HeisenBase: Understanding Database Performance in Haskell

Dept. of CIS - Senior Design 2012-2013

Zachary Wasserman
zwass@seas.upenn.edu
Univ. of Pennsylvania
Philadelphia, PA

Susan Davidson
susan@cis.upenn.edu
Univ. of Pennsylvania
Philadelphia, PA

ABSTRACT

This work aims to develop a performant relational database management system, HeisenBase, in the functional programming language Haskell. Much work has been published with the intent to create sensible interaction layers between the Haskell programming language and existing database software. We build the DBMS itself in Haskell in order to understand the benefits and drawbacks of a Haskell-based implementation.

Our research has led to a proof-of-concept DBMS implemented in Haskell. This DBMS supports basic relational queries and joins over data tables. We present metrics investigating the unique performance characteristics of the system, and analysis of the technology we developed.

1. INTRODUCTION

Below we present an introduction to the motivation (Sec. 1.1), approach (Sec. 1.2), challenges (Sec. 1.3) and contributions (Sec. 1.4) of this work.

1.1 Motivation

Database management systems (DBMS) pervade the landscape of tools used by software engineers. Relational databases like MySQL have over 400 developer years put into reliability and fast performance [16]. Vast amounts of effort have been put into understanding the performance of these systems, both from an academic perspective, as well as industry. Typically, databases such as MySQL are written in low level systems languages such as C and C++ [16].

We look to approach the problem of implementing a relational database through a very different style of programming. By attempting to implement a usable DBMS in Haskell, we hope to further understand the implications of systems programming in Haskell, as well as how lazy evaluation effects performance.

There is little existing work focusing on systems development in Haskell. The language serves mostly as a unified platform for research in functional programming languages. Other projects, such as a Haskell implementation of an operating system kernel demonstrate the feasability of building high-performance, low-level systems using the language [7].

One motivating factor for exploring a Haskell implementation of a DBMS is the relative ease by which the strongly-typed Haskell programs can be proven correct. With strong guarantees from the type system, and innovative infrastructure for testing, Haskell has earned a reputation for producing correct programs. Prior research has gone as far as formally verifying correctness through the assistance of automated theorem provers [1].

Another is the unique performance characteristics of the language itself. Different runtime evaluation semantics lead to surprising performance characteristics which can be applied to optimizing a database system for certain specific workloads.

1.2 Approach

The core of this work is a proof-of-concept database implementation. Through adaptations of traditional database development techniques, we create a database supporting a variety of data retrieval operations.

Additional tooling provides insight into the unique performance characteristics of this system. This is presented by plots of the prototype's real world performance.

Testing through Haskell's standard testing toolkit demonstrates correctness of the database and indices.

1.3 Challenges

Haskell is a relatively new language. Though it is well proven for use in functional programming research, the language has seen little use in industry and systems programming. This work represents an early foray into relational database development using the language.

Challenges arise from dealing with the language enforcement of purity, as a database is a highly stateful piece of software. In addition, lazy evaluation causes very different performance characteristics from the programming languages typically used to implement database systems.

1.4 Contributions

- A proof of concept Haskell relational database implementation.
- An in-memory Haskell implementation of a B+ Tree for database indexing.
- Findings on database performance in the context of Haskell's lazy evaluation.
- Recommendations for harnessing Haskell's strengths to implement future relational database systems.

2. BACKGROUND

In Section 2.1 we provide an introduction to the theory of databases, followed by an introduction to Haskell in Section 2.2.

2.1 Databases

A database is a system used to allow access to large sets of interrelated data. DBMS users should be "protected from having to know how the data is organized in the machine," while able to access their data in the required manner. A *relational* database is one in which data is described by its "natural structure," or the relationships between data sets. Data can be queried individually, or connected by its relationships [4].

Databases use indices in order to optimize certain types of queries. Without indices, many queries require fully enumerating the stored data when only a single record is needed. Common index types are tree-based and hash-based indices [15].

Projects throughout academia and industry rely on databases for data archiving, indexing, analysis, and retrieval.

2.2 Haskell

Haskell is a "polymorphically statically typed, lazy, purely functional language" [12] designed to promote further research and education in functional languages, as well as for applications including "development of large systems" [14].

Haskell utilizes polymorphism in order to provide facilities for building useful abstractions while implementing software systems. Primarily it uses parametric polymorphism [8], allowing definition of functions and data structures that can operate on arbitrary types of data (subject to these data types meeting certain restrictions).

Static typing in Haskell provides for stronger assurances of correctness in software developed using the language. Together with the polymorphism facilities, this system is both precise and flexible.

Haskell's lazy evaluation is one of its most touted features, yet it provides an extra challenge when implementing a database. Laziness allows the language runtime to evaluate function arguments at most once. This "at most once" semantic can result in some structures never being evaluated at all [9]. In Haskell, these properties also allow for useful definitions of infinite data structures and computations. Due to lazy evaluation, the tail of an infinite list, for example, can be cut off leaving only the (finite) beginning elements of that list.

The last important feature of Haskell, purity, comes hand in hand with lazy evaluation. Haskell is a purely functional language, meaning that all functions in Haskell are side effect free (they will always return the same value, given the same input, and will not modify the program state). This is an important property when we have lazy evaluation, as otherwise it would be difficult for the programmer to reason about when side effects occur. To allow for useful computation (such as input/output), Haskell uses a construct called "monads" to separate the stateful computations from the purely functional computations [13].

Monads are a useful abstraction in pure functional programming languages, allowing the language to "integrate the benefits of the pure and impure schools" [18]. A monad is simply a piece of data conforming to a certain interface (the functions of the `Monad` typeclass, along with their corresponding laws), allowing sequencing operations (such as passing of program state) to be applied to it. Thus, functions can be chained in very particular ways to provide useful abstractions to the programmer [13]. Haskell's standard library comes with a variety of polymorphic monad implementations providing facilities such as error handling, simulated stateful computation, and input/output.

In addition to Haskell's strong static type system, the QuickCheck library is often used for program verification. QuickCheck provides infrastructure for property based testing – Randomized test cases run against predefined properties of the system, surfacing edge cases, and problems with both the specification and software itself [3]. We use the QuickCheck library to ascertain the correctness of the HeisenBase system.

3. RELATED WORK

There is a large body of work related to HeisenBase. Section 3.1 discusses databases, and Section 3.2 provides more about Haskell.

3.1 Databases

The earliest work on relational databases was [4]. In this publication, Codd defined the relational algebra, a system for defining queries that can be made on relational data. He also introduced the idea of the DBMS as a black box for storage and retrieval of data. Our system attempts to fit this black box model, not requiring the user to consider system internals while using it to store data. We also use the relational algebra in our internal representation and optimization of queries.

Later work by Bayer and McCreight established the B Tree as an efficient data structure for evaluation of relational algebra queries [2]. The B Tree is a generalization of binary trees that allows for branching beyond a factor of two [5]. Using a B Tree, the DBMS can have $O(\log n)$ access time to arbitrary data, with a large base on the logarithm reducing the number of disk accesses required before reaching the requested data.

Continued investigation of B Trees led to a variant, the B+ Tree that has become a standard for implementation of databases. In a B+ Tree, all data is stored at leaf nodes, which allows for efficient sequential access of data, in addition to random access [5]. B+ Trees are used in many popular databases such as Oracle and MSSQL in the modern day [15]. In accordance with the standard of using the B+ Tree for data indexing, HeisenBase does the same.

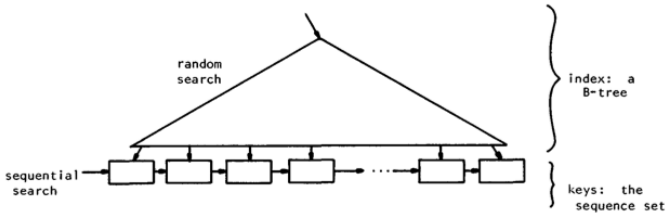

Figure 1: Structure of a B+ Tree [5]

Figure 1 demonstrates the structure of a B+ tree. Notice that random access can be performed by the $O(\log n)$ tree traversal from the top arrow, or sequential access through

the left arrow. Also, sequential access can begin from any randomly accessed node.

More modern work by Graefe discusses techniques for optimizing query evaluation. He presents an overview of a number of algorithms for computing joins with efficient use of the disk [6]. We considered these algorithms in implementing the query optimization step of the system.

3.2 Haskell

A great deal of effort has been published in understanding the performance of Haskell programs. Due to language features like lazy evaluation, this has been a fruitful area for research.

Okasaki [11] discusses data structures for efficient computation in general (pure) functional programming languages. He asserts that "only a small fraction of existing data structures are suitable for implementation in functional languages," and provides data structures with efficient applications in functional languages. There is also a focus on methods for understanding algorithmic complexity in the presence of lazy evaluation. Though this work does not offer focus on Haskell or databases in particular, the general perspective on data structure implementation in a lazy functional language provided guidance in our B+ Tree implementation.

Techniques for optimizing performance of Haskell code are discussed in [17]. Though lazy evaluation provides a useful facility for abstraction, we need to circumvent the mechanism when we determine that performance is actually being degraded. Tibell's work [17] discusses techniques for forcing strict evaluation in the presence of these conditions. We find this applicable in lazy-evaluation of the B+ Tree index. Forcing strict evaluation during B+ Tree construction can prevent long delays from lazily postponed index evaluation when queries are executed.

Other work has been published demonstrating Haskell's viability as a language for serious systems development. In [7], an interface is defined through monads for operating system level operations in Haskell. The resulting system is a functioning operating system written entirely in Haskell. Much of this work was focused on utilizing Haskell's type system for verification of the system. The authors conclude that more work must be done to understand the performance implications of using Haskell at such a low level. We endeavor to give further understanding of this low-level performance through the HeisenBase implementation.

Much work has been performed attempting to create interfaces between Haskell programs and existing DBMS systems (Imperative language implementations such as MySQL). Unique challenges are presented in this work due to Haskell's strict type system. Systems like HDBC [13] connect Haskell programs with relational databases that use structured query language (SQL) for their queries. Some support is also provided for expressing in Haskell, rather than in SQL the queries that should be run on the database. HaskellDB [10] actually creates a domain specific language within Haskell for expressing these queries.

Unpublished work on Github uses Haskell to implement a database. This database, Siege, is a key/value style database, and not fully relational [19]. It implements the Redis protocol enabling fast key/value access. HeisenBase, on the other hand, is a fully relational database, providing efficient join operations among others.

4. SYSTEM MODEL

We are building a relational database in Haskell, with the aim to better understand how Haskell's language features effect performance in a DBMS. This includes an implementation of a B+ Tree for data indexing, a parser for our SQL-like query language, as well as a query planner/evaluator that can execute the parsed queries. These components all come together in a read-evaluate-print-loop called the HeisenBase shell.

```
Customer(cid:integer, name:string)
Order(oid:integer, cid:integer, item:string)
```

Figure 2: Example schema

Figures 2 and 3 demonstrate a potential schema and query in HeisenBase. This syntax is modeled after SQL, and the query shown in Fig. 3 is one of the more complex queries the system is intended to handle.

```
SELECT *
FROM Customer JOIN Order
WHERE Customer.name = 'Jones'
```

Figure 3: Example query

Benchmarks allow us to understand how design tradeoffs impact the performance of the system. This helps to illuminate both how these types of systems can best be designed given the particular features of Haskell, and the potential for real world database solutions to be implemented in a language like it.

Figure 3 shows how the components of the system interact to provide the described functionality. The main focus of the project is on the query evaluator, B+ tree index, and language parsing, with the benchmark system enabling evaluation of query performance.

5. RESULTS

Our work has led to a fully functional relational database proof-of-concept in Haskell. The datatypes supported by the system are discussed in Section 5.1. We utilize a B+ Tree implementation discussed further in Section 5.2. Interactions with the system itself are supported through the HeisenBase Shell, discussed in Section 5.3. A Python based wrapper has been built to test and record performance benchmarks for the completed system (Sec. 5.4). Throughout all of this work, the most consistent challenge was managing state in the pure environment of Haskell.

5.1 Data Types

HeisenBase supports a small set of the most critical data types. These include boolean values, integers, floating point numbers, and strings. With this, the system should support nearly all the data that a user might need to store.

By defining a new Haskell datatype for storing these values, we simplify the types involved in the index and evaluator. Figure 5 shows the definition of this type.

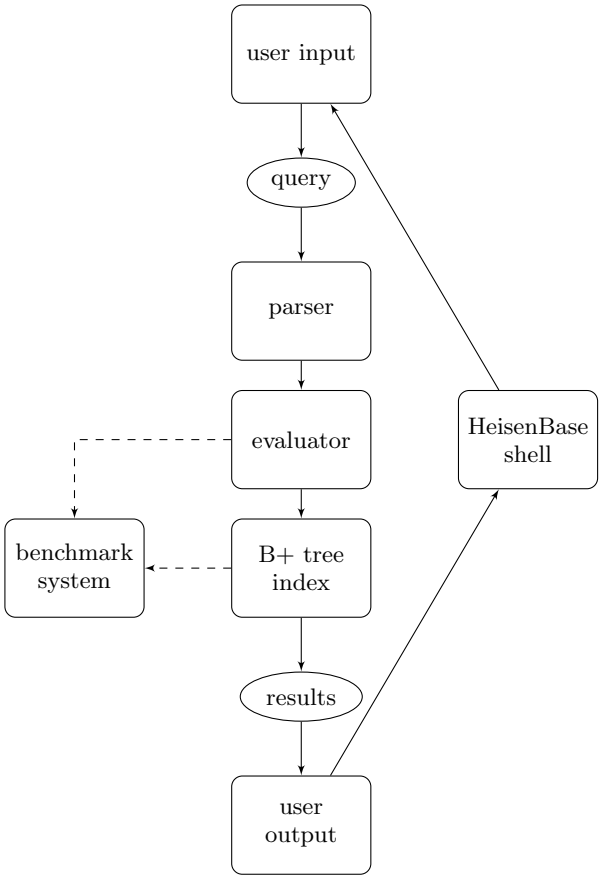

Figure 4: Execution flow of HeisenBase query

```
data DataType =
  DBool Bool
  | DInt Int
  | DFloat Float
  | DString String
  deriving (Eq, Ord, Read)
```

Figure 5: Haskell data type for column values

5.2 B+ Tree

The implementation of our B+ Tree index avoids stateful computation entirely. It is a persistent implementation of the B+ Tree structure, never modifying the existing tree, but instead returning a new tree with the appropriate modifications. This is a common idiom in pure functional data structures, and Okasaki believes with lazy evaluation it can result in efficient overall performance [11].

When evaluation is performed lazily, the entire tree may be "built," without any computation actually taking place. Thus it is relatively inexpensive to be constantly creating new copies of the data structure.

Figures 6 and 7 illustrate how the evaluation of the index might proceed after construction and query evaluation.

The representation of the B+ Tree is implemented as a recursive data type in Haskell, shown in Figure 8.

As the B+ Tree is responsible for indexing the data stored in the database, its correctness is critical. We have used

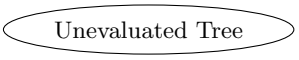

Figure 6: B+ Tree after construction, before lookup

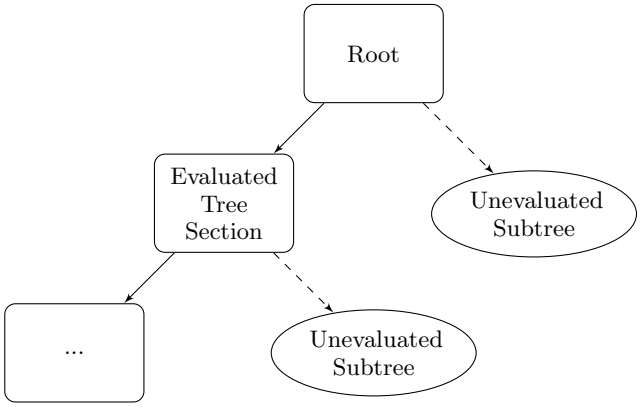

Figure 7: B+ Tree after lookup in left subtree

property based testing with Haskell's QuickCheck library to verify our tree implementation. This enables us to execute randomized testing to verify that the tree is balanced (also important for performance), and that every inserted element can be retrieved.

Despite its stateless, persistent implementation, monads are used in the B+ Tree to provide serialization to/from disk. We have defined an instance of the `Binary` typeclass for serialization of our tree structure to/from binary bytestrings. With this, we can use the IO monad to write a tree to file, or read a tree back from an existing file.

We have encountered the unfortunate consequences of lazy evaluation through stack space overflows when a massive number of database insertions are performed sequentially. Because the system will never evaluate the built up B+ Tree computations until we need the results (e.g. a lookup is performed), we encounter a "space leak", in which the computations build up until the stack overflows. To get around this, we periodically write the tree back to disk, forcing evaluation at that time.

5.3 HeisenBase Shell

To demonstrate the capabilities of the HeisenBase system, we have created a shell for interaction with the database itself. This shell provides support for creation of databases, along with the full set of implemented queries.

These queries include indexed selection (supported by the B+ Tree index), non-indexed selection (linear scans through the tree), aggregate selection, and indexed joins.

A parser was implemented using Haskell's Parsec library.

```
data BTree =
  Branch { size :: Int,
           children :: [(Maybe DataType, BTree)] }
  | Leaf { size :: Int,
           entries :: [(DataType, [DataType])] }
  deriving (Show, Eq, Ord, Read)
```

Figure 8: Haskell data type for B+ Tree

```
type DBState a b c =
  StateT (BTree a b)
  (ReaderT ReaderState IO) c
```

Figure 9: The `DBState` mega monad for managing state within the HeisenBase shell

This library enabled us to do parsing and lexing of query strings using the high-level Haskell abstractions of Monads and Applicative Functors.

In the shell we create a customized monad for handling our state using Haskell standard library monad transformers. Our monad is a basic `IO` monad wrapped by a `ReaderT` monad transformer wrapped by a `StateT` monad transformer. With this custom "mega-monad", we can support shell IO operations, state changes in the database, and an environment for storing read-only database state. Figure 9 shows the Haskell code defining this monad.

We also wrote a small amount of parsing code for the basic operations currently supported. These use the standard library parsing, and provide minimal error handling. Eventually, parsing for the full supported SQL language will be handled using the Parsec library.

Using the serialization functions provided by the B+ Tree implementation, the shell session can work with an existing database file, or create a new one.

5.4 Benchmarking

In order to benchmark the completed system, we have built a Python script that measures HeisenBase performance through inter-process communication. Essentially, we open a Unix pipe and communicate with the HeisenBase shell over that channel

Using this system, we have been able to perform insertions and selections on B+ Trees of size up to 1,000,000 keys.

In Section 6, we explain some of the benchmarking results collected so far.

5.5 Query Evaluator

This system is used to optimize the queries run through the HeisenBase system.

We implemented the query evaluator using the relational algebra, and techniques of query optimization discussed in [4], [6] and [15]. The job of the query evaluator will is to understand how to structure queries internally in order to maximize the efficiency with which the system can execute them. As in Codd's black box theory of the database, the user should not need knowledge of how the data is structured internally in order to access the data efficiently.

In particular, the query evaluator utilizes the B+ Tree index whenever possible to improve the performance of queries. The results of the parsed query are provided to the evaluator, which takes advantage of knowledge of the internal state of the system to retrieve results.

6. SYSTEM PERFORMANCE

Using the benchmarking tool described in Section 5.4, we were able to collect a variety of metrics for the performance of the HeisenBase system.

In Figure 10, we can see the performance of HeisenBase as the number of elements stored in the index increases. These represent 100 trials for each database size, throwing out the top and bottom 10% of data points to account for interference by the operating system. The database in each trial has already been loaded from disk, and queried once to force evaluation.

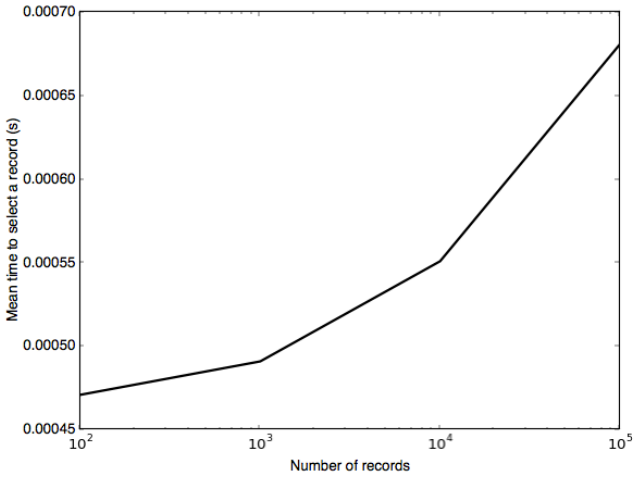

Figure 10: Database size vs. Query performance

Figure 11 shows the time taken in loading the entire database, and querying every key contained. This graph appears entirely linear on linear axes. This can be explained partially due to the fact that the number of queries and volume of data loaded increase linearly with the size of the database. We presume that high overheads on our inter-process communication during the benchmarking step mask small differences in performance during this test.

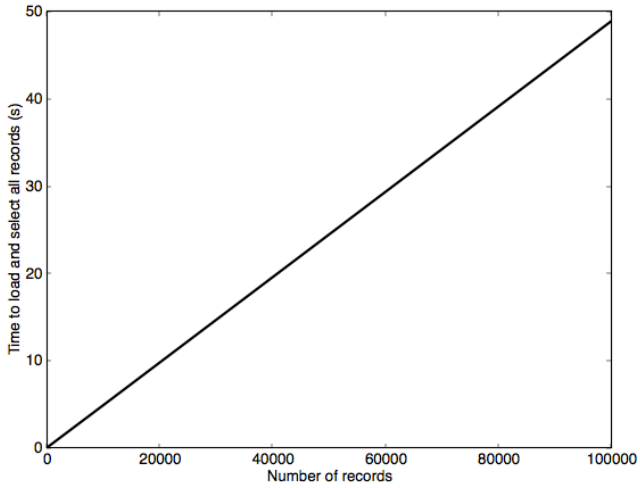

Figure 11: Database size vs. performance of loading and querying the entire dataset

Lastly, Figure 12 shows a good intuition for how lazy evaluation actually impacts the performance of the HeisenBase system. We can see in this trial that the first query takes an order of magnitude longer to execute than the following queries. This can be explained by the forced evaluation of the IO and deserialization required to actually evaluate the index tree. In subsequent lookups, the tree has already been evaluated, and traversals are quite fast.

For this particular trial, we ran 20 consecutive queries on a database with 10,000 entries, though similar results were found with different database sizes and number of trials.

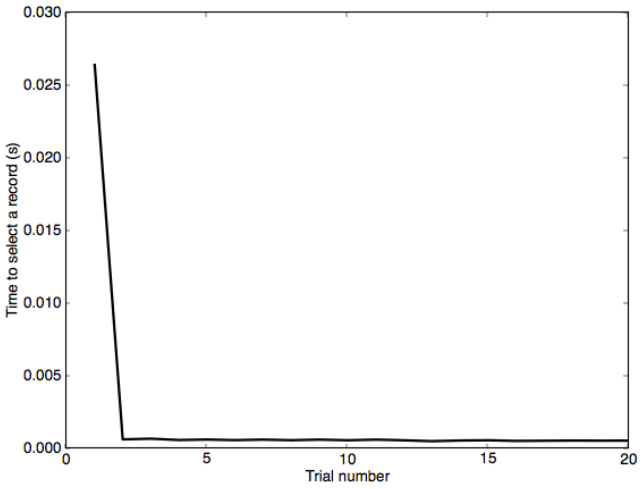

Figure 12: Query number vs. query performance (Sample database with 10,000 entries)

It is not possible to compare the performance of HeisenBase with existing database systems. Due to the vast amount of developer time put into those systems, the paradigms they use for storage, retrieval and caching of their data prevent us from making any direct comparison with the proof-of-concept HeisenBase system.

Rather than focus on a comparison with existing systems, we would like to encourage analysis of the potential for the unique performance characteristics afforded by Haskell in implementing future systems. Figure 12 in particular highlights this interesting performance.

Future Haskell database implementations could be tailor-made to support specific workloads with very high performance. Given a workload with a huge number of insertions, and a small number of lookups, the lazy evaluation could drastically reduce peak latency. Repeated lookups are afforded very fast performance, and construction of the indices for newly inserted data can be delayed to a lower traffic time.

7. ETHICS

While the HeisenBase system demonstrates correct performance through the Haskell type system and QuickCheck library, it is not ready for real world use. Since databases play a central role in many mission-critical systems, they must be solid in their guarantees about the integrity of data they store. HeisenBase cannot yet make these guarantees.

There is no support for database transactions, and no logging to guarantee consistency in the presence of catastrophic system failures. These features are essential to protecting data in production systems. Companies can lose money, customers, and reputation when their databases fail. Individuals can be personally harmed if their data is lost.

With no security features, HeisenBase is a real risk for storing sensitive data. Conventional database systems provide separation of roles, encryption and other features for protecting data from invasive users. This is a protection that the system makes no effort to provide.

Should the work be used as a datastore for future research, these integrity issues could have wide-ranging ramifications. Scientists depend on their computer systems to be reliable beyond the current guarantees of HeisenBase.

The HeisenBase system is intended as a proof-of-concept for Haskell databases. It demonstrates feasability, and desirable performance properties that can be derived from the use of Haskell. Further work should be done to build a production-ready database for real world use.

8. CONCLUSION

We have demonstrated the viability of Haskell for implementation of large scale, relational database systems. This work provided both a proof-of-concept implementation of a DBMS, as well as recommendations for the usefulness of Haskell in implementing this type of system.

Haskell provides a rich foundation from which to build correct, performant computing systems. HeisenBase is only the beginning of Haskell's use in this style.

9. REFERENCES

[1] Andreas Abel, Marcin Benke, Ana Bove, John Hughes, and Ulf Norell. Verifying haskell programs using constructive type theory. In *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, Haskell '05, pages 62–73, New York, NY, USA, 2005. ACM.

[2] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Inf.*, 1:173–189, 1972.

[3] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM.

[4] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[5] Douglas Comer. Ubiquitous B-Tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979.

[6] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, June 1993.

[7] Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew Tolmach. A principled approach to operating system construction in Haskell. In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, ICFP '05, pages 116–128, New York, NY, USA, 2005. ACM.

[8] Based on a paper by Simon Peyton Jones Haskell Project. Haskell: Introduction. `http://www.haskell.org/haskellwiki/Polymorphism`, 2012.

[9] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, September 1989.

[10] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Proceedings of the 2nd conference on Domain-specific languages*, DSL '99, pages 109–122, New York, NY, USA, 1999. ACM.

[11] C. Okasaki. *Purely Functional Data Structures*. PhD

thesis, Carnegie Mellon University, 1996. http://www.cs.cmu.edu/~rwh/theses/okasaki.pdf.

[12] Haskell Project Based on a paper by Simon Peyton Jones. Haskell: Introduction. http://www.haskell.org/haskellwiki/Introduction, 2012.

[13] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008.

[14] Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. http://www.haskell.org/definition/.

[15] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, 3 edition, 2003.

[16] Black Duck Software. MySQL project summary. http://www.ohloh.net/p/mysql, 2012.

[17] Johan Tibell. High-performance Haskell. In *Commercial Users of Functional Programming*. CUFP, October 2010.

[18] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, UK, 1995. Springer-Verlag.

[19] Daniel Waterworth. Siege. https://github.com/DanielWaterworth/siege, 2012.