Senior Project

Model Database Engine

Isabelle Sanford
Advisor: Geoffrey Towell

Submitted in Partial Fullfillment of the Requirements of the BA in Computer Science
Bryn Mawr College
Spring 2021

Abstract

The internal components of databases are generally very complicated in order to make the database secure, scalable, and efficient. This project is a model database engine, for both educational use and the ability to compare specific design and implementation decisions. Its aims were to be functional, simple, modifiable, and easy to evaluate.

Acknowledgements

I want to thank Professor Geoffrey Towell for advising me through the difficulty of this project and over the course of my computer science experience. Thanks to all my professors for being understanding and encouraging me to pursue my interests. And thank you to family and friends for supporting me through my time at Bryn Mawr.

Contents

1 Introduction

Databases are a vital part of everyday life, and a tool that many programmers interact with regularly throughout their careers. But modern database engines are extremely complex and difficult to understand, because they're made to be safe, fast, generalized, and scalable. This is great for end users. They don't care about the inside of a database. However, some people do need to actually understand what's happening and how the engine fits together - e.g. database admins, data scientists, and the database engineers.

Unfortunately, education about database internals is analagous to that of operating systems or compilers - the theory isn't especially hard to teach, but practical examples are a different story. Looking at the complete internals of a modern database is way too complex. Even examining individual parts of a theoretical engine is difficult because of how intertwined everything is in a commercial database product. The natural answer, then, is a simple (but complete) 'model' database engine with clear internals. And that would be somewhat helpful in itself, but ideally you'd also want the person using this as a learning platform to be able to see the difference with or without certain features or optimizations. These desires suggest the usefulness of a model database engine that's easily modifiable and provides a substantial evaluation functionality. The aim of this project is to build such a system.

2 Solution

The model database engine has four major requirements.

1. The database engine must be **functional**, i.e. a user can submit a SQL query[1] and the engine will return the appropriate data. This includes creating tables, inserting data, modifying or deleting data, and querying the database.

2. The engine must be **simple** and understandable. This means good coding style in general (in terms of variable names, encapsulation, comments, etc) to explain what does what, but it also means opting for simple or naive implementations over technically-better but much more complex ones. This can also mean not handling some edge cases; but if the engine breaks it should do so in an understandable and obvious way. Unhandled problems should also be documented.

3. The database engine must be easily **modifiable**. One of the goals of this engine is to be able to easily try different implementations of database parts, to compare the performance of one implementation against another. That means strong encapsulation, so that changing one object internally does not affect other parts of the database. Particular attention should be paid to what other common implementations look like, to ensure that those features would be reasonably easy to add.

4. The engine must be able to **evaluate** its processing and performance. Changing pieces of the database is essentially useless unless you can see the effects. So all parts of the engine should have extensive logging, and it should return top-level results to the user automatically.

Note that this database engine is intended for testing and education; it does not have many of the security features most databases do (which is partly by design - those features can make the engine

[1] For at least a subset of SQL; the entire language doesn't have to be implemented.

substantially more complex). So it should not be used to store sensitive data. Neither should this database engine be used to store data that will be relied upon later. Even without alteration, the engine does not guarantee that data will retain accurate values, be retrievable, save updates made to it, etc.

3 Design

This database engine follows the basic structure of any relational database, and this section is primarily an outline of what that structure entails. Ultimately, the first underlying concern in every design choice for a database engine is how often the database talks to the disk. Retrieving information from the disk is much slower than any other operation the engine might do. A large portion of any database management system exists purely for minimizing those reads and writes.

The second consideration is database integrity. Any database engine tries to ensure that data is not corrupted or lost if client programs are interrupted or run concurrently, or if the database suffers failure. This consideration is focused much less in my database, because integrity concerns add a lot of complication obscuring the underlying data structure. The hooks for implementing full integrity checking exist in this engine. But they are only hooks without significant implementation, because integrity is one of the main reasons commercial databases are so difficult to understand.

The basic function of each component in Figure 1 is given below. More specific implementation details are given in the next section.

1. File Manager & Pages

The file manager is the component that talks directly to the disk. It does that by dividing files into blocks, which are sections of the file that are the same length as the input/output page provided by the operating system. This means block accesses will be equivalent to page accesses by the OS, which ensures we control when and where the OS actually accesses the disk.

The blocks themselves have basic utilities for the engine to use: at any given spot in the block, they can read or write whatever types the engine supports (integers, strings, etc.).

2. Log manager & Transactions

The log manager is notable in that it is not particularly concerned with disk accesses at all. Its function is to keep the data safe and accurate. Usually, this means dealing with concurrency issues, thread locking, potential crashes, etc. The log manager is so named because it keeps a log file in which it records *transactions*, which are discrete requests from a client that the database performs all at once.

3. Buffer Manager

The operating system of a computer keeps a set of pages in active memory so that it doesn't have to ask the disk every time if the page is in frequent use - this is the OS cache. And while operating

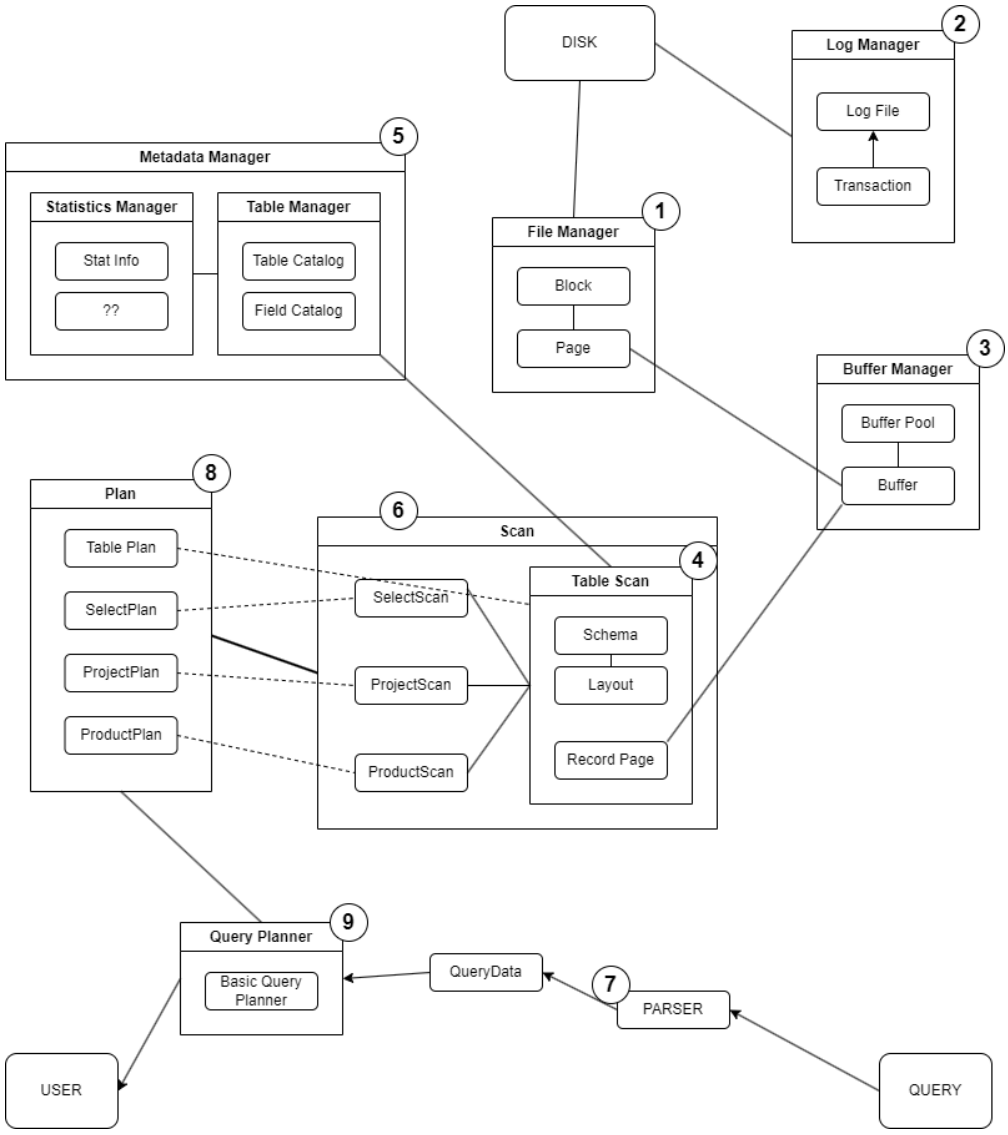

Figure 1: The basic structure of the database engine. Numbers correspond to descriptions of each component on the following pages. For simplicity, the diagram does not show some duplicate structures used to update the database (create tables, insert data, etc).

systems are fairly decent at guessing which pages will and won't be used again, we can do much better with the knowledge of what's happening inside the database. So the Buffer Manager has a buffer pool, consisting of a list of buffers, which each contain a page. When a page is in active use, it is considered *pinned*. So when a query asks for some particular page, that page is read into a buffer (if it is not already in one), and given a pin. When that query finishes, it removes its pin from that buffer. If a buffer has no pins (i.e. no query is using the page it holds), it is considered unpinned, but it is not written to the disk immediately (or discarded if it involved only reading and no writing). Instead, unpinned buffers stay in the buffer pool until a query requests a page that's not currently held in the pool. At that point, the buffer manager picks an unpinned buffer to remove and replace with the new requested page. This ensures that pages in regular use are not often read from or written to the disk, but instead stay easily accessible in the computer's RAM.

4. Schema/Layout, Record Page, Table Scan

The page that a buffer holds is called a Record Page, which holds a block worth of information and provides the functionality to access each record within that block. But a Record Page only contains part of a table, unless the table is quite small, and so it is used by something else - the Table Scan - to interact with the whole table. A Table Scan iterates through each record in the table (switching between Record Pages as needed) and can return or update the values within. It's similar to an iterator (like a tokenizer, maybe, or something reading through a CSV), in that it has a current record at which it is pointed, and all values it's asked for come from that record until it's told to move elsewhere.

In order to know where and what to read, though, the Table Scan needs to know how the table is set up. If it's asked to return the value of a specified field, it needs to know the type and location of that field, or it'll return bytes of gibberish. So there needs to be a way to formally specify that structure, which in this database is done in a Schema and a Layout. The Schema is provided by the user when creating a table, and specifies the name and type of each field. The Layout is an internal version with additional information which the database engine creates from the Schema. Layouts contain the Schema itself, along with a map from each field to its length (in bytes), and an integer representing the sum of all those lengths, or the *slot size* of each record. The Table Scan uses this to know where the start of the next record is, and exactly what offset from that point to ask for if asked for a specific field.

5. Metadata (Table Manager & Stats Manager)

The Metadata Manager keeps track of information about the database, and contains several sub-managers.

The Table Manager is where the database engine keeps track of its tables and their fields, with two special tables which catalog the tables and what fields each table contains (including their types). These catalogs are used to construct Schema/Layouts for the Table Scans to use.

The Stats Manager keeps track of statistical information about each table. This includes stats like the number of records in a table, the number of distinct records in each field, and the number of

blocks a table takes up. This is useful particularly for query optimization, e.g. for deciding in what order to execute operations depending on table lengths.

Databases will often contain other managers here for other features which have not been implemented in this particular database (e.g. indices, views).

6. Scans (select/project/product)

A Scan is the output of a relational algebra operation. A Select Scan, for instance, takes a predicate and an underlying scan (maybe a Table Scan) as input, and then calls on it return in the same way they would for a Table Scan. That is, asking for the `next()` record in any scan will return a record, regardless of what the underlying scan type is. A query is essentially a tree of scans inside each other, where the outermost scan asks its inner scan for the next record all the way down to the most internal scan, which is always a Table Scan.

Here, a Select Scan corresponds to filtering out rows (i.e. the `WHERE X = Y` part), a Project Scan filters columns (i.e. the `SELECT A, B, C` part[2]), and a Product Scan combines two tables together to get a combined bigger table (this is the `FROM T, U, V`).

7. Parser

The parser translates a SQL string into a form the database can understand. The subset of SQL allowed by this engine is quite limited, and mainly consists of the following:

```
SELECT col1 [, col2 , ...]  FROM table1 [ , table2 , ...]  [WHERE <Predicate>]
```

Here a `<Predicate>` can consist of any number of *terms*, connected by `AND`s. A term is either in the form of `field1 = field2` or `field1 = const1` (e.g. `COLUMNB = 'e'`).

The only other functionality is similarly simple CREATE, UPDATE, INSERT, and DELETE statements. The parser takes strings that are valid within these rules and converts them into Data objects (QueryData, CreateData, ...) specifying the information that the Plans will use to perform relational algebra.

8. Plans (table/select/etc)

A Plan is a piece of metadata that takes part of a query and decides how to implement it as a Scan. This is especially notable with a Product Plan, because the direction in which a product is taken can have a substantial impact on the number of reads/writes, depending on which table is used first. So a Product Plan takes two sub-Plans (with their associated Scans), decides which direction to cross them, and then its `open()` method returns the Scan it decided on.

[2]That naming is indeed confusing; it follows that of the text which this structure is based upon.

9. Planner

Finally, the Planner puts everything together. It takes a piece of QueryData from the parser and turns it into a set of Plans attached to the appropriate Scans, which will appear to the client as a TableScan with the appropriate results. (Or, if the SQL string is an update command rather than a query, it'll insert/delete/modify data as required.)

4 Implementation & Modification

Below are both notes on the specific implementation of each component, and different modifications that could be made to them.

1. File Manager / Page

This database only has two types: INT and VARCHAR. This helps with simplicity, since there tends to be a lot of duplication in the code with one not-quite-identical function per type.

Modification: Some other types (like FLOAT) would have to be implemented here, while others like DATE could be built on top of the two current types. Block size is also in theory modifiable, although anything other than matching the OS page size should be substantially worse.

2. Log manager / transactions

This is one of the most complicated parts of a commercial database; this model database pares it down to nearly nothing. Users still use and commit transactions, and the log manager does write them to a log file, but not in a way that ensures database integrity. There's just enough to it that transactions could be implemented fully without having to overhaul the rest of the database, but that's all.

Modification: honestly not really implemented enough to be tested. There are efficiency improvements that can be made for logging, but (I think?) too small to be really useful. If concurrency were implemented, there'd be a lot more to play with here.

3. Buffer Manager (/ pool / caching)

A buffer manager is created when a database is initialized. It holds a list of buffers (the number of which can be specified at creation) and tracks how many are available (unpinned). Each individual Buffer tracks the block/page it holds, the transaction it was used in, and the number of pins it has. The notable implementation choice here is in how the buffer manager chooses which unpinned buffer to replace with an uncached page. This engine uses the most naive strategy possible, i.e. it picks the first unpinned buffer in its list of buffers. There are several more optimal algorithms here, which can make a notable difference in speed.

Modification: The size of the buffer pool and the algorithm for replacing unpinned pages are the two interesting modifications for testing or evaluation.

4. Schema/Layout, Record Page, Table Scan

A Record Page only contains the ID of a particular block and a Layout object to specify what fields are where and how big a record is. This database stores records in the simplest way possible: a record cannot be split over two Record Pages, fields are assigned to bytes in the order given by the client, and strings are always allocated the entire space which the user specifies as a maximum. These decisions make RecordPages very simple, but potentially very space inefficient (e.g. if a user allocates `VARCHAR(200)` but every string in the table is less than 10 characters long). Space inefficiency means that fewer records will fit in a given block, or equivalently that tables will require more blocks to store all their records. This in turn leads to time inefficiency, because more blocks in a table means more disk reads and writes.

Modification: Changing the way that records are structured/stored in a file is the most notable implementation choice, which could be improved not only with regards to the space inefficiency discussed above, but also by implementing indices.

5. Metadata (Table Manager & Stats Manager)

The Metadata Manager is an object which serves as a wrapper around several managers the database uses. This database has a Table Manager and a Stats manager, which are detailed below. Other possible managers could be held inside the Metadata Manager, associated with features unimplemented in this database (e.g. views or indices).

The Table Manager contains two tables: a table catalog and a field catalog. The table catalog is a list of every table in the database (including itself and the field catalog) and their associated record lengths. The field catalog has a row for every field in every table, along with the field's type, the table it belongs to, its offset from the beginning of a record, and its own length. Together, these two catalogs are used by the table manager to store and provide information about tables to the rest of the engine. The Table Manager has three external functions: providing the Layout of a specified table (which it constructs from the field catalog), creating a table (by adding its information to both catalogs), and printing a formatted version of a specified table.

The Stats Manager holds a map of every table name (which it gets from the Table Manager) to a StatInfo object about the table. It has a `numcalls` variable that keeps track of each time it's asked to provide statistics, and every *n* calls (default *n* is 100) it runs a `refreshStatistics` method (which can also be called externally) that recalculates all of its StatInfo. Note that `numcalls` will not only be from requests by the client; statistical information is also used by the Planner when deciding how to construct queries.

A StatInfo object holds three pieces of information: the number of blocks a table is using (effectively the file size, or the number of disk reads required to go through the whole table), the number of records in the table (so the number of distinct iterations required to go through each row), and the number of distinct values for each column/field. The number of distinct values is held in a map from field names to an integer that's supposed to represent the number of distinct values; this database engine uses the incredibly naive strategy of guessing the number of distinct values based on the number of records. So for any field in a given table, the Stats Manager will return `distincts[anyfield] = numrecs / 3`.

7. Parser

The parser takes a SQL string as input and returns a QueryData (or UpdateData) object. This process is pretty standard tokenization and parsing based on a specified strict grammar. The string is divided into three parts: the `selectList`, which is the list of fields after the `select` keyword; the `tableList`, which is the list of tables after the `from` keyword; and the `predicate` which is everything in the `where` clause. That predicate is further broken down into *terms* that connect with `AND` booleans. Terms are fields or constants connected with an equals sign (except `const1 = const2` is not a valid term; at least one side of the term must be a field).

Modification: As noted previously, the grammar allowed for this database engine is very limited, so there's plenty of room for adding features here. But if the parser is making any difference whatsoever in the engine's performance, something is very wrong.

8. Plans (table/select/etc)

Plans are very simple objects containing a piece of data and another Plan (except for a TablePlan, the innermost plan for any query, which contains a table name and its layout). Similar to Scans, Plans have a shared set of methods which act like a single Table Plan from the outside. Those methods are statistical information (used to choose which plans are most optimal), a Schema, and an `open()` method that returns a Scan that can be iterated through. A Select Plan stores a predicate, a Project Plan stores a Schema (with only the fields that the query asked for), and a Product Plan stores the two plans of the tables it's combining and their combined Schema.

Modification: mostly better statistical information that would be more useful for the Planner (but even then that might be something for the Stats Manager?).

9. Planner

In this engine, the planner takes all the products first, then does all the selects (filters out rows), then the projects (takes only the requested columns). This is decidedly not optimal, but it is simple.

Modification: Lots of testing / optimization here around how queries are structured. There's a QueryPlanner interface object, which is what the client sees, and the BasicQueryPlanner implements it and is used currently but could be switched out without difficulty.

5 Evaluation

This database engine's evaluation criteria are exactly those laid out in the Solution section. Those are:

1. **Functional**: The engine can create tables, insert and delete data, and return accurate results given a valid SQL string.
2. **Simple**: The database engine is about as simple as possible while still staying functional. It has no concurrency or integrity safeguards making it inscrutable, only two data types, and only the most minimal SQL grammar.

3. **Modifiable**: Every component of the database engine is encapsulated such that changes in one component's implementation will not affect any other component. The components and their connections are standard database structure, so the engine could theoretically could be modified all the way into a database with all the features of commercial ones.
4. **Evaluable**: The engine has an inbuilt statistics manager which can be used to see information about tables and queries (and could be extensively modified to provide more information). It can also of course be timed with Go's inbuilt timing functionality. This includes the ability to time individual components, since the database internals are so exposed and accessible.

6 References

Sciore, Edward. (2020). *Database Design and Implementation*. Springer.