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 (and colleagues’) recent research interests, this year will include frequent discussions of

- fault tolerance to thousands or millions of processors mostly for web processing
- secure database outsourcing
- “array databases,” the extension of relational systems to support ordered data such as time series in finance, network management etc.

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.


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

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 through group bys and nested queries, the implementation of B-tree and hash indexes, and third normal form.

2.3 Course Requirements

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 partner and sign both of your names to a single submitted homework. Both of you will receive the grade that the homework merits.

3 Syllabus — times are estimated

1. Overview of transaction processing, distributed systems, and tuning (1 week) (Reading: Bernstein, Hadzilacos, and Goodman: chapter 1)
2. Principles of concurrency control for centralized, distributed, and replicated databases. (3 weeks) (Reading: Bernstein, Hadzilacos, and Goodman: chapter 2 (2.1, 2.2, 2.3, 2.5) chapter 3, chapter 5 (5.1, 5.4))

3. Principles of logging, recovery, and commit protocols. (3 weeks) (Reading: Bernstein, Hadzilacos, and Goodman: chapters 6, 7, 8)

4. Database Tuning (7 weeks) (Shasha and Bonnet: one chapter per week)

   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 in two parts with the last part due December 6, 2010. It will be graded by the last class at which point you will have nothing more to do. Some possible project topics (you choose one) are:

1. Distributed replicated concurrency control and recovery. You may do this in a team of two.

2. An experimental of tuning issues on a real system.

4.1 Project 1 basics — Replicated Concurrency Control and Recovery (RepCRec for short)

This project consists of two parts, the first one is to be completed by November 15 and the second by December 6.
4.1.1 Part 1 – due November 15, 2010

Implement a distributed concurrency control algorithm and commit algorithm with replication. Let the variables in the database be $x_1, \ldots, x_{20}$ (that is, only 20 variables in whole database — the numbers between 1 and 20 will be referred to as indexes below). Let there be 10 sites, numbered from 1 to 10. A copy of a variable is indicated by a dot. Thus, $x_{6.2}$ is the copy of variable $x_6$ at site 2. The odd indexed variables exist at only one site each (i.e. index number mod 10 plus 1, and so variable $x_1$ would exist only at site 2). Even indexed variables are replicated at all sites. Each variable $x_i$ is initialized to the value $10^i$.

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. (If two transactions have the same age then neither waits for the other.) This implies that your system must keep track of the oldest transaction time of any transaction holding any given lock.

(Optimization observation: 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.

You may assume that the processors work in lock-step. That is, you may assume that all operations on a single line in the input files for your simulation occur concurrently and that we will ensure they don’t conflict.

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 file.) 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 given transaction. (Obviously, some of these operations may be blocked due to conflicting locks.) Input is of the form:

begin($T_1$) means that $T_1$ begins

beginRO($T_3$) means that $T_3$ is read-only and begins
R(T1, x4) says transaction 1 wishes to read x4 (provided it can get the locks or provided it doesn’t need the locks (if T1 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 copies of x6 (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.

Non-transaction commands (dump, fail, and recover) should be processed in a given timestep by the transaction manager before the transaction manager processes the transaction commands for that timestep. This is important to ensure that your simulation produces identical output to what is expected by the grader’s tests.

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 get the lock on x4

end(T3)

Your program should consist of two parts: the first part is 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 receives a read request from transaction T and cannot service 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 (we will guarantee that this doesn’t happen in our tests), so the buffer size for messages from any single transaction can be of size 1.

The second part of your program is a set of data and lock managers, one for each site, which 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. This makes all non-replicated variables available for reads and writes immediately upon recovery. 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 lecture notes on recovery when using the available copies algorithm). If all sites for a given replicated variable fail at once, that variable will be completely unavailable for reading until it has been written (think about why this should be true).
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.

4.1.2 Part 2 – due December 6, 2010

At this stage, your project should be ready to read the input files and execute concurrent workloads correctly, even in the presence of fail()’s.

In the next phase of the project, we would like the even numbered variables to be replicated in a different way. Recall that, up until now, they were in all sites. We would like their location and replication factor to be determined dynamically by a location map, which is itself a variable. Let us call it M. For simplicity, let us assume M is a global variable that is not at any site.

The location map contains a list of entries in the form “variable number : list of sites”. For instance, in the first portion of the project, M would have been:

\[
\begin{align*}
2: & \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10 \\
4: & \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10 \\
& \ldots \\
20: & \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10 \\
\end{align*}
\]

The map we will be using here initially assigns each variable to three consecutive sites in a round robin fashion.

\[
\begin{align*}
2: & \ 1 \ 2 \ 3 \\
4: & \ 4 \ 5 \ 6 \\
& \ldots \\
8: & \ 10 \ 1 \ 2 \\
& \ldots \\
\end{align*}
\]

Make sure to initialize M as above.

In what follows, we describe two situations where you will be manipulating M.
When a site fail()s, we need to update M by removing that site’s number from the list of sites. If a site is down for more than three time step, the system assumes that it won’t come back anytime soon. More precisely, if a site \( n \) fails at timestep \( t \), then when the system reaches timestep \( t + 3 \) the transaction manager marks site \( n \) as having undergone a long fail(). Some variables would become under-replicated: in our case, fewer than three replicas. You should identify the variables affected by a long fail(), pick new sites to store them, and adjust M accordingly. This rebalancing should occur at the beginning of timestep \( t + 3 \), before the transaction manager processes any transaction actions for that timestep.

Conversely, when a site recover()s you should look at M and determine whether other sites have more variables’ replicas than others. If so you should move some replicas to the newly recovered site so to as to restore load balance. Again, make sure to adjust M accordingly.

### 4.1.3 Running the programming project

You will demonstrate the project to our very able teaching assistant. You will have one hour to do so. The test should take only 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 though you can bring in a laptop.

The teaching assistant will make various testing scripts available early in the semester, as well as the expected output from those tests. When grading the final project, additional un-released testing scripts will be used as well. For your creative contribution (see below), you should also provide some testing scripts of your own that you will show to the teaching assistant and that exercise the functionality you have implemented.

The teaching assistant 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 teaching assistant may ask you how you would change the code to achieve some new function.
4.1.4 Documentation, Code Structure, and Creative Contribution

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. It should also include your creative contribution (discussed below).

The submitted code similarly should include as a header to each major function: 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. If you program in Java, please use JavaDocs for documentation.

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

Creative Contribution As this is a capstone course, each student (even if you’re working in a team of two) is expected to go beyond the basic design to an enhanced design. Here are some possible enhancements. If you would like to suggest others, I would be glad to listen.

i) Set this up as a multi-process application so the teaching assistant can force a failure by killing a process and all data structures are held in files.

4.2 Possibility 2 — Benchmarking/Tuning Project

Take a section of the tuning book and see whether its recommendations make quantitative sense on a real system that is available to you at a job. Use substantial relations, e.g. hundreds of millions of rows and up. Specify the database management system, operating system, hardware platform including disks, memory size, and processor. Back up your conclusions with graphs drawn from real data.

Code documentation should include a discussion of the major application
functions, their inputs, outputs and side effects (including to the database). If you use Java, please use JavaDocs. If your application is secret, you may have to sanitize the functions somewhat to avoid revealing corporate secrets.

You should be prepared to discuss your project with me or potentially in front of the class. As your creative contribution, you will try to discover a new tuning principle and apply it to your problem (this goes for each student if you are working in a team of two).

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.

Last class in October: design document.

Project is due on December 6. 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

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 before you go on. If you don’t have access to a real database systems having hundreds of millions of rows or more, please do the programming project above.

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

December 6, 2010: final report to me and begin to set up appointment for testing with grader.