting;

dtrace:::BEGIN {
        starttime=0;
}

/*
 * hotspot:::thread-start, hotspot:::thread-stop probe arguments:
 *  arg0: char*,        thread name passed as mUTF8 string
 *  arg1: uintptr_t,    thread name length
 *  arg2: uintptr_t,    Java thread id
 *  arg3: uintptr_t,    native/OS thread id
 *  arg4: uintptr_t,    is a daemon or not
 *
 * Flag threads named "Worker" as interesting.
 */
hotspot$target:::thread-start
/arg1 >= 7 && stringof(copyin(arg0,6)) == "Worker"/
{
    self->thread_name = stringof(copyin(arg0, arg1));
    self->interesting = 1;
```

```
    starttime = ( starttime ? starttime : timestamp);

    printf("Interesting thread started: %s\n", self->thread_name);
}

sched:::on-cpu
/self->interesting/
{
        self->oncputime = timestamp;
}

sched:::off-cpu
/self->interesting && self->oncputime/
{
        @timing[self->thread_name, "running"] = sum(timestamp - self->↩
           oncputime);
        self->oncputime = 0;
}

sched:::off-cpu
/self->interesting && curlwpsinfo->pr_state == SSLEEP/
{
    /*
     * We're sleeping.  Track our sobj type.
     */
    self->bedtime = timestamp;
    self->sobj = curlwpsinfo->pr_stype;
}

sched:::off-cpu
/self->interesting && curlwpsinfo->pr_state == SRUN/
{
    self->bedtime = timestamp;
}

sched:::on-cpu
/self->bedtime && !self->sobj/
{
    @timing[self->thread_name, "preempted"] = sum(timestamp - self->↩
       bedtime);
    self->bedtime = 0;
}

sched:::on-cpu
/self->sobj/
{
    @timing[self->thread_name, self->sobj == SOBJ_MUTEX ? "kernel-level ↩
       lock" :
        self->sobj == SOBJ_RWLOCK ? "rwlock" :
        self->sobj == SOBJ_CV ? "condition variable" :
        self->sobj == SOBJ_SEMA ? "semaphore" :
        self->sobj == SOBJ_USER ? "user-level lock" :
        self->sobj == SOBJ_USER_PI ? "user-level prio-inheriting lock" :
        self->sobj == SOBJ_SHUTTLE ? "shuttle" : "unknown"] =
        sum(timestamp - self->bedtime);

    self->sobj = 0;
```

```
    self->bedtime = 0;
}

:::END
{
        printf("END\n");
        normalize(@timing, 1000000);
        printa(@timing);
}

syscall::rexit:entry,
syscall::exit:entry
/pid == $target/
{
   exit(0);
}
```

A.2 DTrace wait time and blocking objects script

Listing A.2 is a DTrace program that aggregates the time spent waiting for the interesting worker threads and prints sorted lists top blocking objects and methods.

This script, as well as the next one, were both based on examples included with the Java SE 6 development kit for Solaris, and adapted by us.

The following copyright notice applies to the original script:

Listing A.2: DTrace wait time and blocking objects script

```
#!/usr/sbin/dtrace -Zs

self string thread_name;
self char* str_ptr;
self uint thread_id;
self uint java_thread_id;
self uint interesting;

dtrace:::BEGIN {
        starttime=0;
}
```

```
/*
 * hotspot:::thread-start, hotspot:::thread-stop probe arguments:
 *  arg0: char*,        thread name passed as mUTF8 string
 *  arg1: uintptr_t,    thread name length
 *  arg2: uintptr_t,    Java thread id
 *  arg3: uintptr_t,    native/OS thread id
 *  arg4: uintptr_t,    is a daemon or not
 *
 * Flags threads named "Worker" as interesting.
 *
 */
hotspot$target:::thread-start
/arg1 >= 7 && stringof(copyin(arg0,6)) == "Worker"/
{
    self->thread_name = stringof(copyin(arg0, arg1));
    self->interesting = 1;

    starttime = ( starttime ? starttime : timestamp);

    printf("Interesting thread started: %s\n", self->thread_name);
}

hotspot$target:::thread-start
/!(arg1 >= 7 && stringof(copyin(arg0,6)) == "Worker")/
{
    self->thread_name = stringof(copyin(arg0, arg1));

    self->interesting = 0;

    starttime = ( starttime ? starttime : timestamp);

    printf("Other thread started: %s\n", self->thread_name);
}


hotspot$target:::monitor-contended-enter
/* /self->interesting/ */
{

    self->str_ptr = (char*) copyin(arg2, arg3+1);
    self->str_ptr[arg3] = '\0';
    self->class_name = (string) self->str_ptr;

    self->waitstamp = timestamp;

    monitors[arg1] = self->class_name;

    monitors_enter[arg1] = arg0;
    @blockees[jstack(5)] = count();
    @blockers[self->class_name] = count();
}

hotspot$target:::monitor-contended-entered
/* /self->interesting/ */
{
    monitors_entered[arg1] = arg0;

        self->ts_entered = timestamp;
```

```
    @waittime[self->class_name] = sum((self->ts_entered - self->waitstamp↩
        )/1000000);
        @totalwaittime[self->thread_name] = sum((self->ts_entered - self↩
            ->waitstamp)/1000000);

}

sched:::on-cpu
/self->interesting/
{
        self->oncputime = timestamp;
}

sched:::off-cpu
/self->interesting && self->oncputime/
{
        @timing[self->thread_name, "running"] = sum(timestamp - self->↩
            oncputime);
        self->oncputime = 0;
}

sched:::off-cpu
/self->interesting && curlwpsinfo->pr_state == SSLEEP/
{
    /*
     * We're sleeping.  Track our sobj type.
     */
    self->bedtime = timestamp;
    self->sobj = curlwpsinfo->pr_stype;
}

sched:::off-cpu
/self->interesting && curlwpsinfo->pr_state == SRUN/
{
    self->bedtime = timestamp;
}

sched:::on-cpu
/self->bedtime && !self->sobj/
{
    @timing[self->thread_name, "preempted"] = sum(timestamp - self->↩
        bedtime);
    self->bedtime = 0;
}

sched:::on-cpu
/self->sobj/
{
    @timing[self->thread_name, self->sobj == SOBJ_MUTEX ? "kernel-level ↩
        lock" :
        self->sobj == SOBJ_RWLOCK ? "rwlock" :
        self->sobj == SOBJ_CV ? "condition variable" :
        self->sobj == SOBJ_SEMA ? "semaphore" :
        self->sobj == SOBJ_USER ? "user-level lock" :
        self->sobj == SOBJ_USER_PI ? "user-level prio-inheriting lock" :
        self->sobj == SOBJ_SHUTTLE ? "shuttle" : "unknown"] =
        sum(timestamp - self->bedtime);
```

```
    self->sobj = 0;
    self->bedtime = 0;
}

:::END
{
        printf("END\n");
        normalize(@timing, 1000000);
        printa(@timing);

                printf("--------------------\n");

                printf("\nTOTAL WAIT TIME, BY THREAD:\n");
                printa(@totalwaittime);

            trunc(@blockers, 20);
            printf("\nTOP BLOCKING OBJECTS:\n");
            printa(@blockers);

            trunc(@waittime, 20);
            printf("\nTOP BLOCKING OBJECTS BY WAIT TIME:\n");
            printa(@waittime);

            trunc(@blockees, 20);
            printf("\nTOP BLOCKING METHOD SIGNATURES:\n");
            printa(@blockees);

}

syscall::rexit:entry,
syscall::exit:entry
/pid == $target/
{
   exit(0);
}
```

A.3 DTrace wait time by class

Listing A.3 is a DTrace program that aggregates wait time and count by class in addition to total wait time by thread and blocking objects/methods.

Listing A.3: A DTrace hack

```
#!/usr/sbin/dtrace -Zs

/*
#pragma D option quiet
#pragma D option defaultargs
#pragma D option aggrate=100ms
*/

self string thread_name;
self char* str_ptr;
/*self uint thread_id;
self uint java_thread_id;*/
```

```
self uint interesting;
self uint offcputime;

dtrace:::BEGIN {
    starttime=0;
}

/*
 * hotspot:::thread-start, hotspot:::thread-stop probe arguments:
 *  arg0: char*,        thread name passed as mUTF8 string
 *  arg1: uintptr_t,    thread name length
 *  arg2: uintptr_t,    Java thread id
 *  arg3: uintptr_t,    native/OS thread id
 *  arg4: uintptr_t,    is a daemon or not
 */
hotspot$target:::thread-start
/arg1 >= 7 && stringof(copyin(arg0,6)) == "Worker"/
{
    self->str_ptr = (char*) copyin(arg0, arg1+1);
    self->str_ptr[arg1] = '\0';
    self->thread_name = (self->str_ptr != NULL ? (string) self->str_ptr: ↩
       "foo" );

    /*self->thread_name = stringof(copyin(arg0, arg1));*/
    /*    self->java_thread_id = arg2;
        self->thread_id = arg3;

    printf("thread-start: id=%d, is_daemon=%d, name=%s, os_id=%d\n",arg2,↩
         arg4, self->thread_name, arg3);

    threads[arg2] = self->thread_name;
    */
    self->interesting = 1;

    starttime = ( starttime ? starttime : timestamp);

    printf("Interesting thread started: %s\n", self->thread_name);
}

hotspot$target:::thread-start
/!(arg1 >= 7 && stringof(copyin(arg0,6)) == "Worker")/
{
    self->str_ptr = (char*) copyin(arg0, arg1+1);
    self->str_ptr[arg1] = '\0';
    self->thread_name = (self->str_ptr != NULL ? (string) self->str_ptr: ↩
       "foo" );

    /*self->thread_name = stringof(copyin(arg0, arg1));*/

    self->interesting = 0;

    starttime = ( starttime ? starttime : timestamp);

    printf("Other thread started: %s\n", self->thread_name);
}


hotspot$target:::monitor-contended-enter
```

```
 /self->interesting/
{
    /* (uintptr_t thread_id, uintptr_t monitor_id,
       char* obj_class_name, uintptr_t obj_class_name_len) */

    self->str_ptr = (char*) copyin(arg2, arg3+1);
    self->str_ptr[arg3] = '\0';
    self->class_name = (string) self->str_ptr;

    self->waitstamp = timestamp;

    /* this->where = (string) jstack(1); */

    /*monitors[arg1] = self->class_name;

    monitors_enter[arg1] = arg0;
    @blockees[jstack(5)] = count();
    @blockers[self->class_name] = count();*/

    /* printf("%s: -> enter monitor (%d) %s\n",
        threads[arg0], arg1, monitors[arg1]); */
}

hotspot$target:::monitor-contended-entered
/* /self->interesting/ */
/self->interesting/
{
    /* (uintptr_t thread_id, uintptr_t monitor_id, char* obj_class_name,
        uintptr_t obj_class_name_len) */

    /*monitors_entered[arg1] = arg0;*/

    self->ts_entered = timestamp;
    @classwait["cont" , self->class_name] = sum((self->ts_entered - self↩
       ->waitstamp)/*/1000000*/);
    @threadwait["cont"/*, self->thread_name*/] = sum((self->ts_entered - ↩
       self->waitstamp));
    @classcount["cont", self->class_name] = count();
    @threadcount["cont"/*, self->thread_name*/] = count();

    /* printf("%s: <- entered monitor (%d) %s\n",
        threads[arg0], arg1, monitors[arg1]); */
}

hotspot$target:::monitor-wait
/self->interesting/
{
    self->waitstamp = timestamp;

    self->str_ptr = (char*) copyin(arg2, arg3+1);
    self->str_ptr[arg3] = '\0';
    self->class_name = (string) self->str_ptr;
}

hotspot$target:::monitor-waited
/self->interesting/
{
    @classwait[".wait()", self->class_name] = sum((timestamp - self->↩
```

```
        waitstamp)/*/1000000*/) ;
    @threadwait[".wait()"/*, self->thread_name*/] = sum((timestamp - self↩
        ->waitstamp));
    @classcount[".wait()", self->class_name] = count();
    @threadcount[".wait()"/*, self->thread_name*/] = count();
    self->waitstamp = 0;
}

:::END
{
        printf("END\n");
        printf("--------------------\n");

        normalize(@classwait, 1000000);
        printf("\nTOTAL WAIT TIME, BY CLASS:\n");
        printa(@classwait);

        printf("\nTOTAL WAIT COUNT, BY CLASS:\n");
        printa(@classcount);

        normalize(@threadwait, 1000000);
        printf("\nTOTAL WAIT TIME, BY THREAD:\n");
        printa(@threadwait);

        printf("\nTOTAL WAIT COUNT, BY THREAD:\n");
        printa(@threadcount);
/*
        trunc(@blockers, 20);
        printf("\nTOP BLOCKING OBJECTS:\n");
        printa(@blockers);

        trunc(@waittime, 20);
        printf("\nTOP BLOCKING OBJECTS BY WAIT TIME:\n");
        printa(@waittime);

        trunc(@blockees, 20);
        printf("\nTOP BLOCKING METHOD SIGNATURES:\n");
        printa(@blockees);
*/

}

syscall::rexit:entry,
syscall::exit:entry
/pid == $target/
{
   exit(0);
}

/* vim: set bg=dark: */
```

Appendix B

DTrace results

Results from running our DTrace scripts.

B.1 DTrace monitor contention

This listing is the output from the DTrace program showing the top blocking objects and methods in reverse order.

Listing B.1: DTrace monitor contention

```
TOP BLOCKING OBJECTS:

Monitor class                                        Contention count
  org/apache/derby/impl/services/locks/ActiveLock                   2
  java/lang/ref/ReferenceQueue$Lock                                 2
  org/apache/derby/impl/services/cache/CachedItem                   2
  org/apache/derby/impl/services/reflect/ReflectLoaderJava2         3
  java/io/PrintStream                                               5
  org/apache/derby/impl/store/raw/data/StoredPage                  10
  sun/misc/Launcher$AppClassLoader                                 10
  org/apache/derby/impl/sql/GenericPreparedStatement               10
  org/apache/derby/impl/store/raw/xact/XactFactory                 11
  org/apache/derby/impl/store/raw/xact/TransactionTable            29
  java/util/Hashtable                                              34
  org/apache/derby/impl/store/raw/data/AllocationCache             36
  org/apache/derby/impl/services/cache/Clock                      257
  org/apache/derby/impl/services/locks/SinglePool                 577
  org/apache/derby/impl/services/locks/LockSet                   7851

TOP BLOCKING OBJECTS BY WAIT TIME:

Monitor class                                          Wait time (ms)
  java/lang/ref/ReferenceQueue$Lock                                 2
  org/apache/derby/impl/services/locks/ActiveLock                   2
  org/apache/derby/impl/services/cache/CachedItem                   6
  org/apache/derby/impl/store/raw/data/StoredPage                  11
  org/apache/derby/impl/services/reflect/ReflectLoaderJava2        13
  org/apache/derby/impl/store/raw/data/AllocationCache            110
  org/apache/derby/impl/sql/GenericPreparedStatement              120
  org/apache/derby/impl/store/raw/xact/TransactionTable           129
```

```
  java/util/Hashtable                                              198
  org/apache/derby/impl/store/raw/xact/XactFactory                 350
  sun/misc/Launcher$AppClassLoader                                 660
  java/io/PrintStream                                              709
  org/apache/derby/impl/services/cache/Clock                      1380
  org/apache/derby/impl/services/locks/SinglePool                 3137
  org/apache/derby/impl/services/locks/LockSet                   16194

TOP BLOCKING METHOD SIGNATURES:

  org/apache/derby/impl/services/locks/LockSet.lockObject(Ljava/lang/↩
     Object;Lorg/apache/derby/iapi/services/locks/Lockable;Ljava/lang/↩
     Object;ILorg/apache/derby/iapi/services/locks/Latch;)Lorg/apache/↩
     derby/impl/services/locks/Lock;*
               55

  org/apache/derby/impl/services/cache/Clock.find(Ljava/lang/Object;)Lorg↩
     /apache/derby/iapi/services/cache/Cacheable;*
               56

  org/apache/derby/impl/services/locks/LockSet.lockObject(Ljava/lang/↩
     Object;Lorg/apache/derby/iapi/services/locks/Lockable;Ljava/lang/↩
     Object;ILorg/apache/derby/iapi/services/locks/Latch;)Lorg/apache/↩
     derby/impl/services/locks/Lock;*
               56

  org/apache/derby/impl/services/cache/Clock.release(Lorg/apache/derby/↩
     iapi/services/cache/Cacheable;)V*
               65

  org/apache/derby/impl/services/locks/SinglePool.lockAnObject(Ljava/lang↩
     /Object;Ljava/lang/Object;Lorg/apache/derby/iapi/services/locks/↩
     Lockable;Ljava/lang/Object;ILorg/apache/derby/iapi/services/locks/↩
     Latch;)Lorg/apache/derby/impl/services/locks/Lock;*
               79

  java/util/Hashtable.get(Ljava/lang/Object;)Ljava/lang/Object;*
               84

  org/apache/derby/impl/services/locks/SinglePool.unlock(Ljava/lang/↩
     Object;Ljava/lang/Object;Lorg/apache/derby/iapi/services/locks/↩
     Lockable;Ljava/lang/Object;)I
               85

  org/apache/derby/impl/services/locks/SinglePool.lockObject(Ljava/lang/↩
     Object;Ljava/lang/Object;Lorg/apache/derby/iapi/services/locks/↩
     Lockable;Ljava/lang/Object;I)Z
              103

  org/apache/derby/impl/services/locks/LockSpace.unlockGroup(Lorg/apache/↩
     derby/impl/services/locks/LockSet;Ljava/lang/Object;)V
              110

  org/apache/derby/impl/services/locks/LockSet.unlock(Lorg/apache/derby/↩
     iapi/services/locks/Latch;I)V*
              153

  java/util/Hashtable.get(Ljava/lang/Object;)Ljava/lang/Object;*
```

```
            221

  org/apache/derby/impl/services/locks/SinglePool.lockAnObject(Ljava/lang↩
     /Object;Ljava/lang/Object;Lorg/apache/derby/iapi/services/locks/↩
     Lockable;Ljava/lang/Object;ILorg/apache/derby/iapi/services/locks/↩
     Latch;)Lorg/apache/derby/impl/services/locks/Lock;
            260

  org/apache/derby/impl/services/locks/SinglePool.unlatch(Lorg/apache/↩
     derby/iapi/services/locks/Latch;)V
            264

  org/apache/derby/impl/services/locks/SinglePool.unlock(Ljava/lang/↩
     Object;Ljava/lang/Object;Lorg/apache/derby/iapi/services/locks/↩
     Lockable;Ljava/lang/Object;)I
            330

  org/apache/derby/impl/services/locks/SinglePool.latchObject(Ljava/lang/↩
     Object;Lorg/apache/derby/iapi/services/locks/Lockable;Ljava/lang/↩
     Object;I)Z
            459

  org/apache/derby/impl/services/locks/LockSet.lockObject(Ljava/lang/↩
     Object;Lorg/apache/derby/iapi/services/locks/Lockable;Ljava/lang/↩
     Object;ILorg/apache/derby/iapi/services/locks/Latch;)Lorg/apache/↩
     derby/impl/services/locks/Lock;*
            559

  org/apache/derby/impl/services/locks/LockSet.unlock(Lorg/apache/derby/↩
     iapi/services/locks/Latch;I)V*
           1714

  org/apache/derby/impl/services/locks/LockSet.lockObject(Ljava/lang/↩
     Object;Lorg/apache/derby/iapi/services/locks/Lockable;Ljava/lang/↩
     Object;ILorg/apache/derby/iapi/services/locks/Latch;)Lorg/apache/↩
     derby/impl/services/locks/Lock;*
           2840
```

B.2 DTrace measurements for the DERBY-1704 patch

These are test results from our DTrace monitor contention program for "vanilla" Derby 10.2 and 10.2 built with the patch from Derby JIRA issue 1704 [15].

B.2.1 Monitor contention for "vanilla" Derby 10.2

Listing B.2 is cleaned up output from our DTrace program tracing a benchmark of the "vanilla" version of Derby 10.2.

Listing B.2: DTrace, monitor contention, "vanilla" Derby 10.2

```
TOP BLOCKING OBJECTS:

 java.lang.Class                                                 1
```

```
 org.apache.derby.impl.services.reflect.ReflectLoaderJava2          4
 sun.misc.Launcher$AppClassLoader                                   6
 java.lang.Object                                                   7
 java.io.PrintStream                                                7
 java.lang.ref.ReferenceQueue$Lock                                 25
 org.apache.derby.impl.services.locks.ActiveLock                   32
 org.apache.derby.impl.store.raw.xact.XactFactory                  53
 org.apache.derby.impl.sql.GenericPreparedStatement               102
 org.apache.derby.impl.store.raw.data.StoredPage                  111
 org.apache.derby.impl.store.raw.data.AllocationCache             401
 org.apache.derby.impl.store.raw.xact.TransactionTable            583
 org.apache.derby.impl.services.cache.CachedItem                  607
 org.apache.derby.impl.services.locks.SinglePool                 2948
 org.apache.derby.impl.services.cache.Clock                     12681
 java.util.Hashtable                                            22209
 org.apache.derby.impl.services.locks.LockSet                   93754


TOP BLOCKING OBJECTS BY WAIT TIME:

 java.lang.Class                                                    1
 java.lang.Object                                                  26
 org.apache.derby.impl.services.locks.ActiveLock                   42
 java.lang.ref.ReferenceQueue$Lock                                 46
 org.apache.derby.impl.store.raw.data.StoredPage                  161
 org.apache.derby.impl.store.raw.xact.XactFactory                 229
 org.apache.derby.impl.services.reflect.ReflectLoaderJava2        323
 org.apache.derby.impl.sql.GenericPreparedStatement               368
 java.io.PrintStream                                              660
 org.apache.derby.impl.services.cache.CachedItem                  712
 org.apache.derby.impl.store.raw.data.AllocationCache             734
 sun.misc.Launcher$AppClassLoader                                 745
 org.apache.derby.impl.store.raw.xact.TransactionTable           1879
 org.apache.derby.impl.services.locks.SinglePool                 7262
 org.apache.derby.impl.services.cache.Clock                     22726
 java.util.Hashtable                                            27144
 org.apache.derby.impl.services.locks.LockSet                  139154


TOP BLOCKING METHOD SIGNATURES:

 void LockSet.unlock(Latch, int)
                                                                  481
 Lock LockSet.lockObject(Object, Lockable, Object, int, Latch)
                                                                  502
 void LockSet.unlock(Latch, int)
                                                                  528
 Object java.util.Hashtable.get(Object)
                                                                  586
 void LockSet.unlock(Latch, int)
                                                                  614
 boolean SinglePool.latchObject(Object, Lockable, Object, int)
                                                                  647
 Lock LockSet.lockObject(Object, Lockable, Object, int, Latch)
                                                                  707
 <Unknown>
                                                                  735
 Lock LockSet.lockObject(Object, Lockable, Object, int, Latch)
```

```
                                                                  880
 Lock SinglePool.lockAnObject(Object, Object, Lockable, Object, int, ↩
    Latch)
                                                                  907
 Conglomerate RAMAccessManager.conglomCacheFind(TransactionManager, long)
                                                                 1063
 Lock LockSet.lockObject(Object, Lockable, Object, int, Latch)
                                                                 1074
 Lock LockSet.lockObject(Object, Lockable, Object, int, Latch)
                                                                 1275
 int LockSpace.unlockReference(LockSet, Lockable, Object, Object)
                                                                 3798
 Cacheable Clock.find(Object)
                                                                 4504
 void Clock.release(Cacheable)
                                                                 4767
 Lock LockSet.lockObject(Object, Lockable, Object, int, Latch)
                                                                 8713
 Object Hashtable.get(Object)
                                                                20863
 void LockSet.unlock(Latch, int)
                                                                26401
 Lock LockSet.lockObject(Object, Lockable, Object, int, Latch)
                                                                40856
```

B.2.2 Monitor contention for Derby 10.2 with the DERBY-1704 patch

Listing B.3 is cleaned up output from our DTrace program tracing a benchmark of Derby 10.2 with the DERBY-1704 patch.

Listing B.3: DTrace, monitor contention, Derby 10.2 with the DERBY-1704 patch

```
TOP BLOCKING OBJECTS:

 org.apache.derby.impl.services.reflect.ReflectLoaderJava2         3
 java.io.PrintStream                                               4
 sun.misc.Launcher$AppClassLoader                                  5
 java.lang.ref.ReferenceQueue$Lock                                11
 org.apache.derby.impl.services.locks.ActiveLock                  36
 java.lang.Object                                                 47
 org.apache.derby.impl.store.raw.xact.XactFactory                 68
 org.apache.derby.impl.sql.GenericPreparedStatement               87
 org.apache.derby.impl.store.raw.data.StoredPage                 187
 org.apache.derby.impl.store.raw.data.AllocationCache            521
 org.apache.derby.impl.services.cache.CachedItem                1020
 org.apache.derby.impl.store.raw.xact.TransactionTable          1075
 org.apache.derby.impl.services.cache.Clock                    22619
 java.util.Hashtable                                           49441

TOP BLOCKING OBJECTS BY WAIT TIME:

 java.lang.ref.ReferenceQueue$Lock                                25
 org.apache.derby.impl.services.locks.ActiveLock                  58
 java.lang.Object                                                212
 org.apache.derby.impl.store.raw.data.StoredPage                 368
 org.apache.derby.impl.store.raw.xact.XactFactory                395
```

```
 org.apache.derby.impl.services.reflect.ReflectLoaderJava2       409
 sun.misc.Launcher$AppClassLoader                                548
 org.apache.derby.impl.store.raw.data.AllocationCache            869
 org.apache.derby.impl.sql.GenericPreparedStatement             1311
 org.apache.derby.impl.services.cache.CachedItem                1363
 java.io.PrintStream                                            2012
 org.apache.derby.impl.store.raw.xact.TransactionTable          2759
 org.apache.derby.impl.services.cache.Clock                    46187
 java.util.Hashtable                                           78386

TOP BLOCKING METHOD SIGNATURES:
 Cacheable Clock.find(Object)
                                                                 261
 void Clock.release(Cacheable)
                                                                 279
 Object Hashtable.get(Object)
                                                                 327
 Cacheable Clock.find(Object)
                                                                 328
 boolean FileContainer.pageValid(BaseContainerHandle, long)
                                                                 407
 boolean CachedItem.unkeep()
                                                                 409
 void Clock.release(Cacheable)
                                                                 412
 void CachedItem.setUsed(boolean)
                                                                 417
 <Unknown>
                                                                 658
 Object Hashtable.get(Object)
                                                                 740
 Object Hashtable.get(Object)
                                                                 756
 Object Hashtable.get(Object)
                                                                 758
 void TransactionTable.add(Xact, boolean)
                                                                 847
 Lock LockSet.lockObject(Object, Lockable, Object, int, Latch)
                                                                 991
 Conglomerate RAMAccessManager.conglomCacheFind(TransactionManager, long)
                                                                2165
 void LockSet.unlock(Latch, int)
                                                                3779
 Lock LockSet.lockObject(Object, Lockable, Object, int, Latch)
                                                                4390
 Cacheable Clock.find(Object)
                                                                8184
 void Clock.release(Cacheable)
                                                                8525
 Object java.util.Hashtable.get(Object)
                                                               34236
```