Lecture 26

COMPILER DESIGN
Announcements

• HW6: Analysis & Optimizations
  – Alias analysis, constant propagation, dead code elimination, register allocation
  – **Due**: Tuesday, December 17\textsuperscript{th} at 23:59
  – *May submit by Thursday, December 19\textsuperscript{th} (i.e., today) at 23:59 w/ penalty*

• Final Exam
  – Scheduled for Friday, January 31\textsuperscript{st}, 9-11 AM
  – A bit more on it later
What have we learned?
Where else is it applicable?
What next?

SUMMARY
Final Exam

- **Cover** everything until (and including) garbage collection & the 6 HWs
  - Lexing (regular expressions, DFA/NFA, lexer generator)
  - Parsing (top-down/bottom-up parsing, including LL, LR, & LALR)
  - Scope, type-checking, inference rules
  - Lambda calculus, closure conversion
  - Objects, inheritance, types, implementation of dynamic dispatch
  - Basic optimizations
  - Dataflow analysis (forward vs. backward, fixpoint computations, etc.)
    - Liveness, reaching definitions, available expressions, very busy expressions
    - Constant propagation
    - MOP solutions, distributed vs. non-distributed analyses
  - Linear-scan and graph-coloring register allocation
  - Control flow analysis
    - Loops, dominator trees
  - Garbage collection (mark & sweep, stop & copy, reference counting)

- **Exam focus and format**
  - Focus on the theory side, but may also include materials specific to the HWs
  - Focus on simple answers, computation, multiple choice, etc.
  - Sample exams will be posted on the course Moodle
Why Compiler Design?

• You will learn:
  – Practical applications of theory
  – Parsing
  – How high-level languages are implemented in machine language
  – (A subset of) Intel x86 architecture
  – A deeper understanding of code
  – A little about programming language semantics
  – Functional programming in OCaml
  – How to manipulate complex data structures
  – How to be a better programmer

• Did we meet these goals?
Materials we didn’t cover

• We skipped materials at many levels
  • Concrete syntax/parsing
    – Much more to the theory of parsing … LR(*)
    – Good syntax is art, not science!
  • Source language features
    – Exceptions, advanced type systems, type inference, concurrency
  • Intermediate languages
    – Intermediate language design, bytecode, bytecode interpreters, just-in-time compilation (JIT)
• Compilation
  – Continuation-passing transformation, efficient representations, scalability
• Optimization
  – Scientific computing, cache optimization, instruction selection/optimization, …
• Runtime support
  – Advanced garbage collection algorithms
Where to go from here?

• Major relevant conferences
  – Programming Language Design and Implementation (PLDI)
  – Principles of Programming Languages (POPL)
  – Object Oriented Programming Systems, Languages & Applications (OOPSLA)
  – International Conference on Functional Programming (ICFP)
  – International Symposium on Code Generation and Optimization (CGO)
  – International Conference on Compiler Construction (CC)
  – European Symposium on Programming (ESOP)
  – ...

• Technologies / Open-Source Projects
  – Yacc, lex, bison, flex, ...
  – LLVM (low level virtual machine) → MLIR
  – Java virtual machine (JVM), Microsoft’s Common Language Runtime (CLR)
  – WebAssembly
  – Languages: OCaml, F#, Haskell, Scala, Go, Rust, …?
Where else is this stuff applicable?

• General programming
  – In C/C++, better understanding of how the compiler works can help generate better code
  – Ability to read assembly output from compiler
  – Experience with functional programming can give different ways to think about how to solve a problem

• Writing domain specific languages
  – lex/yacc very useful for little utilities
  – understanding abstract syntax and interpretation

• Understanding hardware/software interface
  – Different devices have different instruction sets, programming models
Thesis Projects

• A wealth of possible projects suitable for BSc (and MSc) theses
  – Language design & implementation
  – (Futuristic) Programming methodologies, environments and tools
  – Program analysis, verification, and testing
  – Software (including emerging software) quality, reliability, and security
  – Education technologies (for compilers, programming, and CS in general)
  – ...

• Watch for an email/announcement from Manuel & get in touch w/ us

• Successful projects will likely lead to
  – Nice research results
  – Publications in the top CS conferences
    • PLDI, POPL, OOPSLA, ICSE, FSE, ...
  – More importantly, a taste of research & some tangible real impact
Thanks!

• To all the TAs for doing a great job helping with the course!

• To Dr. Grosser & Dr. Leopoldeser for the nice guest lectures!

• To you all for taking and participating in the class!

• How can we improve the course for future offerings?
  – Please let me know in the course evaluations!

• Happy Holidays, and good luck with your exams in early 2020!
Testing Database Management Systems via Pivoted Query Synthesis

Manuel Rigger

December 19, 2019
Compiler Design
How is this Related To Compiler Design?

- You can apply the concepts and principles of the Compiler Design class in many scenarios
- Configuration file parsers, input validation, embedded scripting languages, ...
- Testing Database Management Systems!
Who has used at least one of these Database Management Systems?
Databases are Used Ubiquitously

“SQLite is the most used database engine in the world. SQLite is built into all mobile phones and most computers and comes bundled inside countless other applications that people use every day.”

https://www.sqlite.org
### Relational Data Model

<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>A cute toast cat</td>
<td><img src="image1.png" alt="Cat" /></td>
</tr>
<tr>
<td>Dog</td>
<td>Cute dog pic</td>
<td><img src="image2.png" alt="Dog" /></td>
</tr>
<tr>
<td>Cat</td>
<td>Cat plants (cute!)</td>
<td><img src="image3.png" alt="Cat" /></td>
</tr>
</tbody>
</table>
Relational Data Model

<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>A cute toast cat</td>
<td><img src="image1" alt="Cat picture" /></td>
</tr>
<tr>
<td>Dog</td>
<td>Cute dog pic</td>
<td><img src="image2" alt="Dog picture" /></td>
</tr>
<tr>
<td>Cat</td>
<td>Cat plants (cute!)</td>
<td><img src="image3" alt="Cat plants picture" /></td>
</tr>
</tbody>
</table>

A database schema describes the **tables (relations)** in the database.
Structured Query Language (SQL) is a declarative DSL to query and manipulate data.

```
SELECT picture, description
FROM animal_pictures
WHERE animal = 'Cat'
AND description LIKE '%cute%'
```
Database Management Systems

```
SELECT * FROM <table>
WHERE <cond>
```

Client Application  ➔  Database Management System (DBMS)  ➔  Database

- row₁  ➔  <cond>
- row₂  ➔  <cond>
- row₃  ➔  ¬<cond>
Goal

Aim: Detect logic bugs in DBMS
Database Management Systems

SELECT * FROM <table>
WHERE <cond>

Client Application

Database Management System (DBMS)

Database

row₁
<cond>
row₂
<cond>
row₃
¬<cond>

row₁
<cond>
row₂
<cond> ✔
Database Management Systems

```
SELECT * FROM <table>
WHERE <cond>
```

Client Application  ➔  Database Management System (DBMS)  ➔  Database

- row₁
- row₂
- row₃

- <cond>
- ¬<cond>
Example Bug: SQLite

CREATE TABLE t1(c1, c2, c3, c4, PRIMARY KEY (c4, c3));
INSERT INTO t1(c3) VALUES (0), (0), (0), (0), (0), (0),
(0), (0), (0), (0), (NULL), (1), (0);
UPDATE t1 SET c2 = 0;
INSERT INTO t1(c1) VALUES (0), (0), (NULL), (0), (0);
ANALYZE t1;
UPDATE t1 SET c3 = 1;
SELECT DISTINCT * FROM t1 WHERE t1.c3 = 1;

ANALYZE gathers **statistics about tables**, which are then used for query planning.
Example Bug: SQLite

```
CREATE TABLE t1(c1, c2, c3, c4, PRIMARY KEY (c4, c3));
INSERT INTO t1(c3) VALUES (0), (0), (0), (0), (0), (0), (0), (0), (0), (0), (NULL), (1), (0);
UPDATE t1 SET c2 = 0;
INSERT INTO t1(c1) VALUES (0), (0), (NULL), (0), (0);
ANALYZE t1;
UPDATE t1 SET c3 = 1;
SELECT DISTINCT * FROM t1 WHERE t1.c3 = 1;
```

<table>
<thead>
<tr>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>0</td>
<td>1</td>
<td>NULL</td>
</tr>
<tr>
<td>0</td>
<td>NULL</td>
<td>1</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>1</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Expected result set

<table>
<thead>
<tr>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>0</td>
<td>1</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Actual result set

<table>
<thead>
<tr>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>0</td>
<td>1</td>
<td>NULL</td>
</tr>
</tbody>
</table>

A bug in the skip-scan optimization caused this logic bug.
Challenges

• DBMS are tested well
Databases are Tested Well

SQLite (~150,000 LOC) has **662 times** as much test code as source code

SQLite is **extensively fuzzed** (e.g., by Google’s OS-Fuzz Project)

SQLite’s test cases achieve **100% branch test coverage**

SQLite’s performs **anomaly testing** (out-of-memory, I/O error, power failures)


[https://www.sqlite.org/testing.html](https://www.sqlite.org/testing.html)
Challenges

• DBMS are tested well
• Fuzzers are ineffective in finding logic bugs
Existing Work: Fuzzers and Query Generators

Fuzzers are effective in detecting bugs that result in crashes

AFL, SQLSmith, QGEN (Poess et al. 2014), ...
Challenges

• DBMS are **tested well**
• Fuzzers are **ineffective** in finding logic bugs
• Knowing the **precise result set** for a query is **difficult**
Test Oracle for Finding Logic Bugs

**Test oracle problem**: For a given input, determine whether the **system works as expected**.
Differential Testing

Differential testing applies only when systems implement the same language.

RS_1 = RS_2 = RS_3?
Problem: Differential Testing

- DBMS-specific SQL
- Common SQL Core

**Problem:** The common SQL core is small
Differential Testing: RAGS (Slutz 1998)

“[Differential testing] proved to be extremely useful, but only for the small set of common SQL”
Constraint Solving (Khalek et al. 2010)

Idea: Use a solver to generate queries, generate data, and provide a test oracle

Could reproduce already reported bugs, injected bugs, but only one (potentially) new bug
Challenges

• DBMS are tested well
• Fuzzers are ineffective in finding logic bugs
• Knowing the precise result set for a query is difficult

The problem of automatically testing DBMS has not yet been well addressed
Approach: Pivoted Query Synthesis

Pivoted Query Synthesis is an automatic testing approach that can be used to effectively test DBMS.

Pivoted Query Synthesis (PQS)
Idea: PQS

Idea: Construct an automatic testing approach considering only a single row

<table>
<thead>
<tr>
<th>Column_0</th>
<th>Column_1</th>
<th>Column_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td><strong>Value_{i,0}</strong></td>
<td><strong>Value_{i,1}</strong></td>
<td><strong>Value_{i,2}</strong></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Intuition

- **Simple** approach and test oracle

```sql
SELECT * FROM <table>
WHERE <cond>

<cond>?
```
Intuition

- **Simple** approach and test oracle
- **Same effectiveness** as checking all rows

```
SELECT * FROM <table>
WHERE <cond>
```

<table>
<thead>
<tr>
<th>Column_0</th>
<th>Column_1</th>
<th>Column_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Value_{i,0}</td>
<td>Value_{i,1}</td>
<td>Value_{i,2}</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- ✔️
- ✔️
- ✔️
Intuition

• **Simple** approach and test oracle
• **Same effectiveness** as checking all rows
Intuition

- **Simple** approach and test oracle
- **Same effectiveness** as checking all rows
- **Precise** oracle for a single row
Approach
Database Generation

To explore “all possible database states” we randomly create databases

<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>A cute toast cat</td>
<td>![Cat Image]</td>
</tr>
<tr>
<td>Dog</td>
<td>Cute dog pic</td>
<td>![Dog Image]</td>
</tr>
<tr>
<td>Cat</td>
<td>Cat plants (cute!)</td>
<td>![Cat with Plants Image]</td>
</tr>
</tbody>
</table>
Pivot Row Selection

Randomly Generate Database

Select Pivot Row

<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>A cute toast cat</td>
<td><img src="image1.png" alt="Cat" /></td>
</tr>
<tr>
<td>Dog</td>
<td>Cute dog pic</td>
<td><img src="image2.png" alt="Dog" /></td>
</tr>
<tr>
<td>Cat</td>
<td>Cat plants (cute!)</td>
<td><img src="image3.png" alt="Cat Plants" /></td>
</tr>
</tbody>
</table>
Query Generation

Randomly Generate Database

Select Pivot Row

Generate Query for the Pivot Row

```
SELECT picture, description
FROM animal_pictures
WHERE animal = 'Cat'
    AND description LIKE '%cute%'
```
Verifying the Result

Randomly Generate Database

Select Pivot Row

Generate Query for the Pivot Row

Verify that the Pivot Row is contained

\[
\text{SELECT picture, description FROM animal_pictures WHERE animal = 'Cat' AND description LIKE '%cute%'}
\]

result set

<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>A cute toast cat</td>
<td>![Cat picture]</td>
</tr>
<tr>
<td>Cat</td>
<td>Cat plants (cute!)</td>
<td>![Cat plants picture]</td>
</tr>
</tbody>
</table>

pivot row

<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>Cat plants (cute!)</td>
<td>![Cat plants picture]</td>
</tr>
</tbody>
</table>
Verifying the Result

Randomly Generate Database → Select Pivot Row → Generate Query for the Pivot Row → Verify that the Pivot Row is contained

**SELECT** picture, description
**FROM** animal_pictures
**WHERE** animal = 'Cat'
AND description LIKE '%cute%'

<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>A cute toast cat</td>
<td></td>
</tr>
<tr>
<td>Dog</td>
<td>Cute dog pic</td>
<td></td>
</tr>
</tbody>
</table>

pivot row

<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>Cat plants (cute!)</td>
<td></td>
</tr>
</tbody>
</table>

pivot row $\not\in$ result set
Verifying the Result

Randomly Generate Database → Select Pivot Row → Generate Query for the Pivot Row → Verify that the Pivot Row is contained

pivot row $\notin$ result set

The “containment oracle” is PQS’ primary oracle
Approach

1. Randomly Generate Database
2. Select Pivot Row
3. Generate Query for the Pivot Row
4. Verify that the Pivot Row is contained
Approach

1. Randomly Generate Database
2. Select Pivot Row
3. Generate Query for the Pivot Row
4. Verify that the Pivot Row is contained
Approach

1. Randomly Generate Database
2. Select Pivot Row
3. Generate Query for the Pivot Row
4. Verify that the Pivot Row is contained
Approach

Randomly Generate Database

Select Pivot Row

Generate Query for the Pivot Row

Verify that the Pivot Row is contained

How do we generate this query?
How do we Generate Queries?

```
SELECT picture, description
FROM animal_pictures
WHERE
```

Generate an **expression** that **yields TRUE** for the pivot row.
How do we Generate Queries?

1. Randomly Generate Expression
2. Evaluate Expression on Pivot Row
3. Modify expression to yield TRUE
4. Use in WHERE clause
Random Expression Generation

We first generate a random expression

animal_pictures

animal | description | picture

https://www.sqlite.org/syntax/expr.html
Random Expression Generation

```
animal = 'Cat'
AND description LIKE '%cute%
```
Random Expression Generation

\[
\text{animal} = 'Cat' \quad \text{AND} \quad \text{description} \text{ LIKE 'cute'}\
\]

Evaluate the tree based on the pivot row.
Random Expression Evaluation

AND

= 'Cat'

LIKE '%cute%'

Constant nodes return their assigned literal values

<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>Cat plants (cute!)</td>
<td></td>
</tr>
</tbody>
</table>
Random Expression Evaluation

AND

= 'Cat'

'Cat plants (cute!)

LIKE

'Cat'

description

'Cat'

'Cat'

'%cute%'

'%cute%'
Random Expression Evaluation

AND

TRUE

TRUE

= 'Cat' 'Cat' 'Cat plants (cute!)'

LIKE

'en
descri ption'

'cute%'

Compound nodes compute their result based on their children

<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>Cat plants (cute!)</td>
<td></td>
</tr>
</tbody>
</table>
Random Expression Evaluation

\[
\text{TRUE} \quad \text{AND} \quad \text{TRUE} \quad \text{TRUE}
\]

\[
\text{animal} \rightarrow \text{Cat'} \quad \text{LIKE} \quad \text{description} \rightarrow \text{Cat'} \quad \text{LIKE} \quad \text{picture} \rightarrow \text{Cat plants (cute!)}
\]

\[
\text{'Cat'} \quad \text{=} \quad \text{'Cat'} \quad \text{LIKE} \quad \text{'%cute%'}
\]

\[
\text{TRUE} \quad \text{AND} \quad \text{TRUE} \quad \text{TRUE}
\]
Query Synthesis

```
SELECT picture, description
FROM animal_pictures
WHERE animal = 'Cat' AND description LIKE '%cute%'
```
Random Expression Evaluation

What about when the expression does not evaluate to TRUE?

```
AND
  /
 TRUE
 /
 TRUE
 /
 TRUE
 /
 animal = 'Cat'
 /
 'Cat (cute!)
 /
 description LIKE '%cute%'
```

<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>Cat plants</td>
<td>(cute!)</td>
</tr>
</tbody>
</table>
Random Expression Evaluation

What about when the expression does not evaluate to TRUE?

animal = 'Dog'

<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>Cat plants (cute!)</td>
<td></td>
</tr>
</tbody>
</table>
Random Expression Rectification

```c
switch (result) {
    case TRUE:
        result = randexpr;
    case FALSE:
        result = NOT randexpr;
    case NULL:
        result = randexpr ISNULL;
}
```
Random Expression Rectification

```java
switch (result) {
    case TRUE:
        result = randexpr;
    case FALSE:
        result = NOT randexpr;
    case NULL:
        result = randexpr ISNULL;
}
```

animal = 'Dog'

<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>Cat plants (cute!)</td>
<td>![Picture of a cat]</td>
</tr>
</tbody>
</table>
Random Expression Rectification

```java
switch (result) {
    case TRUE:
        result = randexpr;
    case FALSE:
        result = NOT randexpr;
    case NULL:
        result = randexpr ISNULL;
}
```

```
NOT(animal = 'Dog')
```

<table>
<thead>
<tr>
<th>animal</th>
<th>description</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>Cat plants (cute!)</td>
<td></td>
</tr>
</tbody>
</table>
How do we Generate Queries?

```
SELECT picture, description
FROM animal_pictures
WHERE NOT(animal = 'Dog')
```
Evaluation
Tested DBMS

PostgreSQL

SQLite

MySQL

We tested these (and other DBMS) in a period of 3-4 months
DBMS

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Popularity Rank</th>
<th>LOC</th>
<th>First Release</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DB-Engines</td>
<td>Stack Overflow</td>
<td>LOC</td>
<td>First Release</td>
</tr>
<tr>
<td>SQLite</td>
<td>11</td>
<td>4</td>
<td>0.3M</td>
<td>2000</td>
</tr>
<tr>
<td>MySQL</td>
<td>2</td>
<td>1</td>
<td>3.8M</td>
<td>1995</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>4</td>
<td>2</td>
<td>1.4M</td>
<td>1996</td>
</tr>
</tbody>
</table>
## DBMS

<table>
<thead>
<tr>
<th>DBMS</th>
<th>DB-Engines</th>
<th>Stack Overflow</th>
<th>LOC</th>
<th>First Release</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>11</td>
<td>4</td>
<td>0.3M</td>
<td>2000</td>
<td>19 years</td>
</tr>
<tr>
<td>MySQL</td>
<td>2</td>
<td>1</td>
<td>3.8M</td>
<td>1995</td>
<td>24 years</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>4</td>
<td>2</td>
<td>1.4M</td>
<td>1996</td>
<td>23 years</td>
</tr>
</tbody>
</table>
DBMS

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Popularity Rank</th>
<th>LOC</th>
<th>First Release</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DB-Engines</td>
<td>Stack Overflow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLite</td>
<td>11</td>
<td>4</td>
<td>0.3M</td>
<td>2000</td>
</tr>
<tr>
<td>MySQL</td>
<td>2</td>
<td>1</td>
<td>3.8M</td>
<td>1995</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>4</td>
<td>2</td>
<td>1.4M</td>
<td>1996</td>
</tr>
</tbody>
</table>
Bugs Overview

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Fixed</th>
<th>Verified</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>65</td>
<td>0</td>
</tr>
<tr>
<td>MySQL</td>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Sum</td>
<td>85</td>
<td>14</td>
</tr>
</tbody>
</table>

**99 real bugs**: addressed by code or documentation fixes, or verified as bugs
Bugs Overview

The SQLite developers quickly responded to all our bug reports → we focused on this DBMS

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Fixed</th>
<th>Verified</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>65</td>
<td>0</td>
</tr>
<tr>
<td>MySQL</td>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Sum</td>
<td>85</td>
<td>14</td>
</tr>
</tbody>
</table>

Real Bugs
## Bugs Overview

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Fixed</th>
<th>Verified</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>65</td>
<td>0</td>
</tr>
<tr>
<td>MySQL</td>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Sum</td>
<td>85</td>
<td>14</td>
</tr>
</tbody>
</table>

All MySQL bug reports were **verified quickly**.
## Bugs Overview

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Fixed</th>
<th>Verified</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>65</td>
<td>0</td>
</tr>
<tr>
<td>MySQL</td>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Sum</td>
<td>85</td>
<td>14</td>
</tr>
</tbody>
</table>

MySQL’s trunk is **not available**, and it has a long release cycle.
Bugs Overview

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Fixed</th>
<th>Verified</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>65</td>
<td>0</td>
</tr>
<tr>
<td>MySQL</td>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Sum</td>
<td>85</td>
<td>14</td>
</tr>
</tbody>
</table>

We found the fewest bugs in PostgreSQL and not all could be easily addressed.
## Oracles

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Containment</th>
<th>Error</th>
<th>SEGFAULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>46</td>
<td>17</td>
<td>2</td>
</tr>
<tr>
<td>MySQL</td>
<td>14</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>1</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Sum</td>
<td>61</td>
<td>34</td>
<td>4</td>
</tr>
</tbody>
</table>
## Oracles

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Containment</th>
<th>Error</th>
<th>SEGFAULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>46</td>
<td>17</td>
<td>2</td>
</tr>
<tr>
<td>MySQL</td>
<td>14</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>1</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Sum</td>
<td>61</td>
<td>34</td>
<td>4</td>
</tr>
</tbody>
</table>

Our *Containment* oracle allowed us to detect **most errors**
Result: Bug in SQLite3

```
CREATE TABLE t0(c1 TEXT PRIMARY KEY) WITHOUT ROWID;
CREATE INDEX i0 ON t0(c1 COLLATE NOCASE);
INSERT INTO t0(c1) VALUES ('A');
INSERT INTO t0(c1) VALUES ('a');
```

An index is an auxiliary data structure that should not affect the query’s result
Result: Bug in SQLite3

```
CREATE TABLE t0(c1 TEXT PRIMARY KEY) WITHOUT ROWID;
CREATE INDEX i0 ON t0(c1 COLLATE NOCASE);
INSERT INTO t0(c1) VALUES ('A');
INSERT INTO t0(c1) VALUES ('a');
```

Real Bugs

Containment
Oracle

<table>
<thead>
<tr>
<th>c1</th>
</tr>
</thead>
<tbody>
<tr>
<td>'A'</td>
</tr>
<tr>
<td>'a'</td>
</tr>
</tbody>
</table>
Result: Bug in SQLite3

CREATE TABLE t0(c1 TEXT PRIMARY KEY) WITHOUT ROWID;
CREATE INDEX i0 ON t0(c1 COLLATE NOCASE);
INSERT INTO t0(c1) VALUES ('A');
INSERT INTO t0(c1) VALUES ('a');

SELECT * FROM t0;

SQLite failed to fetch 'a'!
Result: Bug in PostgreSQL

CREATE TABLE t0(c0 INT PRIMARY KEY, c1 INT);
CREATE TABLE t1(c0 INT) INHERITS (t0);
Result: Bug in PostgreSQL

CREATE TABLE t0(c0 INT PRIMARY KEY, c1 INT);
CREATE TABLE t1(c0 INT) INHERITS (t0);

INSERT INTO t0(c0, c1) VALUES(0, 0);

<table>
<thead>
<tr>
<th>t0</th>
<th>c0</th>
<th>c1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>t1</th>
<th>c0</th>
<th>c1</th>
</tr>
</thead>
</table>
Result: Bug in PostgreSQL

CREATE TABLE t0(c0 INT PRIMARY KEY, c1 INT);
CREATE TABLE t1(c0 INT) INHERITS (t0);
INSERT INTO t0(c0, c1) VALUES(0, 0);
INSERT INTO t1(c0, c1) VALUES(0, 1);

The inheritance relationship causes the row to be inserted both in t0 and t1

Real Bugs

Containment
Oracle
Result: Bug in PostgreSQL

CREATE TABLE t0(c0 INT PRIMARY KEY, c1 INT);
CREATE TABLE t1(c0 INT) INHERITS (t0);
INSERT INTO t0(c0, c1) VALUES(0, 0);
INSERT INTO t1(c0, c1) VALUES(0, 1);

SELECT c0, c1 FROM t0
GROUP BY c0, c1;

PostgreSQL failed to fetch the row 0 | 1
Result: Bug in MySQL

```
CREATE TABLE t0(c0 TINYINT);
INSERT INTO t0(c0) VALUES(NULL);
```

<table>
<thead>
<tr>
<th>t0</th>
<th>c0</th>
<th>NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>NULL</td>
</tr>
</tbody>
</table>
Result: Bug in MySQL

```
CREATE TABLE t0(c0 TINYINT);
INSERT INTO t0(c0) VALUES(NULL);

SELECT * FROM t0
WHERE NOT(t0.c0 <=> 2035382037);
```

The MySQL-specific equality operator `<>` malfunctioned for large numbers
# Oracles

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Containment</th>
<th>Error</th>
<th>SEGFAULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>46</td>
<td>17</td>
<td>2</td>
</tr>
<tr>
<td>MySQL</td>
<td>14</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>1</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Sum</td>
<td>61</td>
<td>34</td>
<td>4</td>
</tr>
</tbody>
</table>

We also found many bugs using an *Error* oracle.
SQLite3 Bug

CREATE TABLE t1 (c0, c1 REAL PRIMARY KEY);

INSERT INTO t1(c0, c1) VALUES (TRUE, 9223372036854775807), (TRUE, 0);

UPDATE t1 SET c0 = NULL;

UPDATE OR REPLACE t1 SET c1 = 1;

SELECT DISTINCT * FROM t1 WHERE (t1.c0 IS NULL);

The INSERT and UPDATE statements corrupted the database

Database disk image is malformed
Anecdote

Understanding GCC Builtin to Develop Better Tools

Manuel Rigger
Johannes Kepler University Linz
Austria
manuel.rigger@jku.at

Bram Adams
Polytechnique Montréal
Canada
bram.adams@polymtl.ca

Stefan Marr
University of Kent
United Kingdom
s.marr@kent.ac.uk

Hanspeter Mössenböck
Johannes Kepler University Linz
Austria
hanspeter.moessenboeck@jku.at

ABSTRACT
C programs can use compiler builtins to provide functionality that the C language lacks. On Linux, GCC provides several thousands of builtins that are also supported by other mature compilers, such as Clang and ICC. Maintainers of other tools lack guidance on whether and which builtins should be implemented to support popular projects. To assist tool developers who want to support GCC builtins, we analyzed builtins in 4,913 C projects from GitHub. We found that 37% of these projects relied on at least one builtin. Supporting an increasing proportion of projects requires support of an exponentially increasing number of builtins; however, implementing only 19 builtins already covers 30% of the projects. Since we found that many builtins in our corpus remained unused, the effort needed to support 90% of the projects is moderate, requiring about 110 builtins to be implemented. For each project, we analyzed the evolution of builtin use over time and found that the majority of projects mostly added builtins. This suggests that builtins are not a legacy feature and must be supported in future tools. Systematic testing of builtin support in existing tools revealed that many lacked support for builtins either partially or completely; we also discovered incorrect implementations in various tools, including the formally verified CompCert compiler.

CCS CONCEPTS

KEYWORDS
GCC builtins, compiler intrinsics, C GitHub projects

Error: Database disk image is malformed

SQLite3 Database
Oracles

We found only a low number of crash bugs, likely because DBMS are fuzzed extensively.

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Containment</th>
<th>Error</th>
<th>SEGFAULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>46</td>
<td>17</td>
<td>2</td>
</tr>
<tr>
<td>MySQL</td>
<td>14</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>1</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Sum</td>
<td>61</td>
<td>34</td>
<td>4</td>
</tr>
</tbody>
</table>
Average Number of Statements

Half of all bugs can be reproduced with only 4 SQL statements.
SQLite3 Bug with a Single Statement

SELECT '' - 2851427734582196970;

-2851427734582196936

Subtracting a large integer from a string resulted in an incorrect result
Discussion

• Are the bugs relevant?

<table>
<thead>
<tr>
<th>Severity Level</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical</td>
<td>14</td>
</tr>
<tr>
<td>Severe</td>
<td>8</td>
</tr>
<tr>
<td>Important</td>
<td>14</td>
</tr>
</tbody>
</table>

The SQLite developers (inconsistently) assigned severity levels
Discussion

• Are the bugs relevant?
• Statement coverage

Low coverage 20%-50%, DBMS provide a lot more than pure database management
Discussion

• Are the bugs relevant?
• Statement coverage
• Implementation effort

4,000-6,000 LOC per DBMS \(\rightarrow\) significantly smaller than the DBMS
Discussion

• Are the bugs relevant?
• Statement coverage
• Implementation effort
• Limitations

- Aggregate and window functions
- Difficult-to-implement functionality
Compiler Design and DBMS

• Similarly to compilers, DBMS implement many optimizations
  • Which can be buggy too!

• You could all have implemented the approach yourself
  • Coming up with the “right” conceptual idea is hard

• Bachelor theses in these and other areas
Aim: Find Logic Bugs in DBMS

Challenge: Precise Oracle is Difficult to Construct

Idea: Consider Only a Single Row

Create Expressions that Yield TRUE for the Pivot Row

PQS is Highly Effective

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Fixed</th>
<th>Verified</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>65</td>
<td>0</td>
</tr>
<tr>
<td>MySQL</td>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Sum</td>
<td>85</td>
<td>14</td>
</tr>
</tbody>
</table>

99 real bugs: addressed by code or documentation fixes, or verified as bugs