

# Advanced Database Systems, CSCI-GA.2434-001

## New York University, Fall 2011

instructor: Dennis Shasha  
shasha@cs.nyu.edu  
212-998-3086  
Courant Institute  
New York University  
251 Mercer Street  
NY, NY 10012 USA

Office Hours: 9 PM on Tuesdays or 5 PM Thursdays by appointment

September 27, 2011

## 1 Goals

To study the internals of database systems as an introduction to research and as a basis for rational performance tuning.

The study of internals will concern topics at the intersection of database system, operating system, and distributed computing research and development. Specific to databases is the support of the notion of transaction: a multi-step atomic unit of work that must appear to execute in isolation and in an all-or-nothing manner. The theory and practice of transaction processing is the problem of making this happen efficiently and reliably.

Tuning is the activity of making your database system run faster. The capable tuner must understand the internals and externals of a database system well enough to understand what could be affecting the performance of a database application. We will see that interactions between different levels of the system, e.g., index design and concurrency control, are extremely important, so will require a new optic on database management design as well as introduce new research issues. Our discussion of tuning

will range from the hardware to conceptual design, touching on operating systems, transactional subcomponents, index selection, query reformulation, normalization decisions, and the comparative advantage of redundant data. This portion of the course will be heavily sprinkled with case studies from database tuning in biotech, telecommunications, and finance.

Because of my recent research interests, this year will include frequent discussions of

- “array databases,” the extension of relational systems to support ordered data such as time series in finance, network management etc.
- “secure database outsourcing,” the ability to use an outsourced database while preserving privacy.

## 2 Mechanics

YOU MUST BE ENROLLED IN THIS CLASS TO SIT IN ON THE LECTURES.

### 2.1 Texts and Notes

The first text will be used for the first half of the course and the second text in the second half. The notes will be used throughout the course.

- *Concurrency Control and Recovery in Database Systems* by Bernstein, Hadzilacos, and Goodman, Addison-Wesley, 1987. ISBN 0-201-10715-5 Now available for free at: <http://research.microsoft.com/pubs/ccontrol/>
- *Database Tuning : Principles Experiments and Troubleshooting Techniques* Dennis Shasha and Philippe Bonnet, Morgan Kaufmann Publishers, June 2002, ISBN 1-55860-753-6, Paper, 464 Pages.

There are also three optional books which are very nicely written:

- *Transaction Processing: Concepts and Techniques* by Jim Gray, Andreas Reuter 1002 pages ; Publisher: Morgan Kaufmann; 1st edition (1993) ISBN: 1558601902

- *Principles of Distributed Database Systems* M. Tamer Ozsu and Patrick Valduriez, Prentice-Hall, 1999. A nice introduction to the general problem of distributed database systems.
- *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery* Gerhard Weikum, Gottfried Vossen The Morgan Kaufmann Series in Data Management Systems, Jim Gray, Series Editor May 2001, 944 pages \$79.95, ISBN 1-55860-508-8 If you want to be a world authority on these topics.

## 2.2 Prerequisites

Fundamental Algorithms I plus Data Base Systems I or equivalent (first 6 chapters of Ullman). If you don't have the database prerequisites, then you may take the course, but you must be responsible for understanding material covered in Database I: a reading knowledge of SQL and basic familiarity with indexes and third normal form.

## 2.3 Course Requirements

three problem sets (40%), project (60%).

LATE HOMEWORKS OR PROJECTS WILL NOT BE ACCEPTED without a note from your physician or from your employer. (We will discuss the solutions on the day you hand in the assignment. That's why I don't want any late homeworks. As for projects, this is a question of fairness.)

On the other hand, collaboration on the problem sets IS allowed. You may work together with one other student and sign both of your names to a single submitted homework. Both of you will receive the grade that the homework merits. So, you may work alone or in a team of two, but no team larger than two.

## 3 Syllabus — times are estimated

1. Overview of transaction processing, distributed systems, and tuning (1 week)

2. Principles of concurrency control for centralized, distributed, and replicated databases. (3 weeks)
3. Principles of logging, recovery, and commit protocols. (3 weeks)
4. Database Tuning (7 weeks)

Tuning principles.  
 Hardware, operating system, and transaction subsystem  
 Transaction Chopping  
 Index tuning  
 Tuning relational systems  
 Tuning data warehouses  
 Troubleshooting  
 Case Studies from Wall Street and Elsewhere

5. Special topics: array databases, special indexes, time series.

## 4 Project

Your project is due the second to last class (December 6, 2011). It will be graded by the last class at which point you will have nothing more to do. You have three possible projects:

1. Distributed replicated concurrency control and recovery. You may do this in a team of two.
2. An experimental study of tuning issues on a real system.
3. Deployment of an encrypted transactional store built on top of Scalaris and integration with SQLite.

### 4.1 Possibility 1 — Replicated Concurrency Control and Recovery (RepCRec for short)

Implement a distributed concurrency control algorithm and commit algorithm with replication. Variables  $x_1, \dots, x_{20}$  (that is, there are only 20 variables in whole database — the numbers between 1 and 20 will be referred to as indexes below). Sites are 1 to 10. A copy is indicated by a dot.

Thus,  $x_{6.2}$  is the copy of variable  $x_6$  at site 2. The odd indexed variables are at one site each (i.e.  $1 + \text{index number} \bmod 10$ ). Even indexed variables are at all sites. Each variable  $x_i$  is initialized to the value  $10i$ .

Implement the available copies approach to replication using two phase locking (using read and write locks) at each site and validation at commit time.

Avoid deadlocks using the wait-die protocol in which older transactions wait for younger ones, but younger ones never wait for older ones. No two transactions will have the same age. This implies that your system must keep track of the oldest transaction time of any transaction holding a lock.

(Possible optimization: If  $T_2$  is waiting for a lock on  $x$  and  $T_3$  later arrives and is also waiting for a lock on  $x$  and  $T_3$  is younger than  $T_2$  and the lock  $T_2$  wants conflicts with the lock that  $T_3$  wants, then you may if you wish abort  $T_3$  right away. Alternatively, you can delay the decision until  $T_2$  actually acquires the lock and abort  $T_3$  then.)

Read-only transactions should use multiversion read consistency.

Input instructions come from a file in or the standard input, output goes to a file out. (That means your algorithms may not look ahead in the input.) Input instructions occurring in one step begin at a new line and end with a carriage return. Thus, there will be several operations in each step, though at most only one per transaction. Some of these operations may be blocked due to conflicting locks. You may assume that the processors work in lock-step That is, you may assume that all operations on a single line occur concurrently. When running our tests, we will ensure that operations occurring concurrently they don't conflict with one another.

Input is of the form:

`begin(T1)` says that  $T_1$  begins

`beginRO(T3)` says that  $T_3$  is read-only

`R(T1, x4)` says transaction 1 wishes to read  $x_4$  (provided it can get the locks or provided it doesn't need the locks (if  $T_1$  is a read-only transaction)). It should read any up (i.e. alive) copy and return the current value.

`W(T1, x6,v)` says transaction 1 wishes to write all available copies of  $x_6$  (provided it can get the locks) with the value  $v$ .

`dump()` gives the committed values of all copies of all variables at all sites, sorted per site.

dump(i) gives the committed values of all copies of all variables at site i.  
dump(xj) gives the committed values of all copies of variable xj at all sites.

end(T1) causes your system to report whether T1 can commit.

fail(6) says site 6 fails. (This is not issued by a transaction, but is just an event that the tester will execute.)

recover(7) says site 7 recovers. (Again, a tester-caused event) We discuss this further below.

A newline means time advances by one. A semicolon is a separator for co-temporaneous events.

Example (partial script with six steps in which transactions T1 and T2 commit, and one of T3 and T4 may commit)

```
begin(T1)
begin(T2)
begin(T3)
W(T1, x1,5); W(T3, x2,32)
W(T2, x1,17); — will cause T2 to die because it cannot wait for an older
lock
end(T1); begin(T4)
W(T4, x4,35); W(T3, x5,21)
W(T4,x2,21); W(T3,x4,23) — T4 will die freeing the lock on x4 allowing
T3 to finish
```

Your program should consist of two parts: a single transaction manager that translates read and write requests on variables to read and write requests on copies using the available copy algorithm described in the notes. The transaction manager never fails. (Having a single global transaction manager that never fails is a simplification of reality, but it is not too hard to get rid of that assumption.)

If the TM requests a read for transaction T and cannot get it due to failure, the TM should try another site (all in the same step). If no relevant site is available, then T must wait. T may also have to wait for conflicting locks. Thus the TM may accumulate an input command for T and will try it on the next tick (time moment). While T is blocked (whether waiting for a lock to be released or a failure to be cleared), no new operations for T will

appear, so the buffer size for messages from any single transaction can be of size 1.

A data and lock manager at each site performs concurrency control. You should implement a simple message buffer at each site. In one step each working DM reads its message buffer from the TM in that step, performs some processing and perhaps responds to the TM. The TM won't send more than one message to a DM in one step though that message may contain several operations each from a different transaction.

Failures are indicated only by the fail statement. The site should forget any previous messages sent to it (because these are held in volatile storage) and should forget lock information.

If a site fails and recovers, the DM would normally perform local recovery first (perhaps by asking the TM about transactions that the DM holds pre-committed but not yet committed), but this is unnecessary since, in the simulation model, commits are atomic with respect to failures. Therefore, all non-replicated variables are available for reads and writes. Regarding replicated variables, the site makes them available for writing, but not reading. In fact, reads will not be allowed until a committed write takes place (see notes on recovery when using the available copies algorithm).

During execution, your program should say which transactions commit and which abort and for what reason. For debugging purposes you should implement the command `querystate()` which will give the state of each DM and the TM as well as the data distribution and data values. Finally, each read that occurs should show the value read.

#### **4.1.1 Running the programming project**

You will demonstrate the project to our very able graders. You will have one hour to do so. The test should take a few minutes. The only times tests take longer are when the software is insufficiently portable. The version you send in should run on departmental servers or on your laptop.

The grader will ask you for a design document that explains the structure of the code. The grader will also take a copy of your source code for evaluation of its structure. Finally, the grader may ask you how you would change the code to achieve some new function.

### 4.1.2 Documentation and Code Structure

Because this class is meant to fulfill the large-scale programming project course requirement, please give some thought to design and structure. The design document submitted in late October should include all the major functions, their inputs, outputs and side effects. If you are working in a team, the author of the function should also be listed. The submitted code similarly should include as a header: the author (if you are working in a team of two), the date, the general description of the function, the inputs, the outputs, and the side effects.

In addition, you should provide a few testing scripts in plain text to the grader in the format above and say what you believe will happen in each test.

## 4.2 Possibility 2 — Benchmarking/Tuning Project

Use everything you learn in class or from the tuning book and try to improve a real system that is available to you. Use substantial relations, e.g. 100 million rows and up. Specify the database management system, operating system, hardware platform including disks, memory size, and processor. For each attempted improvement, specify what happened to the system (whether it became faster or slower) and try to form a hypothesis as to why that occurred.

You should be prepared to discuss your project with me or potentially in front of the class.

## 4.3 Possibility 3 — Encrypted Key-Store Integrated with SQLite

In this project, you will implement private key encryption on top of Scalaris and allow the following key-value accesses:

1. accept a private key from the user
2. use the private key to encrypt the value of a key-value pair
3. insert a key-value pair to Scalaris
4. append to the value of a key-value pair in Scalaris



5. read a key-value pair from Scalaris
6. begin a transaction
7. end a transaction

You will then deploy SQLite and make it possible to decrypt and instantiate a SQLite instance with the value of a key-value pair in Scalaris and to encrypt and save an entire SQLite instance inside the value of a key-value pair in Scalaris.

## **5 Project Schedule — depends on project you choose**

### **5.1 Schedule for Programming Project**

Last class in September: Letter of intent that you are going to do programming project. Partner chosen if any. Please send this letter to radheshg@nyu.edu

Last class in October: design document. Please send this document to st1376@nyu.edu

Project is due on December 6, 2011. Please send your code to radheshg@nyu.edu Between December 6 and December 13, your project will be graded. You will make an appointment that week with the grader. We will figure out a randomized way to do this.

### **5.2 Schedule for Benchmarking/Tuning Project and for Scalaris/SQLite project**

Last class in September: project outline (should fit on one page). Tuning problem you intend to address. System you plan to use and experimental question you plan to ask. This must be approved by me (Shasha) before you go on.

Last Class in October: status report. How are you doing? Any show-stoppers.

December 6, 2011: final report to me and begin to set up appointment for testing with grader.