

Programming Project II
January 24th, 2012

Purpose of the Project:

The goal of this project is that you develop a Concurrent Index (CI) mechanism (and pertinent auxiliary structures) to help retrieve and manipulate data items from a relational table. The CI can be any *disk-resident* structure of your own choice: *ISAM*, *B+tree*, *Extensible Hashing*, and *Linear Hashing*. The key requirement is that CI enables fast retrieval of data items (records) in a file of data (relation); CI has to also support simplified database operations on a data file of records.

Outline:

User access and manipulation to the CI and ultimately of the data file will have take place through *transactions* that support the following set of operations:

1. *b_xaction*: a transaction commences.
2. *insert a-data-record*: a record is inserted in the data file. The record features a *key* (of your own choice such as numeric, alphanumeric, etc.) that uniquely identifies the record among those already in the file. The insert operation might create the expansion of the CI (index-file) in order to accommodate for a new key value. The nature of the record is a matter of your own choice as well and remains fixed in terms of length and content at all times.
3. *delete key*: the record with identifier *key* in both data-file as well as index-file have to be removed. You have to adhere to the properties and requirements of your selected indexing structures (for example, if you use B+Tree then the utilization of nodes has to remain higher than 50% at all times).
4. *lookup key*: retrieve and display the entire record that is uniquely identified by the key in question.
5. *update-a-data-record-with key*: retrieve the record (as above), update some of its non-key element(s), and finally, store the result (flush to the disk).
6. *range key1 key2*: retrieve all records having key value between *key1* and *key2*. This might not be a feasible operation in some of the CI structures.
7. *wait x*: the transaction waits doing nothing for *x* seconds. This allows for imposing “artificial delays” (may be very handy in testing things out).
8. *abort*: the transaction unilaterally decides to abort. No long term effect should take place on both the data file and the CI index.
9. *commit*: all (potential) changes become permanent and are reflected on the disk. Next instruction to be executed is always *e_xaction*.
10. *e_xaction*: terminate transaction.
11. *bulk-load file-of-records*: insert all records that appears in file provided as argument in the command.

12. bulk-delete file-of-keys: delete all records whose key appears in the file given as argument in the command.

Data records and index-values can be “simultaneously” accessed by multiple *concurrent* application programs (that can be realized as threads or processes) simply called *clients*. Such clients are essentially programs or entry points to the your system that help launch *transactions*.

We can assume that *transactions* “know” which specific data objects they intend to read/write one at a time; there is however no global a-priori knowledge of all the data (records, indexed values) to be accessed and/or manipulated while a transaction is in progress.

Transactional processing will have to be “handed over” to the core of your system (also called *engine*) for execution. In this respect, you will have to develop an interface positioned between the user(s) who initiate transactions and the Concurrency Control Manager (CCM). CCM has to provide orderly access and manipulation of required structures and data elements. The manager should coordinate concurrent accesses of disk-resident data records and keys (in the index file CI). As mentioned above, each data record has a format of your own choice and features a unique identifier (such as a customer id number) that may be integer, alphanumeric, etc.

A transaction is not and cannot be aware ahead of time of all the specific operations that is about to execute (i.e., reads/writes from/to the disk) during its lifetime. Operations within a transaction will be carried out one at a time and at the end writes to permanent storage occur just after **commit** (i.e., rigorous commit). The execution of transactions always follows the *ACID*-ic model.

The *CCM* should feature the following:

- Serializable access to records should be based on any version of the two-phase ($2-\phi$) locking protocol that allows for maximum concurrency [BHG97]. Also, you should attempt to provide for a flexible yet correct mechanism to increase concurrent access within the index.
- The *CCM* should be able to detect deadlocks and successfully recover if required by aborting one or more of the threads that are involved. A heuristic of your choice should be used to recover from a deadlock.
- Your *CCM* should preferably function atop a simplified buffer area (of frames) that can “host” up to a maximum (finite) number of pages. Pages (or frames) do have a fixed size at all times and this is provided at program initiation time. A page should be able to at least host one (or more) records. Also, the number of buffers available should be passed to the server as a parameter at the time of its invocation. While pages are in memory, they stay there for as long as it is necessary in order for the respective transaction to complete its work.
- Ideally, your *CCM* should be capable of producing recoverable schedules from failures and/or unilateral transaction aborts.

Overall, your engine should be realized as a multi-threaded server that accepts requests from multiple simultaneously running clients. Each client furnishes one transaction at a time. Thus, clients essentially generate the “traffic” of *reads* and *writes* for data records and key values to data file and CI respectively and interface with CCM to have their operations carried out in *ACID*-ic fashion.

Overall Organization:

The *multi-threaded server* either retrieves or writes data objects back to the disk on behalf of clients. We assume that all data records have legitimate content (value) making together a single file. How matters are

organized within a file, it is entirely your own choice.

Clients create sequences of operations that insert, lookup, update and delete records, from the disk-resident file(s). Once data are in memory, they remain in place so that the work of involved transaction(s) is completed. Insertions/deletions (updates) to data remain in buffer frames and are all flushed out to disk at commit time. Evidently, transactions that do not eventuate due to deadlock or unilateral elect to abort have no effect on the disk-resident structures and data.

The server should comply with the requirements of (a version of) 2- ϕ locking protocol (*rigorous 2- ϕ* is a plausible choice). Our main goal is to provide maximum serializable access to all records going always through the CI. To this end, you should think of ways to demonstrate in a comprehensive manner the *correct* operation of your engine and clients as soon as testing of your work begins.

Figure 1 depicts the functionality as well as the likely position of the various components your engine has

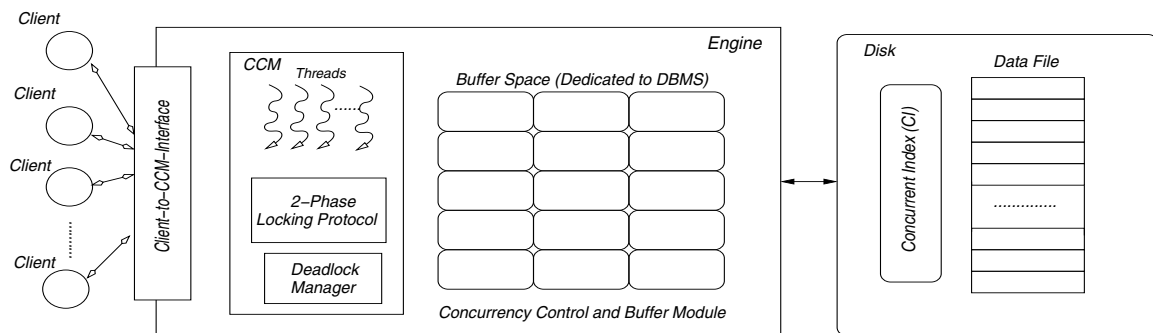


Figure 1: A likely organization of various functional elements

to implement. The *Concurrency Control Manager (CCM)* contains all the essential features that implement the concurrency control policies, the component that helps recover from deadlocks, multiple threads for the realization of the work that the clients ask the server to do on their behalf, and finally the basic interaction to the simplified buffer manager. The latter features a constant number of frames and may work with a scheduling policy of your own choice.

Constraints:

1. You can pick **one** partner to work with and implement the project. No parties of three or more are allowed.
2. You can use any language you want and you any computing environment you are more comfortable with for development.
3. If you consider developing B+Tree structures follow the routines we talked about in class [Del12] or those discussed in [Jan95]. Alternatively, you can carry out an algorithm that you will have to describe in your own write-up.
4. You may use the *Pthreads* library [LB98] to implement the engine and its synchronization primitives.
5. The functionality and design of the clients and of the internal structures are entirely up to you. The same goes for the organization and maintenance of the data file (which actually should be as simple as possible).
6. The maintenance of *wait-for* graphs and any other meta-data you elect to use should be part of your engine design.

7. It is rather *important* that you can automate the function (ie, operation) of clients so that you can test your engine in the presence of tens (or even hundreds) of concurrent transactions.

What you need to submit:

- All your work in a tar-ball.
- A design document where you present your ideas/decisions about your program(s). This should be only a few pages long.
- A methodology (i.e., a disciplined way) that helps demonstrate the correctness of concurrent transactions.
- Sample runs that show the concurrent execution of your program(s).
- Demonstration of deadlock cases and recovery from such situations.

Deadline and Demonstration:

1. The deadline of the project is March 15th or earlier.
2. You will have to demonstrate your work.

References

- [BHG97] P.A. Bernstein, V. Hatzilakos, and N. N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, available from the web page of the course, Reading, MA, 1997.
- [Del12] A. Delis. B+Tree Iterative Insertion and Deletion Routines. Dept. of Informatics, Univ. of Athens, January 2012. <http://cgi.di.uoa.gr/~ad/MDE515>.
- [Jan95] J. Jannink. Deletion in B+-trees. *ACM SIGMOD RECORD*, 24(1):33–38, 1995.
- [LB98] B. Lewis and D.J. Berg. *Multithreaded Programming with Pthreads*. Prentice Hall International, Mountain View, CA, 1998.