NTNU
Innovation and Creativity

Apache Derby SMP scalability

Investigating limitations and opportunities for improvement

Anders Morken
Per Ottar Ribe Pahr

Master of Science in Computer Science
Submission date: June 2007
Supervisor: Svein-Olaf Hvasshovd, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

Investigate the scalability of Derby on multi-CPU and multi-core systems for an increasing number of concurrent threads. Use a read-only (SELECT) workload. Identify performance limiting factors and bottlenecks. Previous studies have identified latching as a potential area of resource contention. Investigate the cost of latching and concurrent entry to the B-Tree access structure and explore possible improvements.

Assignment given: 20. January 2007
Supervisor: Svein-Olaf Hvasshovd, IDI

Abstract

This report investigates the B-Tree access method of Apache Derby. Apache Derby is an open source Java database system. The detailed focus of the report is on performance aspects of the Derby page latch implementation. Our focal point is the interaction between the B-Tree access method and page latching, and the impact of these components on the ability of Derby to scale on multiprocessor systems.

Derby uses simple and – in the single-threaded case – inexpensive exclusive-only page latches. We investigate the impact on scalability of this design, and contrast it with a version of Derby modified to support both shared read-only and exclusive page access for lookups in index structures. This evaluation is made for single-threaded as well as multi-threaded scenarios on multiprocessing systems.

Based on analyses of benchmark results and profiler traces, we then suggest how Derby may be able to utilize modern Java locking primitives to improve multiprocessor scalability.

Preface

This master's thesis is the result of our work in the course *TDT4900 Master Thesis, Engineering Programme* at the Department of Computer and Information Science, IDI, at the Norwegian University of Science and Technology, NTNU.

We have worked with Apache Derby, an open source database system, to identify and understand performance bottlenecks and improve scalability on SMP systems. This thesis builds upon our work in the autumn project for the course *TDT4740 Database Technology and Distributed Systems, Specialization*.

The assignment was given in cooperation with Sun Microsystems' Trondheim office. Sun also helped us in initial discussions around the problem definition, and provided us with access to an 8-way UltraSPARC system and a Niagara system for testing. We would like to thank them, and in particular Olav Sandstaa, Øystein Grøvlen and Knut Anders Hatlen for their input, advice and time.

We also consider ourself very lucky to be advised by professor Svein-Olaf Hvasshovd, whose insights, advice and helpful feedback have been invaluable in the work on this thesis.

Anders would also like to thank Hege for proofreading, patience and being such a great girlfriend!

June 5, 2007

Per Ottar Ribe Pahr	Anders Morken

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

This chapter is the introduction to the thesis. We will describe the background of the project, the problem definition, our contributions and the structure of the thesis.

1.1 Project context

The course *TDT4900 Master Thesis, Engineering Programme* concludes the 5 year master's degree in Computer Science at NTNU. The master's thesis is the result of work during the final semester, corresponding to 30 credit points[1].

We worked with Apache Derby and scalability in our "autumn project" for the course *TDT4740 Database Technology and Distributed Systems, Specialization* in the autumn semester of 2006. The project report is available online at [83]. The autumn project was intended as a pre-study for a master's thesis, and we chose to continue our work on Apache Derby. In the autumn project we concluded that the scalability of Apache Derby on SMP systems is limited, pointed out several performance bottlenecks and suggested possible changes to improve scalability.

1.2 Motivation

The development of Apache Derby has focused on creating a lightweight database system with a small memory footprint. This makes Derby ideal for use as an embedded database in a Java application. Full transactional support and good support for SQL features has set it apart from other lightweight databases. Thus, performance and scalability have not been top priorities. Nor have other advanced features, e.g. replication or clustering, which are not available in Derby at the moment[2].

For embedded use in an application where access to the database is single threaded, the data volume small and the complexity of queries low, this is not an issue. On the other hand, there are many environments where Derby could benefit from improved multithreaded scalability. Derby is frequently used as a backend for web applications,

[1] ECTS [43] credit points. An academic year of full-time study equals 60 credit points.
[2] It is possible to implement replication and clustering at the JDBC layer with Sequoia [90].

either in embedded or network server mode. Web applications are multithreaded by nature, they must be able to handle multiple simultaneous requests. Web servers are also often SMP or CMT (Chip Multi-Threading) systems, so scalability in these environments is important. Depending on the application, the database might well be the limiting factor of how many connections the web server can handle.

The development in the microprocessor industry has recently focused on multicore architectures [12]. Increasing operating frequency and performance for a single core has proven difficult, so multicore CPUs are seen as the way to execute an even higher number of instructions per second. This means that applications must be parallelized, i.e., multithreaded, to be able to benefit from multicore CPUs and provide the expected performance. This is a challenge for programmers. They have to write efficient programs that scale well on parallel systems.

If Derby is to provide good performance on modern hardware and in multithreaded applications, scalability is important. We found our work with the scalability of Derby in the autumn project interesting, and we wanted to continue to work with Derby. We want to further examine the scalability properties of Derby, and find ways to improve its multithreaded performance.

1.3 Problem definition

The problem definition for this thesis was formulated as:

> *Investigate the scalability of Derby on multi-CPU and multi-core systems for an increasing number of concurrent threads. Use a read-only (`SELECT`) workload. Identify performance limiting factors and bottlenecks. Previous studies have identified latching as a potential area of resource contention. Investigate the cost of latching and concurrent entry to the B-Tree access structure and explore possible improvements.*

1.4 Our contributions

We have analyzed and benchmarked Derby to understand how it scales on SMP and multicore CMT systems. A benchmark application has been developed, and some of the recent and proposed changes to improve performance have been benchmarked. A read-only workload was used.

Benchmarks and analyses point out the root node of the B-Tree index as a performance bottleneck. We suggest shared latches as a way to improve index lookup performance for a read-only, index intensive workload.

We have made an experimental implementation of shared, i.e., read/write, latches to replace the exclusive latches in Derby. We have benchmarked our modified version of Derby against two other versions. Support for shared latches improves performance in high concurrency scenarios.

1.5 Thesis structure

This chapter has given an introduction to the thesis.

Chapter 2 will introduce the reader to the "state of the art". An introduction to Apache Derby is given, and some work on the performance and scalability of Derby presented. This includes results from our previous "autumn project". Then we discuss concurrency control and access method concurrency in database systems, with focus on B-Tree indexes, and mechanisms for concurrency control in hardware and the Java programming language.

Chapter 3 describes our problem, goals, hypothesis, test environment and the resources available for the project.

Chapter 4 describes our contributions, i.e., the experimental implementation of support for shared latches, and a benchmark application.

Chapter 5 presents the results of benchmarking and profiling several different versions of Derby, including our experimental implementation of shared latches support.

The results obtained are discussed and analyzed in Chapter 6. We will explain the benchmark and profiling results, and discuss factors that limit the scalability of Derby.

Finally, Chapter 7 presents the conclusions of this thesis, and suggests possible improvements and future work.

Chapter 2

State of the art

This chapter is intended to introduce the reader to the state of the art. We give an introduction to Apache Derby and present some previous work on Derby performance and scalability. Then we will describe some of the problems associated with access methods and concurrency in database systems, and provide some insight to the problems of multithreading and concurrency control, both at the Java level and how this relates to the hardware Java is running on.

The reader is assumed to be somewhat familiar with database systems in general. For an introduction to the fundamentals of a database system see our report from the "autumn project" [83] and the references there.

2.1 Apache Derby

Apache Derby [5] is an Open Source [77] database management system (DBMS) written entirely in the Java [59] programming language. The project is run by the Apache Software Foundation (ASF) [9], and is distributed under the Apache License [10].

Derby is designed as a lightweight DBMS, with a small code[1] and memory footprint. This makes it easy to embed in Java applications. Despite being lightweight, Derby supports SQL [58], and has full transactional support. In addition to embedded mode, there is also a Derby network server and client. Derby implements the JDBC [60] standard for connectivity.

2.1.1 Background

What is now known as Derby started out as JBMS when Cloudscape Inc. began development of a Java database system in 1996. The first release was in 1997. Informix Software, Inc. acquired Cloudscape in 1999. IBM then acquired the core database assets[2] of Informix in 2001, and the database was renamed IBM Cloudscape.

[1] The JAR file for the core derby engine and JDBC driver is 2.2MB for a normal build. The network server requires another 200KB.

[2] The applications and tools part of Informix was split off as Ascential Software. Ascential Software was acquired by IBM in 2005.

In 2004, IBM decided to open the source code of Cloudscape, and donated the code to the ASF. The project was named Derby, a subproject of the Apache DB [27] project. Sun Microsystems joined the development, and included Derby in their stack of Java products, under the name Java DB. As of Java 6, Java DB is included with the Java Development Kit (JDK).

IBM has recently announced [21] that it will discontinue the IBM Cloudscape product, but continue to support the Apache Derby project. IBM will stop selling and marketing Cloudscape 10.0 and 10.1 June 13, 2007. The products will not be supported after September 30, 2008.

2.1.2 Outline of Derby

As mentioned, Derby can run in either embedded or client/server mode. In embedded mode the Derby engine runs in the same Java Virtual Machine (JVM) as the application. In client/server mode the Derby engine and network server run in one JVM, and client applications that run in other JVMs can access the Derby network server over a TCP/IP network. Figure 2.1 illustrates Derby in embedded mode and Figure 2.2 illustrates operation in client/server mode. Derby supports JVM version 1.3 and higher.

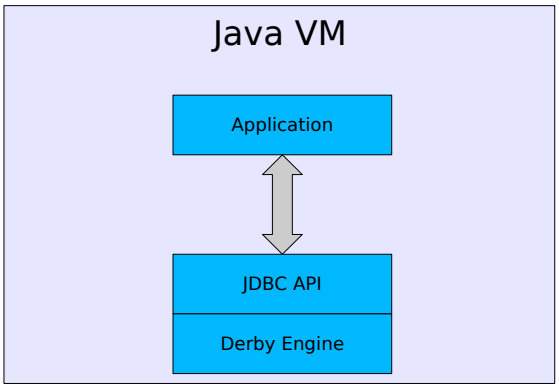

Figure 2.1: Derby Embedded

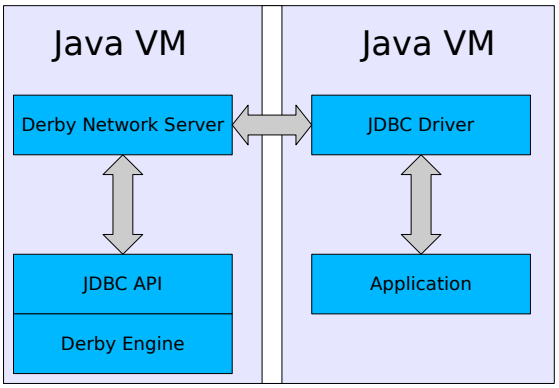

Figure 2.2: Derby Client/Server

The interface between the Derby engine and client applications is provided by an implementation of the JDBC standard. JDBC is used both for embedded connections to the database and for network operation. Strict adherence to standards is important in the development of Derby. This is also true for the SQL standards [58]. Derby supports a subset of SQL, see [92] for a list of SQL features supported by Derby.

Network communication in client/server mode is based on the Distributed Relational Database Architecture (DRDA) [23], which is a standard published by The Open Group [78]. The DRDA architecture was originally developed by IBM [28], and it is also used by the DB2 DBMS [55].

For transactional support, satisfying the ACID[3] properties, Derby does logging and recovery based on the Algorithm for Recovery and Isolation Exploiting Semantics

[3] Atomicity, Consistency, Isolation, Durability. These are required properties for a transaction processing system. Precise definitions of these terms are given in [57].

(ARIES) [74] protocol. ARIES uses Write Ahead Logging (WAL) and a *steal/no-force*[4] policy. Recovery is done using *repeating history* and undo for uncomitted transactions. A description of the Derby implementation of ARIES can be found in [34]. The use of ARIES means that concurrency control in Derby is based on locking[5].

2.1.3 Derby architecture

Derby is based on a modular architecture. A module provides a specific service. The interface of a module is specified by Java interfaces, and the implementation code is separated from the interfaces. In the Derby source code the interface definitions for the internal Derby API can be found in the package `org.apache.derby.iapi`, and the implementation code is found in `org.apache.derby.impl`. Note that this is for the *internal* API and implementation of Derby. The external API available to the application programmer is the JDBC standard API only, Derby does not provide a custom API or extensions of the standard.

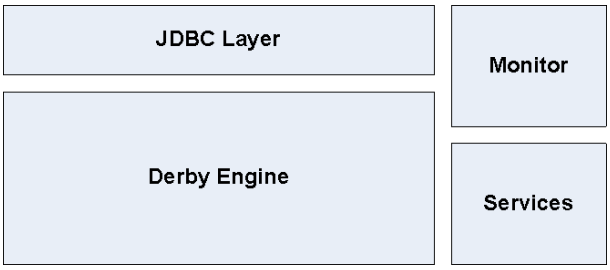

Figure 2.3: Derby Architecture

Figure 2.3 shows an overview of the Derby architecture. The core of Derby is the database engine, which handles SQL and storage. All database access must go through the JDBC layer. Other functions, e.g., lock management or buffer management, are provided as services. The monitor subsystem handles loading (booting) of services when required. This allows for the existence of different implementations of a service, and the correct one to load may be determined at run time. E.g., different versions of JDBC may be used depending on the JVM version Derby is running under.

2.1.4 Derby internals

Figure 2.4 illustrates some of the important subsystems in the Derby architecture. We will describe some of the most important and relevant subsystems of Derby.

[4] A *steal* policy means that pages updated by a transaction can be written to disk before the transaction is committed. This allows for greater IO flexibility and better performance, but undo operations may have to be performed during recovery. A *no-force* policy means that updates done by a transaction does not have to be forced to disk before the transaction can commit, it is enough that the operation is logged to stable storage. Again this allows for better performance, but redo of operations may have to be performed during recovery.

[5] As opposed to alternative methods for concurrency control. An introduction to concurrency control is given in Section 2.3. Concurrency control methods and a performance analysis are presented in [3]. For a description of multi-version concurrency control see [16].

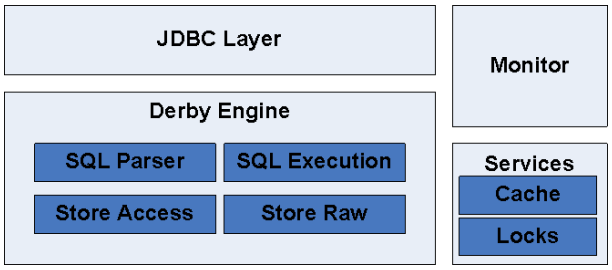

Figure 2.4: Derby Subsystems

JDBC

The JDBC driver is found in the package `org.apache.derby.jdbc`. Derby provides both embedded and network client JDBC drivers. The JDBC layer is the interface to the core engine, which handles parsing and execution of SQL queries.

SQL

Execution of an SQL statement consists of generating a query tree from the statement, validation of the query, optimization and generation of Java byte code for execution. Query optimization and access path selection is described in [89]. A description of the Derby optimizer design can be found in [33]. Java byte code for the selected query plan is generated, and the generated Java class can then be loaded and executed. This is handled by the SQL execution service, implemented by the package `org.apache.derby.impl.sql.execute`. The results from the query are then made available as a JDBC `ResultSet`.

Lock manager

The Derby lock manager is specified by the interfaces in the `org.apache.derby.iapi.services.locks` package. The `LockFactory` interface defines how locks can be obtained and released for objects implementing the `Lockable` interface. This allows the implementation of specific locking policies for different objects.

A lock is associated with a *"compatibility space"*, which is used to determine if a lock request is compatible with locks already held for an object. As of Derby 10.3, the development trunk, this is specified by the `org.apache.derby.iapi.services.locks.CompatibilitySpace` interface. The owner of a compatibility space is usually a transaction.

The `LockFactory` interface is implemented by `org.apache.derby.impl.services.locks.SinglePool`. This class holds a single pool of all the locks in the system, i.e., a single `LockSet` object. A `HashMap`[6] is used to store the pool of locks, and all access to this object is protected by synchronization. This means that the lock table is a global synchronization point, i.e., all lock requests are serialized.

[6] `java.util.HashMap`, included in the Java Standard Edition API.

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

Table 2.1: Simple lock compatibility matrix

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

Table 2.2: Container lock compatibility matrix

Lock types

Shared, exclusive and update lock types are available for data locks. The compatibility of these lock types are shown in Table 2.1. These locks have a scope of either row, range, i.e., multiple rows, or table. Automatic escalation of multiple row locks to a table lock is performed. The escalation threshold is a tunable parameter.

The implementation differs from this simplified view of locks. There is a difference between container, i.e., a table or index, and row locking. Hierarchical locking is used for container locks, i.e., there are *intention* lock types. Hierarchical locking and intention locks are explained in [47].

The lock types for container locking are defined in the interface `org.apache.derby.iapi.store.raw.ContainerLock`. The available types are Container Intent Shared (CIS), Container Intent Exclusive (CIX), Container Shared (CS), Container Update (CU) and Container Exclusive (CX). The compatibility matrix for these lock types is reproduced in Table 2.2.

Support for different isolation levels [13] complicates the implementation of row locking. Separate lock types exist for level 3 isolation and for level 2 and lower levels of isolation. The lock types defined in `org.apache.derby.iapi.store.raw.RowLock` are Row Shared level 2 (RS2), Row Shared level 3 (RS3), Row Update level 2 (RU2), Row Update level 3 (RU3), Row Insert Previous key (RIP), Row Insert (RI), Row Exclusive level 2 (RX2) and Row Exclusive level 3 (RX3). The compatibility matrix for row locks is shown in Table 2.3.

Locking and isolation in Derby is explained in the Derby Developer's Guide [32]. For more in-depth information about the implementation, see the Derby Javadoc, available online at [6], or the Derby source code [8].

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

Table 2.3: Row lock compatibility matrix

Store

The Derby store consists of the raw and access storage layers. The raw store provides page based storage in files. The access layer uses the raw store and provides an interface for accessing conglomerates, i.e., tables or indexes, and rows. Locking is performed at the access level, using the service provided by the lock manager.

Indexes in Derby are provided by a B-Tree [22] implementation. The `org.apache.derby.impl.store.access.btree` package implements a regular B^+-Tree, i.e., all pointers are stored at the leaf nodes. The B-Tree implementation in Derby is explained in more detail at [82]. Note that the `BTree` class defines the root page of the B-Tree to be the first page in the container used to store the index. The first page in a container is defined as page number 1 in the `ContainerHandle` interface. The B-Tree index is always stored in a separate container, i.e., file.

Concurrent access to the B-Tree index is made safe by the use of page level latches[7]. The B-Tree scan protocol requests latches in a top-down and left-to-right order, and at most two latches can be held at any time. This prevents deadlocks involving latches during B-Tree operations. If latches are held for a parent and a child node during B-Tree traversal, then the latch for the parent node has to be released before another latch can be acquired. This is called a *lock-coupling* algorithm. Derby uses *data-only locking* [73], so locks are not used for B-Tree keys. Locks are obtained only for the data rows the index points to.

The raw storage layer provides page based storage. This service is used by the access layer. Pages are organized in files and latches are used for mutual exclusion at the raw page level. Latches in Derby are exclusive, there are no shared/read-only latches. In the current release version[8] of Derby, latches are implemented using the lock manager. This means that the overhead for obtaining or releasing a latch is comparable to that of obtaining or releasing a lock. This is contrary to the traditional assumption that latches should be an order of magnitude cheaper than locks [74]. Using the lock manager also means that latching will contribute to the contention on the global lock table.

[7] Latches are "lightweight" locks used to ensure physical consistency, usually at a disk page level. An explanation of latches is provided in [74].

[8] 10.2 is the current stable branch. As of May 1 2007 the latest official release is 10.2.2.0, which was released on Dec 12 2006.

In the development trunk of Derby, the implementation of latching has been changed to not use the lock manager. Each page, implemented by `org.apache.derby.impl.store.raw.data.BasePage`, will instead keep track of its latch status. Latch acquisition and release is then synchronized on the page object itself instead of on the global lock management subsystem.

Even though there is no global synchronization to obtain a latch, there might still be contention for the page itself, i.e., for a "hot spot" database record or part of an index. A typical example of this is the root node of a B-Tree. All threads accessing a table through the index will have to obtain a latch on the root node, thus causing contention. The Java `wait()/notify()` protocol [98] is used when several threads are waiting to obtain a latch on the same page, so there is no fair queue. It is considered unlikely that this will cause problems with starvation, since latches are only held for a short time.

Cache management

The cache management service in Derby is defined by interfaces in the `org.apache.derby.iapi.services.cache` package, and the implementation is found in `org.apache.derby.impl.services.cache`. The `Clock` class implements the CLOCK cache replacement algorithm. The CLOCK policy is described in [19]. The cache implementation uses a synchronized `Hashtable`, so access to the cache is serialized. This synchronization is for the cache itself only, thread safety for objects retrieved from the cache must be handled separately.

Some work has been done to improve the cache manager in Derby and evaluate other cache replacement algorithms. A Google Summer of Code project in 2006 investigated possible improvements to the Derby cache manager. Results and a report from the project can be found in the Apache Derby Wiki [35].

2.2 Apache Derby performance and scalability

This section presents a selection of work on the performance and scalability of Derby. We will comment on some performance analyses and benchmark results. We will also present our work and conclusions from the "autumn project". For general performance and tuning tips for Derby we refer to the Tuning Derby [100] guide.

2.2.1 Sun's Apachecon US 2005 presentation

At Apachecon [11] US 2005, Sun Microsystems gave a presentation on the performance of Derby [7]. A quick introduction to the Derby architecture was given, and a performance evaluation presented. Tips are provided for host system configuration, and for tuning Derby for better performance. This is illustrated with benchmarks. It is also pointed out that use of *prepared statements* [20] and writing efficient SQL, using indexes where appropriate, is crucial for performance.

The second part of the presentation is a performance comparison of Derby to two other open source database systems, PostgreSQL [87] and MySQL [75]. Both the embedded

and client/server versions of Derby are tested. No configuration or tuning for performance was done for any of the systems, except that the database buffer was set to 64MB and the log kept on a separate disk from the database. Tests were run for a main-memory database (10MB) and an on-disk database (10GB), and with 1-100 concurrent clients. Two different workloads were used, TPC-B [99] like and single record `SELECT`.

For the TPC-B like load and a large on-disk database, Derby outperforms MySQL and PostgreSQL. For the in-memory database MySQL wins, while PostgreSQL performs poorly for the update intensive TPC-B like benchmarks. In the single record `SELECT` benchmarks, Derby is outperformed by both competitors for the in-memory database. For the on-disk database, Derby provides higher throughput than MySQL, but lower than PostgreSQL.

The embedded and client/server versions of Derby provides similar throughput, except for the main-memory database with a single record `SELECT` load. Measurements show that there is higher CPU usage for client/server Derby. A count of the packets sent in networked operation shows that Derby sends and receives four packets per transaction for the single record `SELECT` load, compared to two packets per transaction for the competing systems. This extra network overhead explains why the client-/server version of Derby performs poorly for such workloads.

2.2.2 PolePosition benchmarks

The open source PolePosition [85] benchmark is a suite of tests for database and ORM[9] systems. Benchmarks are implemented and results published for several open source products. There are no results available for commercial database systems. This is because the licenses for such systems generally disallow the publication of benchmark results.

PolePosition provides a framework for writing and running tests, collecting results and presenting the results as graphs. Several benchmarks are implemented. A test is called a "circuit", and these circuits implement different workloads. E.g., the "Bahrain" circuit is a benchmark of write, query, update and delete operations on flat objects and the "Imola" circuit is a read-only benchmark that retrieves objects by their native ID. The published results for the benchmark suite is available online at [86].

The relevance of the PolePosition benchmark for comparison of Derby with other products has been questioned. The tested systems have different transactional support and ACID compliance. Derby has full transactional support, and the default isolation level is "read committed". This means that the results are not necessarily directly comparable.

The PolePosition source code also reveals that some of the tests are not using prepared statements. For Derby this has a huge impact for queries that are executed multiple times. When interpreting the PolePosition results one has to consider that the different systems are not providing the same transactional consistency, and that details in the implementation of a benchmark can affect the results.

[9] Object-Relational Mapping [1]. ORM systems provide automatic translation between an object model and a relational database, i.e., they allow the transparent use of a relational database as the persistent backend of an object oriented application.

2.2.3 Oracle Berkeley DB Java Edition vs Derby

A paper [81] published by Oracle in November, 2006, compares Derby to Oracle Berkeley DB Java Edition [80]. Berkeley DB Java Edition is an open source, lightweight, embedded database system. It has a feature set and interface similar to the original Berkeley DB[10]. Berkeley DB Java Edition has support for ACID transactions, but it is not a relational database. Data is stored as simple key/value pairs and there is no query language support. Thus it is quite limited, compared to Derby, which provides SQL. Berkeley DB also lacks network support.

Benchmarks comparing performance for insert, update and delete operations are presented. Object persistence performance is benchmarked using the Direct Persistence Layer (DPL) [15] with Berkeley DB Java Edition and the Hibernate [51] ORM framework with Derby. While Hibernate does full translation of objects to a relational model and SQL, DPL only provides object persistence. The overhead in using Hibernate and Derby will be higher, but it provides a more flexible solution, and the data can also be accessed through SQL.

Berkeley DB Java Edition and Apache Derby are quite different products. As the paper points out, benchmarking them is an "apples to oranges" [88] comparison. The choice between Derby and Berkeley DB Java Edition for a certain application is more likely to be about the features needed, not about performance. Berkeley DB Java Edition performs better than Derby, but has a minimal feature set. One of the Derby developers has commented on the paper. His post, and a reply from one of the Berkeley DB Java Edition developers, is available online at [26].

2.2.4 DERBY-1704

The Derby development community has been looking into the performance issues with the lock manager. DERBY-1704 [29] is a ticket in the Apache Derby project's issue tracker, JIRA[11]. An experimental patch to split the hash tables in `LockSet` and `SinglePool` into 16 tables each, to eliminate global contention, was made. Benchmarks show performance increases for multiple clients, without negative impact for single-threaded applications. Note that these results were obtained when latching was done using the lock manager.

The work on DERBY-1704 has resulted in the removal of the `Hashtable` in `SinglePool` from the current development version of Derby, but these changes are not yet in the release version. `LockSet` remains a global synchronization point, but changes are proposed for an implementation using `ConcurrentHashMap`, available in Java 1.5, to improve multithreaded performance.

An experimental implementation of `LockSet` using `ConcurrentHashMap` is made available in DERBY-2327 [31]. Benchmarks show throughput improvements, especially for lock intensive loads. There is no significant reduction in throughput for a single client.

[10] An open source embedded database, originally developed at U.C. Berkeley and then by Sleepycat Software, which was acquired by Oracle in 2006. For more information about Berkeley DB and its history, see [14].

[11] JIRA [61] is a bug and issue tracking system for use in project management. JIRA is made by Atlassian Software Systems.

This patch is not yet committed to the development version of Derby.

2.2.5 DERBY-2107

As mentioned in Section 2.1.4, the implementation of latches in Derby has been changed from using the lock manager to setting a latch directly on the page object in question. This change is the result of the work on the DERBY-2107 [30] issue. The changes have been approved and included in the current development version of Derby.

The separation of page latches from the lock manager will reduce the load at the contention point in the lock manager, and improve performance for latch-intensive loads. Note that cheaper latching will not eliminate the problem of contention for the page to be latched, e.g., the root node of a B-Tree will still be a contention point even if it is cheaper to obtain a latch on the page.

2.2.6 Our "autumn project"

We did a project [83] on Apache Derby SMP scalability for our "autumn project" in 2006. In that project we benchmarked, profiled and analyzed Derby to understand its multithreading scalability properties. We used a single record `SELECT` benchmark, DTrace[12] and profiling to locate points of resource contention that limit scalability. We also analyzed the lock manager to understand how locking is performed for `SELECT`, `UPDATE` and `DELETE` operations.

A summary of the results and suggested improvements from our "autumn project" is included here.

Findings

The scalability of Derby on SMP systems is poor. For a `SELECT` only workload there is a slight increase in throughput from one to two threads, but as more than two threads are added the number of transactions executed per second decreases. At eight concurrent threads the throughput is lower than for a single thread. This indicates that contention for resources is serializing the load.

The lock manager in Derby is identified as a contention point that impedes multithreaded performance. Global synchronization on the lock tables results in Java monitor contention. A lot of CPU time is wasted on threads either waiting for synchronization monitors, or executing explicit `wait()` calls, when trying to obtain a lock. While the database level locks are necessary to provide consistent transaction processing, it should be possible to reduce the contention caused by Java synchronization.

The use of the lock manager to implement latches causes performance problems for latch intensive workloads, e.g., index lookup. For the `SELECT` workload we observe that all the top ten locks ordered by wait time are latches. This is because of latches

[12] DTrace [42], short for Dynamic Tracing, is a tracing framework built into Sun Microsystem's Solaris 10 [91] operating system. It provides the possibility for real-time instrumentation of both user and kernel level code. This is useful for profiling and analyzing performance problems.

being obtained for the pages in the B-Tree during index traversal. The time required to obtain a lock and a latch was measured using a profiler, and found to be respectively 68µs and 55µs on average. This can be compared to 1.5µs on average to acquire a `java.util.concurrent.locks.ReentrantLock`[13] on the same system.

Experiments with the DERBY-1704 patch, that splits the hash tables in the lock manager, show that the cache manager is also a point of contention. The `Clock` class keeps the cached pages in a synchronized hash table, and synchronization is also needed when updating the state of the cache. The situation with the cache manager is much like that of the lock manager, it is a global synchronization point. When contention in the lock manager is reduced, the bottleneck moves to the cache manager.

Conclusions

We found that Derby scales poorly on SMP systems. For a high number of concurent threads there is a negative speedup. The lock manager seems to be a major performance bottleneck for a `SELECT` only workload. This is due to global synchronization points resulting in serialization of lock and latch requests. The cache manager is also identified as a performance bottleneck, again due to global synchronization across all threads.

Several possible improvements are suggested. Some of these have already been implemented, or work is in progress by the Derby community. Some possible changes are:

- Splitting the `Hashtable` objects in the lock manager.
- Move latching out of the lock manager.
- Implement read/write latches.
- Let transactions keep track of their held locks.
- Use of concurrent data structures and lock-free algorithms.
- Reduce synchronization in the cache manager.
- Reimplementation of the `CLOCK` algorithm or implementation of other cache replacement algorithms.
- Look into the use of synchronization and synchronized data structures elsewhere in the Derby code.

2.3 Concurrency control

If a database or transaction processing system is to support multiple users simultaneously, some sort of concurrency control is needed to provide transactional consistency.

[13] Available in Java 1.5 and newer versions.

A lot of research effort has been put into the development of efficient concurrency control algorithms. A description of the problems associated with concurrency, and the consistency requirements for transaction processing, can be found in [46].

We will give a short presentation of concurrency control mechanisms and the associated performance implications.

2.3.1 Concurrency control methods

Concurrency control methods can be classified as either *blocking*, i.e., methods that use some form of locking, or *non-blocking*, also known as optimistic, methods. The traditional locking methods will result in transactions being blocked, waiting for the owner of the lock, if there is a lock conflict. Execution cannot resume until the owner releases the lock, i.e., when the transaction commits or aborts if a strict 2PL[14] protocol is used. This means poor throughput, if transactions spend a lot of time waiting in a blocked state. If coarse grained locking is used, it might result in a lot of unnecessary waiting. Fine grained locking, on the other hand, gives extra overhead for the locking protocol. Examples of database systems that use a locking approach to concurrency control are IBM's DB2 [55], Oracle [25] and, as explained in Section 2.1, Apache Derby.

The argument for optimistic concurrency control methods is that locking results in too much overhead, if it is only needed in the worst case. The optimistic approach is to let transactions execute without locking, and then, in a *validation phase*, verify that no conflicting operations were performed, before the results are written and the transaction commits. If a conflict is detected between two transactions, one of the transactions is aborted and restarted. Optimistic methods, and two algorithms for validation, are presented in [64]. Mimer SQL [71] is a DBMS using an optimistic concurrency control algorithm. Optimistic concurrency control in Mimer is described in [70].

Another approach is multiversion concurrency control (MVCC) [16]. In MVCC rows are not overwritten when they are updated. Instead the old version is kept, the new version of the row is written to another location, and rows are timestamped. A transaction will read the most recent version of a row with a timestamp older than the transaction. Thus read operations are never blocked. Write operations can either be validated for consistency at commit time, i.e., like the optimistic methods, or locking can be used in combination with multiversioning. The combination of timestamping and locking is known as a *mixed method*. PostgreSQL [87] is an example of a DMBS using a MVCC scheme.

2.3.2 Performance of concurrency control schemes

The idea behind optimistic concurrency control methods is to provide better performance when locking is not really needed. On the other hand, resources will be wasted when transactions have to be restarted. When transactions are restarted due to conflicts they have to perform operations again, thus wasting CPU time. Different con-

[14]Two Phase Locking. A transaction has a *growing phase* where locks can be acquired, and a *shrinking phase* where locks can be released. No more locks can be acquired when the shrinking phase has been entered. *Strict* 2PL requires that all locks are held until the transaction either commits or aborts.

currency control strategies are studied in [3], and results of performance simulations are presented.

It is shown that blocking methods perform better than restart-oriented methods, when resources are limited. This is because they conserve resources by waiting, contrary to the optimistic methods, where more work has to be performed due to restarting transactions. Under the assumption of infinite resources, or low resource utilization, the optimistic algorithm performs best. This is because it allows a higher degree of concurrency, and, under low utilization, wasted resources is not a problem. For a practical system, with limited resources and designed for high utilization, a blocking concurrency control strategy seems to be the best choice. As a result of this, almost all DBMS implementations use a blocking approach to concurrency control.

2.4 Access method concurrency

In addition to providing transactional consistency, a database system also needs forms of concurrency control to ensure consistency of internal data structures, e.g., indexes. Insert, update and delete operations may modify the structure of the index. When such modifications are done, it is important that concurrent access to the index does not expose an inconsistent state.

In this section we give a brief introduction to the problems associated with concurrent access to index structures, describe some of the algorithms for B-Tree concurrency, and investigate the performance of some B-Tree variants. This will allow us to evaluate the concurrency control mechanism used in the B-Tree index implementation in Derby.

2.4.1 Introduction to index concurrency

A DBMS provides indexed access methods to improve performance, i.e., the cost, in IO operations, to retrieve a single row is likely to be significantly lower for an indexed lookup than for a table scan. In the single client, i.e., single-threaded, case it is that easy. A lower number of page accesses when using an index means better performance.

In a multi threaded scenario, however, it is more complicated. Concurrency control is needed to make sure that the consistency of the index is maintained, and that index lookups return the correct results. This concurrency control requirement may lead to contention. Even though the problem of two threads competing for access to the whole table, as would be the case if table scans were used, is reduced, there may still be contention on the index. The index lookup might be a performance bottleneck, if several simultaneous transactions are accessing a table via the same index.

For a hierarchical index, e.g., a B-Tree, the highest level may be a contention point. This is because all index lookups start with reading the root node of the tree. If shared locks are used, the impact of this is reduced, as the root node is seldom updated. Insert or delete operations, that modify the structure of the index, require mutual exclusion to ensure the consistency of the index. These operations require exclusive locks for structure modifications, while read-only search operations acquire only shared locks. For workloads that result in a large number of index structure modifications, contention

may be a more significant problem.

2.4.2 B-Tree concurrency

The problem of concurrency control in B-Trees has been studied extensively. Many algorithms, locking protocols and modifications of the B-Tree have been proposed, some of them are quite complex. An example is the B-link Tree, proposed in [68]. We are primarily concerned with the B^+-Tree variant, because this is the B-Tree variant used in Derby, but for the purpose of discussion other variants and algorithms will be described. B-Tree variants and algorithms for concurrency control are described in [93]. We will present some of the algorithms here.

Naive approach

The simplest approach to concurrency control in B-Trees is to exclusively lock all parts of the tree that might be modified. If a node may be modified, this means that exclusive access to the entire sub-tree rooted at the node must be obtained. Because structure modifications might propagate up the tree, there is a risk that the whole tree must be locked. This simple solution provides a low level of concurrency, because locking a sub-tree, or in worst cases the whole tree, in exclusive mode will block other operations. If the index is not in main memory, it will be very inefficient to keep the entire index locked while waiting for an IO operation to fetch the next node.

For an index that is kept entirely in main memory, the naive approach to concurrency may give satisfactory performance for single CPU systems. In this case the overhead to obtain and release a lock may be of the same order of magnitude as the index operations. If no IO operations may be performed during index operations, it may be more efficient to just lock the entire tree, compared to setting a high number of locks for each node involved in the index operation. For scalability on a multi CPU system it may be necessary to use multiple entry points to the tree structure, multiple trees or other access methods.

Bayer-Schkolnick algorithms

The Bayer-Schkolnick algorithms for concurrent B-Tree modification use a hierarchical locking protocol with S, X, IX and SIX locks[15]. Structure modifications in the Bayer-Schkolnick algorithms are done with exclusive access to the entire scope of the update, essentially as one atomic operation with respect to other concurrent operations. Read-only index searches use S locks and *lock-coupling*[16] during tree traversals. This ensures that no conflicting modifications can be performed, because the exclusive lock types are incompatible with shared locks. Because lock-coupling is used, a consistent tree will be observed during a traversal.

For updates there are several variants of the Bayer-Schkolnick algorithm. They are known as B-X, B-SIX and B-OPT. The B-X algorithm uses X locks and lock-coupling for

[15] These lock types are explained in [47].

[16] A lock-coupling algorithm holds at most two locks during tree traversal. The lock held for the current node is released as soon as the lock for the child node has been obtained.

updates. The problem with this approach is that an exclusive lock must be obtained for all nodes in the search path of an update operation. Requiring X locks for nodes that may not be modified is inefficient. Especially for the frequently accessed root page this leads to contention.

The B-SIX variant uses SIX locks and lock-coupling during tree traversal for update operations, and upgrades to X locks when reaching a node that must be updated. This allows a higher degree of concurrency, because SIX locks are compatible with S locks. There will be no conflicts with read operations other than for the nodes that actually have to be modified. This approach is still inefficient for concurrent updates, which will be conflicting also at the higher levels of the tree, even if the nodes will not actually be updated.

B-OPT is an "optimistic" version of the algorithm, which assumes that node splits are not likely to happen. IX locks and lock-coupling are used during traversal for update operations. When reaching the leaf node an X lock is acquired. If the child node is safe[17], the modification can be performed at the leaf level. If the leaf is not safe, the locks will be released and the algorithm restarts from the root node like the SIX variant, acquiring an exclusive lock the first time an unsafe node is encountered. For B-Trees with high *fanout*[18], page splits and merges are assumed to be rare. This version of the algorithm is expected to perform well, even if it risks an extra traversal in worst cases.

Top-down algortihms

In *top-down* algorithms preparatory page splits and merges are performed. If an update operation encounters a full node during traversal, the node will be split. Correspondingly, a delete operation will merge nodes if a node with only one entry[19] remaining is found. This approach may split or merge nodes that otherwise might not have been modified, but it ensures that parent nodes are always safe, so splits or merges can be performed at lower levels without risk of changes propagating up the tree.

For updates, the top-down variants can be classified as the Bayer-Schkolnick algorithms, i.e., TD-X, TD-SIX and TD-OPT. TD-X uses X locks and lock-coupling. Before releasing the lock on the parent node during lock-coupling, a split or merge can be performed if an unsafe node was reached. TD-SIX is a variation of the algorithm using SIX locks, that will be upgraded to X locks only when needed, like the B-SIX algorithm.

The "optimistic" version of the top-down algorithm, TD-OPT, uses S locks and lock-coupling on the first pass, assuming that structure modifications will most likely not be needed at the higher levels. The leaf is locked with an X lock, and if it is unsafe the algorithm restarts, using the TD-SIX protocol on the second pass.

The important improvement in the top-down algorithms, as opposed to the Bayer-Schkolnick algorithms, is that modifications are reduced to a series of operations in-

[17] A *safe* node in a B-Tree is a node that will not have to be split or merged if an insert or delete occurs.

[18] *Fanout* is the number of pointers per node in the B-Tree. A high fanout means a higher number of children per node, and a lower tree height for a given number of keys. For B-Trees stored in large disk blocks the fanout can typically be quite large, a fanout of 100 or more is not uncommon.

[19] In practical implementations of B-Trees nodes are usually not required to be at least half full. Merges are only performed when nodes become empty. This improves efficiency without wasting too much space.

volving only two nodes, i.e., the node to be split or merged, and the parent node. Thus exclusive access is not needed for the entire subtree of the node to be updated. This allows for higher concurrency, because concurrent modifications are allowed in the parts of the tree structure that are not directly affected by the update.

The B-link Tree

The B-link Tree [68] is a B-Tree variant where nodes at each level[20] are linked together in increasing key order, i.e., each node is linked to its right sibling. Each node also contains a *high key*, which is the highest key value of the sub-tree below the node. The links and the high key value allow page splits to happen in two steps, this is called *half-splits*. This means that in the B-link Tree algorithms, structure modifications can be broken up into sub-operations that affect only nodes at one level.

When splitting a node, it is locked in exclusive mode. Then space for a new node is allocated, and the *higher* half of the keys in the node are copied to the new node. The right link of the original node is pointed at the new node, and the new node points to the node that was linked from the original node. Then the link from the parent node must be updated, and a link to the newly created node inserted. This requires locking the parent in exclusive mode. But since the parent was not locked when the split was performed, other operations might follow an inconsistent link to the newly split node, which has now been unlocked. This situation is resolved by always comparing the search value to the high key when reaching a node. If the search value is greater than the high key, the node has been split. Then the link to the next page at the same level must be followed, this is called a *link chase*.

The original B-link algorithm [68] gives no procedure for merging nodes, instead nodes are allowed to underflow in the case of delete operations. An algorithm for merging nodes in a corresponding two step manner, termed a *half-merge*, is given in [65]. This algorithm requires the introduction of a new pointer, called the *outlink*. In a merge operation, the keys in the node on the right side is moved to the node on the left, and the *outlink* from the right node pointed at the left node, i.e., where the keys were moved. This allows searches that reach the removed node, before it has been unlinked from the parent, to follow the link to the newly merged node containing the keys.

Two variants of the B-link algorithm are LY (for Lehman-Yao) and LY-LC. In the LY algorithm, searches use S locks during traversal. Locks on nodes are released *before* acquiring a lock on the next node, this differs from the lock-coupling approach. Update operations follow the same protocol during traversal, but when the leaf to be updated is reached, the S lock is released, and an attempt made to acquire an X lock on the leaf. At most one lock is held at any time.

When the X lock has been granted, the page might have been modified, in this case link chases will be performed until the correct node is found. X locks are used during these link chases, but locks are released before the next one is acquired. If the update requires a split or a merge, the parent node will also have to be updated. Then the lock is released, before the higher-level node that was used is locked in X mode. Note that this node may have been updated, so link chasing may be necessary to find the correct

[20] As opposed to the B^+-Tree, where only nodes at the leaf level are linked, in increasing key value order, to allow efficient value range scans.

node to update.

The LY B-link algorithm allows operations to see inconsistent states. This is because lock-coupling is not used, but a node is unlocked before the next one is locked. The next node may already be locked, or a concurrent operation can obtain a lock on it, thus leaving the node in a conflicting state when the lock is granted. E.g., an insert operation might find the key it is inserting existing at a higher level, because a delete operation on the key in question has not yet propagated up the tree. This is solved by restarting update operations when such inconsistencies are detected. There is no guarantee that an operation does not have to be restarted multiple times.

The possibility of inconsistent situations is resolved by the LY-LC algorithm. In this algorithm, update operations release the lock on a node that has been split or merged after the lock to update the parent has been acquired. Updaters retain an S lock on newly split or merged nodes when requesting an X lock on the parent node to be updated. This can be explained as "reverse" lock-copling, i.e., on the way up the tree instead of down. This bottom-up approach to structure modifications allows higher concurrency.

OPT-DLOCK

A new algorithm, called OPT-DLOCK, is introduced in [93]. As the name implies, it is an "optimisic" algorithm, in the sense that it assumes that splits or merges will most likely not have to be performed. The algorithm depends on deadlock detection to decide if a restart is required. Updaters use S locks, that are kept until a safe node is reached, during traversal. A node is only considered safe if it is safe for both insert and delete operations. When the leaf is reached, it is locked in X mode. If the leaf is unsafe, the lock is released and the highest level S lock still held is upgraded to an X lock, then X locks are acquired for the lower-level nodes in the scope of the update, and the updates performed.

If two updaters are in conflict, they will have the same high-level safe node. This means that there will be a deadlock, when both operations try to upgrade their locks on the node to an X lock. Because the conflicting operations must have the same high-level safe node, deadlocks can be detected locally at nodes. When a deadlock is detected, one of the operations is restarted. To avoid starvation, priority is given to operations by timestamping.

ARIES/KVL and ARIES/IM

Two methods for improved concurrent performance of B-Tree operations, based on the ARIES protocol, are presented in [73]. They are ARIES/KVL (ARIES using Key-Value Locking) and ARIES/IM (ARIES for Index Management). In both algorithms, a B-Tree variant where leaf nodes are linked to both their left and right siblings is used. This allows efficient range scans. Structure modifications are performed bottom-up from the leaf nodes.

In ARIES/KVL, lock-coupling (latch coupling) and S locks are used for tree traversal. An exclusive latch is obtained for the leaf node before any modification to the

tree structure, and this is held until all changes caused by the modification have been performed.

During modification, another thread might read an inconsistent pointer from a node, resulting from a split or merge at a lower level that has not yet propagated up the tree. This is solved by setting a bit, called the *Structure Modification Bit* (SM_Bit), on the modified pages. This bit is set when the modification of a leaf node is initiated, indicating that the page is under modification.

When structure modifications propagate up the tree, the SM_Bit is set for all the pages affected by the modification. This allows the detection of an inconsistent state. If inconsistencies are encountered, the search is restarted. Remembering the parent nodes that have been visited, and their version number, allows the search to be restarted from the lowest possible consistent level in the tree, i.e., a restart from the root node can be avoided in most cases. When the structure modification is complete, the SM_Bit is reset for all the pages that was updated.

ARIES/IM uses the same protocol for latching and index traversal as ARIES/KVL, but provides a more efficient logical locking protocol. Locks are obtained for individual instances of keys, instead of key values as in ARIES/KVL. This allows a higher level of concurrency for non-unique indexes. ARIES/IM requires a minimal number of locks, and thus provides a higher level of concurrency.

There are two variants of ARIES/IM, one using *data-only locking*, the other uses *index-specific locking*. If the transactional unit of locking is a page, data-only locking will lock the page containing the data value. If the unit of locking is a row, only the specific data row will be locked. Index-specific locking, on the other hand, differentiates between a lock on a key value and a lock on the corresponding data value. Data-only locking allows for a reduction in the number of locks required, while index-specific locking provides higher concurrency in some cases, at the expense of more locking overhead.

2.4.3 Performance of B-Tree variants

In [93] several B-Tree concurrency control algorithms are discussed, and a performance simulation is done for varying system parameters, e.g., the fanout of the B-Trees, number of CPUs and disks, and buffer size. Different workloads simulating low, moderate and high contention scenarios are used. The performance of the algorithms are evaluated when the level of concurrency increases.

Low contention

Simulations are first run for a low contention scenario. A workload consisting of 80% search operations, 10% random inserts and 10% random deletes is used. The experiment is configured with a buffer pool holding about 75% of the index. For a system with one CPU and a single disk, the different algorithms give approximately the same throughput. Increasing the concurrency level from 1 to 4 gives only a slight increase in throughput, and all algorithms saturate at this level. This is because the single disk becomes a bottleneck.

For a single CPU system with 8 disks all algorithms reach a higher throughput. The

lock-coupling, i.e., pessimistic, algorithms give about half the throughput of the optimistic and B-link algorithms. For the optimistic variants disk utilization is high, but for the pessimistic algorithms disk utilization is less than half of the maximum. This indicates that the pessimistic algorithms are limited by lock contention, not by the available resources. The waiting time to obtain locks for the root node is shown to be high compared to the response time of the transactions for the lock-coupling algorithms, i.e., the root node is a bottleneck.

Moderate contention

A workload with 100% inserts is used for a moderate contention scenario. Again a buffer pool that can fit 75% of the initial index is used. The difference between the algorithms is again small for a system with only a single disk. For 8 disks the B-link algorithms perform best, closely followed by the optimistic variants. The pessimistic algorithms perform much worse.

The reason for the B-link algorithms giving better throughput is that they manage to utilize the disks completely, while the optimistic algorithms utilize only about 80% of the maximum disk capacity. Of the optimistic algorithms, the highest peak throughput is provided by the top-down variant. This is because the time spent waiting for locks is lower with the TD-OPT than with B-OPT. TD-OPT results in a higher number of restarts, but this is not a problem when throughput is limited by disk capacity.

To evaluate the performance of an in-memory index, simulations for the moderate contention workload are also run for a scenario with infinite resources, i.e., unlimited CPU and disk capacity. Under these conditions, B-OPT performs better than TD-OPT. This is because TD-OPT performs a higher number of restarts, and the overhead of a restart is higher in the case of infinite resources.

High contention

A high contention scenario was simulated with a workload consisting of 50% appends and 50% random searches. Appends are done in increasing key order, this creates a high level of contention for the rightmost nodes in the tree. The difference between the algorithms is again small for a single CPU system with one disk. This is also the case if the number of disks is increased to 8. If infinite resources are assumed, the B-link algorithms perform better than the other variants, showing improved throughput at high levels of concurrency. There was no significant difference between the LY and LY-LC variants of the B-link Tree.

Simulations are also run for the high contention workload when the whole index is in main memory. In this case, TD-SIX, a pessimistic algorithm, performs better than the B-link and optimistic variants for a single CPU system with one disk. This is because the high number of link chases for the B-link variants, caused by the high contention append workload, results in much overhead at high levels of concurrency.

For the optimistic variants, a high number of restarts results in much overhead. For the B-link variants, multiple link chases may be necessary because of the high contention append workload. The paper points out that for an index that can be kept in main memory only, the overhead of link chases or restarts is of the same order of mag-

nitude as the response time of an operation. This means that a high number of link chases or restarts will give a relatively high increase in response time, thus reducing throughput. If a higher number of CPUs is used, or infinite resources assumed, the B-link and optimistic variants perform better than the pessimistic algorithms, also for an in-memory index. This is because the impact of wasted resources for link chases or restarts is reduced when the CPU is not saturated.

Conclusions

The conclusions of the paper is that the B-link variants generally perform best for different system configurations and workloads. An algorithm that does lock-coupling with exclusive locks will result in bad performance, because this reduces parallelism. Bottlenecks form at the root node of B-trees if it is locked in exclusive mode. The ARIES/IM algorithm is predicted to perform close to the B-link algorithms if several page splits at one time are allowed. No further explanation is given for this assumption. However, the ARIES variants are similar to the B-link algorithms, performing structure modifications in a bottom-up manner.

2.4.4 The Derby B-tree protocol

The B-Tree scan algorithm in Derby uses lock-coupling, as explained in Section 2.1.4. The B-Tree locking protocol uses the same page level latches that are used to ensure low level physical consistency of all data pages in Derby. This means that there are only exclusive latches. Latching in Derby was explained in Section 2.1.4. This means that all index operations will have to obtain exclusive access to the root node at the start of an index operation, i.e., the root node will be a bottleneck for high levels of concurrency. As concluded in Section 2.4.3, lock-coupling with exclusive latches is inefficient.

The Derby protocol for updates in the B-Tree is an optimistic variant, even though the exclusive latches make it inefficient. In the first pass, updates are not performed until the leaf is reached. If the leaf is unsafe for the operation, the algorithm releases locks and restarts from the root node, this time splitting or merging the internal nodes that are unsafe. After the internal nodes have been updated, the algorithm does a normal restart of the operation. Because locks are released after the internal node update phase, there is no guarantee that other transactions have not modified the tree, and the leaf again has become unsafe. Thus several restarts might be required.

The Derby B-Tree algorithm is likely to provide poor performance for high levels of concurrency. This is because it uses exclusive locking and lock-coupling. The possibility of multiple restarts of operations that involve structure modifications may also give extra overhead.

2.5 Multithreading and concurrency control

Derby executes all JDBC calls in the thread they were made from. As its clients are often multithreaded, Derby also needs to maintain thread safety. In addition to thread

safety, Derby needs to maintain transactional isolation, which inherently works at a level higher than threads, using different synchronization primitives and strategies. While not directly related, the implementation of transactional isolation needs to use lower-level thread synchronization primitives to ensure a coherent view of data.

To fully understand the performance implications of multithreaded Derby running on a multiprocessor machine, it is important to understand the performance implications of low-level synchronization primitives, as well as how the implementation uses them. In this section we will therefore describe some hardware features used for synchronization, the synchronization primitives that are available in Java and how they are implemented.

2.5.1 Hardware support for multithreaded software

Different hardware architectures present different low-level primitives for locking, but the general building block is an operation that permits a read and subsequent write to a memory location to happen atomically. We will discuss such atomic instructions, and instruction reordering, which can have unwanted effects for thread synchronization, and how memory barriers can be used to resolve this.

Locks and atomic instructions

Compare-And-Swap (CAS) is one of the more commonly available atomic memory access primitives. It is available in the SPARC, i.e., the `CASA` instruction [104], and Intel x86, i.e., the `CMPXCHG` instruction [24], architectures.

The MIPS[21] approach to atomic memory access is the *Load Linked / Store Conditional* instruction pair, which is interesting because it nicely illustrates how the atomic access property is handled in hardware. A "linked" load will set a "link bit" in addition to performing the load. This link bit will be checked by the conditional store operation, which will fail if the link bit is unset. The hardware will take care of unsetting the link bit if the underlying memory location is touched by another processor, i.e., if the cache line is invalidated, or if *bus snooping*[22] reveals that someone else has written to the memory location.

This points to another problem with lock contention on multiprocessors. A heavily contended lock on a shared-memory multiprocessor will not only force threads to wait for resources, it also consumes system resources. Because blocking a thread using the OS is an expensive operation, it is common to repeatedly attempt to acquire the lock a few times, known as *spinning* on the lock, before yielding. With multiple processors contending for a lock, this causes the processors to contend for ownership of the cache line containing the lock. This causes cache coherency traffic on the CPU/memory interconnect. More on this can be found in papers such as [17] and [4].

[21] MIPS [62], for Microprocessor without Interlocked Pipeline Stages, is a RISC processor architecture by MIPS Technologies [72].

[22] Bus Snooping is a technique used to ensure cache coherence in multiprocessor systems. Processors listen for broacasts about cache line status on a common bus, and invalidate and update lines as necessary.

Out of order execution and memory barriers

Instruction reordering can happen both at compilation time and execution time. Modern compilers reorder instructions as part of the optimization, an example of this is given in [63]. Modern processors also reorder instructions dynamically, e.g., to reduce the amount of time spent waiting for memory loads and stores [50].

While this reordering carefully avoids changing the final result of the computation, it can affect the order of reads and writes as seen from other concurrent threads of execution, thus leading to very challenging debugging sessions.

To regain control of when memory is read and written, special instructions known as memory barriers can be inserted into the instruction stream, which prevents reordering of loads and stores across the barrier.

One example of a place where such a barrier is needed is at the end of a critical section. A write barrier is needed *before* the critical section ends, to make sure that any stores intended to be protected by the critical section are not reordered and done outside of the critical section.

2.5.2 Java threading and thread synchronization

The Java language provides support for multithreading. An execution thread in Java is represented by the `java.lang.Thread` class. Java provides synchronization primitives, to enable mutual exclusion for threads, and also higher level locking primitives in the `java.util.concurrent` package.

To understand the implications of multithreading and thread synchronization in Java we will discuss some of these features, and how they are implemented in the current Sun JVM.

The Java Memory Model

The Java Memory Model is defined in chapter 17.4 of the Java Language Specification, Third Edition [45]. The memory model specifies the rules that decide whether a particular execution of a program is valid or not. This approach, as opposed to defining how a program should be executed, gives implementations a fair amount of leeway to optimize and reorder instructions, still ensuring that the observed behavior of software is correct for the given rules.

Java monitors and synchronization

The principal concurrency control mechanism in Java is the *monitor*, a term coined in [52]. Java monitors are not "true" monitors in this sense, see [48] and [49]. Conceptually, every Java object has a monitor associated with it. A Java monitor is essentially a combined mutual exclusion lock and condition variable. We consider a full discussion of monitors beyond the scope of this thesis.

The `synchronized` keyword is used to enter, i.e., lock and leave, i.e., unlock, monitors. Either a method or a code block can be synchronized. The language and com-

piler are constructed in a way that ensures that monitors will always be symmetrically locked. Monitors will always be released as the control flow leaves the synchronized block or method.

This property of Java's object locking means that it is unsuitable for some applications, such as lock-coupling in B-trees. However, when combined with condition variable functionality, implementing a simple old-fashioned lock, using monitors and a few state variables, is straightforward.

The performance of Java synchronization has been the subject of much research. There has been a significant focus of engineering effort, particularly in VM implementations with mature JIT compilers, where code execution is fast enough to make synchronization overhead a significant factor.

JVM implementation of synchronization primitives

JVMs are free to implement Java monitors however they like, as long as the requirements of the Java Memory Model are fulfilled. For a practical example, we will take a look at how Sun Microsystems' current releases of the HotSpot[23] Java VM implements monitors to see how the issues discussed above influence monitors and scalability.

Implementations of synchronization primitives have evolved as Java VM technology has matured. JVMs prior to HotSpot placed relatively little emphasis on synchronization performance, but it has received significant work in newer versions. [2] and [37] are technical reports from 1999 and 2006, respectively, that discuss how JVMs can implement synchronization primitives as efficiently as possible.

Both papers tell us about the costs of synchronization and how the authors seek to minimize the impact of those costs, and exploit the characteristic uses of synchronization. For example, David Dice, one of the authors of [37], discusses the costs of HotSpot 1.6's locking techniques in a weblog post at [39], and the gory details can be found in the HotSpot source code, available at [53].

Current JVM releases use very cheap locking primitives in the simple and common case of a single thread working on an object without contention, called *biased locking*, and use increasingly heavy methods for more complex, but less common, situations. While lock biasing [37] and lock inflation [2] technically are two separate techniques, they are employed together by the HotSpot JVM.

We see in [54] that in the optimum *fast-path*[24], i.e., a lock biased towards the locking thread, locking entails essentially a few loads and compares to verify that the lock remains biased towards the current thread. This avoids the potentially high latency of the CAS operation in a common case.

CAS operations will be used when the owner of a lock is changed, but as long as a lock is uncontended only a single data word is necessary to maintain the lock state. A full lock data structure will not be used before a lock is contended, i.e., threads have to

[23] HotSpot is Sun's current implementation of the JVM. It was originally developed by Longview Technologies, LLC, which was acquired by Sun Microsystems in 1997. It became the default Sun JVM for Java 1.3. The latest version is 1.6, released December 11, 2006, as part of the Java 6 platform.

[24] The term *fast-path* means that the execution of complex synchronization code is skipped in the uncontended case. This is explained in [37].

wait, or when `wait()` is called on the monitor of an object.

Higher-level locking primitives

The `java.util.concurrent` package, formalized in JSR-166 [66], was added to Java in the 1.5 release. It contains a number of more advanced concurrency primitives, as an extension of the monitors specified by the language. It includes new versions of locking primitives such as locks, read/write locks and semaphores. It also adds data structures intended to aid concurrent programming, e.g., `ConcurrentHashMap`, a concurrent version of `java.util.Hashtable`, and `ConcurrentLinkedQueue`, a thread-safe queue based on a wait-free algorithm [69].

Our primary interests in the `java.util.concurrent` package are the locks, specifically `ReentrantLock` and `ReentrantReadWriteLock`. These classes provide a mutual-exclusion and a shared/exclusive lock, respectively. Both are implemented using the `AbstractQueuedSynchronizer` helper class. For a further discussion of the implementation of the `java.util.concurrent` package, we recommend [67].

These locks provide memory model semantics similar to `synchronized`, but do not force locking and unlocking to be balanced, or even to happen in the same method. Implemented mostly in Java, these primitives are not as finely tuned as Java synchronization, but they can still be a performant alternative where `synchronized` is not applicable. This is discussed in [38].

Chapter 3

Problem elaboration

Our task is to investigate the scalability properties of Derby and determine possible improvements. Our goal is to improve performance by modifying Derby to support read/write latches. A proof of concept implementation of shared latches will allow evaluation of the performance impact of latches on scalability. We will benchmark different modifications of Derby to understand the impact of changes to the locking system. We will limit the scope of our benchmarks to a read-only workload.

In this chapter we will describe the problem and our goals, our hypothesis, how we will run test benchmarks, and the available resources and environment for this project.

3.1 Scalability

We are interested in the scalability of Derby for a multithreaded workload. We will study how throughput, i.e., the number of transactions executed per second, develops for an increasing number of concurrent threads. We will run benchmarks both on a SMP system with 8 CPUs, and on a single CPU CMT system with 8 cores and simultaneous execution of 4 threads per core.

We will compare different modifications to Derby that have been made to improve performance. The impact of these patches on benchmark results will be evaluated. We will also run benchmarks with both version 1.5 and 1.6 of Sun's implementation of the Java Virtual Machine. This is intended to show the performance impact of the enhancements to monitors and synchronization in the 1.6 JVM.

Applying the DERBY-1704 [29] patch, described in Section 2.2.4, to the development trunk of Derby will allow us to benchmark a version of Derby where the problem of contention in the lock manager has been reduced. This should make the problem of latch contention more obvious. We will benchmark a version of Derby with the DERBY-1704 patch applied to the current development trunk. This will allow us to evaluate the impact of latch contention, especially for the pages in the B-Tree index.

3.2 B-Tree access and page latching

One of our conclusions in the autumn project was that the way latches are implemented in Derby is inefficient. We will investigate the effect the implementation of latches outside the lock manager has on multithreaded performance.

We also concluded that an implementation of shared latches should improve performance for latch-intensive workloads. The root node of the B-Tree index is a significant point of contention. All threads accessing the database through the index have to obtain an exclusive latch on the root node.

Our hypothesis is that an implementation of read/write latches should provide a significant performance gain for a high number of concurrent clients when running a read-only workload, because the bottleneck effect at the root node can be reduced or removed. If shared read-only access to the root node is allowed, for a read-only workload, threads do not have to wait to obtain exclusive access.

We will provide an experimental implementation of shared latches and benchmark the modified version of Derby. This will allow us to study how reduced contention on the B-Tree index affects performance.

3.3 Workload

As our task specifies, we study the performance and scalability of Derby for a read-only workload. We use a very simple workload, where transactions run a single `SELECT` operation. We are primarily interested in the cost of B-Tree index lookups, so we choose an index intensive workload. Therefor we use a `SELECT` statement that selects only the primary key for a single row.

The primary key is indexed[1], i.e., the actual database does not have to be read if an indexed access method is used, because the primary key is contained in the index. Since only the value of the primary key column is selected, an index lookup to see if the value is present is sufficient to execute the query. This access path will be preferred by the optimizer.

We define a database schema with rows of approximately 100 bytes. Each row has an `INTEGER` primary key value and a `CHAR` value. We will use a test database with 100 000 rows for our benchmarks. The read-only benchmark retrieves the indexed primary key for random values in the valid key range. The details of the benchmark application are described in section 4.1.

3.4 The B-Tree index

Knowledge about the number of rows to be indexed allows us to reason about the size and structure of the B-Tree index for the primary key in our test database.

[1] Derby creates indexes for primary key values automatically. This is because indexes are used to enforce the primary key constraint. Such indexes are called *backing indexes*.

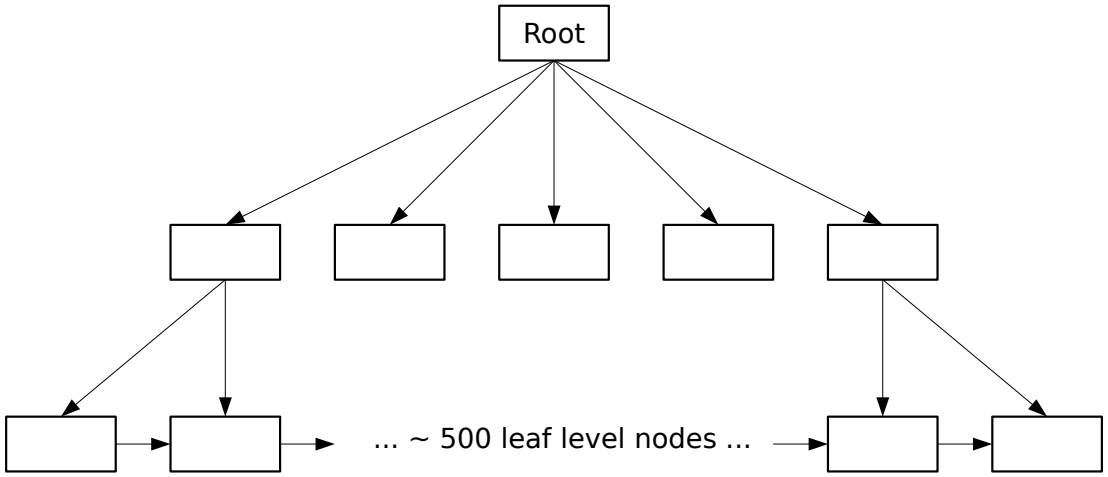

Figure 3.1: B-Tree structure

The block size used for the B-Tree nodes in Derby is 4KB. An index entry must contain a 4 byte integer and a pointer to a block at the next level, or to the data block for leaf nodes. Allowing for at least 8 bytes for the pointer, and allowing for some overhead, give an estimated entry size of 20 bytes per index record. 20 byte entries give a fanout of $\frac{4KB}{20B} \approx 200$. For 100 000 rows, this means that the leaf level will have at least $\frac{100000}{200} = 500$ blocks.

Since the leaf level is is at least 500 blocks, this requires more entries at the next level than the root block alone can hold. There must therefore be an intermediate level, i.e., the B-Tree will have 3 levels.

If we have 500 leaf nodes, this requires at least $\lceil \frac{500}{200} \rceil = 3$ second level nodes. If the second level nodes are half full, it would require 5 nodes. If the leaf nodes are not full, i.e., there are significantly more than 500 leaf nodes, more than 3 second level nodes would also be needed. This of course assumes that there are not enough leaf nodes to warrant a 4th level, with a fanout of 200 this would mean more than about $200^2 = 40000$ leaf nodes. Figure 3.1 illustrates a possible 3 level B-Tree structure, with 5 blocks on the second level.

If the leaf level blocks are half full, 1000 leafs would be needed. Up to 1000 leaf blocks can be supported by 5 second level blocks. Depending on the order in which keys are inserted, and the algorithm for inserting and splitting, blocks may end up either full or half full, or in an arbitrary state in between for random inserts.

Inserting data into our test database is done sequentially in increasing key order. We have investigated the Derby source code to understand how B-Tree inserts are performed. For splits when the key to be inserted is the highest value in the block, the B-Tree insert algorithm in Derby splits the block by only moving the highest key to the new block[2]. This approach to inserts is used regardless of what B-Tree level the insert happens at. This means that the blocks in the B-Tree index for our test database will be full, except for the rightmost blocks and the root node.

[2]This is implemented in the classes `BranchControlRow` and `LeafControlRow` in the package `org.apache.derby.impl.store.access.btree` in the Derby source code [8].

3.5 Test systems

We will give a brief description of the two systems we have used to benchmark Derby in this project. The systems were made available to us by Sun Microsystems, Inc. in Trondheim. One is a traditional multiprocessor machine, and the other a multicore CMT system.

Both systems were running the Solaris 10 [91] operating system. For our benchmarks we used versions 1.5 and 1.6 of Sun's JVM.

3.5.1 Sun Fire V880

The Sun Fire V880 [97] is a SMP system with support for up to 8 CPUs. We used a configuration with 8 CPUs and 32GB main memory. The CPUs are 1.2GHz UltraSPARC III [102] processors. The UltraSPARC III is a 64 bit CPU based on the SPARC V9 architecture. The SPARC V9 architecture uses big-endian instructions, but can access data both in little-endian and big-endian mode.

3.5.2 Sun Fire T2000

The Sun Fire T2000 [96] server has one UltraSPARC T1 [103] CPU, also known as "Niagara". The T1 is the latest[3] UltraSPARC processor. It is based on the Ultrasparc Architecture 2005 specification, which is compliant with the SPARC V9 architecture. The T1 is both a multi-core and a multithreading CPU. It has 8 cores and each core can execute 4 simultaneous threads. This means that the T1 can execute 32 simultaneous threads. Sun has named its CMT technology "CoolThreads". The T1 CPU and the T2000 server architecture are described in [95]. Our test system was equipped with 16GB of main memory.

Each core is simpler than a contemporary CPU, and the single-thread performance offered is lower than for other recent processors, e.g., compared to the single threaded performance of the UltraSPARC III CPUs in the V880. For multithreaded workloads however, the T1 will provide good throughput. The T1 can only be used in single CPU configurations, SMP is not supported. The T1 processor architecture and source code has been released under an open source license in the OpenSPARC [79] project.

One weakness with the T1 architecture is that the CPU only has a single floating point unit. This means that all threads performing floating point operations have to share the same FPU. I.e., floating point operations are serialized and performance is poor. Thus the T1 is not suited for scientific or other applications that require good floating point performance. For its inteded server market, however, this is not a problem since applications rarely rely on FPU performance. E.g., webservers and databases are typically multithreaded applications that require good integer performance. This makes the T1 well suited for use in such systems.

[3]The successor to the T1, known as T2, is planned for a launch in the second half of 2007. The T2 will have 8 cores, each capable of executing 8 threads, for a total of 64 simultaneous threads.

3.6 Environment

For this project we have used workstations provided by IDI, NTNU. We have used mostly free/open software and tools. We will describe our available resources and some of the important software used in this project.

3.6.1 Workstations

IDI has provided us with office space in a computer lab and workstations. Our development workstations are 3GHz Intel Pentium 4 machines with 1GB of main memory. We have used Ubuntu Linux [101] as the operating system for these workstations. For Java we have used Sun's JDK implementations for Linux. We have used Java versions 1.5 and 1.6 for development and testing. Building Derby also requires Java 1.3 and 1.4, and some additional libraries.

3.6.2 Software

To aid us in the work with the Java source code for Derby, and the benchmark application, we have used two different Java IDEs[4]. For revision control we have used Subversion. We will give a brief description of these tools here.

NetBeans

NetBeans [76] is an open source Java IDE developed by Sun Microsystems. NetBeans is also a framework for development of graphical Java applications. In addition to the expected IDE features, NetBeans provides a powerful profiler. The profiler can be of great help in the analysis and optimization of Java applications.

The NetBeans Profiler reports run-time statistics on CPU and memory usage, and can be set up record timing information for specific methods. Information on active threads is also provided, and thread state information is collected over the lifetime of all threads. It is possible to attach the profiler to an application running in a remote JVM. When the profiler is not attached, there is no instrumentation overhead.

IntelliJ IDEA

IntelliJ IDEA [56] is a proprietary and commercial Java IDE by JetBrains. IntelliJ IDEA provides a powerful source code editor with support for "intelligent" code completion and code refactoring. It has a powerful debugger with a graphical interface.

The debugger can be attached to an external JVM running in debug mode. Breakpoints can be set or code executed step by step while the execution stack trace and state of

[4] An IDE, short for Integrated Development Environment, is an application that simplifies software development by integrating a source code editor, compiler and debugging tools. IDEs also often support version control and integration with other software development tools.

objects and variables can be explored. This has allowed us to attach the debugger to a JVM running Derby and study execution in great detail.

We have used the time limited demo version of IntelliJ IDEA for this project.

Subversion

For revision control for our files we have used the Subversion (SVN) [94] Version Control System (VCS). SVN is also used for version control by the Derby project, and the Derby source code is hosted at the Apache SVN repository.

SVN is developed by CollabNet, Inc. and released under the open source Subversion License[5]. SVN is designed as a replacement for the traditional CVS. It has most of the features from CVS. SVN also has a more extensive and flexible feature set. This includes better metadata support, with versioning.

[5]The Subversion License is a modification of the BSD License. Reuse and redistribution is freely permitted if the license is retained and distributed with the modified version. Some trademark clauses apply.

Chapter 4

Implementation

We have implemented a benchmark application for the workload described in Section 3.3, and an experimental patch for shared latches. In this chapter we will present our implementations. The benchmark application is used to test our modified version and several other variants of Derby.

4.1 Benchmarks

For benchmarking we have used a simple test database and a Java application. This section describes the workload and the benchmark application.

4.1.1 Test database and workload

For benchmarking we use a simple database definition. The test database has one table with ∼100 byte rows. The SQL query for creating this table is included in Listing 4.1.

```
CREATE TABLE test(
        A INTEGER NOT NULL,
        B CHAR(96) FOR BIT DATA NOT NULL,
        PRIMARY KEY(A)
);
```

Listing 4.1: SQL for test database creation

We have used a default table size of 100000 rows. This is tunable in the benchmark application. The workload is implemented as a tunable number of threads, each running transactions consisting solely of executing the query

```
SELECT a FROM test WHERE a = ?;
```

for random values within the range of `a`. This is run in a loop a configurable number of times per active thread.

4.1.2 Benchmark application

We have implemented a benchmark application called *selectload*. The program consists of a main class called `Select`, an abstract `Worker` class with the extension classes `SelectWorker`, `UpdateWorker` and `DeleteWorker`, and the `TransactionStats` class. The source code has been enclosed with this report in a .zip file, see Appendix C.1.

The `Select` class is the main class of the application. It is responsible for initializing the test database if specified. Before the benchmark run there is an optional "warmup" phase. This consists of first executing a `SELECT * FROM test;` query, and then running a warmup `Worker` thread for the specified number of transactions. This warmup phase should ensure that the database and index are cached both by Derby and in the operating system's file system cache.

The mentioned `SELECT` query used for the read-only select workload is implemented by the `SelectWorker` class. This is the default workload for the benchmark application. We also implemented the classes `UpdateWorker` and `DeleteWorker` to study how `UPDATE` and `DELETE` queries are executed by Derby.

Each `Worker` thread has a `TransactionStats` object that is used to keep track of statistics for the number of transactions executed, time elapsed, response time, etc. At the end of a benchmark run statistics for all `Worker` threads are collected and aggregated.

The total time that threads have spent waiting to latch a page can optionally be collected on a per-page basis. Note that the total time spent waiting to obtain latches may be higher than the elapsed wall clock time, if many threads are waiting to latch the same page at the same time.

Measuring the latch wait time was made possible with a modification of Derby, using the debug system, which, in combination with our benchmark application, allows for the collection of the time spent in `wait()`[1] calls on page objects to obtain latches. I.e., only the time spent in `wait()` calls for a contended latch is measured, not the time to obtain an uncontended latch. This modification of Derby is included with our shared latches patch.

4.2 Shared latches

As described in Section 2.1.4, Derby page consumers latch pages to receive exclusive access to the in-memory page data structure. The logic for latching is implemented in the `org.apache.derby.impl.store.raw.data.BasePage` class. We decided to experiment with shared latches after preliminary benchmarks, the results of which will be presented in Section 5.1, revealed very large wait times spent in the `BasePage.setExclusive()` method, escpecially for B-Tree root pages.

This section will describe the goals for our implementation of shared latches, some of the challenges met during programming, and the resulting patch with support for shared latches. This patched version of Derby was later used for benchmarking to evaluate the performance impact of shared latches.

[1] The use of `wait()` when obtaining latches in Derby was explained in Section 2.1.4. For more about the `wait()` and `notify()` methods in Java thread synchronization, see [98].

4.2.1 Implementation goals

Our stated goal to "*explore* possible improvements" guided our implementation towards a simple one. We focused on making Derby work "good enough" to run our simple workload – with little emphasis on the requirements of production quality code, or even the ability to run workloads other than our own.

As we wanted to measure the impact of removing the latch bottleneck in the Derby B-Tree implementation, our effort would be concentrated on the latch implementation in `BasePage` and the B-Tree code in `org.apache.derby.impl.store.access.btree`.

In the spirit of "do the simplest thing that possibly might work", and in the hope of getting an indication of the cost of latching by comparing regular Derby to a Derby entirely without latching, an initial attempt was made to make the methods `BasePage.setExclusive()` and `BasePage.releaseExclusive()` no-ops, and thus in effect disable latching entirely.

However, we found that even with a read-only workload, the implementation of `BasePage` required some latching to be done.

4.2.2 Challenges

One of the significant problems with the change from exclusive ownership of pages to shared ownership is the inherent assumption of single-theaded access while a page is latched.

The original simplified latching code, which did not involve the full locking subsystem to implement latches, actually piggybacked latching status on a member variable of the `BasePage` class, `BasePage.owner`. When a page was latched, `owner` was set to point to the `ContainerHandle` representing the opened B-tree instance to the page consumer.

The associated logic to determine if a page was latched or not was:

- `owner` $\neq$ `null` $\Rightarrow$ page is latched.
- `owner` $=$ `null` $\Rightarrow$ page is unlatched.

As the `owner` field is used in subsequent calls to `BasePage` while the page is latched, not setting the owner field, as we did when making latch operations a no-op, caused `NullPointerException`s to be thrown in testing.

While not a showstopper, these and other assumptions around the use of the `owner` field meant that we had to investigate how the field was used. We also looked for other assumptions, regarding thread safety and data sharing, that would no longer be valid when multiple readers may be accessing a page simultaneously.

The other significant thread safety assumption was a page internal `org.apache.derby.iapi.services.io.ArrayInputStream` instance, one for each `BasePage` instance. It is used to read data values from the internal `byte[]` data array, and one part of its internal state is an array index pointer which is moved as data is read from the

array. Multiple readers using the same pointer will "leapfrog" through the data and bad things will happen.

A remedy for this problem was required if shared latching was to work at all. We tried various ways to handle this, including creating a new `ArrayInputStream` for each method invocation requiring one, giving each thread a private `ArrayInputStream` using `java.lang.ThreadLocal`, and pooling and recycling the `ArrayInputStream` instances. The patch as delivered uses the first method - creating new instances for each invocation.

Changing `BasePage` to support both shared and exclusive latches started with embedding a `ReentrantReadWriteLock` and modifying the methods `setExclusive()`, `setExclusiveNoWait()`, and `releaseExclusive()` in `BasePage` to use the exclusive write lock provided by `ReentrantReadWriteLock`. Corresponding `setShared()`, etc., methods were created to use the shared read lock. This was the easy part.

Modifying the B-Tree implementation to use shared latches was done without regard for correctness in complex cases involving tree modification. This should not be a problem in the case of a read-only workload, as no structure modifications are necessary.

The interfaces and abstraction layers separating the B-tree code from the latching methods of `BasePage` also required modification to support two different modes of latching. This involved modifying method signatures to add a `boolean shared` flag, and some modification of the logic of methods that maintain internal latching state.

4.2.3 Result

The end result is available as a patch produced with the `svn diff`[2] command. It can be applied to the Derby source code, revision 501369, with the `patch` utility [84]. The patch is enclosed with this report, see Appendix C.2 for more information.

This is an experimental patch, database consistency is not guaranteed, and it is not appropriate for inclusion in the Derby project in its current form.

In addition to the changes necessary to implement SX latches, we also added some instrumentation code to measure the time spent waiting for latches. As we utilize features only available in Java 1.5 and Java 1.6, other changes were necessary to make Derby compile cleanly with Java 1.5 compilers. This was unrelated to our goal, but still necessary. Java 1.5 and 1.6 compatibility was necessary to enable us to experiment with modern Java language features, such as atomic operations and `ReentrantReadWriteLock`.

The changes made to the Derby code are summarized by the output from the `diffstat` utility [41], included in Listing 4.2. The file `BasePage.java` received the largest amount of modification. This is natural, as it contained the original latching code, and it was also the most natural place to put latch instrumentation code.

The changes to the lock management code in `java/engine/org/apache/derby/impl/services/locks` are forward-ports of the `split-hashtables.diff` patch in

[2] `svn diff` is the SVN version of the `diff` [40] program, showing the difference between the local copy of a file and the latest revision in the repository, or between two specified revisions.

```
java/engine/org/apache/derby/iapi/store/raw/ContainerHandle.java                          |   19
java/engine/org/apache/derby/iapi/store/raw/Page.java                                     |   24
java/engine/org/apache/derby/impl/services/build.xml                                      |    8
java/engine/org/apache/derby/impl/services/locks/Deadlock.java                            |   41
java/engine/org/apache/derby/impl/services/locks/LockSet.java                             |  163 ++--
java/engine/org/apache/derby/impl/services/locks/LockSpace.java                           |    2
java/engine/org/apache/derby/impl/services/locks/LockTableVTI.java                        |   26
java/engine/org/apache/derby/impl/services/locks/SinglePool.java                          |  102 +-
java/engine/org/apache/derby/impl/store/access/btree/BTreeController.java                 |   15
java/engine/org/apache/derby/impl/store/access/btree/BTreeCostController.java             |    6
java/engine/org/apache/derby/impl/store/access/btree/BTreeMaxScan.java                    |    2
java/engine/org/apache/derby/impl/store/access/btree/BTreePostCommit.java                 |    7
java/engine/org/apache/derby/impl/store/access/btree/BTreeScan.java                       |   11
java/engine/org/apache/derby/impl/store/access/btree/BranchControlRow.java                |   15
java/engine/org/apache/derby/impl/store/access/btree/ControlRow.java                      |   54 -
java/engine/org/apache/derby/impl/store/access/btree/D_BTreeController.java               |   12
java/engine/org/apache/derby/impl/store/access/btree/LeafControlRow.java                  |    9
java/engine/org/apache/derby/impl/store/access/btree/OpenBTree.java                       |   12
java/engine/org/apache/derby/impl/store/access/btree/index/B2IFactory.java                |   15
java/engine/org/apache/derby/impl/store/access/btree/index/B2IRowLocking3.java            |    6
java/engine/org/apache/derby/impl/store/access/btree/index/B2IUndo.java                   |   12
java/engine/org/apache/derby/impl/store/build.xml                                         |    6
java/engine/org/apache/derby/impl/store/raw/data/AllocPage.java                           |    6
java/engine/org/apache/derby/impl/store/raw/data/BaseContainer.java                       |   49 -
java/engine/org/apache/derby/impl/store/raw/data/BaseContainerHandle.java                 |    9
java/engine/org/apache/derby/impl/store/raw/data/BasePage.java                            |  425 ++++------
java/engine/org/apache/derby/impl/store/raw/data/CachedPage.java                          |   21
java/engine/org/apache/derby/impl/store/raw/data/FileContainer.java                       |   61 -
java/engine/org/apache/derby/impl/store/raw/data/RAFContainer4.java                       |    2
java/engine/org/apache/derby/impl/store/raw/data/StoredPage.java                          |   43 -
java/shared/build.xml                                                                     |   10
java/shared/org/apache/derby/shared/common/sanity/SanityManager.java                      |   10
java/testing/org/apache/derbyTesting/functionTests/tests/derbynet/DerbyNetAutoStart.java |    4
java/testing/org/apache/derbyTesting/unitTests/store/T_QualifierTest.java                 |    6
tools/ant/properties/compilepath.properties                                               |    2
tools/ant/properties/modern.properties                                                    |    1
36 files changed, 676 insertions(+), 540 deletions(-)
```

Listing 4.2: `diffstat` output showing the files changed as part of the work

DERBY-1704. We added these to our source tree manually as they were not yet in the mainline trunk when we started our work. We still wanted to remove the known bottleneck in the lock management subsystem to enable us to better observe the behavior of the latches.

`java/engine/org/apache/derby/impl/store/access/btree` contains the Derby B-Tree implementation, and changes in this part of the source are naturally related to the use of shared latches in B-Tree traversal, as well as the associated interface definition changes which required changes to interface consumers.

4.3 Summary

We have implemented a benchmark application, and a patch that adds experimental support for shared latches to Derby.

The benchmark application will be used to test the multithreaded scalability of different versions of Derby for a read-only workload that selects a single, indexed primary key value.

Support for shared latches was implemented to study the performance impact of removing the bottleneck at the root node of the B-Tree index. The experimental patch will allow us to run benchmarks, but it is not ready for production use. The patch uses `java.util.concurrent.locks.ReentrantReadWriteLock` to implemented support for shared / exclusive latches, and requires Java 1.5 or newer versions.

Benchmark results will be presented in Chapter 5.

Chapter 5

Benchmark results

In this chapter we present the results of our benchmarks.

At first we did some preliminary benchmarks to evaluate the split hash tables patch from DERBY-1704 [29], described in Section 2.2.4. The changes introduced by this patch remove the Java level synchronization bottleneck in the lock manager, thus making the effects of latch contention more obvious. Measuring the wait time to obtain latches will show if latch contention is a problem.

After implementing experimental support for shared latches, we benchmarked three different versions of Derby, running in both Java 1.5 and Java 1.6. The versions tested are the "old" version of the Derby development trunk, where latching is done using the lock manager, a "new" version of the trunk that separates latches from the lock manager, and our experimental version with shared latches not using the lock manager.

All versions of Derby used for benchmarking have been compiled with the "insane" build option, i.e., they do not include asserts and debugging information. This code, available in a "sane" build is useful for testing and debugging during development, but it causes extra overhead. Therefor, "insane" builds are preferred in production environments when high performance is required. For more information about this, and how to compile Derby from source, see the Building Derby guide [18].

We also include some results from profiling Derby.

5.1 Preliminary benchmarks

These are the results from the preliminary benchmarks. The version of Derby used is the development trunk as of January, 2007, with the DERBY-1704 patch applied to split the hash tables in the lock manager. Note that some work was needed for the patch to apply cleanly, as the development trunk of Derby had changed since the patch was made available.

Benchmarks were run on both the 8 CPU V880 system and on the T2000 multicore system. Java version 1.6 was used for these benchmarks. A total number of 800 000 transactions were run for each test, i.e., the number of transactions executed per thread depended on the number of active threads. A warmup phase, executing 100 000 trans-

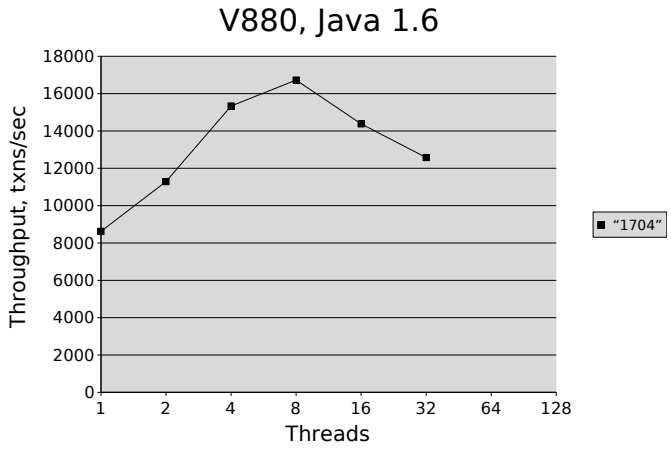

Figure 5.1: Throughput; Derby with "1704" patch; 8-way SMP system

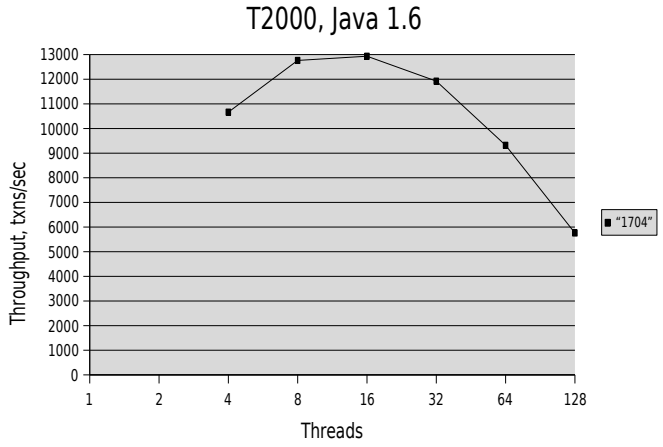

Figure 5.2: Throughput; Derby with "1704" patch; T2000 CMT system

actions, was used in all the tests. Timing information for latch wait time per page was collected.

5.1.1 Throughput and response time

Figure 5.1 shows the throughput, i.e., number of simple `SELECT` transactions executed per second, for a varying number of concurrent threads on the 8-way V880 system. Throughput increases until there is 8 concurrent threads, i.e., one thread per CPU. However, the scaleup when adding threads is limited. There is only a 30% increase in throughput for 2 threads compared to 1. The throughput for 8 threads is less than double the throughput for 1 single thread. When the concurrency level is increased beyond the number of available CPUs, there is a drop in throughput.

To study contention for a higher number of concurrent threads we also tested this version of Derby on the T2000 system. We ran benchmarks for up to 128 concurrent

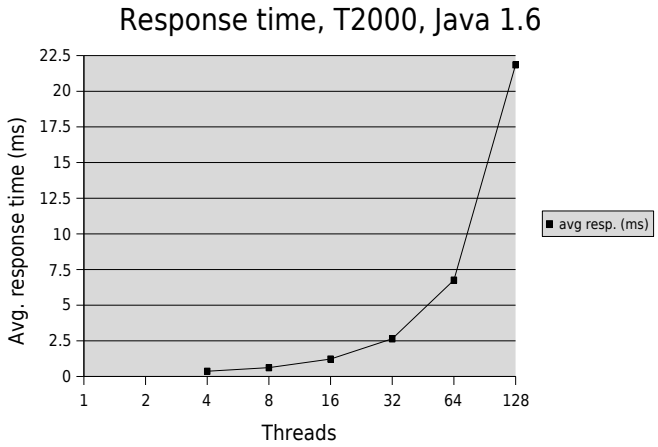

Figure 5.3: Response time; Derby with "1704" patch; T2000 CMT system

threads. The results are shown in Figure 5.2. We see that throughput increases up to 16 concurrent threads. For 32 threads there is a drop, and then throughput sharply declines for 64 and 128 concurrent threads.

Note that the increase in throughput from 8 to 16 concurrent threads is small. There is also a drop for 32 concurrent threads, even though the T1 CPU should, in theory, be able to execute 32 simultaneous threads.

The average response time for the tests on the T2000 systems have been graphed in Figure 5.3. The increase in response time is nearly linear[1] for up to 16 concurrent threads. When the number of active threads is increased above 32, the response time increases dramatically. This indicates that transactions spend a lot of time queued, waiting for latches or locks.

The raw numbers from these benchmarks are included in Appendix A.

5.1.2 Latch wait time

For these benchmarks we also measured the aggregated wait time for threads to obtain latches on pages. This is optional with our benchmark application, but requires a modified version of Derby. As explained in Section 4.1.2, this is the time spent in `wait()` calls for a contended page, i.e., it does not include the cost of acquiring a latch in the uncontended case. Collecting the wait time for all contended pages will show if there is significant contention for specific pages. We will look at the results for a single thread, 2 threads and 32 threads for the 8-way SMP system.

For a single thread there is, obviously, no wait time to obtain latches. This is because there cannot be any contention to obtain a latch if there is only a single active thread. When increasing the number of active threads more wait time is observed. We have chosen to include results from 2 and 32 concurrent threads on the 8-way SMP system to illustrate this.

[1]Note that while the curve on the graph in Figure 5.3 may look exponential, the X axis is logarithmic – we benchmarked threads in numbers of powers-of-two. See Table in Appendix A

Page #	**Total wait time, µs**
Page(1,Container(0, 977))	647134
Page(129,Container(0, 977))	451776
Page(263,Container(0, 977))	11719
Page(393,Container(0, 977))	9729
Page(133,Container(0, 977))	7903
Page(486,Container(0, 977))	111
Page(512,Container(0, 977))	95
Page(505,Container(0, 977))	93
Page(183,Container(0, 977))	90
Page(371,Container(0, 977))	87

Table 5.1: Latch wait time; 2 threads; 8-way SMP system

Page #	**Total wait time, µs**
Page(1,Container(0, 977))	1130868904
Page(129,Container(0, 977))	2343688
Page(263,Container(0, 977))	1807935
Page(133,Container(0, 977))	1485839
Page(393,Container(0, 977))	968103
Page(273,Container(0, 977))	831267
Page(210,Container(0, 977))	19860
Page(250,Container(0, 977))	13898
Page(90,Container(0, 977))	13038
Page(355,Container(0, 977))	11890

Table 5.2: Latch wait time; 32 threads; 8-way SMP system

The top 10 contended pages for the test with 2 concurrent threads on the 8-way SMP system is included in Table 5.1. The elapsed real time for this test was 70.847 seconds. The aggregated wait time for the most contended page was 647 134 µs, or 0.647 s. Note that page number 1 in a B-Tree container is the root node, as explained in Section 2.1.4.

The total time spent waiting to obtain latches in this test was 1130ms, or 1.13s. The time spent waiting for the root node thus accounts for $\frac{0.613s}{1.13s} \approx 0.542$, or 54.2% of the time spent waiting for latches. Note that the time spent waiting for the second most contended page, which is page #129, was 0.452 s. This is of the same order of magnitude as the time spent waiting for latches on the root node.

For 32 threads the wait time for the top 10 contended pages are shown in Table 5.2. There is significantly more waiting time for 32 concurrent threads than for 2 threads. The aggregated waiting time for the most contended page, again page number 1, works out to 1 130 868 904µs, or 1130.869s. The elapsed wall clock time for the benchmark was 63.613 seconds.

For this case, with 32 active threads, the total "thread time", i.e., aggregated run time for all threads, is $32 \cdot 63.613s = 2035.616s$. The fraction of total run time spent waiting to obtain a latch on the root node then is $\frac{1130.869s}{2035.616s} \approx 0.555$, i.e., 55.5% of the total time.

The total time spent waiting to obtain latches in this case was 1 140 910ms, or 1140.091s. Thus the fraction of latch wait time that is spent waiting for the root node is $\frac{1130.869s}{1140.091s} \approx 0.992$, or 99.2% of total latch wait time is for the root node of the B-Tree. Note that the time spent waiting to latch the second most contended page, again page #129, was 2.34s.

Figure 5.4 illustrates the results from the test with 32 threads, included in Table 5.2, on a chart with a logarithmic scale. This cleary shows that the wait time for the most contended page is several orders of magnitude larger than for the second most contended page. Then there are several pages with wait time of the same order, before there is another considerable drop-off.

The time spent waiting for latches is related to the response time of transactions. If many transactions wait to obtain a latch on the same page, a considerable amount of time will be spent in the queue, as the time to obtain a latch is proportional to the queue length[2]. The apparent contention for the root page of the B-Tree is also illustrated by the response times of the transactions. For 2 concurrent threads on the 8-way system the average response time was 0.175 ms, for 32 threads it had increased to 2.525 ms. As was illustrated by Figure 5.3, the response time suffers even more when the concurrency level is increased above 32, but the times obtained on the T2000 system are not directly comparable.

5.2 Shared latches benchmarks

We benchmarked our experimental modification of Derby against an "old" version of the development trunk, without the patch that removes latching from the lock man-

[2]The queue to obtain a latch in the version of Derby used in these benchmarks is a fair queue, since the lock manager is used for latches. Even if a non-fair queue was used, the *average* time spent waiting would be proportional to the number of waiters.

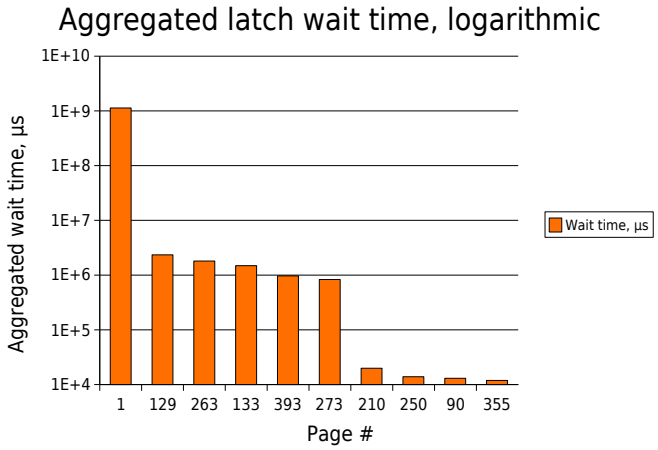

Figure 5.4: Latch wait time; 32 threads; 8-way SMP system

ager, and a clean version of the development trunk with the newest modifications, i.e., latching is done without the lock manager and the lock manager code has been cleaned up.

We label our version "SX", for Shared/eXclusive latches. The unmodified versions are named "OLD" and "NEWCLEAN" respectively. "OLD" corresponds to SVN revision 500250 of the Derby source code, "NEWCLEAN" is revision 521680, and our "SX" patch was made with revision 501369 as the starting point.

The benchmark results are presented here as graphs. The raw numbers are included in Appendix B.

5.2.1 8-way SMP results

Here we present the benchmark results obtained on the 8-way SMP V880 system. Benchmarks were run for Java 1.5 and Java 1.6.

Java 1.5

Benchmark results for Java 1.5 on the V880 are shown in Figure 5.5. We can see that for a single thread the NEWCLEAN version of Derby provides the best throughput, 8209 transactions per second, followed by the OLD version, 7273 transactions per second, and our experimental version, SX, 6362 transactions per second.

At 4 concurrent threads our SX version provides the best throughput, at 11789 transactions per second, compared to 11415 and 10745, respectively, for the OLD and NEWCLEAN versions. The peak throughput is at 8 concurrent threads, where the SX version executes 15032 transactions per second.

For a higher number of concurrent threads there is a drop in throughput for all versions. The drop in throughput is less for the SX version than for the other versions. The throughput curve for SX is closer to the ideal flat line for max throughput when increasing concurrency. At 128 threads the throughput is 12928 transactions per sec-

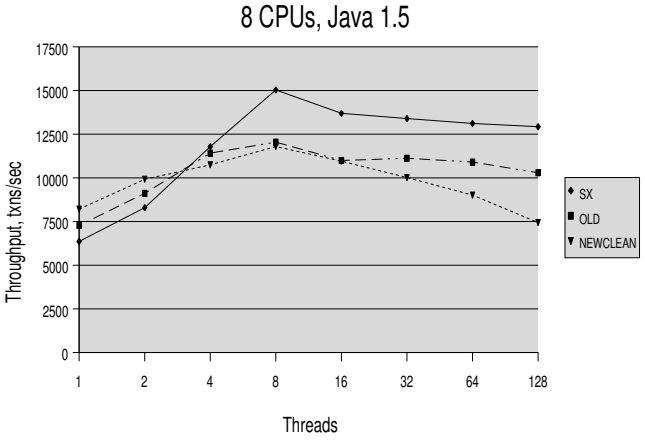

Figure 5.5: Throughput; 8-way SMP; Java 1.5

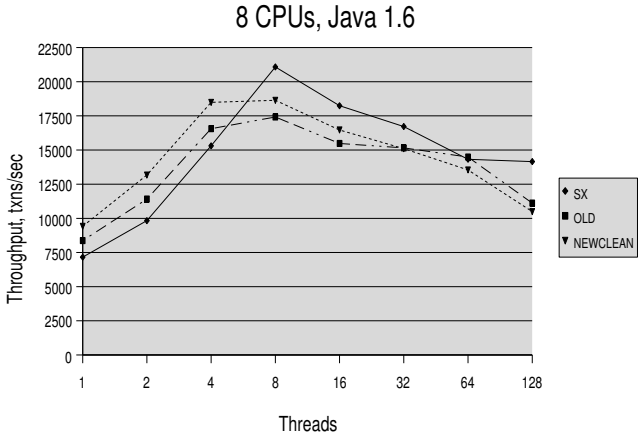

Figure 5.6: Throughput; 8-way SMP; Java 1.6

ond for the SX version, 10291 transactions per second for the OLD version, and the NEWCLEAN version gives the lowest throughput at 7444 transactions per second.

Java 1.6

Results for Java 1.6 are shown in Figure 5.6. Java 1.6 provides higher throughput for all versions and concurrency levels. The relative performance of the versions is similar in the single-threaded case. The NEWCLEAN version provides the best throughput, at 9417 transactions per second, the OLD version 8374 transactions per second, and our SX version 7162 transactions per second.

At 8 concurrent threads the SX version provides the best throughput at 21084 transactions per second. The NEWCLEAN and OLD versions follow, with 18642 and 17419 transactions per second, respectively. Again the peak throughput is reached for 8 concurrent threads.

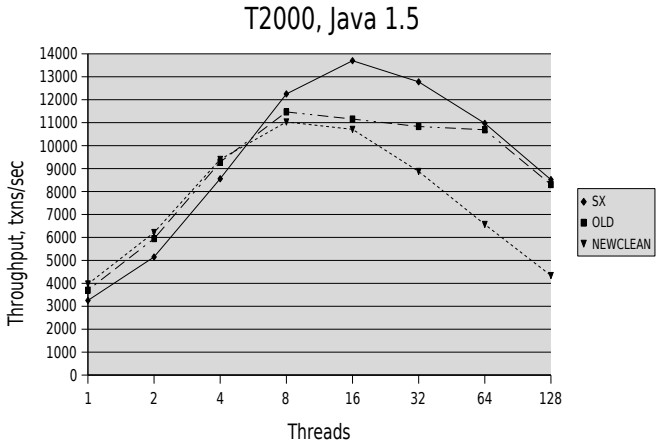

Figure 5.7: Throughput; T2000 CMT; Java 1.5

The throughput drops off when increasing the number of threads above 8. At 128 concurrent threads the throughput is 14156 transactions per second for the SX version, 11117 transactions per second for the OLD version, and the NEWCLEAN version is again lowest at 10492 transactions per second.

5.2.2 T2000 CMT results

We ran the same benchmarks on the T2000 multicore system, both for Java 1.5 and 1.6. The results are presented here.

Java 1.5

The benchmark results for Java 1.5 on the T2000 system is shown in Figure 5.7. For a single thread the NEWCLEAN version is fastest, at 3971 transactions per second, followed by OLD at 3705 and our SX version at 3254. Note that this is about half the throughput of the corresponding benchmark on the 8-way SMP system.

Peak throughput is reached at 16 concurrent threads for all versions. At this concurrency level the SX version is the best at 13701 transactions per second, followed by the OLD version at 11158 and the NEWCLEAN at 10708.

Througput drops off for all versions as the concurrency level is increased beyond 16. At 128 threads, the SX version executes 8518 transactions per second, closely followed by the OLD version at 8305 transactions per second. There is a considerable drop in performance for the NEWCLEAN version, which manages only 4342 transactions per second for 128 concurrent threads.

Java 1.6

Figure 5.8 shows the benchmark results with Java 1.6 on the T2000 system. For a single thread, the situation is similar to the Java 1.5 results, but the throughput a little higher.

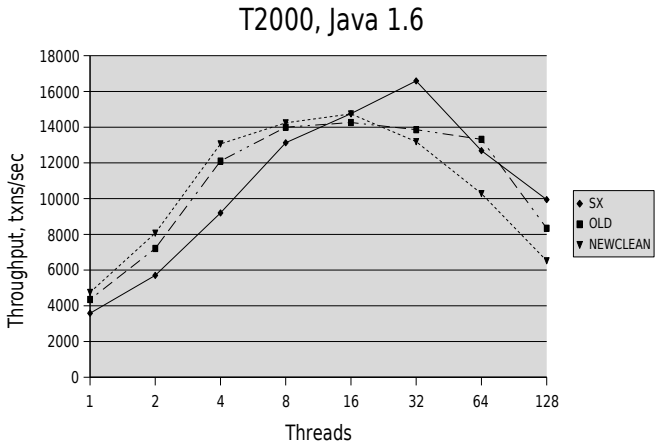

Figure 5.8: Throughput; T2000 CMT; Java 1.6

The NEWCLEAN version is the best at 4752 transactions per second, then the OLD version at 4354, and our SX version at 3585.

At 16 concurrent threads the SX version is the fastest, at 14756 transactions per second, compared to 14744 for the NEWCLEAN version and 14263 for the OLD version. This is the peak throughput both for the NEWCLEAN and OLD versions. Our version, on the other hand, reaches peak throughput at 32 concurrent threads when running in Java 1.6. Then the throughput is 16588 transactions per second for our SX version, while it has dropped to 13188 and 13858, respectively, for the NEWCLEAN and OLD versions.

For more than 32 threads throughput decreases for all versions. At a concurrency level of 128, the SX version gives a throughput of 9944 transactions per second. This is followed by the OLD version at 8341, and the NEWCLEAN at 6522 transactions per second.

5.3 Profiling results

As discussed in Section 5.2, we see that in the single-threaded cases our SX patch is significantly outperformed by the baseline, i.e., OLD, build as well as a newer trunk build, i.e., NEWCLEAN.

A performance difference of $\frac{8374-7162}{7162} = 16.9\%$ is significant, and warranted further investigations. We used the NetBeans Profiler [76], explained in Section 3.6.2, in an attempt to pinpoint how and where the SX version spent more time and resources than the OLD and NEWCLEAN versions.

5.3.1 Gathering the data

Figure 5.9 shows a screenshot of the profiler while browsing a profiling snapshot of a run with our SX build. This run executed just 20 000 transactions in a single thread.

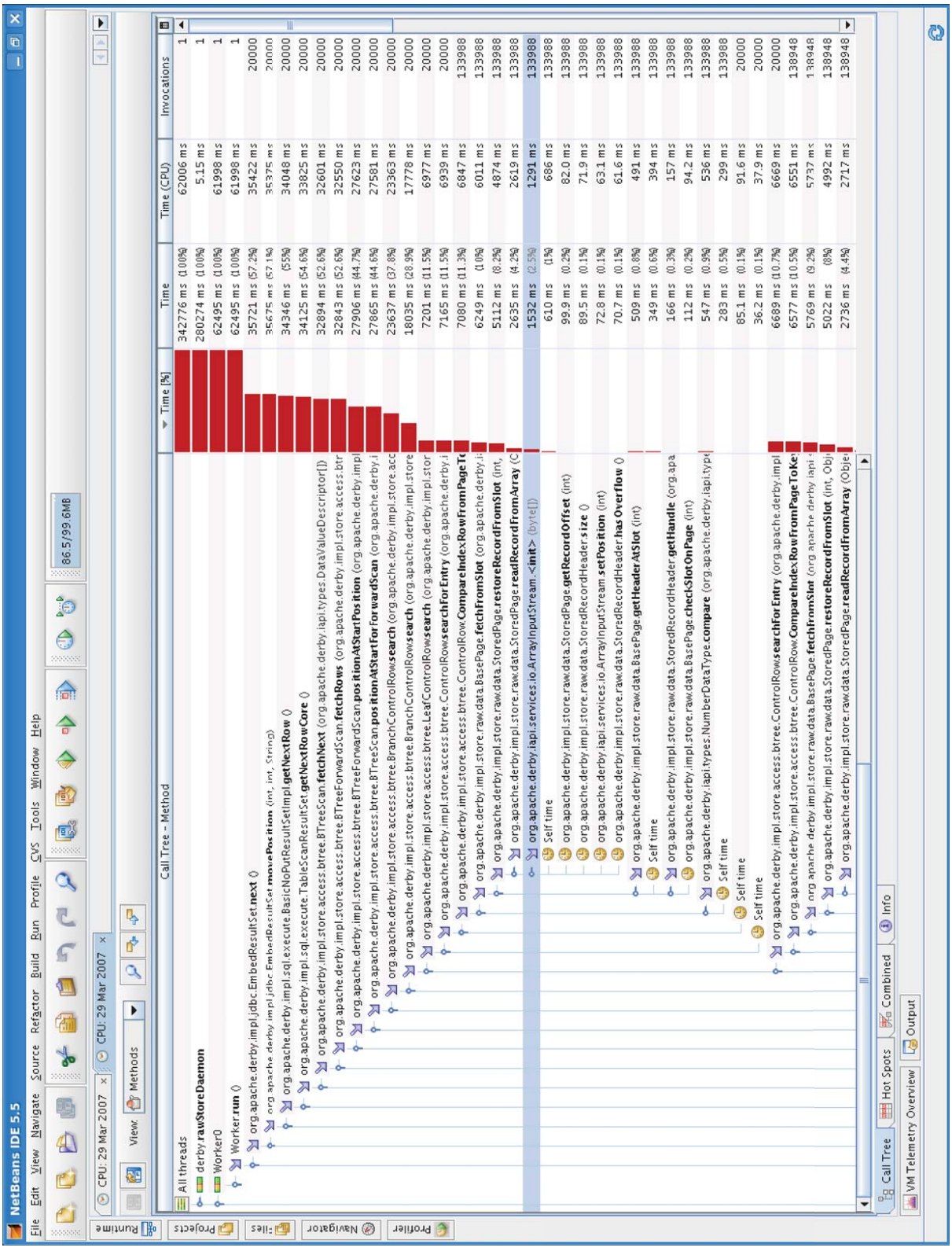

Figure 5.9: Netbeans profiler; drilldown to `ArrayInputStream.<init>()`

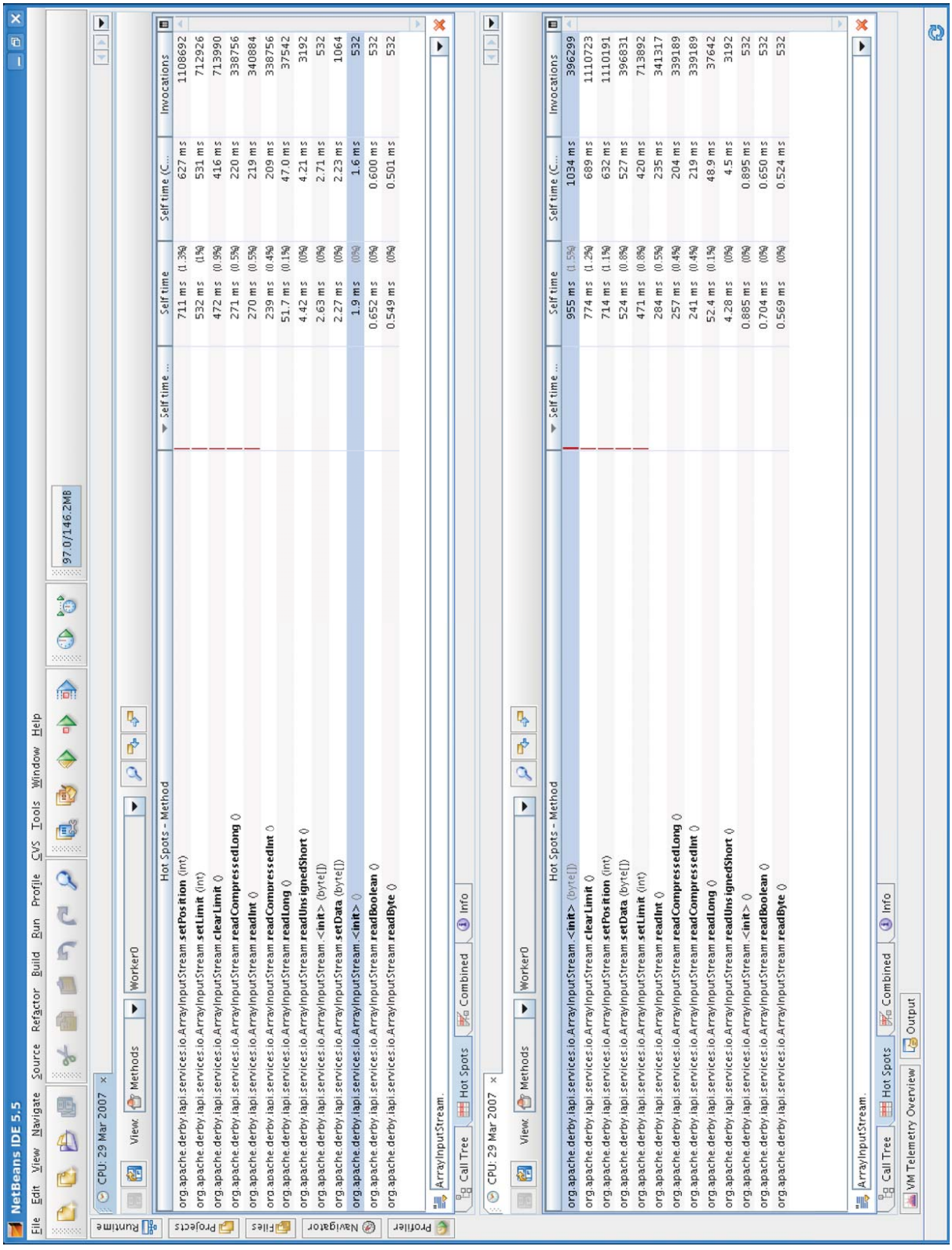

Figure 5.10: Netbeans profiler; comparing time spent in the `ArrayInputStream` class between OLD (above) and SX (below)

This is much less than for the benchmark runs, but as profiling overhead causes very large slowdowns, even a relatively short run with 20 000 transactions takes a while to complete.

We made profiling snapshots of both the SX, OLD and NEWCLEAN builds executing 20 000 transactions, and they were all collected on the 8-way SPARC machine mentioned earlier. We also limited profiling to classes in the `org.apache.derby.*` package hierarchy, to reduce profiling overhead. The disadvantage of this is that time spent in the Java class libraries on behalf of Derby is not included in the timing statistics.

Profiling snapshots for the three different versions have been enclosed with this report. See Appendix C.3 for a description.

Our examination of the gathered data primarily focused on finding the differences between the SX and OLD versions, more specifically pinpointing any performance regressions from OLD to SX. The immediate observation is that the worker thread, i.e., Worker0, spends 55 372 ms of CPU time running the OLD version, and 62 495 ms running the SX version. This means that the SX version spent 62 495 - 55 372 = 7132 ms more CPU time, i.e., about 12.8% more, than the OLD version.

5.3.2 The cost of thread safety

As Figure 5.9 suggests, the creation of new `ArrayInputStream` objects was identified as a significant cost in the SX version, when compared to the OLD version.

We do not count the time spent allocating and garbage collecting these objects, as the profiling data do not include time spent in the allocator or garbage collector. The additional memory management may thus affect the total run time and CPU consumption of the tests, but it should not affect the CPU timing data collected by the profiler.

Figure 5.10 displays the difference in execution time spent inside methods of `ArrayInputStream` between the OLD and SX versions, as well as execution counts in both cases. Note that Figure 5.10 does not include the time spent in methods called by the listed methods, so the total cost of the listed method calls is higher.

This is illustraded by Figure 5.9, where 133 988 invocations of the `ArrayInputStream` constructor takes 1532 ms including subcalls, while Figure 5.10 shows that it takes just 955 ms for 396 299 invocations when its subcalls are not counted.

When counting the additional invocations of `ArrayInputStream.<init>`, we find at least 394 165 more invocations for SX compared to OLD. The total cost of these invocations appear to be 4161.42 ms[3].

This is a significant portion of the 7132 ms difference between OLD and SX, but there is still about 3000 ms to account for.

5.3.3 The cost of S/X latches

We then turn our attention to the latching code itself, i.e., `BasePage.setExclusive()` and its new partner `BasePage.setShared()`. In the OLD profiling snapshot, we

[3] $1532ms + 1383ms + 6.42ms + 195ms + 484ms + 152ms + 195ms + 214ms = 4161.42ms$

find 60 000 invocations of `setExclusive()`, with an aggregated time consumption of 635.5 ms [4].

In the profiling snapshot of the SX version, many, but not all, of the invocations of the `BasePage.setExclusive()` method have been replaced by `BasePage.setShared()`. For a fair comparison we summarize the time spent acquiring both S and X latches. In total, we count 141 ms [5] spent acquiring X latches, and 2195 ms [6] acquiring S latches. This adds up to a total of 2336 ms of CPU time for the same number of latch acquisitions.

The SX patch thus spent about 1700 ms [7] more than the OLD version on latch acquisitions alone.

Examining the time spent unlatching, we find that OLD spends a total of 1610 ms[8] releasing latches, while SX spends 1902 ms [9] releasing shared latches and 204.4 ms [10] releasing exclusive latches. This adds up to $(204.4ms + 1902ms) - 1610ms = 496.4ms$ more time spent releasing latches in the SX build compared to the OLD version.

The differences accounted for so far, i.e., 4161.4 ms (`ArrayInputStream.<init>`) + 1700.5 ms (acquiring latches) + 496.4 ms (releasing latches), amount to a total of 6358 milliseconds. This is about 89% of the total CPU time difference of 7123 ms between OLD and SX.

While looking for further differences is possible, we are approaching the point of diminishing returns. Looking for small differences apart from the large ones we already have identified in a large pile of numbers is very time consuming, so we have ended our search here.

5.3.4 Summary of profiling results

We have found that our changes to support shared latches have incurred a significant cost in the single threaded no-contention case, and identified two major expenses: Creation of `ArrayInputStream` objects (4 CPU seconds on top of original 55 second total CPU time for this workload), and more expensive latches (2.2 CPU seconds of additional CPU time compared to OLD - original latches used only about 2.2 seconds of CPU time in total).

The additional object initialization and the fact that our `ReentrantReadWriteLock` based latches are twice as expensive as the original simple latches thus explains nearly 90% of the additional CPU cost.

[4] Sum of time use in all invocation points: $206ms + 1.66ms + 157ms + 50.8ms + 220ms = 635.5ms$
[5] $4.86ms + 136ms = 141ms$, for 5024 invocations.
[6] $804ms + 570ms + 821ms = 2195ms$, for 54 976 invocations
[7] $2336ms - 635.5ms = 1700.5ms$
[8] $519ms + 550ms + 541ms = 1610ms$, for 60 000 invocations.
[9] $509ms + 702ms + 691ms = 1902ms$, for 54 976 invocations.
[10] $197ms + 7.4ms = 204.4ms$, for 5024 invocations.

5.4 Summary

The preliminary benchmarks have shown that the scalability of Derby is limited for a high number of concurrent threads. Measurements of the time spent waiting for contended latches show that the root node of the B-Tree index is a performance bottleneck at high levels of concurrency.

Benchmarks of the patched version with support for shared latches, against two other variants of Derby, show that the SX patch performs worse for a single thread and low levels of concurrency. The SX patch scales better at high levels of concurrency, and provides the highest peak throughput in all benchmarks. Running Derby with Java 1.6 provides significant performance gains compared to Java 1.5.

Profiling results show that most of the extra overhead in the shared latches version can be attributed to the construction of `ArrayInputStream` objects, and the higher cost involved in latch / unlatch operations when `ReentrantReadWriteLock` is used. Combined, these effects constitute nearly 90% of the extra time spent in the single-threaded invocation of Derby with the SX patch.

We will provide further comments and our analysis of these results in Chapter 6.

Chapter 6

Analysis

In this chapter we present our analysis of the benchmarks and results presented in Chapter 5. We will discuss the preliminary benchmarks, the benchmarks of our modified version of Derby with support for shared latches, and the profiling results.

We will give our interpretation and explanation of the results with basis in the concurrency theory and description of Derby presented in Chapter 2. Understanding the limiting factors for scalability will allow us to evaluate our experimental version of shared latches, and also suggest other possible improvements.

6.1 Preliminary benchmarks

The results from the preliminary benchmarks were presented in Section 5.1. Our goal for these benchmarks was to understand how the current version of Derby scales for a multithreaded workload. We are particularily interested in the cost of latching and its impact on multithreaded scalability.

Testing the "1704" version, explained in Section 2.2.4, with a latch intensive workload should give a good indication of the limits to scalability. The patched version reduces, but does not completely remove, contention in the lock manager, so the impact of latch cost should be more obvious. This is because the global synchronization point to obtain and release database locks is removed. Since the lock manager is used to set latches, the cost to obtain a latch is high. This was explained in Section 2.2.6.

The `SELECT` only workload we have used for our benchmarks does an index lookup on a single record only. Since only the primary key value is selected, it can be returned directly after the index lookup, the database does not have to be touched. Database locks has to be obtained, but for a `SELECT` only workload these will be shared locks – there should be no conflicting database locks in our benchmarks. However, there may be overhead and contention in the lock manager for setting and releasing database locks, because of the synchronization in the lock manager.

Since we are running a `SELECT` only workload, there should not be any structure modifications in the B-Tree index. All index operations will be read only searches. The key values for lookups are random, so this should be a "best case" workload with regards to B-Tree contention.

This background knowledge allows us to analyze the results from the preliminary benchmarks. We will comment on the throughput and response times, and on the time spent waiting for latches.

6.1.1 Throughput and response time

We will first comment on the results for throughput and response time for a varying number of concurrent threads. All tests were run with Java 1.6.

V880 8-way SMP system

For the benchmark on the 8-way SMP system the throughput increases up to 8 concurrent threads, i.e., one active thread per CPU. However, the scaleup is not as good as might be expected. An ideal scenario would be doubled throughput when the number of threads is doubled, as long as CPU or other resources are not limiting factors. However, this perfect linear scalability, as described in [36], can not be expected in a real world scenario, because of different kinds of overhead involved in thread synchronization.

As pointed out in Section 5.1.1, the increase in throughput when moving from 1 to 2 threads is only 30%. 8 concurrent threads, on an 8 CPU system, provides only double the throughput compared to a single thread. This clearly shows that the scalability of Derby is limited, even for a low number of concurrent threads.

When increasing the number of active threads to 16 and 32 there is a considerable drop in throughput. The ideal scenario when increasing the concurrency level, after maximum throughput is reached, is a flat throughput curve. In such a scenario, the response time of individual transactions would increase, as more transactions are queued at higher levels of concurrency, but the throughput should not decrease. I.e., if there is no overhead for queuing and synchronization of a higher number of threads, the system should be able to process the same number of transactions per second, also for a higher concurrency level.

Because there is overhead involved in queuing and synchronization, a flat throughput curve for higher levels of concurrency can not be expected in a practical scenario. However, the goal is a curve that is close to flat, i.e., the decrease in throughput when the concurrency level is increased, beyond the point of maximum throughput, should be as small as possible.

The drop in throughput for increased levels of concurrency is significant. For 16 concurrent threads the throughput is lower than for 4 threads, and for 32 threads the throughput is only slightly higher than for 2 threads. It seems that scalability is limited to a concurrency level matching the number of CPUs for this system, however the CPU utilization was not very high, even for a higher number of concurrent threads. This suggests that synchronization overhead and contention are limiting factors for scalability.

T2000 CMT system

To evaluate scalability for an even higher number of concurrent threads, we also tested this version of Derby on the T2000 multicore system. It should be able to execute 32 simultaneous threads.

On the T2000 there is only a marginal increase in throughput for 16 threads compared to 8. This shows that the scalability beyond 8 threads is limited also on hardware that supports concurrent execution of more than 8 threads. Increasing the number of threads to 32 gives a reduction in throughput, even though the system should support 32 concurrent threads. Again, CPU utilization is not the limiting factor.

A further increase in concurrency level to 64 or 128 threads gives a sharp decline in throughput. For 128 concurrent threads the throughput is less than half of the maximum throughput reached at 16 threads. This shows that this version of Derby scales poorly for a high number of concurrent threads. This is also consistent with our earlier observations from the "autumn project".

Response time

Another important performance metric is transaction response time. For applications that require a low response time, such as interactive applications, it does not matter if the total throughput is good, if the response time for transactions is too high. Our benchmarks show that response time increases dramatically for a high number of concurrent threads.

When the number of threads is increased above the concurrency level that provides maximum throughput, there is a seemingly exponential increase in response time. This increase is caused by transactions being queued, waiting for synchronization or contended objects. Because the throughput decreases for a high number of concurrent threads, this will have an adverse effect on response time.

When the concurrency level is increased, there are more transactions waiting. The related drop in throughput leads to even more queuing, causing the observed increase in response time for high concurrency levels.

6.1.2 Latch wait time

The measurements of time spent waiting[1] to obtain latches show that there is significant contention for latches at higher concurrency levels.

Because our workload only reads the index, we know that any observed latch contention will be for the B-Tree index, not for the pages in the database itself. This is confirmed by the results: All observed latch contention, i.e., calls to `Object.wait()` when obtaining a latch in these benchmarks was for pages in Container(0, 977). This was the file containing the B-Tree index in these benchmarks. This is also true for the results for other concurrency levels, there was no latch contention observed for any page not in Container(0, 977).

[1] This is, as explained in Section 4.1.2, the time spent in calls to the `wait()` method on contended latches, not the time to acquire a latch in the uncontended case.

Because we are interested in how Derby performs in the case of high concurrency, we look at these numbers for 32 concurrent threads on the V880 system. For reference, we also included the result from the test with 2 concurrent threads. For a single thread, there can be no contention to obtain latches.

2 threads

For 2 concurrent threads, some contention was observed. 0.647s was spent by threads waiting to latch page #1, i.e., the root node. This is comparable to the time spent waiting for the second most contended page, which was 0.452s. It is likely that a "true" bottleneck does not form at this low level of concurrency, because there can only be one thread waiting while the other active thread has the latch.

Since lock-coupling is used, latches will not be held for a long time. This is because the node is only used to look up the next node in the index traversal, no modification will be performed, and there are no results to return or other work to be performed during the index lookup. Thus with a maximum of one other thread waiting, i.e., there is only one other working thread, and because latches are held for a short time, the root node is unlikely to be a limiting bottleneck.

This is confirmed by the observations. There is some contention, but the waiting time for contended latches is not significant compared to the total execution time. The difference between the root node and the second most contended latch is also small. If the root node was a limiting factor, it would have a significantly higher wait time compared to the other nodes. There is some wait time, but bottleneck behavior is not observed for 2 concurrent threads.

32 threads

The high contention scenario is illustrated by the test with 32 concurrent threads, i.e., 4 threads per CPU in the V880 system. The root node is again the most contended node, with an aggregated wait time of 1130.869s. This is 99.2% of the total time spent waiting for contended latches, and also a significant share of the total thread life time. It is clear that the root node is a bottleneck, and probably a limiting factor for performance, at this level of concurrency.

With 32 concurrent threads it is more likely that a transaction will have to wait to latch the root node of the B-Tree index, compared to the case with 2 concurrent threads. This is because the formation of a queue is more likely with a higher number of active threads that can possibly be queued. Because *all* threads have to access the root node as part of the index lookup, it is highly likely that a queue will form at the root node.

The time spent in a queue is proportional to the queue length. This is under the assumption that the processing rate is independent of the queue length. If the processing rate is affected by the queue length, the time spent in the queue, and thus the response time, will be even higher for a large number of queued threads. Thus it is more likely that a thread will be queued if a queue has first formed, because of the overhead involved in synchronization of the queued threads.

The root node stands out as a bottleneck when compared to the other pages in the

B-Tree index. This is because it is a single point which all transactions have to access in exclusive mode. Assuming a good random distribution of search keys, and k nodes on the second level of the B-Tree, each of the second level nodes will be accessed $\frac{N}{k}$ times, where N is the total number of index searches. The root node will be accessed N times.

This is a classic bottleneck situation, where contention for pages in the later stages of the index search will be lighter than for the root node. If we assume that the time a latch is held for a second level node is about the same as for the root node, this means that the "processing rate" at which threads latch, fetch the correct pointer for the index key, and unlatch the node will be comparable for the root node and the second level nodes.

For k second level nodes, and an equal key distribution in the B-Tree, i.e., a probability of $\frac{1}{k}$ for a random index search accessing a specific second level node, this should give an average queue length of $\frac{L}{N}$ at the second level nodes, L being the queue length at the root node. For a high number of second level nodes, and if the time each thread holds a latch for a second level node is comparable to the root node, it is unlikely that there will be any significant queuing at the second level. This is exactly what we observe, with 99.2% of the latch wait time being for the root node.

Looking back at Figure 5.4, which displays the wait times for the top 10 contended pages on a logarithmic scale, the root page, i.e., Page #1, stands out with a wait time several orders of magnitude higher than the other nodes. Then there are 5 nodes with comparable wait times, before there is another drop of about two orders of magnitude to the wait times for the rest of the nodes.

According to the argument presented in Section 3.4, the B-Tree index for this database should have 3 levels. The 5 nodes on the "second tier" in Figure 5.4 suggests that there are 5 nodes on the second level of the B-Tree. Thus we seem to have an index structure like the one that was sketched in Figure 3.1.

In Section 5.3.3 we found that a 20 000 transaction benchmark run acquires and releases latches a total of 60 000 times. This fits neatly with a three-level B-Tree as well.

6.1.3 Preliminary benchmarks wrapup

These benchmarks have shown that the root node of the B-Tree index is a performance bottleneck when running a high number of concurrent threads and a latch-intensive workload. Reducing the impact of the root node bottleneck should improve performance.

An implementation of shared latches is one way to achieve this for a read-only workload. If only index searches are performed, no exclusive latches would have to be obtained. Thus shared latches should provide better performance for a read-only workload, because the bottleneck at the root node is removed. However, there would still be some overhead involved in setting and releasing latches, even though shared latches allows concurrent access to the root node.

6.2 Shared latches benchmarks

To evaluate the performance of our patch for shared latches we benchmarked it against two other versions of Derby. Both were recent development versions, the "old" version using the lock manager for latches, while the "new" version does locking directly on the page objects, separate from the lock manager. This removes the global synchronization involved in latching.

We expected our patch to provide som extra overhead in the single threaded case, but we believed that shared latches would improve the multithreaded scalability for latch-intensive workloads. This should be beneficial with our read-only test workload, which only does B-Tree index lookups.

Our patched version was based on the version with split hash tables in the lock manager, that was tested in the preliminary benchmarks. This means that the results obtained can be compared to the preliminary benchmarks for Java 1.6.

We tested these versions of Derby on the V880 and T2000 systems for a varying number of concurrent threads. All tests were run with both Java 1.5 and Java 1.6, to evaluate the performance impact of the JVM, and to see if the scalability characteristics for the different versions changed when running in another JVM. We expected Java 1.6 to perform better, because the multithreaded workload and heavy use of synchronization should benefit from the improvements to concurrent performance in Java 1.6.

6.2.1 V880 8-way SMP system

As explained in Section 5.2, we labeled the version with our shared latches patch "SX", while the old version of the development trunk was labeled "OLD" and the slightly newer version "NEWCLEAN". Here we will discuss our findings on the V880 SMP system, presented in Section 5.2.1, both for Java 1.5 and 1.6.

Java 1.5

The benchmarks with Java 1.5 on the 8-way SMP system show that our SX version does indeed perform worse than the OLD and NEWCLEAN versions for a single thread, with the best throughput being provided by NEWCLEAN. The situation is much the same for 2 concurrent threads, with throughput increasing a little for all versions. However, for 4 concurrent threads the results are interesting. Then our SX version provides the best throughput, and OLD is now faster than NEWCLEAN. This indicates that our version does provide a performance improvement in a contended scenario.

For 8 concurrent threads there is little improvement over 4 threads for the OLD and NEWCLEAN versions, while our SX version reaches a max throughput of 15 032 transactions per second. This is a 24.8% improvement over the 12 047 transactions per second throughput of the OLD version, while the NEWCLEAN version provides a slightly lower throughput of 11 795 transactions per second for 8 threads.

When the number of threads is increased above 8, throughput declines for all versions. The NEWCLEAN version seems to be the worst for a high number of threads, while

the performance drop for a high number of threads is less for the OLD and SX versions. Our SX version has a throughput curve that is nearly flat from 16 to 128 concurrent threads. This shows that the cost of synchronization for a high number of threads does not have a huge impact on performance.

Java 1.6

For Java 1.6 the relative results are similar for 1 and 2 threads. NEWCLEAN is the fastest version, followed by OLD, while our SX version provides the lowest throughput. The increase in throughput from 1 to 2 threads is similar for all versions, a bit less than 40%, with NEWCLEAN showing the best improvement. However, the throughput for all versions is significantly higher with Java 1.6 than with 1.5. The single threaded throughput for NEWCLEAN was 8 209 transactions per second with Java 1.5, this is improved to 9 417 with Java 1.6, i.e., a 14.7% increase. Scalability from 1 to 2 threads is also improved with Java 1.6, for NEWCLEAN the improvement is 39.8% with Java 1.6 compared to 20.9% with Java 1.5.

As opposed to the results for Java 1.5, NEWCLEAN is still the fastest version at 4 concurrent threads, again followed by OLD, while SX still provides the lowest throughput. However, the performance increase from 2 to 4 threads is highest for our SX version, at 55.7%, while OLD and NEWCLEAN follow with increases of 45.1% and 40.4%, respectively. It is worth noting that all versions show an increased relative improvement compared to the move from 1 to 2 threads, but for NEWCLEAN this is only a marginal improvement.

At 8 concurrent threads SX performs best, reaching a maximum throughput of 21 084 transactions per second, i.e., a 37.7% increase in throughput from 4 threads. For NEWCLEAN and OLD however, the curves have flattened, with increases of 0.8% and 5.2%, respectively. This shows that the SX version scales to a higher maximum throughput also with Java 1.6.

The curves seem to decline steeper than for the Java 1.5 benchmarks, when the number of concurrent threads is increased to 16 and beyond. However, the throughput is higher with Java 1.6 than with 1.5 for all concurrency levels in these benchmarks. With Java 1.6, the curves for the different versions are closer to each other when the concurrency level is increased to 16 and beyond. The NEWCLEAN version is again the slowest at high levels of concurrency, and SX the fastest. At 64 concurrent threads the results are close, with the OLD version edging out SX, with a throughput of 14 488, compared to 14 328. At 128 threads, SX again provides significantly better throughput.

Again the results are consistent with our hypothesis that the SX version should provide better throughput at high levels of concurrency, even though it performs worse for a low number of threads. This is because the extra overhead is outweighed by the gains from increased concurrency with shared latches.

Looking at the average response times, included in Appendix B, we note that the response times for the SX version with Java 1.6 is worse at 64 and 128 threads than with Java 1.5. Although the average response times are worse, Java 1.6 provides the best throughput. For 1 to 32 concurrent threads, however, Java 1.6 gives the best response times for the SX version. For the OLD and NEWCLEAN versions, Java 1.6 provides better response times at all concurrency levels.

We will get back to the differences between running Derby with Java 1.5 and 1.6.

6.2.2 T2000 CMT system

We ran corresponding benchmarks on the T2000 system, again both with Java 1.5 and 1.6. The T2000 system provides lower single-thread throughput than the V880, but its multithreading, multicore architecture should allow scaling to a higher number of concurrent threads.

Here we comment on the results presented in Section 5.2.2.

Java 1.5

The situation for a low number of concurrent threads is similar on the T2000 system. For 1 to 4 concurrent threads, NEWCLEAN is faster, closely followed by OLD. SX provides a little lower throughput, but scales correspondingly to the other versions. For the V880 system SX was the fastest at 4 threads with Java 1.5, this is not the case on the T2000.

When increasing the number of concurrent threads to 8, the SX version provides the best throughput, and OLD is now better than NEWCLEAN. This is comparable to the V880 results, but the relative performance difference between SX and the other versions is not as large for 8 threads on the T2000.

However, when further increasing the concurrency level to 16, we observe different behavior on the T2000. The SX version scales further, showing a performance improvement of 22.8% from 8 to 16 concurrent threads. The OLD and NEWCLEAN versions, however, have slightly decreasing throughput, but are still close to each other.

When the number of active threads is increased above 16, all versions have decreasing throughput. The NEWCLEAN version provides the worst performance for a high number of threads, while the OLD version has a close to flat throughput curve from 8 to 64 threads, only showing a slight decrease. From 64 to 128 threads, however, there is a significant decrease also for the OLD version. The SX is the fastest at all concurrency levels above 8, but for 64 and 128 threads OLD provides close to the same throughput.

The situation on the T2000 is much the same as on the V880 with Java 1.5. The NEWCLEAN version is the best for a low number of concurrent threads, while our SX version scales best when the concurrency level is increased. The OLD version is also faster than NEWCLEAN for 8 and more concurrent threads.

More interestingly, this shows that on the T2000 system, the SX version is able to increase throughput up to 16 concurrent threads. This is different from the behavior on the V880 system. However, running with Java 1.5, performance does not scale up to 32 threads, even though the T2000 should support 32 concurrent threads. The OLD and NEWCLEAN versions, however, reach their maximum throughput already at 8 concurrent threads, compared to 16 threads for the SX version.

At 128 threads we also observe the highest performance improvement for SX compared to the other versions. SX provides 96% higher throughput than the NEWCLEAN version.

Java 1.6

Again, the situation is similar for a low number of concurrent threads, also with Java 1.6 on the T2000 system. For 1 to 4 threads NEWCLEAN gives the best throughput, followed by OLD and our SX version. For 8 threads there is a difference in behavior when running Java 1.6. NEWCLEAN is still the best at 8 threads, providing marginally better throughput than the OLD version, and with SX a little bit lower.

When the concurrency level is increased to 16, SX marginally beats the performance of the NEWCLEAN version. NEWCLEAN is faster than OLD at 16 concurrent threads, as was also the case for 16 threads with Java 1.6 on the V880 system. Both NEWCLEAN and OLD show slight improvements in throughput when the concurrency level is increased from 8 to 16 threads. This shows that the scalability characteristics are different on the T2000, also when running Java 1.6.

More noteworthy, the SX version provides a further increase in throughput when moving from 16 to 32 concurrent threads. This shows that with Java 1.6 it is possible to exploit the support for 32 concurrent threads on the T2000. The OLD and NEWCLEAN versions, however, give lower throughput at 32 threads. The OLD version is also faster than NEWCLEAN at 32 threads, showing that it does indeed perform better for a high concurrency level, also with Java 1.6 on the T2000.

At 64 concurrent threads, the OLD version actually provides the best throughput. This is similar to the situation with Java 1.6 on the V880 system. A further increase in the concurrency level to 128 threads shows that SX is again the fastest, followed by OLD and NEWCLEAN.

The throughput when running with Java 1.6 is better than for Java 1.5 for all concurrency levels, also on the T2000 system. However, at 128 concurrent threads it is close to the throughput with Java 1.5 for the OLD version.

We have shown that when running with Java 1.6 on the T2000 system, the version of Derby patched to support shared latches will actually show a performance improvement when scaling to 32 concurrent threads. Removing the bottleneck at the root node in the B-Tree index will allow better scalability for a read-only, index lookup intensive workload. With Java 1.6, the support for 32 concurrent threads on the T2000 will allow a performance increase for up to 32 threads with this version.

6.2.3 Java 1.5 vs 1.6

As explained in Section 2.5.2, a lot of effort has been put into the performance of multithreading and thread synchronization in the JVM implementations from Sun. Version 1.6 of the HotSpot VM should provide improved performance for multithreaded applications that use synchronization or the primitives provided by the `java.util.concurrent` package a lot. With this in mind, we expected Java 1.6 to provide better performance with Derby, because the code makes heavy use of synchronization.

This is confirmed by our benchmarks. Java 1.6 provides better throughput for all tests with all versions of Derby in these benchmarks. However, Java 1.6 does not solve the problem of performance degradation at high concurrency levels. At high concurrency levels, the time spent waiting at contention points is likely to be more significant than

the overhead involved in acquiring a monitor or lock. Running Derby with Java 1.6 will reduce some of the overhead involved in synchronization and locking, but it can not solve the problem of resource contention.

6.2.4 Shared latches wrapup

The benchmark results have shown the multithreaded performance characteristics of our modified version of Derby with support for shared latches. Here we will try to summarize our findings and also compare the results to the preliminary benchmarks, to evaluate the performance gain of the shared latches version compared to the version with the split hash tables in the lock manager, which was the starting point for our patch.

Shared latches and multithreaded scalability

As expected, the extra overhead causes our shared latches version to perform slightly worse for a single-thread and low levels of concurrency. Except for the test with Java 1.5 on the V880 system, the other versions beat the SX version at 4 concurrent threads.

When increasing the number of concurrent threads to 8, the SX version provides the best throughput in all tests, except for on the T2000 system with Java 1.6. With Java 1.6 on the T2000 the SX version is the fastest at 16 concurrent threads, and provides peak throughput at 32 threads. This is opposed to the other versions, which do not scale beyond 16 threads even on the T2000.

Except for the special case of 64 threads with Java 1.6 on the T2000, the SX version provides the best throughput for all tests with 16 or more concurrent threads. For the V880 system, SX is fastes from 8 threads, and even from 4 threads with Java 1.5.

This shows that there is a lot to be gained with support for shared latches, when scaling to a high number of concurrent threads. A clean and simplified implementation of support for shared latches should be able to reduce the overhead, thus reducing or eliminating the performance penalty for a single or few concurrent threads.

SX vs preliminary benchmarks

The results for the SX patch can also be compared to the results from the preliminary benchmarks, presented in Section 5.1 and discussed in Section 6.1. The preliminary benchmarks were done for a version of Derby with the “1704” patch, as explained in Section 2.2.4. This modified version was the starting point for our patch, so the results should be comparable. This concerns only Java 1.6, as we only used that version of the JVM in the preliminary benchmarks.

On the V880 system, there was a performance decrease of 17% percent for SX compared to the preliminary benchmarks for a single thread. The max throughput was reached for 8 threads in both benchmarks on the V880, the SX patch provides an improvement of 26% compared to the preliminary benchmarks. Comparing single thread to peak throughput at 8 threads, the preliminary benchmarks showed an increase of 1.9x, while the SX version scales to 2.9x the throughput for a single thread.

On the T2000 system, the peak throughput was reached at 16 concurrent threads in the preliminary benchmarks. The SX patch peaks at 32 threads when running with Java 1.6, for an improvement in peak throughput of 28%.

6.3 Profiling results

The profiling results show that most of the overhead incurred when running the SX patch with a single thread, compared to the OLD version, can be attributed to the creation of new `ArrayInputStream` objects and to time spent in the method calls to latch and unlatch pages.

6.3.1 Object creation – `ArrayInputStream`

More than half of the difference in CPU consumption between OLD and SX in our single-threaded profiling was spent running the constructor for `ArrayInputStream`, constructing nearly 400 000 instances of this class. This is wasteful, to put it mildly, but it was a simple way to replace the thread-unsafe shared per-page `ArrayInputStream`.

We have carried out some quick tests to see if primitive object pooling would alleviate the costs of creating `ArrayInputStream` objects, but failed to see improvements despite cutting down the number of allocations by nearly a factor of thousand.

The added costs of managing the pool, using data structures from the Java standard library, such as `ConcurrentLinkedQueue`, to avoid pool monitor contention, and resetting the objects for reuse outweighed the savings from object allocation and construction. This is in line with recommendations on object pooling in modern JVMs, as explained in [44], where object pooling is discouraged due to the low overhead associated with short-lived allocations.

Relieving this problem will probably require modifying `ArrayInputStream`, either making it thread safe or making the constructor used in performance critical loops as fast as possible.

6.3.2 Expensive latches

In section 5.3.4 we found that the latches we implemented using `ReentrantReadWriteLock` were about twice as CPU-intensive, in the single-threaded, non-contended case, as the simple latches implemented using Java `synchronized`, a state variable and `wait()/notify()`.

This is not a very surprising observation. It is common to see a simple system outperform a more complex system in simple cases. In the uncontended case, the original latch implementation only has to test and set a single variable, protected by a Java monitor, which has been highly optimized by the JVM developers.

While it is generally accepted that more complex synchronization primitives are more expensive than simple ones, it does not mean that complex synchronization primitives should be avoided. However, it may be prudent to restrict their use to places where

the benefits outweigh the cost.

In the case of `ReentrantReadWriteLock`, we can expect those places to be where many read operations would contend for a lock, e.g., the upper levels of a B-Tree in a scenario with multiple clients accessing the same tree. However, in our implementation these locks are used for all pages in all cases, not necessarily just for the pages where the benefit is tangible.

One could imagine a scenario where pages are assigned a lock implementation based on their contents. A non-leaf B-Tree page would get a `ReentrantReadWriteLock`, while leaf pages would contain a `ReentrantLock`, and simply map requests for shared access to requests for exclusive access.

Optimizing even more aggressively we could skip latching entirely while running single client workloads. This would be quite benchmark- and developer-friendly.

Advanced versions of per-page switchable lock implementations could have on-the-fly replacement of lock implementations with automatic detection of contention. Such a system would escalate the lock type if it detected readers contending for the cheap write lock. It would probably be an interesting challenge to develop and optimize such a system.

6.4 Summary

This section is intended to summarize our analysis. We review the findings for the preliminary benchmarks, shared latches benchmarks and profiling.

6.4.1 Preliminary benchmarks

The preliminary benchmarks showed that Derby scales poorly for a high number of concurrent threads on both the V880 and T2000 systems. Peak throughput was reached at 8 concurrent threads on the V880, and for 16 concurrent threads on the T2000. The response time increases dramatically when the concurrency level is increased.

The measurements of time spent waiting for latches on a per page basis shows that contention for the root node of the B-Tree index is a significant performance overhead. The root node is a bottleneck for index intensive workloads. An implementation of shared latches should allow more concurrency for read-only index lookups.

6.4.2 Shared latches

The modified version of Derby with support for shared latches was expected to perform less well for a low number of threads, but scale better for high levels of concurrency. This is because the implementation adds overhead to obtain latches, but shared latches allow more concurrency for index lookups. The benchmarks confirmed these assumptions.

The SX patch provides the best throughput for high levels of concurrency on both the SMP and multicore test systems. For the other two versions of Derby that was

tested, NEWCLEAN is best at a low number of threads, while OLD is the best for high concurrency levels.

Java 1.6 provides better performance than Java 1.5 for all test cases. With Java 1.6, the SX patch is able to scale further on the T2000 system reaching peak throughput at 32 concurrent threads.

The SX patch provides an improvement in peak throughput over the "1704" patched version, tested in the preliminary benchmarks. The improvement was 26% for 8 threads on the V880 system, and 28% for 32 threads on the T2000 system, where the "1704" reached peak throughput at 16 threads.

6.4.3 Profiling results

The results of profiling show that the overhead added in the SX patch is largely due to the creation of a large number of `ArrayInputStream` objects, and the higher cost of setting and releasing latches based on `ReentrantReadWriteLock`.

The first of these two problems is probably the most straightforward to remedy, thread safety can be achieved by other means than simply creating a new `ArrayInputStream` for every use.

Reducing the cost of latching may be harder, as `ReentrantReadWriteLock` is included with the Java runtime, and implementing a cheaper version just for Derby may not be realistic. We suggest looking at ways to use it more sparingly, to minimize the impact of the higher overhead of this type of lock.

Chapter 7

Conclusion and further work

In this chapter, we present the conclusion on the basis of the obtained results and analysis. We then suggest further work on the shared latch implementation, and on improving the scalability of Derby and the B-Tree index.

7.1 Conclusion

The Derby B-Tree algorithm uses exclusive latches and a lock-coupling algorithm. With basis in the theory presented in Chapter 2, we can conclude that this is inefficient. The algorithm implements a top-down approach to structure modifications, and structure modification operations may be restarted multiple times.

The preliminary benchmarks showed that the multithreaded scalability of Derby is limited. We identified the root node of the B-Tree index as a performance bottleneck for high concurrency levels, when running a read-only workload that performs a single record index lookup. This prompted us to implement an experimental patch for shared latches, to evaluate its performance impact for multithreaded workloads.

For a high number of concurrent threads, the version with shared latches scales better than the other two benchmarked versions of Derby. Our version with shared latches provides higher peak throughput than the other versions for all benchmarks on both the V880 and T2000 test systems, both running with Java 1.5 and 1.6.

With 128 concurrent threads and Java 1.5 on the T2000 system, we observe the best performance improvement. The shared latches version provides 96% higher throughput than the "new" version of the Derby development trunk.

Java 1.6 provides a significant performance gain when running Derby, both for single-threaded and high concurrency workloads. The best improvement is observed for the "new" version with 4 concurrent threads on the V880 system, Java 1.6 provides an increase in throughput of 72% compared to 1.5.

However, the version of Derby with support for shared latches provides less throughput for single-threaded and low concurrency workloads. This is due to overhead incurred by the more expensive shared/exclusive latches, implemented with use of the `ReentrantReadWriteLock` java class.

Profiling shows that about 90% of the extra overhead, in single-threaded execution,

for the shared latches version is caused by the creation of `ArrayInputStream` objects and the use of `ReentrantReadWriteLock`, which is more expensive than exclusive locking.

7.2 Further work and improvements

The experimental implementation of shared latches needs further work before it is ready for production use with Derby. There are also possibilities to further extend the patch to provide better performance both in single- and multithreaded cases.

We will describe possible improvements to shared latches support, the Derby B-Tree index and the general scalability of Derby. These are more specific than the suggestions from our autumn project, summarized in Section 2.2.6.

7.2.1 Shared latches

Our patch has allowed us to evaluate the performance impact of shared latches, but it is experimental and not ready for production use. The patch should be cleaned up, or reimplemented, to improve code quality and ensure consistency for all workloads.

The overhead when running a single-thread workload should be reduced. This can be achieved by more efficient solutions to thread safety for page access, i.e., without the need to create many `ArrayInputStream` objects, and by looking into the use of `ReentrantReadWriteLock`. A different locking strategy for single-threaded workloads is one possibility, another is escalation to shared lock types when contention is detected.

7.2.2 B-Tree algorithm

With support in the relevant theory, we have concluded that the B-Tree algorithm used in Derby is inefficient. The B-Tree algorithm should be reimplemented to provide higher concurrency for index operations. Some of the B-Tree variants presented in Section 2.4.2 may be viable alternatives. The implementation of an advanced concurrent B-Tree algorithm, based on one of these variants, requires support for shared latches. Therefore we would recommend that an effort to improve concurrency for the B-Tree access method starts with an efficient, production quality implementation of shared latches.

Bibliography

[1] Shailesh Agarwal, Christopher Keene, and Arthur M. Keller. Architecting object applications for high performance with relational databases. In *Proc. of OOPSLA Workshop on Object Database Behavior, Benchmarks, and Performance*, October 1995.

[2] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, YS Ramakrishna, and D. White. An Efficient Meta-lock for Implementing Ubiquitous Synchronization. 1999.

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

[4] T.E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[5] Apache Derby. `http://db.apache.org/derby/`. Accessed 20 April 2007.

[6] Apache Derby JavaDoc. `http://db.apache.org/derby/javadoc/`. Accessed 30 April 2007.

[7] Apache Derby Performance. `http://wiki.apache.org/apachecon-data/attachments/Us2005OnlineSessionSlides/attachments/ApacheCon05usDerbyPerformance.pdf`. Accessed 1 May 2007.

[8] Apache Derby: Source Code. `http://db.apache.org/derby/dev/derby_source.html`. Accessed 30 April 2007.

[9] The apache software foundation. `http://www.apache.org/`. Accessed 19 April 2007.

[10] Apache License, Version 2.0. `http://www.apache.org/licenses/LICENSE-2.0`. Accessed 19 April 2007.

[11] ApacheCon Conferences. http://apachecon.com/. Accessed 1 May 2007.

[12] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, et al. The Landscape of Parallel Computing Research: A View from Berkeley. Technical report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006.

[13] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1–10, 1995.

[14] Berkeley DB - Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Berkeley_DB`. Accessed 1 May 2007.

[15] Berkeley DB Java Edition Direct Persistence Layer Basics. `http://www.oracle.com/database/docs/BDB-JE-DPL-Basics-Whitepaper.pdf`. Accessed 1 May 2007.

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

[17] L.N. Bhuyan, Q. Yang, and D.P. Agrawal. Performance of Multiprocessor Interconnection Networks. *Computer*, 22(2):25–37, 1989.

[18] Building Derby. `http://svn.apache.org/repos/asf/db/derby/code/trunk/BUILDING.txt`. Accessed 28 May 2007.

[19] 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.

[20] D. D. Chamberlin, M. M. Astrahan, W. F. King, R. A. Lorie, J. W. Mehl, T. G. Price, M. Schkolnick, P. Griffiths Selinger, D. R. Slutz, B. W. Wade, and R. A. Yost. Support for repetitive transactions and ad hoc queries in system r. *ACM Trans. Database Syst.*, 6(1):70–94, 1981.

[21] Changes in Cloudscape Availability and Support. `http://www-1.ibm.com/support/docview.wss?rs=636&uid=swg21256502`. Accessed 25 April 2007.

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

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

[24] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, 2006.

[25] Database 10g | Oracle Database - The First Database Designed for Grid Computing. `http://www.oracle.com/database/index.html`. Accessed 22 May 2007.

[26] 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 May 2007.

[27] DB Apache Project. `http://db.apache.org/`. Accessed 20 April 2007.

[28] DBAzine.com: DRDA. `http://www.dbazine.com/db2/db2-mfarticles/mullins-drda`. Accessed 23 April 2007.

[29] [#DERBY-1704] Allow more concurrency in the lock manager - ASF JIRA. http://issues.apache.org/jira/browse/DERBY-1704. Accessed 2 May 2007.

[30] [#DERBY-2107] Move page latching out of the lock manager - ASF JIRA. https://issues.apache.org/jira/browse/DERBY-2107. Accessed 2 May 2007.

[31] [#DERBY-2327] Reduce monitor contention in LockSet - ASF JIRA. https://issues.apache.org/jira/browse/DERBY-2327. Accessed 2 May 2007.

[32] Derby Developer's Guide. http://db.apache.org/derby/docs/dev/devguide/. Accessed 30 April 2007.

[33] Derby Optimizer Design. http://db.apache.org/derby/papers/optimizer.html. Accessed 25 April 2007.

[34] Derby Logging and Recovery. http://db.apache.org/derby/papers/recovery.html. Accessed 23 April 2007.

[35] DerbyLruCacheManager - Db-derby Wiki. http://wiki.apache.org/db-derby/DerbyLruCacheManager. Accessed 3 May 2007.

[36] David DeWitt and Jim Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

[37] Dave Dice, Mark Moir, and William III Scherer. Quickly Reacquirable Locks. http://home.comcast.net/~pjbishop/Dave/QRL-OpLocks-BiasedLocking.pdf, 2006.

[38] David Dice. java.util.concurrent ReentrantLock vs synchronized() - which should you use? http://blogs.sun.com/dave/entry/java_util_concurrent_reentrantlock_vs, 2006. Accessed 22 May 2007.

[39] David Dice. Let's say you're interested in using HotSpot as a vehicle for synchronization research. http://blogs.sun.com/dave/entry/lets_say_you_re_interested, 2006. Accessed 22 May 2007.

[40] diff - Wikipedia, the free encyclopedia. WikimediaFoundation,Inc. Accessed 26 May 2007.

[41] DIFFSTAT - make histogram from diff-output. http://invisible-island.net/diffstat/diffstat.html. Accessed 26 May 2007.

[42] Sun Microsystems – BigAdmin: DTrace. http://www.sun.com/bigadmin/content/dtrace/. Accessed 02 May 2007.

[43] EUROPA - Education and Training - Socrates programme - ECTS - European Credit Transfer and Accumulation System. http://ec.europa.eu/education/programmes/socrates/ects/index_en.html. Accessed 25 May 2007.

[44] Brian Goetz. Java theory and practice: Urban performance legends, revisited. http://www.ibm.com/developerworks/java/library/j-jtp09275.html. Accessed 30 May 2007.

[45] J. Gosling, B. Joy, G.L. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2005.

[46] J. Gray. Notes on Database Operating Systems. In *volume 60 of Lecture Notes in Computer Science*, pages 393–481. Springer Verlag, 1978.

[47] 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.

[48] P.B. Hansen. *Operating System Principles*. Prentice Hall PTR Upper Saddle River, NJ, USA, 1973.

[49] P.B. Hansen. Java's insecure parallelism. *ACM SIGPLAN Notices*, 34(4):38–45, 1999.

[50] John L. Hennessy and David A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[51] hibernate.org - Hibernate. `http://hibernate.org/`. Accessed 1 May 2007.

[52] C.A.R. Hoare. Monitors: An Operating System Structuring Concept. *Communications*, 1974.

[53] HotSpot Java virtual machine source code. `http://openjdk.java.net/groups/hotspot/`. Accessed 21 May 2007.

[54] HotSpot Java virtual machine source code, biased locking code. `http://opengrok.neojava.org/hotspot/xref/src/cpu/i486/vm/assembler_i486.cpp#4076`. Accessed 21 May 2007.

[55] IBM Software - DB2 Product Family - Family Overview. `http://www-306.ibm.com/software/data/db2/`. Accessed 23 April 2007.

[56] IntelliJ IDEA :: The Most Intelligent Java IDE. `http://www.jetbrains.com/idea/`. Accessed 14 May 2007.

[57] International Organization for Standardization. *ISO/IEC-9804:1998: Information technology – Open Systems Interconnection – Service definition for the Commitment, Concurrency and Recovery service element*, 12 1998.

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

[59] Java Technology. `http://java.sun.com`. Accessed 19 April 2007.

[60] JDBC Documentation. `http://java.sun.com/j2se/1.5.0/docs/guide/jdbc/`. Accessed 19 April 2007.

[61] JIRA - Bug tracking, issue tracking and project management software. `http://www.atlassian.com/software/jira/`. Accessed 15 May 2007.

[62] G. Kane and J. Heinrich. *MIPS RISC architecture*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1992.

[63] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 224–234, 1992.

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

[65] Vladimir Lanin and Dennis Shasha. A symmetric concurrent b-tree algorithm. In *ACM '86: Proceedings of 1986 ACM Fall joint computer conference*, pages 380–389, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[66] Doug Lea. Java Specification Request 166: Concurrency Utilities. `http://www.jcp.org/en/jsr/detail?id=166`, 2004.

[67] Doug Lea. The java. util. concurrent synchronizer framework. *Science of Computer Programming*, 58(3):293–309, 2005.

[68] Philip L. Lehman and s. Bing Yao. Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Syst.*, 6(4):650–670, 1981.

[69] M.M. Michael and M.L. Scott. Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms. 1995.

[70] Mimer Developers - Features - Transaction Concurrency - Optimistic Concurrency Control. `http://developer.mimer.com/features/feature_15.htm`. Accessed 27 May 2007.

[71] Mimer SQL - Mimer Information Technology. `http://www.mimer.com/`. Accessed 27 May 2007.

[72] MIPS Technologies - At the Core of the User Experience.®. `MIPSTechnologies, Inc.` Accessed 25 May 2007.

[73] C. Mohan. Concurrency control and recovery methods for B+-tree indexes: ARIES/KVL and ARIES/IM. In *Performance of Concurrency Control Mechanisms in Centralized Database Systems*, pages 248–306. 1996.

[74] 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.

[75] MySQL AB :: Developer Zone. `http://mysql.org/`. Accessed 4 May 2007.

[76] NetBeans IDE. `http://www.netbeans.org`. Accessed 14 May 2007.

[77] The Open Source Definition. `http://www.opensource.org/docs/osd`. Accessed 19 April 2007.

[78] The Open Group: Enterprise Architecture Standards, Certification and Services. `http://www.opengroup.org/`. Accessed 23 April 2007.

[79] OpenSPARC. `http://www.sun.com/processors/opensparc/`. Accessed 11 May 2007.

[80] Oracle Berkeley DB Java Edition | Oracle Berkeley DB. http://www.oracle.com/technology/products/berkeley-db/je/index.html. Accessed 1 May 2007.

[81] Oracle Berkeley DB Java Edition vs. Apache Derby: A Performance Comparison. http://www.oracle.com/technology/products/berkeley-db/pdf/je-derby-performance.pdf. Published November, 2006.

[82] org.apache.derby.impl.store.access.btree. http://db.apache.org/derby/papers/btree_package.html. Accessed 30 April 2007.

[83] Per Ottar Ribe Pahr and Anders Morken. Apache Derby SMP Scalability. http://base.google.com/base/a/1468576/D10343779029035238114. Published 18 December 2006.

[84] patch(1): apply diff file to original - Linux man page. http://www.die.net/doc/linux/man/man1/patch.1.html. Accessed 26 May 2007.

[85] PolePosition. http://www.polepos.org/. Accessed 1 May 2007.

[86] PolePosition results. http://polepos.sourceforge.net/results/html/index.html. Accessed 1 May 2007.

[87] PostgreSQL: The world's most advanced open source database. http://www.postgresql.org/. Accessed 4 May 2007.

[88] Scott A. Sandford. Apples and Oranges – A Comparison. http://www.improbable.com/airchives/paperair/volume1/v1i3/air-1-3-apples.html. Published in the Annals of Improbable Research Volume 1, Issue 3, May/June 1995.

[89] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD '79: Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34, New York, NY, USA, 1979. ACM Press.

[90] Sequoia: Welcome to the Sequoia Project! http://sequoia.continuent.org/HomePage. Accessed 9 May 2007.

[91] Solaris 10 Operating System. http://www.sun.com/software/solaris/. Accessed 2 May 2007.

[92] SQLvsDerbyFeatures - Db-derby Wiki. http://wiki.apache.org/db-derby/SQLvsDerbyFeatures. Accessed 23 April 2007.

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

[94] subversion.tigris.org. http://subversion.tigris.org. Accessed 19 May 2007.

[95] Sun Fire T1000 and T2000 Server Architecture – Unleashing the UltraSPARC T1 Processor with CoolThreads Technology. http://www.sun.com/servers/coolthreads/coolthreads_architecture_wp.pdf. White Paper, Published December 2005.

[96] Sun Fire T2000 Server. http://www.sun.com/servers/coolthreads/t2000/. Accessed 10 May 2007.

[97] Sun Fire V880 Server. http://www.sun.com/servers/midrange/v880/spec.html. Accessed 10 May 2007.

[98] Threads and Locks. http://java.sun.com/docs/books/jls/second_edition/html/memory.doc.html#29596. Accessed 25 May 2007.

[99] TPC-B. http://www.tpc.org/tpcb/default.asp. Accessed 1 May 2007.

[100] Tuning Derby. http://db.apache.org/derby/docs/dev/tuning/. Accessed 1 May 2007.

[101] Ubuntu Home Page. http://www.ubuntu.com/. Accessed 24 May 2007.

[102] UltraSPARC III processor. http://www.sun.com/processors/UltraSPARC-III/. Accessed 10 May 2007.

[103] UltraSPARC T1 processor. http://www.sun.com/processors/UltraSPARC-T1/. Accessed 10 May 2007.

[104] D.L. Weaver and T. Germond. *The SPARC Architecture Manual: Version 9*. PTR Prentice Hall, 1994.

Abbreviations and Terms

Some abbreviations and terms used in this report.

Term	Definition
2PL	Two Phase Locking
ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
ARIES	Algorithm for Recovery and Isolation Exploiting Semantics
ASF	Apache Software Foundation
CAS	Compare-And-Swap
CMT	Chip Multi-Threading
CPU	Central Processing Unit
CVS	Concurrent Versioning System
DBMS	Database Management System
DPL	Direct Persistence Layer
DRDA	Distributed Relational Database Architecture
ECTS	European Credit Transfer and Accumulation System
FPU	Floating Point Unit
IDE	Integrated Development Environment
IO	Input/Ouput
IP	Internet Protocol
JAR	Java ARchive
JDBC	Java Database Connectivity
JDK	Java Development Kit
JVM	Java Virtual Machine
KVL	Key Value Locking
MIPS	Microprocessor without Interlocked Pipeline Stages
MVCC	Multiversion Concurrency Control
ORM	Object-Relational Mapping
RDBMS	Relational Database Management System
SMP	Symmetric Multiprocessing
SPARC	Scalable Processor ARChitecture
SQL	Structured Query Language
SVN	Subversion
SX	Shared / Exclusive
TCP	Transmission Control Protocol
TPC	Transaction Processing Performance Council
VCS	Version Control System
WAL	Write Ahead Logging

Appendix A

Raw results for “1704” patch

These are the raw numbers from the benchmarks of the “1704” patch.

V880, Java 1.6		
Threads	Throughput	Avg. response time (ms)
1	8624	0.114626
2	11291	0.174921
4	15338	0.258150
8	16728	0.474062
16	14379	1.103958
32	12576	2.525081

T2000, Java 1.6		
Threads	Throughput	Avg. response time (ms)
4	10665	0.371471
8	12764	0.621377
16	12934	1.226974
32	11924	2.641924
64	9321	6.755690
128	5762	21.853692

Appendix B

Raw results for shared latches

These are the raw numbers from the benchmarks of the shared latches patch.

V880, Java 1.5						
	Throughput			Avg. response time (ms)		
Threads	SX	OLD	NEWCLEAN	SX	OLD	NEWCLEAN
1	6362	7273	8209	0.155838	0.135451	0.120340
2	8295	9115	9921	0.238975	0.216482	0.199349
4	11789	11415	10745	0.334594	0.337805	0.367885
8	15032	12047	11795	0.521615	0.655912	0.672250
16	13696	10996	10959	1.146136	1.441663	1.443169
32	13398	11124	10014	2.324168	2.854482	3.132350
64	13113	10906	9014	4.254139	5.811324	7.022955
128	12928	10291	7444	7.975246	11.789528	17.007157

V880, Java 1.6						
	Throughput			Avg. response time (ms)		
Threads	SX	OLD	NEWCLEAN	SX	OLD	NEWCLEAN
1	7162	8374	9417	0.138350	0.118027	0.104768
2	9833	11415	13172	0.201129	0.172911	0.149675
4	15308	16563	18490	0.258619	0.238547	0.213813
8	21084	17419	18642	0.371578	0.454363	0.425914
16	18243	15481	16467	0.860991	1.023511	0.961125
32	16719	15158	15102	1.877340	2.088063	2.105491
64	14328	14488	13544	4.369171	4.358392	4.701581
128	14156	11117	10492	8.823511	11.319624	12.103009

T2000, Java 1.5						
	Throughput			Avg. response time (ms)		
Threads	SX	OLD	NEWCLEAN	SX	OLD	NEWCLEAN
1	3254	3705	3971	0.304863	0.267275	0.249254
2	5149	5949	6214	0.385888	0.333709	0.317768
4	8557	9287	9402	0.464373	0.427413	0.419681
8	12254	11469	11032	0.648994	0.688204	0.716321
16	13701	11158	10708	1.152993	1.424874	1.470348
32	12779	10834	8875	2.385603	2.932017	3.538944
64	10971	10692	6565	5.130610	5.397780	9.594385
128	8518	8305	4342	12.923013	13.348098	29.281882

T2000, Java 1.6						
	Throughput			Avg. response time (ms)		
Threads	SX	OLD	NEWCLEAN	SX	OLD	NEWCLEAN
1	3585	4354	4752	0.276591	0.227365	0.208127
2	5703	7210	8069	0.348074	0.274994	0.245421
4	9197	12104	13064	0.432125	0.327562	0.303570
8	13128	14000	14240	0.605104	0.565213	0.558131
16	14756	14263	14744	1.076278	1.114718	1.079485
32	16588	13858	13188	1.909419	2.297579	2.412880
64	12685	13320	10283	4.407453	4.763810	6.190361
128	9944	8341	6522	11.483683	15.178643	19.491824

Appendix C

Enclosed source code and profiling snapshots

Source code for the benchmark application, the patch for shared latches and profiling snapshots have been enclosed with this report in a .zip file.

C.1 Benchmark application

The source code for the benchmark application is included in the `selectload` directory in the enclosed .zip file.

C.2 Shared latches patch

The patch to add support for shared latches to Derby is included in the `SX-patch` directory in the enclosed .zip file.

The patch can be applied to the Derby development trunk, revision 501369, using the `patch` utility [84]. For information on using SVN to check out the Derby source code, see [8].

C.3 Profiling snapshots

The NetBeans profiler snapshots have been included. They can be found in the `snapshots` directory in the enclosed .zip file.

The profiler snapshots corresponds to the versions of Derby tested in the benchmarks. The files are named `sx-20k.nps`, `old-20k.nps`, and `newclean-20k.nps` respectively, for the SX, OLD and NEWCLEAN versions.