Advanced Database Systems, CSCI-GA.2434-001
New York University, Fall 2024

instructor: Dennis Shasha
shasha@cs.nyu.edu
212-998-3086
Courant Institute
New York University
251 Mercer Street
NY, NY 10012 USA
Office Hour: On Thursdays at 10:30 am to 11:30 am at https://nyu.zoom.us/j/92842629337

September 5, 2024

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. You will even get to program in our system aquery:
https://github.com/josepablocam/aquery2q

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 Available for free from Phil Bernstein's website at Microsoft. You can find a copy in this current directory at ccontrol.zip

There are also optional books which are very nicely written:

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

problem sets (35%), project (65%).

LATE HOMEWORKS OR PROJECTS WILL NOT BE ACCEPTED without a note from your physician or from your employer. (We may 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. Please choose your partner carefully.

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 on December 6, 2024. It will be graded in the following week 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, team of one.
3. A paper on the topics I invite, team of one..

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

In groups of 1 or 2, you will implement a distributed database, complete with serializable snapshot isolation, replication, and failure recovery. If you do this project, you will have a deep insight into the design of a distributed system. Normally, this is a multi-year, multi-person effort, but it's doable because the database is tiny.

Data

The data consists of 20 distinct variables x1, ..., x20 (the numbers between 1 and 20 will be referred to as indexes below). There are 10 sites numbered 1 to 10. A copy is indicated by a dot. Thus, x6.2 is the copy of variable x6 at site 2. The odd indexed variables are at one site each (i.e. 1

+ (index number mod 10)). For example, x3 and x13 are both at site 4. Even indexed variables are at all sites. Each variable xi is initialized to the value $10i$ (10 times i). Each site has an independent Serializable Snapshot Isolation informaiton. If that site fails, the information is erased.

Algorithms to use

Please implement the available copies approach to replication using serializable snapshot isolation (SSI) and validation at commit time. Though I won't require that you implement a distributed version of SSI, please use the abort rule for writes on the Available Copies algorithm anyway (if T writes to a site s and then s fails before T commits, then T should abort at end(T); you need not do that for reads), because a truly distributed implementation would have local information that would disappear on failure. Please use the version of the algorithm specified in my notes rather than in the textbook, just for consistency. Note that available copies allows writes and commits to go to just the available sites, so if site A is down, its last committed value of x may be different from site B which is up.

Detect concurrency-induced abortion conditions due to first committer wins and due to consecutive RW edges in a cycle. There is never a need to restart an aborted transaction. That is the job of the application (and is often done incorrectly).

Here are some optional implementation suggestions that may be helpful. In which situations can a transaction read an item xi?

1. If xi is not replicated and the site holding xi is up, then the transaction T can read the value of xi as of the time that T begins. Because that is the only site that knows about xi.

2. If xi is replicated then T can read xi from site s provided xi was committed at s by some transaction T' before T began and s was up all the time between the time when xi was committed and T began. In that case T can read the version that T' wrote. If every site containing xi failed between the last commit recorded on that site and the time T began, then T can abort because no end(T) will ever be forthcoming. Here is an example showing why: Suppose x has copies on sites A and B. T1 commits x on site A, then site A fails, then site A recovers, then T begins. In the replicated case, site B might have received a write on x, but at the time of the read, site B might be down, so T cannot be sure of the value of x before T begins.

Implementation: for every version of xi, on each site s, record when that version was committed. Also, the TM (transaction manager) will record the failure history of every site. Note though that in real implementations the data manager will record the commit time of each transaction and the recovery time if that data manager is rebooted; that will be an approximation of the failure history.

Examples:

- Sites 2-10 are down at 12 noon. Site 1 commits a transaction T that writes x2 at 1 PM. At 1:30 pm, a (serializable snapshot isolation) transaction T' begins. Site 1 has been up since transaction T commits its update to x2 and the beginning of T'. No other transaction wrote x2 in the meantime. Site 1 fails here at 2 PM. T' can still use the values it had from site 1, because all transactions use serializable snapshot isolation.

- Sites 2-10 are down at 12 noon. Site 1 commits a transaction that writes x2 at 1 PM. Site 1 fails at 1:30 PM. At 2 pm a new transaction T' begins. T' should abort because no site has been up since the time when a transaction commits its update to x2 and the beginning of T'.

Test Specification

When we test your software, input instructions come from a file or the standard input, output goes to standard out. (That means your algorithms should not look ahead in the input.) Each line will have at most a single instruction from one transaction or a fail, recover, dump, end, etc.

The execution file has the following format:

begin(T1) says that T1 begins

R(T1, x4) says transaction 1 wishes to read x4. It should read any up site and return the commited value when T1 started if available. It should print that value in the format
x4: 5
on one line by itself.

W(T1, x6,v) says transaction 1 wishes to write all available copies of x6 with the value v. So, T1 can write to x6 on all sites that are up and that contain x6.

dump() prints out the committed values of all copies of all variables at all sites, sorted per site with all values per site in ascending order by variable name, e.g.
site 1 – x2: 6, x3: 2, ... x20: 3
...
site 10 – x2: 14, ... x8: 12, x9: 7, ... x20: 3
This includes sites that are down. Down sites should not reflect writes when they are down.

in one line.

one line per site.

end(T1) causes your system to report whether T1 can commit or abort in the format:

T1 commits
T1 aborts

Note that T1 will abort if (i) some data item x that T1 has written has also been committed by some other transaction T2 since T1 began, (ii)) any reason having to do with the available copies algorithm. T1 may abort there are two RW conflicts in a row in a serialization graph cycle and T1 is in at least one of those conflicts.

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 in the input means time advances by one. There will be one instruction per line.

Other events to be printed out:
(i) when a transaction commits, the name of the transaction;
(ii) when a transaction aborts, the name of the transaction;
(iii) which sites are affected by a write (based on which sites are up);
(iv) every time a transaction waits because a site is down (e.g., waiting for an unreplicated item on a failed site).

Example (partial script with six steps in which transactions T1 commits, 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)
// T2 should do this write locally (not to the database). Note that at least
// one of T1 and T2 will
// abort when the second one reaches end.
end(T1)
end(T3) // This will commit
end(T2) // This will abort because T1 performed a committed write first and both wrote x1

Design

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 by using a shared disk configuration. For this project, the transaction manager is also fulfilling the role of a broker which routes requests and knows the up/down status of each site.)

If the transaction manager TM requests a read on a replicated data item x for read-write 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. Note that T must have access to the version of x that was the last to commit before T began. (As mentioned above, if every site failed after a commit to x but before T began, then T should abort and will never reach an end(T).)

If a a data manager (DM) fails and recovers, the DM would normally decide which in-flight transactions to commit (perhaps by asking the TM about transactions that the DM holds pre-committed but not yet committed), but this is unnecessary in this project, since, in the simulation model, commits are atomic with respect to failures (i.e., all writes of a committing transaction apply or none do).

Upon recovery of a site s, all non-replicated variables are available for reads and writes. Regarding replicated variables, the site makes them available for writing, but not reading for transactions that begin after the recovery until a commit has happened. In fact, a read from a transaction

that begins after the recovery of site s for a replicated variable x will not be allowed at s until a committed write to x takes place on s.

During execution, your program should say which transactions commit and which abort and why. For debugging purposes you should implement (for your own sake) a command like 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.

If a transaction T writes an item at a site and the site then fails, then T should continue to execute and then abort only at its commit time (unless T is aborted earlier for some other reason).

Output of your program:

Your program output will execute each test case starting with the initial state of the database. Your output should contain: (i) the committed state of the data items at each dump.
(ii) which value each read returns as it happens (iii) which transactions commit and which abort. (iv) anything in the "Other events to be printed out" from above

4.1.1 Running the programming project

If our very able grader has system style problems with your project, he may contact you to work with them to resolve them. You will not be permitted to hand in a new version of the project, but you can ask the grader to change a few things to get the project to run. If you do have such a meeting, you will have 15-30 minutes to present. The test should take a few minutes. The only times tests take longer are when the software is insufficiently portable. You will ensure portability by using Reprozip.

The grader will ask you for a design document that explains the structure of the code (see more below). The grader will also take a copy of your source code for evaluation of its structure (and, alas, to check for plagiarism).

Please note that you ARE permitted to use large language models in your homeworks and projects, but you must identify which code comes from such a model and which prompt you used.

4.1.2 Some Questions and Answers

1. When does the transaction manager learn about failed sites?

   Answer: immediately upon the tick which has say fail(3)

2. When a transaction Tj that has written v into xi aborts, what should the value of xi be?

   Answer: The value before Tj started. Before-image of xi.

3. The reason a transaction Ti must abort when it wrote to a site s that failed between that access and the end time of Ti is that the site s would have lost the fact that another transaction may have committed something on s.

4. When a transaction aborts due to concurrency control or failure reasons, there may not be an end statement from that statement and no other operation from that transaction.

5. If a transaction must read from a particular site and that site is down, the transaction will wait.

6. We will never cause all sites to fail.

4.1.3 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. That is, we should see the major components, relationships between the components, and communication between components. This should work recursively within components. A figure showing interconnections and no more than three pages of text would be helpful.

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.

Finally, you will use reprozip and virtual machines to make your project reproducible even across different architectures for all time. Simply using portable Java or python or whatever is not sufficient. Documentation and packaging using reprozip and doing so correctly constitutes roughly 30 of the 170 points of the project grade.

4.2 Submission Instructions

```
- Sign in to https://brightspace.nyu.edu with your NYU NetID and password.
- Under the 'My Courses' section, select the course 'Advanced Database Systems CSCI-G
- Navigate to the 'Assignment' tab.
- Please submit your source code as text and your executable
using reprozip (which can be made to produce a docker file).
```

4.3 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.

4.4 Possibility 3 — Paper

This year, I invite papers on the topic of a survey of systems for replicated lock-free databases.

If you want to write a paper though, it's a one person job and we need to discuss beforehand. Your model for a survey paper should be ACM Computing Surveys. Also, you should create four non-trivial problems based on the material you read and give the solutions.

5 Project Schedule — depends on project you choose

5.1 Schedule for Programming Project

Last class in September: Letter of intent if you want to do something other than the replicated concurrency control project. Partner chosen if any. We will need to discuss.

Late October: design document. Please submit this document through brightspace as for the final submission.

Project is due on December 6, 2024 at 11:59 pm. Please submit your project to NYU Classes. Between December 6 and December 12, your project will be graded. The grader will reach out to you if there is any issue.

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, 2024: final report/demo to me.