Not-So-Bitter Pill to Swallow: Slipstreaming Memory Safe Programming via Rust as part of a Database Systems Course

Mohammed Suhail Rehman
Department of Computer Science
University of Chicago
Chicago, USA
suhail@uchicago.edu

Aaron Elmore
Department of Computer Science
University of Chicago
Chicago, USA
aelmore@uchicago.edu

Raul Castro Fernandez
Department of Computer Science
University of Chicago
Chicago, USA
raulcf@uchicago.edu

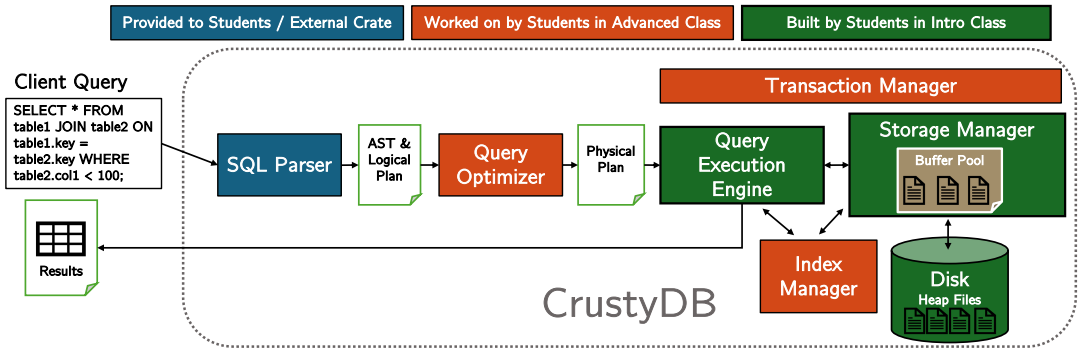

Figure 1: The architecture of CrustyDB, a simple relational database system implemented in Rust.

Abstract

In this paper, we present our experience integrating Rust, a memory-safe systems programming language, into an introductory database systems course project. Our findings indicate that while Rust's steep learning curve posed initial challenges, it significantly enhanced students' understanding of memory safety and systems programming concepts. We also discuss the outcomes of the course, which has now been taught to over 500 students over five separate offerings. While student feedback has been overwhelmingly positive, we provide insights for educators considering Rust for similar systems-oriented elective CS courses.

CCS Concepts

• **Social and professional topics → Computer engineering education**.

Keywords

Databases, Rust, Systems Programming, Memory Safety

ACM Reference Format:
Mohammed Suhail Rehman, Aaron Elmore, and Raul Castro Fernandez. 2025. Not-So-Bitter Pill to Swallow: Slipstreaming Memory Safe Programming via Rust as part of a Database Systems Course. In *Workshop on Data Systems Education: Bridging Education Practice (DataEd '25), June 22–27, 2025, Berlin, Germany*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3735091.3737532

DataEd '25, Berlin, Germany

ACM ISBN 979-8-4007-1918-9/25/06
https://doi.org/10.1145/3735091.3737532

1 Introduction

Databases are a popular elective course in many undergraduate computer science programs, and provide an excellent opportunity to introduce students to system engineering concepts [1], particularly in courses that are centered around designing database systems, rather than using them. This is especially true for undergraduate CS majors at the University of Chicago, as the average student is only able to take two or three systems-oriented elective courses, given the breadth of the undergraduate core curriculum [27]. In 2021, we decided to revamp the existing database systems course at UChicago to sharpen its focus on systems programming, and, in particular, teach students about memory safety in the context of building a complex, multi-threaded, high-performance systems application. Unsafe memory management is rampant in modern software, leading to a wide range of security vulnerabilities, often landing in the top 5 of CWE's Top 25 Most Dangerous Software Weaknesses [25].

As we evaluated various implementation choices for the database system (§ 3.1), we found Rust [14] to be particularly attractive for systems programming tasks that require high performance and low-level control over system resources – a near perfect alignment with the needs of a database system implementation. Rust's memory safety guarantees are enforced at compile time, which eliminates the need for runtime checks and garbage collection, making it suitable for database systems, among other system software [28].

In this paper, we describe our experience designing and teaching a database systems course that uses Rust as the primary systems implementation language. We describe the curriculum design and course materials (§ 3), and the course project, CrustyDB (§ 4, 5, 6).

We also present the results of student feedback (§ 7) that was conducted at the end of the course offerings to evaluate the effectiveness of using Rust in a database systems course. Our experience (§ 8) shows that while students found Rust to be a powerful and expressive language that is well-suited for systems programming tasks, they were mixed about using in using Rust in future projects, in-part due to its steep learning curve.

2 Related Work

Database-related courses represent popular elective options in many CS programs [9], are taught by doing a deep dive into the internals of a database system, often by either working with an existing database such as PostgresSQL [1], or by building a simple database from scratch [6, 20, 21, 24].

Due to the increasing popularity of Rust as a system programming language, several universities offer mini-courses [10, 17, 29] or workshops that teach Rust. However, to our knowledge, only few courses have slip-streamed Rust into an existing systems-based course. An OS course at UVA [8] in 2014 used Rust as the primary programming language to teach operating systems concepts, but has since moved back to Python and C in its latest offering. A similar experimental course was taught at Stanford [4] that used Rust, but focused on developing an OS for embedded hardware, but has also since reverted to C/C++. We also found a similar course at Georgia Tech used Rust to teach OS concepts as recently as 2020 [12].

We have found that it is far more popular to include Rust in practical programming language or programming paradigm courses [2, 7] than to include them within traditional CS system elective courses. The closest course we found to ours is an advanced database systems course at CMU [18] allows students to build database components such as query optimizers and transaction managers in Rust. To the best of our knowledge, our course is the first to use Rust as the *primary* programming language for an introductory database systems course.

3 Curriculum Design

3.1 Design Goals

The *Introduction to Database Systems* course at UChicago was first taught using a custom-built database system called chiDB [24] that was built in C, and was designed to be a simple, extensible database system that was format-compatible with SQLite, using a database machine model. In 2016, the course was restructured to use the popular SimpleDB [21] educational database system. In 2020, the authors decided to build a new database system from scratch, as the limitations of SimpleDB were becoming apparent – the system was not extensible, and the Java-based implementation lacked low-level control over system resources such as memory and IO management and did not allow students to get a deep understanding of systems engineering concepts.

As we evaluated various implementation choices for the database system, we transitioned away from Java, and wanted to use a modern, systems oriented programming language. We considered C++, Go, and Rust, and ultimately settled on Rust as it (a) provided memory safety guarantees without garbage collection, (b) had strong support for concurrency, while not being prescriptive about the concurrency model, and (c) had modern language features, with robust types, generics, pattern matching, functional programming features, and a modern build and dependency management system baked in.

Thus, with the help of multiple research assistants, we built the CrustyDB database system from scratch in Rust, and designed the course project around it. The system was first developed in it's entirety, and then parts of the system were extracted into a set of project milestones that students would implement as part of the course-long project. The result is a fully functional, extensible, database system that is designed to be used as a teaching and research tool for undergraduate and graduate students.

weeks	Topic
1	Introduction to Database Systems, Relational Model
2	SQL, Application Development with Databases
3	Database Design, Storage and File Organization
4	Indexing
5	Query Execution
6	Query Optimization
7	Transactions and Concurrency Control
8	Recovery and Parellel Databases
9	Distributed Databases and non-relational systems

Table 1: Database Topics Covered in our Course

3.2 Course Description and Syllabus

Our course, CMSC 23500: Introduction to Database Systems [15], is a systems elective course in computer science intended for undergraduate CS majors, as well as graduate students at the UChicago. As an advanced systems elective, students are required to complete a four-course introductory sequence in computer programming, which includes two introductory to computer systems courses, based on the popular CS:APP systems course [5]. Two of these classes are in Python and two are in C. CS majors are required to complete a set of systems electives, and this course satisfies ass systems requirement for the CS major.

This course is also cross-listed as a graduate course (CMSC 33550) and is intended for students pursuing an M.S. or Ph.D. degree in computer science, and fulfills a systems breadth requirement for Ph.D. students. This course is a prerequisite for the *Advanced Database Systems* course (CMSC 23530), for students who wish to take a deeper dive into topics such as distributed databases, materialized views, multi-dimensional indexes, cloud-native architectures, data versioning, and concurrency-control protocols. CMSC 23530 also uses CrustyDB as the database project testbed for certain assignments, as mentioned in §4.

The UChicago CS department also offers multiple other graduate database courses for professional CS masters students (MPCS 53001) as well as computational public policy masters students (CAPP 30235), and are intended mostly to train students on the use and practice of database systems, as opposed to designing the internals of a database system.

Our course is designed to introduce students to relational database system concepts and implementation, similar to other database systems courses that follow the textbook by Silbershatz et.al. [23]. Table 1 lists the material covered in the classroom for the course.

Starting from the basics of database systems, we introduce basic relational database theory, SQL, and explain the architecture of a database system, from physical storage, query execution, optimization, and transaction management. Students are evaluated on their understanding of these theoretical concepts via a midterm exam and a final exam, turn a homework related to SQL, and have to complete the CrustyDB project milestones, as described next.

4 CrustyDB's Architecture

CrustyDB (Figure 1) is designed to be a relational database engine built from the ground up in Rust. We have focused on a modular design to allow component re-use for a research system, CrocodileDB [22], to replace parts and enable contributions from undergraduates. The system can be broken down into the following parts:

- **CrustyDB Client**: A simple command-line interface (CLI) program that sends SQL queries as well as database meta-commands (styled after PostgreSQL) to a server process.
- **CrustyDB Server**: The server process listens for incoming client connections, and spawns a separate thread process for each client command. Each client command is routed through the following components:

(1) **Query Parser**: The query parser is responsible for parsing the incoming database meta command and SQL queries, and converting them into a query plan that can be executed by the query executor. CrustyDB uses an external crate called `sqlparser-rs` [26] to parse SQL queries and generate the Abstract Syntax Tree (AST) for the query.
(2) **Query Optimizer**: The query optimizer is responsible for taking the query plan generated by the query parser and generating an optimized query plan that can be executed by the query executor. CrustyDB currently contains a prototype *rule-based optimizer* that can reorder joins and push down selections, and can be used by students and research assistants to experiment with query optimization techniques.
(3) **Query Executor**: CrustyDB uses a simple iterator-based query executor that can execute the query plan generated by the query parser. The iterator model is inspired by the Volcano/Cascades model [11], and is designed to be extensible with new query operators. Some of the query operators such as as the sequential scan operator work with the storage manager to fetch tuples from disk.
(4) **Storage Manager**: The managers described above are designed to be pluggable, with different variants that can be designed and implemented for each. For example, there are three different Storage Manager implementations currently in CrustyDB: The Heapstore (which uses heap file organization using slotted pages), Memstore (purely in-memory storage manager), and the SledStore (a TreeFile storage implementation using Rust's sled [16] library).
(5) **Transaction Manager**: The transaction manager is responsible for managing the lifecycle of transactions, and ensuring that transactions are executed in a manner that is consistent with the ACID properties. Like the storage manager, the transaction manager is designed to be pluggable, with different variants that can be designed and implemented for each. Students that take the *Advanced Database Systems* course are tasked with implementing a simple two-phase locking transaction manager in one of the project milestones.
(6) **Index Manager**: Finally, CrustyDB includes a simple index manager interface that can used to implement different types of indexes, such as B-trees, hash indexes, or bitmap indexes. Students that take the *Advanced Database Systems* course are tasked with implementing a simple B-tree index manager in one of the project milestones.

5 The CrustyDB Assignments

In this section, we now describe the different phases of the CrustyDB project.

5.1 The Rust Primer

We have found that students new to Rust often struggle with the language's unique features, such as its ownership model, borrow checker, lifetime annotations, and variable mutability, particularly in multithreaded programming. To help students get up to speed with Rust, we provide a rust primer (Table 2) that covers the basics of the language. The primer is designed as an online textbook[1] accompanied by a series of exercises that students can complete to reinforce their understanding of the material. We have also minimized the use of advanced memory and ownership features to simplify milestones.

5.2 CrustyDB I: Page Milestone

The first milestone of the CrustyDB project is to implement a simple storage manager that implements the heap file organization using slotted pages. A heap file consists of fixed-size pages, each of which contains a header that stores metadata about the page, and each item stored in the page is stored in a slot that contains a pointer to the item's location in the page. This milestone is a deep dive into Rust's language features, and students need to understand the byte manipulation and serialization features of Rust to implement the storage manager.

5.3 CrustyDB II: Heapstore Milestone

In the second milestone of the CrustyDB project, students continue building the storage manager by implementing a HeapFile interface that can manage heapfiles on disk, and provide an iterative interface to scan tuples from the heapfile. Students have to understand the Rust filesystem API and the Rust I/O model to implement this milestone.

Graduate students taking this class are tasked with an additional requirement to implement a buffer pool manager that can manage pages in memory, and implement a simple clock-based LRU approximation eviction policy to manage the buffer pool.

[1] https://uchi-db.github.io/rust-guide

Topic	Description
Ownership	Rust's ownership model and how it enforces memory safety
Borrowing	How Rust's borrowing system works and how it prevents data races
Lifetimes	Understanding lifetimes and how they are used to ensure memory safety
Mutability	Rules for mutable and immutable references in Rust
Concurrency	Safe concurrency patterns using Rust's ownership and borrowing model
Error Handling	Using Rust's 'Result' and 'Option' types for error handling
Serialization	Techniques for serializing and deserializing data in Rust

Table 2: Topics covered in the Rust primer

5.4 CrustyDB III: Operator Milestone

In this milestone, students are introduced to the iterator model of query execution and are provided with a simple iterator interface (designed as a Rust trait), and are tasked with implementing the following query operators:

- **Nested-Loop Join**: Students implement a simple nested-loop join operator that joins two input streams on a join predicate, and outputs the joined tuples.
- **Hash Join**: Students implement a hash join operator that creates a hash table from one input stream, and probes the hash table with the other input stream to output the joined tuples.
- **Groupby-Aggregate**: Students implement a groupby and aggregate operator that groups input tuples by a group key, and applies an aggregate function to each group.

5.5 CrustyDB IV: Open-Ended

Graduate students taking the course are tasked with an additional open-ended milestone that allows them to extend CrustyDB with one additional feature of their choice:

- **Additional Query Operators**: Students can implement additional operators such as the grace hash join [13], or parallelized join operators.
- **Basic Indexing Structures**: Integrate a hash or tree-based index into the query execution pipeline and develop a basic optimizer to choose when to use the index.
- **Logging**: Implement a simple write-ahead log for updates to the database.
- **Statistics Collection**: Implement a simple statistics collection framework for selectivity estimation and query optimization.
- **Disk-Based Indexes**: Implement a hash or tree-based index that can be stored on disk.
- **Transaction Management**: Implement a 2PL transaction and/or lock manager.

6 Evaluating Student Work

Students in this course are primarily evaluated on their homework, project milestones, as well as a midterm and final exam.

Each project milestone is evaluated using a combination of automated tests and TA/instructor code review. The automated tests are designed to test the correctness of the student's implementation, while the code review is designed to evaluate the quality of the student's code, including factors such as code readability, code organization, and code style. As part of the automated tests, a benchmark suite (Rust's criterion [3] benchmark suite) is run on the student's implementation to evaluate the performance of their code. Each milestone has a set of performance requirements that the student's implementation must meet in order to pass the milestone.

The exams are designed to evaluate the student's understanding of the theoretical concepts covered in the course, including relational algebra, SQL, query execution, query optimization, and transaction management, and is administered on pen-and-paper.

The final grade is calculated as a weighted average of the homework and project milestones (62%), 34% between the midterm and final exam, and 4% for participation in class.

7 Course Outcomes

The Rust-focused redesign of the *Introduction to Database Systems* has been offered on five separate occasions, starting in the winter quarter of 2021, and taught by all three authors in separate sections. The course has been well-received and continues to be a very popular elective course among CS majors at UChicago. At the time of writing, the course as described in this paper has been taught to a total of 509 students, with 448 enrolled in the undergraduate course and 61 in the graduate version. In the latest offering at the time of writing (Spring 2024), the course was taken by 115 students total, (103 undergraduate, 12 graduate).

7.1 Course Feedback

We elicited two forms of feedback from the students. First, as part of our institutional course evaluations, we asked students to provide feedback on the course as a whole, the quality of the lectures, projects and assignments, the effectiveness of the course materials, and the quality of the instruction as well as TA support.

Across the board, the course has strong positive feedback from the students (N=152), with students agreeing or strongly agreeing that the course (a) "was excellent" (78%), (b) "challenged them intellectually" (94%), and, (c) should be recommended to "highly motivated and well-prepared students" (95%).

Second, for each project milestone, we asked students to provide feedback on the specific milestone, including the perceived difficulty, an estimated time to completion, as well as the parts of the milestone they enjoyed and the parts they found challenging. The results of milestone-specific feedback are summarized in § 7.3.

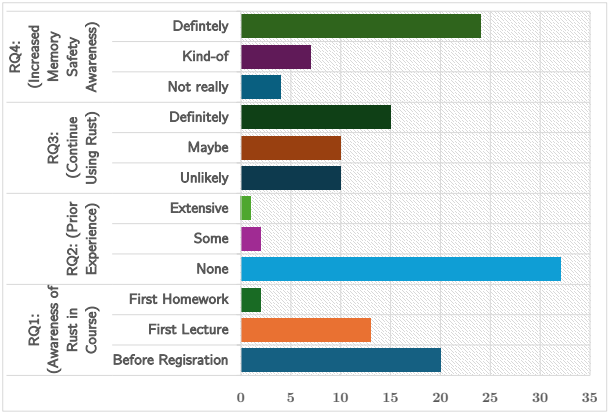

Figure 2: Student responses to Rust questions

7.2 Rust-Specific Feedback

In addition to the overall feedback, we included the following open-ended questions (Rust Questions / RQs) to gauge students' experience with Rust, in the latest offering of the course:

- **RQ1**: When were you first made aware that the course projects would be implemented in Rust?
- **RQ2**: What was your prior experience with Rust before coming into this Class?
- **RQ3**: Do you see yourself continuing to program in Rust?
- **RQ4**: Do you find yourself more aware or less aware about memory safety in Rust after taking this course?

Figure 2 indicates the results for these questions from the student feedback, (N=35). Since this was the fifth time the redesigned course was offered, a significant number of students were aware of the Rust programming requirement, primarily from course feedback of previous students. The vast majority of the students did not have any significant prior experience with Rust, with only three students reported having some experience and only one with significant experience in Rust. Students were divided on whether they would continue to program in Rust, with 20 students indicating that it would be unsure or unlikely, and 15 students indicating that they liked the language enough that they would consider it for future projects. Finally, students were overwhelmingly found themselves more aware of memory safety issues after programming in Rust.

7.3 Feedback from Assignments

We also collected feedback from the students on the Rust primer and the project milestones. The individual feedback was collected and analyzed to understand some of the common themes that emerged:

- **Rust Primer**: Students found the Rust primer to be very helpful, and many students reported that they would not have been able to complete the project milestones without it. Students reported struggling with Rust's ownership model, lifetimes and borrowing system. Students also expressed issues with the type system and error handling but appreciated the compiler's helpful error messages and the relative lack of segmentation faults and memory leaks.
- **Project Milestone 1**: Students expressed challenges with byte manipulation and serialization, and in particular managing memory views and pointers in Rust, while minimizing unnecessary copying.
- **Project Milestone 2**: A common feedback was difficulty in understanding the interior mutability model in Rust, as students were tasked with using reference counting, locks and mutexes to protect access to storage manager's data structures under a multithreaded workload.
- **Project Milestone 3**: Students generally expressed less difficulty at this point in the course, as they were more comfortable with Rust's quirks. Many appreciated the trait system in Rust, and liked some of the flexibility of the built-in data structures such as Rust's HashMap.

8 Lessons Learned

This course (and CrustyDB) is being continually refined and improved based on student feedback and our own experiences. Some of the major lessons we learned from teaching this course include:

- **Rust's Learning Curve**: Rust's learning curve is steep, and students new to Rust often struggle with its ownership model, borrow checker, lifetime annotations, and variable mutability. While the Rust primer helped students get up to speed with Rust, we found that some students still struggled with the language, particularly in getting their code to compile. Students eventually found their footing, and we found that the learning curve was not an impediment to learning core system concepts.
- **Tooling**: Rust's tooling ecosystem is still maturing, and we had some trouble setting up automated testing and benchmarking tools for the students. For example, Rust's automated tests (via `cargo`) were not as mature as Java's JUnit or Python's pytest, even for simple requirements as json output of test results, and we had to rely on external tools like nexttest [19] to generate test results. Similarly, we had trouble setting up a test timeout for the benchmarks, and had to rely on a custom script to kill the benchmark process after a certain time.
- **Use of Generative AI**: The instructors' experience with Rust and generative AI tools such as ChatGPT and Github Copilot seemed to indicate that these tools were not as helpful and effective in writing proper Rust code as they were with Python and SQL. This is likely due to the complexity of the language and significantly less code available in Rust for the models to train on. We warned students ahead of time to be cautious with these tools as they could lead to incorrect or unsafe code, or steer users in the wrong direction when compared to the Rust compiler's suggestions.

Acknowledgments

This course was made possible by the hard work of the teaching and research assistants, including Riki Otaki, Jun Hyuk Chang, Charles Benello, Zhe Heng Eng, Tapan Srivasatava, Rui Liu, Yue Gong, Reilly McBride, Emma Rosenthal, William Ma, and Daisy Barbanel.

References

[1] Anastassia Ailamaki and Joseph M. Hellerstein. 2003. Exposing Undergraduate Students to Database System Internals. *SIGMOD Rec.* 32, 3 (Sept. 2003), 18–20. doi:10.1145/945721.945725

[2] Jonathan Aldrich. 2022. 17-363/17-663: Programming Language Pragmatics. Retrieved June 4, 2025 from https://www.cs.cmu.edu/~aldrich/courses/17-363-fa22/

[3] Jorge Aparicio and Brook Heisler. 2025. Criterion.rs: Statistics-driven Microbenchmarking in Rust. Retrieved June 4, 2025 from https://github.com/bheisler/criterion.rs/

[4] Sergio Benitez. 2018. Stanford CS140e: An Experimental Course on Operating Systems. Retrieved June 4, 2025 from https://cs140e.sergio.bz/

[5] Randal E Bryant and David R O'Hallaron. 2015. *Computer Systems: A Programmer's Perspective* (3 ed.). Pearson, Upper Saddle River, NJ.

[6] CMU Database Group. 2025. The BusTub Relational Database Management System (Educational). Retrieved June 4, 2025 from https://github.com/cmu-db/bustub

[7] Will Crichton. 2019. From Theory to Systems: A Grounded Approach to Programming Language Education. https://arxiv.org/abs/1904.06750. doi:10.48550/arXiv.1904.06750 arXiv:1904.06750 [cs.PL]

[8] David Evans. 2014. Using Rust for an Undergraduate OS Course (Blog Post). https://rust-class.org/pages/using-rust-for-an-undergraduate-os-course.html.

[9] Alan D. Fekete and Uwe Röhm. 2022. Teaching about Data and Databases: Why, What, How? *SIGMOD Rec.* 51, 2 (July 2022), 52–60. doi:10.1145/3552490.3552504

[10] Noah Gift and Alfredo Deza. 2025. Coursera / Duke University: Rust Programming Specialization. Retrieved June 4, 2025 from https://www.coursera.org/specializations/rust-programming

[11] G. Graefe. 1994. Volcano: An Extensible and Parallel Query Evaluation System. *IEEE Trans. on Knowl. and Data Eng.* 6, 1 (Feb. 1994), 120–135. doi:10.1109/69.273032

[12] Taesoo Kim. 2020. Georgia Tech CS-3210: Design-Operating Systems Spring 2020. Retrieved June 4, 2025 from https://tc.gts3.org/cs3210/2020/spring/info.html

[13] Masaru Kitsuregawa, Hidehiko Tanaka, and Tohru Moto-Oka. 1983. Relational Algebra Machine GRACE. In *Proceedings of RIMS Symposium on Software Science and Engineering*. Springer-Verlag, Berlin, Heidelberg, 191–214.

[14] Steve Klabnik and Carol Nichols. 2023. *The Rust Programming Language: 2nd edition.* No Starch Press, San Francisco, CA.

[15] Aaron J. Elmore Mohammed Suhail Rehman. 2023. CMSC 23500/33550: Intro to Database Systems. https://classes.cs.uchicago.edu/archive/2023/spring/23500-1/.

[16] Tyler Neely. 2025. Sled - An Embedded Database. https://github.com/spacejam/sled.

[17] Quan Hao Ng and Daniel Philipov. 2025. UIUC: CS 128 Honors. Retrieved June 4, 2025 from https://honors.cs128.org/

[18] Andy Pavlo. 2024. CMU 15-721 : Advanced Database Systems (Spring 2024). https://15721.courses.cs.cmu.edu/spring2024/.

[19] Rain. 2025. Cargo-Nextest: A next-generation test runner for Rust. https://nexte.st/.

[20] Raghu Ramakrishnan and Johannes Gehrke. 2000. *Database Management Systems* (2nd ed.). McGraw-Hill, Inc., USA.

[21] Edward Sciore. 2007. SimpleDB: a simple java-based multiuser syst for teaching database internals. *SIGCSE Bull.* 39, 1 (March 2007), 561–565. doi:10.1145/1227504.1227498

[22] Zechao Shang, Xi Liang, Dixin Tang, Cong Ding, Aaron J. Elmore, Sanjay Krishnan, and Michael J. Franklin. 2020. CrocodileDB: Efficient Database Execution through Intelligent Deferment. In *CIDR*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p14-shang-cidr20.pdf

[23] Abraham Silberschatz, Henry Korth, and S Sudarshan. 2019. *Database system concepts* (7 ed.). McGraw-Hill, New York, NY.

[24] Borja Sotomayor and Adam Shaw. 2016. Chidb: Building a Simple Relational Database System from Scratch. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16)*. Association for Computing Machinery, New York, NY, USA, 407–412. doi:10.1145/2839509.2844638

[25] Alec Summers, Connor Mullaly, and Steve Christey Coley. 2024. CWE Top 25 Most Dangerous Software Weaknesses. Retrieved June 4, 2025 from https://cwe.mitre.org/top25/index.html

[26] The Apache Software Foundation. 2025. datafusion-sqlparser-rs: Extensible SQL Lexer and Parser for Rust. Retrieved June 4, 2025 from https://github.com/apache/datafusion-sqlparser-rs

[27] The University of Chicago. 2025. The Core Curriculum. https://college.uchicago.edu/academics/core-curriculum.

[28] The White House Office of the National Cyber Director. 2024. Back To The Building Blocks: A Path Towards Secure and Measurable Software. Retrieved June 4, 2025 from https://bidenwhitehouse.archives.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf

[29] Connor Tsui and Jessica Ruan. 2024. CMU 98-008: Intro to Rust Lang. Retrieved June 4, 2025 from https://rust-stuco.github.io/old/f24/