ezDBMS

a simple implementation of RDBMS written in Python

Boyao Liu, Shuheng Gong
Graduate Students of Computer Science
Georgetown University

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ezDBMS

a simple implementation of RDBMS written in Python

***Abstract*—Abstract—The ezDBMS project meets all the grading requirements for the Georgetown COSC-580 final project. In addition to completing it as an assignment, we also gained practical experience in advanced concepts such as Atomicity and Durability, snapshots, rollbacks, and Logging with Log Levels that are commonly used in real-world projects.**

To develop the project, we utilized Antlr, which is the state-of-the-art tool for building DBMS architecture, instead of relying on regular expressions. Furthermore, we implemented a hand-written Treap, which we believe is an elegant solution for optimizing conjunctive and disjunctive condition ordering.

I. Introduction

We have implemented a simplified, single-user, in-memory relational database management system using Python. In this documentation, we will provide an overview of our design decisions and highlight our achievements as well as our deficiencies.

II. Problem Definition

We want to build an in-memory database management system, supporting mainly following features

SQL features

- Schema definition with primary & foreign key identification
- All primary single relation operations including "order by" and "having"
- Supported types: integer & string
- A single two-clause logical conjunction (AND) and disjunction (OR)
- A single aggregation operator (e.g., min, max, sum, etc.)
- A 2-table theta-join

Index support

- Single attribute primary key

Query optimizer

- Sort-Merge vs. Nested-Loop join selection
- Conjunctive and disjunctive condition ordering

Execution engine

- Ability to measure the execution time of any query
- Reasonable input / output format

III. Design Decisions

We try to divide this RDBMS into multiple modules, here comes its architecture:

A. Database Engine

The main entry point of the system waits for raw SQL inputs and calls other modules as needed. It includes a Timer to measure the execution time of the queries.

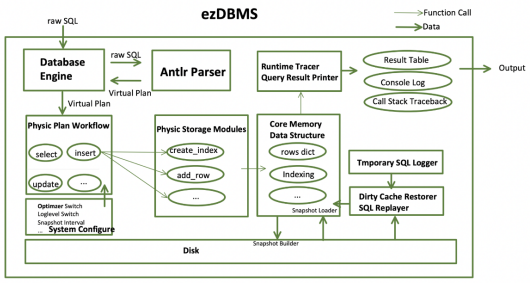

Fig. 1. Architecture of ezDBMS

B. Antlr Parser

Our system includes a parser that is based on code generated by Antlr [2] and follows the grammar of SQLite. This parser can traverse the Abstract Syntax Tree generated from the raw SQL input, extract the required information, and generate a logical execution plan.

We were eager to learn about the techniques used in modern DBMS for implementing their parsers, and we found that using Antlr was a more elegant and maintainable solution compared to using pure Regular Expressions. Therefore, we learned how to use Antlr and incorporated it into our system.

C. Physical Plan Workflow

This module has a highly customized, hand-written workflow for different types of SQL statements. It uses the logical execution plan generated by the parser to call physical storage modules and directly read and write the core data structure in memory.

The most sophisticated part of this module is the hand-written workflow for the $SELECT$ statement, which supports its full range of features.

D. Snapshot Builder&Loader and Lazy File Cleaner

1) Snapshot Builder&Loader: As ezDBMS is mainly an in-memory RDBMS, we designed the Snapshot Builder module to persist our data. This module periodically takes a snapshot of all our in-memory core data structures at a configurable amount of commands. The Snapshot Loader module then loads the snapshot during the system's boot.

Initially, we considered making the Snapshot Builder a separate thread with functions based on time intervals. However, due to time constraints and potential debugging issues with thread locks, we decided against it.

2) Lazy File Cleaner: To ensure that every SQL statement can replay on the latest snapshot and to concentrate disk read and write consumption, we implemented the Lazy File Cleaner module. This module defers the actual file modifications for delete statements until just before building the snapshot. This approach ensures that the delete statements do not interfere with the system's performance during the execution of SQL statements.

E. Exception Handler, SQL Replayer

1) Exception Handler: We implemented an Exception Handler module in ezDBMS to capture all exceptions that may occur during the execution of SQL statements. This module logs the exceptions with red reports at the terminal. In addition to standard exceptions, custom exceptions such as those that trigger foreign key constraints are also raised and handled by this module.

2) SQL Replayer: Similar to AWS Aurora's [3] design principle, which states that "The Log Is The Database", we implemented an SQL Replayer module in ezDBMS. This module is activated when an exception is raised, assuming that the core data structure becomes dirty, and it swipes them out. It then reloads the latest snapshot and replays all SQL statements after that, except for the one that raised the exception. The entire system then recovers to the exact state it was in before the bad SQL execution.

In our design, we did not persist the SQL log because we felt that discarding the log and deferring the commit would be more consistent in a system based on snapshots of a configurable amount of commands rather than a time interval. Since we already use a snapshot builder based on the amount of commands, we can defer the commit and still ensure that there is no data loss once committed.

F. Indexing

The indexing for ezDBMS mainly relies on a Hash table, a B-tree, and a Treap.

1) Hash table: The Hash table supports $O(1)$ lookup of a specific row according to its $uuid$. However, this powerful indexing is not directly exposed to clients because we want to use balanced trees as a general indexing method and avoid the possible linear factor caused by collisions.

2) B tree: B tree [1] solves equal condition on primary key in where-clause with no worse than $O(logn)$. We prepared a showcase to show its efficiency comparing with linear scanning for non-primary key where condition.

3) Treap: The hand-written Treap is designed to update and answer how many values in a range in no worse than $O(logn)$ complexity. This funtion helps to implement condition ordering.

In fact, for an in-memory RDBMS, we think it's reasonable to consider replacing B tree with Treap completely.

G. Constraints

1) Primary Key: Primary key constraint is maintained during insert, we will check if the new value in primary key has showed.

2) Foreign Key: For each relation we maintained two lists of dictionaries

foreign0:[{attr:, rela2:, attr2:},...]
foreign1:[{attr:, rela2:, attr2:},...]

$foreign0$ stores information $attr$ is referring to $rela2$'s primary key $attr2$
$foreign1$ stores information $attr$ is referred by $rela2$'s $attr2$

We can retrieve each foreign key constraint using these two lists. e.g. During operations like deleting a row, we will check if this relation's primary key is referred and showed in that relation, to decide whether it's able to delete it safely. If not, we will raise a foreign key error and tell clients what to do before deleting this row.

H. Optimizers

1) Conjunctive and disjunctive condition ordering: To wisely assign which condition tested firstly for each row, we will evaluate how many rows will satisfy for conditions, e.g. $id > 50$. This feature is supported by Treap with a complexity of $O(logn)$, more specifically, $rank()$ function in Treap.

2) Sort-Merge vs. Nested-Loop join selection: When it comes to join, we will wisely choose to sort then join or just join with a nested loop, according the following inequality.

$$nlogn + mlogm + n + m > n * m$$

I. Core Memory Data Structure

This module functions as the core in-memory storage of the whole system, it includes five parts:

1) baseDBDict: A map indexing $(relationName, uuid)$ to a relation's row, row is stored as a directory mapping attributes to values

2) BTreeDict: A map indexing $(relationName, attribute)$ to a BTree, each node's data of the BTree is a set of $uuid$, to be compatible with non-primary key indexing

3) TreapDict: A map indexing $(relationName, attribute)$ to a Treap root. Treap is a weak self-balanced tree based on randomness.

4) metaDict: A map indexing $relationName$ to the definition of types of the relation's attributes, it's used to guarantee data type is correct.

5) constraintDict: A map indexing $relationName$ to its primary key, foreign key constraint. We can look up which attribute is a primary key of the relation, then to maintain the consistency during inserting and so on. Similar for maintaining foreign key constraint.

IV. Itemization of our achievements

To save space, our implementation completes all grading tasks of Georgetown COSC-580 Project 3 just like we showed during the class demo. We will introduce some selected interesting highlight here.

A. Full Feature Select Supporting

Fig.2 contains an example of a select statement with all features we support.

```
SELECT rela_i_i_1000.key,
       Min(rela_i_i_1000.val),
       Avg(rela_i_i_100000.key),
       Sum(rela_i_i_100000.val)
FROM   rela_i_i_1000
       INNER JOIN rela_i_i_100000
               ON rela_i_i_1000.key = rela_i_i_100000.key
WHERE  rela_i_i_1000.key > 25
       AND rela_i_i_1000.key != 30
GROUP  BY rela_i_i_1000.key
HAVING Min(rela_i_i_1000.val) >= 27
        OR Avg(rela_i_i_100000.val) > 550
ORDER  BY rela_i_i_1000.key ASC
LIMIT  10;
```

Fig. 2. supported full feature select statement

Feature	Time Consumption
Optimization OFF	1.139s
Optimization ON	0.624s

TABLE I. CONDITION ORDERING OPTIMIZER

B. Atomicity at the level of a single SQL statement

We achieved Atomicity at the level of a single SQL statement. Since error may occur during the execution of a SQL statement and it may have modified some data. The memory becomes dirty. We capture all these running time error and swipe out the memory, load the latest snapshot and replay all logs after the snapshot. Then we perfectly clean new SQL statement's effect and achieve atomicity.

C. Durability using snapshot and rollback

All in-memory data will periodically persist to disk. Once commit, no data will be lost.

D. Conjunctive and disjunctive condition ordering

The effect of the optimizer is shown in Table I. It's inspiring to see that we achieved almost exactly twice faster just as expected. i.e. Most rows short circuits during conditional judgment.

E. Join on the largest and largest relations

Thanks to the powerful sort-merge optimization, even the join of two tables of 10^5 rows can be executed very quickly. Table II shows the specific time-consuming.

V. OUR DEFICIENCIES

A. Grammar Deficiencies

- CREATE TABLE, use 'int' other thant 'INT'
- UPDATE TABLE, don't use full name of a column
- USE 'AND OR'
- Must supply ASC or DESC in ORDER BY

Feature	Time Consumption
Optimization OFF	51min
Optimization ON	2.1s

TABLE II. SORT-MERGE OPTIMIZATION

B. Problems related to Antlr's issues

We found two interesting problems during parsing and we believe they're raised cause of Antlr. We are considering to double check and contribute to Antlr by issusing these problems

- Join operations like 'INNER' will be agglutinated to relation 1's name. We manually fix this problem.
- Limit_stmt and Order_by_stmt will be ignored in a SELECT statement containing JOIN but without GROUP BY. We didn't manually fix it.

APPENDIX A
SOURCE CODE

github.com/Drenight/ezDBMS

REFERENCES

[1] Zope Foundation. BTrees, October 2012. [Accessed: Feb. 10, 2023].

[2] Terence Parr. ANTLR 4 Documentation, October 2015. [Accessed: Dec. 5, 2023].

[3] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *SIGMOD 2017*, 2017.