ECSE 4961 Final Project

Ryan Walton

May 2, 2022

1 Introduction

Database logging is a hot topic in database system development recently. The standard of logging, which has been used for many years, is called ARIES. One drawback of the ARIES logging system is the fact that a large log file must be read back upon recovery.

This project aims to provide a modification to the logging scheme, which we will call "Highly Distributed Logging".

2 Theory

2.1 Requirements

The goal of a database logging system is to provide ACID:

- Atomicity

- Consistency

- Isolation

- Durability

In this project, atomicity is assumed at the B+ Tree level, as there are existing implementations maintaining atomicity at this level. Instead, the project is simplified and logging is implemented to provide atomicity at the transaction level.

Consistency is typically maintained by including UNDO log entries in the global log. With highly distributed logging, we can eliminate the need for these log entries. The nature of the distributed logging allows us to never (or very rarely) modify the B+ Tree page entry before the transaction commits.

Isolation is not maintained in this project, it is up to the developer to design isolated transactions.

Durability is typically maintained by including REDO log entries in the global log. With highly distributed logging, we hold these REDO log entries in a short term file system FIFO, and then move these entries into to the B Tree file itself.

2.2 Advantage

Since the distributed logs each take up very little space, this logging scheme leverages the transparent compression feature of new computational storage drives. Since each log is spatially correlated to the page and values it logs, recovery from system failure could potentially be immediate/unnoticeable.

3 Implementation

This project aimed to follow the structure of the block diagram shown in Figure 1. Some of the features were not implemented due to time constraints.

If fully implemented, the system would use an SSD to hold the B Tree, the distributed logs, the short global logs, and relevant transaction status. The database manager would be able to read in data from the B Tree file, including the distributed log, and combine log data with transaction context to create a consistent cached tree node in RAM. Processes would be able to interact with the DBMS and perform modifications to the B Tree.

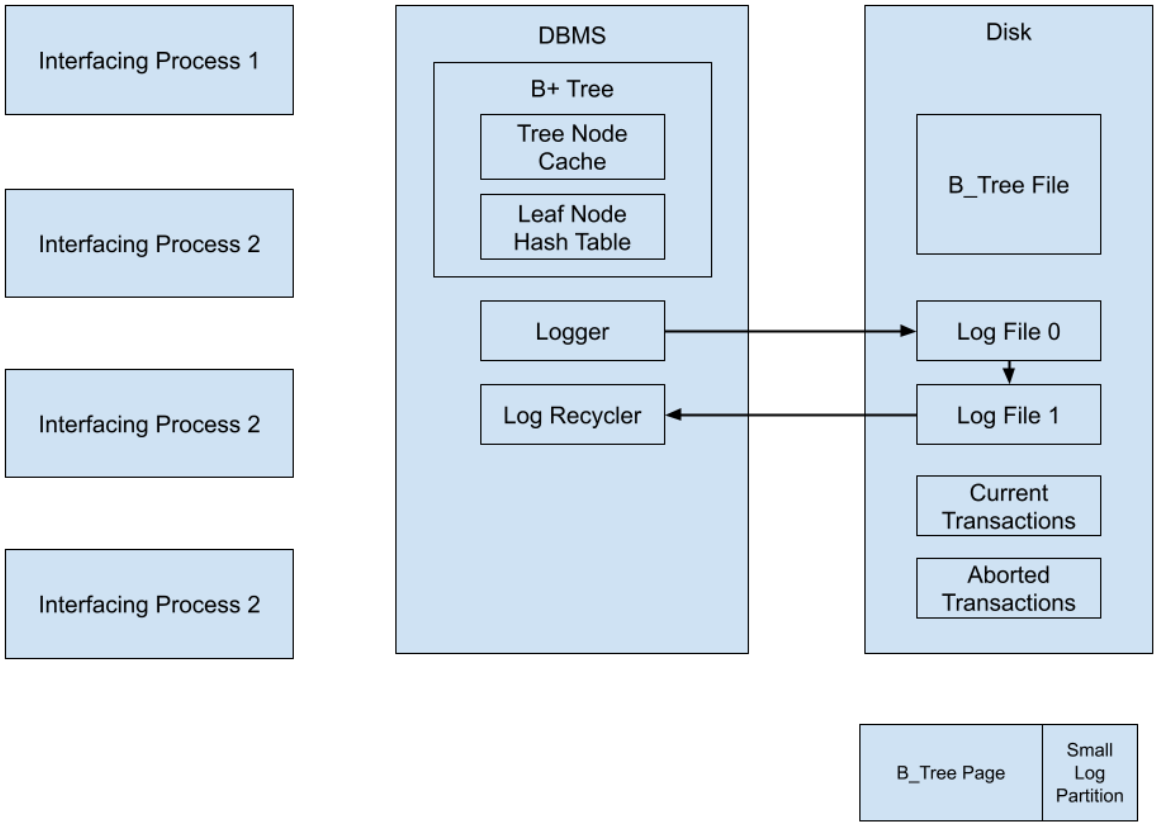

Figure 1: Database System Block Diagram

4 Testing

An extremely simple test of the B Tree and distributed logging is provided in main.cpp. This routine instantiates the B_Tree C++ object with the filename corresponding to the database file. It demonstrates the memory persistence functionality (using aforementioned file). It also shows how insertions can be done, and later undone, if a transaction fails to commit.

To run:

```
g++ *.cpp
./a.out
```

5 Limitations

Some aspects of the project were not yet implemented:

Multithreaded access

Global Log

Also, conceptually there are some loose ends that must be addressed:

B Tree operations not atomic

Undo log required for many updates to a single page before transaction commit

6 Conclusion

This project was a good first exploration into the implementation of database systems, and brought up important issues to take care of regarding database logging, caching, and especially the new method of distributed logging. The project could use some more work, and I believe it would be able to demonstrate the distributed logging functionality more fully.