chidb: Building a Simple Relational Database System from Scratch

Borja Sotomayor
University of Chicago
borja@cs.uchicago.edu

Adam Shaw
University of Chicago
ams@cs.uchicago.edu

ABSTRACT

We present chidb, a medium to large-scale programming project where students implement the main components of a relational database management system, including B-tree data structures for tables and indexes, a database machine with registers and a selection of high-level and low-level instructions, and a SQL compiler targeting that machine. Moreover, chidb's SQL compiler's internal representation is a direct encoding of the relational algebra, whereby the theory that might otherwise be relegated to notes and blackboards is directly connected to practice and experience. The project uses the C programming language and is demonstrably suitable for use in advanced undergraduate courses; we have administered this project through five iterations of our databases course for advanced undergraduates. chidb is freely available online and customizable to suit the needs and tastes of any particular instructor.

Keywords

computer science education; databases; relational database management systems; SQL

1. INTRODUCTION

Teaching subjects that have immediate practical applications, like databases, often involves balancing the practical with the theoretical. Both kinds of knowledge are important; they are, ideally, mixed in such a way as to give students solid instruction in both areas, stinting on neither.

At the University of Chicago, this principle is apparent in many of our systems-oriented courses (Operating Systems, Networks, Databases, *etc.*), most of which involve an extensive hands-on programming project with a strong theoretical connection. Furthermore, these projects are designed to be as realistic as practicable during a quarter-long project. For example, instead of having students in our Operating Systems course implement standalone components of the OS, such as writing a memory allocator as an isolated programming assignment, they work in teams to implement an x86 OS kernel nearly from scratch using the Pintos project [3]. Similarly, students in our Networks class must implement a version of TCP that is RFC-compliant to the extent that, at the end of the quarter, they are able to write socket-based applications (capable of communicating over the Internet) that use their TCP implementation instead of the operating system's.

SIGCSE '16, March 02 - 05, 2016, Memphis, TN, USA
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-3685-7/16/03…$15.00
DOI: http://dx.doi.org/10.1145/2839509.2844638

Of course, few students will be asked in their careers (whether in industry or academia) to write an operating system or their own network stack directly, and doing so arguably takes time away from gaining more immediately applicable skills. Even so, we believe that this approach allows students to understand and appreciate the core concepts of a subject, making the connection between theory and practice clearer, and enabling them to better engage with the subject matter after they complete the class.

Our first foray into this style of teaching began in our major-level *Introduction to Databases* course, where we developed chidb: a quarter-long C programming project where students have to implement a relational database management system (RDBMS) largely from scratch, from the file-based B-trees all the way up to the SQL compiler. By the end of the quarter, students are able to bootstrap their own database (without using any instructor-provided samples) and run a variety of SELECT, INSERT, and CREATE statements on that database within the system they build themselves. More importantly, when students run a SQL statement, they will have written most of the code that is run from the moment the SQL statement is parsed to when individual blocks of the B-tree file are read from disk.

The chidb project makes the following contributions to the teaching of databases:

- Provides a framework for a database course to explore the complete implementation of a realistic RDBMS, including B-trees, its internal architecture, cursors, compilation of SQL queries, and query optimization. Ours is one on the only projects that spans all these layers, providing an integral view of the entire RDBMS.

- Allows students to engage directly with concepts that are typically only covered in the abstract. Notably, our internal representation of SQL queries is based on relational algebra (containing sigma, pi and rho abstract syntax tree nodes, for example), reinforcing in a working system what the students have otherwise seen only on paper. The relational algebra is, furthermore, an especially congenial representation for both reasoning about and implementing query optimizations.

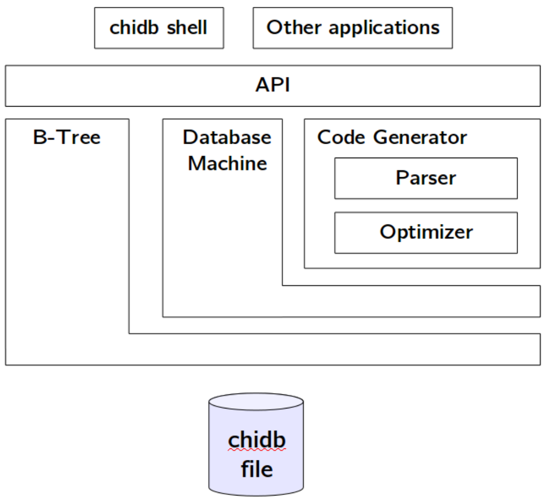

Figure 1: `chidb` architecture

- The project is structured as an open-ended architecture, not as a fixed set of assignments. So while we provide a number of assignments ourselves, it would be relatively easy for other instructors to build additional assignments. All the code is freely available under a BSD license at http://github.com/uchicago-cs/chidb, and we provide documentation on `chidb`'s architecture and assignments at http://chi.cs.uchicago.edu/chidb.

The rest of this paper is structured as follows. We begin by describing the architecture of `chidb` in Section 2, followed by a summary of the assignments we currently use in Section 3, and a discussion of our experience so far with `chidb` in Section 4. Next, we discuss related work in Section 5, and present our conclusions and future work in Section 6.

2. ARCHITECTURE

Consistent with the style of teaching we described in the previous section, `chidb`'s architecture is guided by two main principles:

1. It must replicate, as closely as possible, the architecture of a real RDBMS. We chose to base the architecture of `chidb` on SQLite [2], particularly by using a file format that is a subset of the SQLite file format (all `chidb` files are valid SQLite files, but not the other way around.) This feature provides the benefit that `chidb` databases can be viewed and operated on from within the SQLite shell.

2. It must be structured in such a way that students can focus on database-specific implementation tasks, with an apparent connection to the theory of databases, without getting bogged down in orthogonal and uninteresting implementation details.

In this section, we discuss the `chidb` architecture in its fully implemented state, which is summarized in Figure 1. Section 3 focuses on which implementation tasks could be assigned to students as part of specific assignments.

2.1 File Format

Like SQLite, `chidb` uses a single file to store a relational database. In fact, the `chidb` file format is a subset of the SQLite file format and, thus, a `chidb` file can store any number of tables, physically stored as a B$^+$-tree (referred to as a *table B-tree*), and each table can have records with any number of fields of different datatypes. Indexes are also supported, and are stored as B-trees (referred to as *index B-trees*).

A detailed file format specification is available in the `chidb` website. In summary, a file is divided into equally-sized *pages* (typically 1–4 kilobytes), each storing one *node* of a B-tree. Each node can contain multiple *cells*, with a different cell structure depending on the type (table or index) and level (internal or leaf) of the node. Of special note are the leaf nodes of table B-trees, where the cells contain a $\langle$*key*, *database record*$\rangle$ pair, where each record represents one row of the table. Finally, each B-tree has at least one *root node*, and the location of this root node is stored in a special *schema table* (itself a table B-tree) rooted in page 0 of the file.

Although the file format is remarkably close to the SQLite file format, to the point that valid `chidb` files can be read by SQLite, we do allow students to make the following simplifying assumptions:

- Each table must have an explicit primary key (SQLite allows tables without primary keys to be created), and the primary key must be a single unsigned 4 byte integer field.

- Indexes can only be created for unsigned 4-byte integer unique fields.

- Only a subset of the SQLite datatypes are supported.

- The size of a record cannot exceed the size of a file page (more specifically, SQLite overflow pages are not supported). This effectively also limits the size of certain datatypes (such as strings).

- A user is assumed to have exclusive access to the database file.

These assumptions were made to simplify the implementation of many low-level details, so that students can focus on tasks that directly relate to database concepts (while still requiring a healthy amount of low-level programming). For example, although supporting database records that span multiple pages is a necessary feature in production databases, its implementation requires a fair amount of low-level programming that is algorithmically dull (at least when compared with algorithms for B-tree insertion, query optimization, etc.), and ultimately irrelevant to the main thread of our curriculum.

2.2 B-trees

The B-tree module of `chidb` is the only part of the `chidb` code that reads and manipulates `chidb` files. It has a well-defined API with functions to create new B-trees, insert records, search for records with a given key, *etc.*

2.3 Database Machine

The database machine (or DBM) is a virtual machine specifically designed to operate on `chidb` files, and includes low-level assembly-like instructions such as `Integer`, which loads an integer constant into a register, and `Eq`, which jumps to a specified address if the contents of two registers is the same, as well as high-level database-specific instructions such as `CreateTable`, which creates a new table B-tree and stores its root page number in a register, and `Seek`, which moves a cursor to point to the first entry in a B-tree with a given key k, and jumping to a given address if no such entry exists in the B-tree.

The DBM is inspired by SQLite's VDBE[1] and, in fact, supports many of the same instructions. There are 36 DBM instructions, each with up to four operands: P1, P2, P3, and P4. P1 through P3 are signed 32-bit integers, and P4 is a pointer to a null-terminated string. The implementation of these instructions often requires using functions from the B-tree module.

An instance of the DBM includes the following:

Program A sequence of one or more DBM instructions. The program is typically produced by the code generator, but we also provide the ability to load arbitrary programs easily for the purposes of testing.

Program Counter Keeps track of what instruction is currently being executed. Certain instructions can directly modify the program counter to jump to a specific instruction in the program.

Registers A machine can have an arbitrary number of registers, each of which can contain a 32-bit signed integer, a pointer to a null-terminated string or to raw binary data, or a NULL value.

Cursors A cursor is a pointer to a specific entry in a B-tree.

2.4 SQL Compiler

The SQL compiler consumes plaintext SQL programs and produces DBM programs. We provide a SQL parser that parses a small but significant fraction of the SQL language, including, for example, `WHERE` conditions and natural joins, but excluding certain others like groupings or nested subqueries which are, in our experience, infeasible to implement within our time frame. The internal representation the parser targets is an abstract syntax tree that matches the notation of relational algebra (σ, π, ρ). From a pedagogical standpoint, this has the effect of unifying the theoretical treatment of certain optimizations—pushing sigmas, for example—with their practical implementations.

2.5 API

`chidb` exposes all the above functionality through a simple API that can be accessed from other C programs simply by including a `chidb.h` header file, and linking with a `libchidb` library produced by our Makefile. This means that, given a fully-implemented `chidb` system, `chidb` databases can be used from other C programs in the same way that SQLite databases can be used through the SQLite API[2].

The `chidb` API, shown in Figure 2, provides functions to open and close database files, execute SQL statements, and, in the case of `SELECT` statements, provide access to the results of the query.

[1]https://www.sqlite.org/opcode.html
[2]https://www.sqlite.org/capi3ref.html

```
/* Functions to open and close a database file */
int chidb_open(const char *file, chidb **db);
int chidb_close(chidb *db);

/* Takes a SQL statement and compiles it. The resulting
   database machine (DBM) is stored in the "stmt"
   output parameter */
int chidb_prepare(chidb *db, const char *sql,
                               chidb_stmt **stmt);

/* Frees all the resources associated with a DBM
   previously created with chidb_prepare */
int chidb_finalize(chidb_stmt *stmt);

/* For SELECT queries, step to the next row
   in the results */
int chidb_step(chidb_stmt *stmt);

/* Column access functions */
int chidb_column_count(chidb_stmt *stmt);
int chidb_column_type(chidb_stmt *stmt, int col);
const char *chidb_column_name(chidb_stmt* stmt, int col);
int chidb_column_int(chidb_stmt *stmt, int col);
const char *chidb_column_text(chidb_stmt *stmt, int col);
```

Figure 2: `chidb` API

2.6 Shell

We provide a command-line shell that can be useful for interactively testing a `chidb` implementation (even an incomplete one). It provides the following functionality:

- Running SQL statements and, in the case of `SELECT` statements, displaying their results.
- Running arbitrary DBM programs.
- Parsing SQL and showing the syntax tree produced by the parser.
- An `EXPLAIN` directive which, given a SQL statement, shows the DBM program produced by the code generator.

3. ASSIGNMENTS

The `chidb` architecture can support countless assignments and, in fact, could be used in more that just a databases course. A data structures class could assign the implementation of the B-tree module; an introductory systems course could focus on implementing the DBM; and, a compilers course could focus on scanning, parsing, optimizing, and generating code for the DBM set of instructions. This is not to mention the many ways in which `chidb` could be modified or extended even in the context of a databases course.[3]

We have chosen to structure the `chidb` project somewhat differently in each of the years of our teaching it; the most recent format we used divided the project into four assignments, the first of which is done in pairs, and the latter three done in teams of four students each.

Assignment 1: B-trees Students implement the B-tree module, starting only from the API specification of that module. At the end of this assignment, fully working

[3]For example, `chidb` has never been used in a semester length course, which would necessitate some adaptation.

student implementations (of which there have always been a few) are chosen as reference B-tree implementations for the rest of the quarter, and shared with the rest of the class.

Assignment 2: Database Machine (DBM) This assignment provides students with the instructors' implementation of the core DBM (registers, program loading, etc.), but leaving the implementation of the DBM instructions up to the students. Students must also implement the cursor data structure, and must do so in a way that allows the cursor to move to the next or previous entry in a B-tree in amortized $O(1)$ time.[4]

In the past, students had to implement the entire DBM from scratch themselves. This had the advantage of giving students the freedom to design their implementation however they wanted, within the context of a tight specification, of course, instead of being given it by their instructors. However, as a practical matter, this made it impossible for our graders to automate testing of all the different DBM implementations, since they did not know quite what they would be getting, and we have since standardized the internal representation of the DBM.

Assignment 3: Code Generation Students must implement a code generator which, given a simple `SELECT`, `INSERT`, `CREATE TABLE` or `CREATE INDEX` statement, will generate the appropriate DBM program. A SQL lexer and parser is already provided for them (implemented in `lex` and `yacc`) which produces an abstract syntax tree representation of the SQL statement. Students needed to be able to compile `SELECT` statements with any number of fields or `*` (for which consulting the database schema is necessary), conjunctions of comparisons in their `WHERE` clauses, and two-way natural joins (which also entails examining the database schema), among a few others.

Assignment 4: Query Optimization Students must modify the compiler to optimize certain queries where possible. This includes using indexes when indexes are available, or transforming the tree representation of the query in accordance with certain well-known optimizations, such as sigma-pushing and sigma coalescing.

Students had two or three weeks to complete each of these phases of the project. In all but the last assignment, students are provided with an extensive set of unit tests they can use to test their implementation.

Although this is the way we have structured the project, other instructors could take the `chidb` architecture, described in the previous section, and create new assignments or variations on the existing ones. For example, a possible new assignment could involve adding support for concurrent access and transactions. A variation on an existing assignment could involve providing a complete implementation of the DBM (including all the instructions), to spend more time on the code generation and query optimization assignments, covering a larger portion of the SQL language (a greater variety of `SELECT` statements, for example, or `DELETE` and `UPDATE` statements). Modifying the compiler to do join order analysis, for example, or to support n-way joins, would be natural extensions of the optimization exercises.

[4]See Section 4 for colorful feedback on this point.

4. EXPERIENCE

As mentioned above, our department teaches an advanced undergraduate course entitled *Introduction to Databases.* The prerequisites for the course at our department are three quarters of introductory study in computer science, including two quarters doing a substantial amount of C programming and systems programming, with some assembly language. These prerequisites have worked in the context of our curriculum and our student population, and may not directly translate to other programs of study. In general, students working on this project must have acquired enough knowledge of C programming and software development to tackle a relatively large C programming project, as well as enough of a sense of the low level operations of a machine to understand what code generation requires of them. Furthermore, while students in our course work on `chidb` while simultaneously learning about databases for the first time, `chidb` could also be used in an advanced databases course where students are assumed to already be familiar with fundamental database concepts.

We have taught `chidb` in the databases course each of the last five times we have offered it, in five non-consecutive years between 2009 and 2015. The enrollments in these courses have ranged from a minimum of 6 students in 2010 to a maximum of 62 students most recently in 2015. (Our department's enrollments have, of late, surged across the board.)

There are two ways to discuss our experience of having taught `chidb`. One is from our perspective, and the other is from the students' perspective. We will begin with the students.

Our evidence about students' experience with `chidb` is, admittedly, not statistically rigorous. We have anecdotal evidence from working with the students directly, since the authors have been involved in teaching this course five and three times, respectively, and we also have the students' course evaluations, which are available online internally at our institution.

Although we have not quantified our experience of working with students in person, anecdotally and having interacted with the by now more than 100 students who have taken this course in some form over the last five years, the project is generally well liked and valued by those who have undertaken it. This is not to say it has been universally well liked, because some students' work on this project, for the usual variety of reasons, has been less than stellar. Nonetheless, every time we have taught `chidb`, the project has been the source of great interest and spurred countless interesting discussions about the inner workings of this kind of system. It is a project that appeals very strongly to students who like to roll up their sleeves and write a lot of code!

As recorded in the college's post-course surveys, the students' feelings towards the course are positive, on the whole. The surveys the students are given do not (unfortunately) ask the students outright whether they liked and/or valued the course itself; the questions are mostly about the instructor. Nevertheless, in their comments, students commented favorably on `chidb`:

> The chidb project really is a great learning experience and also is very cool.

They shared impartial observations on its difficulty:

> The projects were very intense and required an immense investment of time.

And they vented their frustrations:

> [The project was] at times an absolute nightmare...

as well as

> I cringe when I hear the word Cursor.

Nevertheless, when the survey asked the students directly "Were the demands of the course reasonable?", in the most recent batch, 34 students responded that yes, the demands were reasonable, and only one responded no[5] (and this is roughly in line with previous years).

As instructors, per our internal measures, we feel the project has been a great success! The fact that it has not received unanimous acclaim from students is, we believe, entirely to be expected, given its difficulty. The students emerge with what we hope and expect is a relatively deep understanding of the diverse machinery that needs to come together for data to be stored on disk in a sensible way and efficiently retrieved by queries. Furthermore, having implemented basic SQL queries "only" up to natural join and indexes, students develop a great appreciation for the achievements of a full SQL compiler. In other words, the knowledge and experience the students gain is well beyond what they would get merely by writing and executing SQL queries, however serpentine and clever those queries might be. When we consider the content of this course alongside the database courses we took at similar points in our educations, what our own students learn from having worked on `chidb` is much richer in depth and detail.

Still furthermore, and looking at this work as part of the students' education more broadly, it is important for students to have grappled with a file format and set of data structures that are engineered for actual use, and are not exercises contrived for the students' benefit. Real data structures (such as B-trees whose nodes fit disk blocks) have a shape, texture and scope that reaches beyond what students are able to work with in the usual sandboxes of their introductory coursework.

5. RELATED WORK

The SimpleDB project [5], while different from `chidb` in nearly every particular, is the most current, most similar work to ours in terms of the role it plays in database instruction. `chidb` was developed roughly concurrently and independently of any knowledge of SimpleDB, so `chidb` is in no way a response to or a critique of SimpleDB. The philosophies of the two projects are quite different. Broadly speaking, `chidb` is in our judgment a narrower project, but a deeper one in its area. `chidb` deals with the particular concerns of storing data in B-trees and compiling SQL queries to operate on that data, in a particular virtual machine framework following SQLite's example. SimpleDB, by contrast, is concerned with the larger scope of database systems, including major units on transactions and concurrency. SimpleDB treats topics that are not part of `chidb` at all, including user authentication, buffer management, deadlock detection, recovery, and more. If these latter elements of database systems are of particular importance to a particular curriculum, then SimpleDB is surely the appropriate choice among these two.

[5]Not all students respond to the surveys, or to every question in the surveys.

```
struct RA_s {
   enum RA_Type t;
   union {
      struct { char *name; } table;
      struct { RA_t *ra; Condition_t *cond; } sigma;
      struct { RA_t *ra; Expression_t *expr_list; } pi;
      struct { RA_t *ra1, *ra2; } binary;
      struct { RA_t *ra;
               Expression_t *to_rename;
               char *new_name;} rho;
   };
   Column_t *columns;
};
```

Figure 3: Excerpt of the RA (relational algebra) data structure

Within the area of overlap between the two projects, there are enough technical dissimilarities between them to make them substantially different from one another:

- `chidb` stores table data, indexes, and even database schema information in B-trees. SimpleDB uses a linear data store for tables.
- `chidb` uses an internal representation that hews closely to relational algebra; see Figure 3 for an excerpt.
- The back end of `chidb` compilation is code generation to a target virtual machine.
- SimpleDB follows a client/server architecture, with databases stored in collections of files; `chidb` is a library that other programs can link with, with each database stored in a single file.

As a practical matter, SimpleDB is written in Java, while `chidb` is in straight C. This might make SimpleDB or `chidb` suitable or unsuitable for different academic programs, depending on which programming languages students know and are expected to use. Unlike `chidb`, SimpleDB is supported by a textbook [6].

Going back a bit further, the Minibase framework [4] is also similar in its educational approach. It is an important predecessor to SimpleDB, and named as such in the abstract of the SimpleDB paper. Like SimpleDB, and like `chidb`, Minibase is designed for students to learn databases by implementing an RDBMS. With respect to its technical details, Minibase implements B$^+$-tree indexes, as `chidb` does, but stores record data in unordered heap files, unlike `chidb`. Furthermore, database queries are implemented in Minibase as different kinds of iterators: iterators for simple queries, for selection, for joins, *etc.* This differs from the compiled-query approach taken by `chidb` and following SQLite.

As far as we can tell, Minibase is not an active project, although we were able to find adaptations of it in recent use (for example, a Java version of Minibase was taught at Purdue as recently as 2012).

MinSQL [7] is a project similar in design and scope to Minibase, although its implementation is in Java. We do not believe this to be an active project. It also worth noting that a course taught at Berkeley and CMU [1] favored modifying the codebase of an actual database system (Postgres, in this case) to working with an educational system (Minibase), on the grounds that the latter was insufficiently representative of real-world software.

6. CONCLUSIONS AND FUTURE WORK

We have described `chidb`, a simple relational database management system that is small enough to comprehend, and even implement, over the course of a quarter or semester, yet complete enough to provide functionality found in many existing database systems, including the ability to run a variety of SQL statements. Furthermore, `chidb`'s architecture lends itself easily to design a variety of programming assignments.

Our experiences using `chidb` have been, overall, positive. However, there are still a number of aspects in which `chidb` can be improved. The biggest gap in `chidb` is its assumption that the database file will be used by a single user; there is no attempt to implement any sort of locking, and concurrent access to the database will likely result in an inconsistent state. This is an arguably fair assumption if we want students to focus on implementing the DBM or the SQL compiler, but it does not meet our standard of producing a system that is as realistic as possible, and it also precludes instructors from designing assignments revolving around transactions.

We have in mind a number of medium-scale improvements to be made in the near future, including the following:

- We have never been fully satisfied with the query optimization assignment, since the SQL queries that are currently supported only lend themselves to some limited optimizations. Also, the result of these optimizations is hard to test automatically. While we consider the first three assignments to be relatively stable, we expect our next efforts will focus on improving the fourth assignment.

- We would like to streamline the data structures that represent relational algebra expressions, and provide better documentation for them.

- Although our suite of tests gives students the ability to load arbitrary DBM programs, we realize that debugging those DBM programs using traditional debuggers can be challenging. We will address this by including a tool that allows students to run a DBM program step by step, providing a dump of the DBM registers and cursors at each step.

Acknowledgments

We owe a large debt of gratitude and inspiration to the creators of SQLite. It is a remarkably robust, elegant, compact system, not to mention an immensely popular one. SQLite has no publications in the academic record, so we are unable to provide a paper citation, but `chidb` is its direct descendant. We acknowledge our colleague Sharon Salveter, who taught the `chidb` course in its first two iterations, and so got the ball rolling with what was at the time untested work. We are also grateful to Allen Nelson, who implemented the current incarnation of the SQL parser that is included with `chidb`. We appreciate the thoughtful comments of our anonymous reviewers; they improved the paper.

7. REFERENCES

[1] A. Ailamaki and J. M. Hellerstein. Exposing undergraduate students to database system internals. *SIGMOD Rec.*, 32(3):18–20, Sept. 2003.

[2] D. R. Hipp. SQLite (http://www.sqlite.org), 2015.

[3] B. Pfaff, A. Romano, and G. Back. The Pintos Instructional Operating System Kernel. *SIGCSE Bull.*, 41(1):453–457, Mar. 2009.

[4] R. Ramakrishnan. The Minibase Home Page (http://research.cs.wisc.edu/coral/mini_doc/minibase.html), 1996.

[5] E. Sciore. SimpleDB: A Simple Java-based Multiuser System for Teaching Database Internals. *SIGCSE Bull.*, 39(1):561–565, Mar. 2007.

[6] E. Sciore. *Database Design and Implementation*. Wiley, 2008.

[7] G. Swart. MinSQL: A simple componentized database for the classroom. In *Proceedings of the 2Nd International Conference on Principles and Practice of Programming in Java*, PPPJ '03, pages 129–132, New York, NY, USA, 2003. Computer Science Press, Inc.