An On-Disk Datalog Engine

by

Krishna Sivakumar

February 1st, 2026

A Thesis submitted to the
faculty of the Graduate School of
the University at Buffalo, The State University of New York
in partial fulfillment of the requirements for the
degree of

Master of Science
Department of Computer Science and Engineering

Acknowledgments

I would like to express my sincere gratitude to my advisor, Dr. Oliver Kennedy,
For his invaluable guidance and support throughout my research endeavor.
I am thankful to the Department of Computer Science and Engineering
at the University at Buffalo
for giving me this opportunity to pursue this work.
I would like to thank my parents, my sister and my friends, who have supported me no matter the circumstance.

Abstract

In-memory Datalog engines such as Soufflé achieve high performance but incur significant memory usage and startup costs when scaling to large analyses. This work presents extensions to Draupnir, a Datalog engine designed to mitigate these limitations by trading some execution speed for persistence and startup latency. Draupnir implements Agralog, a dialect of Datalog based on aggregate-annotated relations, and follows a compilation pipeline that translates high-level queries into typed intermediate representations, logical and physical execution plans, and ultimately bytecode executed by a virtual machine. A persistent storage backend for Draupnir is implemented using RocksDB, a write-optimized key–value store that supports differential updates through merge operators. This backend integrates with Draupnir's cursor and writer abstractions and enables incremental aggregation without requiring read-before-write semantics. To simplify storage backends and reduce serialization overhead, a new addressing scheme based on encoded byte-level tuples is also introduced and adopted throughout the system. The design and implementation of the RocksDB storage layer and the revised cursor addressing mechanism are described in detail. While disk-backed execution introduces additional overhead compared to in-memory systems, this work demonstrates a practical foundation for persistent Datalog evaluation and identifies several directions for future optimization.

Contents

1. Introduction

Multiple interfaces have been developed in the past few decades to query databases. SQL (Structured Query Language) has been incredibly popular, followed by NoSQL, which encompasses key-value interfaces, graph queries and so on.
Datalog is an alternate way to query databases, with roots in Prolog. Queries are defined recursively, wherein the database is first seeded with a few base "facts" and then a query is either derived bottom-up or top-down. Bottom-up queries are derived from the facts by applying a set of rules over and over until a fixed-point is reached where no more new facts can be added to the database, while top-down queries are derived by searching from a particular "goal" or solution.
Souffle is a popular Datalog engine that specializes in running large-scale program analyses. The key contribution of this system is its ability to generate indices on the fly for different sets of data and compile the datalog specification to a C++ specification with access to OpenMP for parallelization. This results in very efficient execution. Souffle also has a number of Datalog extensions to help specify program analysis queries more easily. [[1]]
The main drawback of this system however, is the large amount of RAM usage and the time taken to start up large-scale program analyses.
Aiming to be an improvement on Souffle, This work presents an On-Disk Datalog Engine called Draupnir which implements a dialect of Datalog called Agralog. Agralog or AGRA [[2], [3]] (Aggregate Relational Algebra) operates primarily on Aggregate-Annotated Relations which are functions that map from a tuple of values to an annotation, typically something like a count.
For implementing the storage engine, we chose to use RocksDB, a write-optimized key-value store and fits our use case, as we can update values differentially i.e. update a field in an object or increment a number without having to perform a read. [[4]]
This will slow down query evaluation due to disk reads and writes being slower than memory. However we seek to minimize this with caching and other techniques which are out of the scope of this thesis.

2. Draupnir

2.1. A Brief Introduction to Datalog

Datalog is a way to recursively specify and answer queries, by specifying a set of base "facts" and extending the set of facts with "rules". Datalog has applications in Program Analysis, Declarative Networking, Data Integration, and so on. The set of rules comprise the Intensional Database, and the set of initial facts comprise the Extensional Database. Datalog can be evaluated in bottom-up or top-down fashions. The primary concerns when building out the set of derived rules is to avoid generating the entire search space, and to evaluate rules quickly.[[5]]

A example to showcase Datalog is to answer an all-pairs graph reachability query. Consider a line graph is provided in the form of an adjacency list (pairs of sources to destinations). The query to answer now is a list of all pairs of nodes that can reach each other.

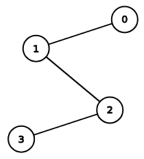

Figure 1: Line Graph

To model this in Datalog, we first need to define the facts of the query, and then define the rules that will produce the answer.

The initial facts are the edges (0, 1), (1, 2), (2, 3).

These facts reside in the relation link(x, y), where x and y are the source and destination edges.

On top of this, two rules are defined:

```
r1: reachable (x, y) := link (x, y)
r2: reachable (x, y) := link (x, z), reachable (z, y)
```

Firstly, any edge in the relation “link” is now in the relation “reachable”. Secondly, if a node z is reachable from x and there is a link from z to y, then the node y is reachable from x.

This rule is now recursively applied over and over until reachable does not change.

The final result is the edge set [(0, 1), (1, 2), (2, 3), (0, 2), (1, 3), (0, 3)].

2.2. Storage and Cursor Interfaces

The storage interface is defined in the following way:

```rust
/// A representation of a dataset
pub trait Storage: std::fmt::Debug {
  /// Retrieve a cursor to the provided dataset with the specified configuration
  fn cursor(&self, configuration: &CursorConfiguration) -> DraupnirResult<Box<dyn
Cursor>>;

  /// Retrieve a writer to the provided dataset for the specified version
  fn writer(&self, version: usize) -> DraupnirResult<Box<dyn Writer>>;

  /// Retrieve the set of supported clusterings; For every clustering listed,
  /// this storage must support index_cursor
  fn supported_clusterings(&self) -> Vec<Vec<usize>>;

  /// Retrieve the set of supported sort orders; For every sort order listed,
  /// this storage must support index_cursor
  fn supported_sorts(&self) -> Vec<Vec<usize>>;
}
```

Listing 1: Specification for the Storage Backend Interface

The storage interface requires a storage backend to be able to produce cursors for reading and writers for writing, along with “capabilities” or guarantees. Storages capabilities are as such:

1. Clustered: The keys are guaranteed to be clustered i.e. keys of a similar rank are next to each other
2. Coalesced: There are no two records with the same key.
3. Resettable: A reading cursor produced from this storage can be reset back to the beginning.

The cursor & writer interfaces are defined as follows:

```rust
pub trait Cursor: std::fmt::Debug {
  /// Seeks to the provided location or a position preceding the successor
  /// of the addressed index.  Returns `true` if the seeked address is
  /// exactly available.
  fn seek(&mut self, address: &EncodedTuple) -> DraupnirResult<bool>;

  /// Seeks to the very first record
  fn seek_to_head(&mut self) -> DraupnirResult<()>;

  /// Seeks to just past the last record
  fn seek_to_tail(&mut self) -> DraupnirResult<()>;

  /// Reads up to one record.  Returns None if no records remain to read
  fn read_one(&mut self) -> DraupnirResult<ReadStatus<Record>>;

  /// Reads a record batch.  Returns None if no records remain to read
  fn read_batch(&mut self, max_size: usize) -> DraupnirResult<ReadStatus<RecordBatch>>;
}

pub trait Writer {
  /// Update the provided record
  fn update_one(&self, record: &Record) -> DraupnirResult<WriteStatus>;

  /// Update a batch of records
  fn update_batch(&self, records: RecordBatch) -> DraupnirResult<WriteStatus>;

  /// Signal that the dataset is 'complete' and that no further updates will be
  /// performed.  You can expect that this will only ever be used for temporary
  /// relations instantiated during execution.  You can also usually expect this
  /// to be a no-op, since most Storage implementations don't care if further
  /// data will be written or not.  However, there are at least two instances
  /// when it's useful for the Storage engine to know when no further writes will
  /// be issued
  /// 1. The relation is a stream, in which case seal() signals EOF
  /// 2. The relation is specialized for assymetric write -> read workloads, and
  ///    seal() signals that the relation should transition from its write- to its
  ///    read-optimized layout.
  fn seal(&self) -> DraupnirResult<()>;
}
```

Listing 2: Specification for the Cursor and Writer Interfaces

The cursor interface should be able to seek to any part of the storage and read one or more records at a time.

A writer can write or update one or more records at a time, and seal off an instance of a storage backend.

Initially cursors used an Address type for seeks, which was just an array of bytes (`Vec<u8>`). There were a few issues with this:

1. Unclear functionality

Addresses served a dual purpose; For any data structure without ordering support, Addresses were a direct index into the contents. This means that they carried no information, in the case of a unbounded buffer. But in the context of a data structure like the IndexedBuffer, Addresses were an encoded array of values.

2. Lack of Metadata

Addresses lacked metadata, making it difficult to understand what they contained. Extracting the bytes of specific fields from an Address required complete type information. But in Draupnir, cursors were sometimes only aware of a subset of keys within the storage. So to extract any information from an Address struct, all parties would need to have all the type information. This was unweildy.

3. Scattered Functionality

There were 5 different methods to deal with Addresses (index_to_adddress, address_to_index, address_to_key, key_to_address & key_ref_to_address), and they were not unified under the Address type. This made working with it pretty hard.

These issues motivated a rewrite of the addressing API.

3. Rewriting Cursor Addressing

The final state of the addressing scheme uses instances of EncodedTuple objects, defined as follows:

```rust
/// The encoded tuple format for the tuple `(C1, C2, ..., CN)` is:
///
/// +------------+----------------+-----+--------------------+----+----+----+----+
/// | |C1|:usize | |C1|+|C2|:usize | ... | |C1|+...+|CN|:usize | C1 | C2 | .. | CN |
/// +------------+----------------+-----+--------------------+----+----+----+----+
///
/// * An array of usize-sized positions within the tuple representing the *end* of the
///   field. (Field |C1| starts at position 0)
/// * A consecutive sequence of actual field data.
///
#[derive(PartialEq, Eq, Clone, Debug)]
pub struct EncodedTuple(Vec<u8>);

impl EncodedTuple {
  /// Returns the offset from which the data of the `i`th field starts, given that there
are `field_count` fields.
  fn field_start(&self, i: usize, field_count: usize) -> usize {
    let base = field_count * size_of::<usize>();
    if i == 0 {
      base
    } else {
      let offset = (i - 1) * size_of::<usize>();
      base
        + usize::from_le_bytes(
          self.0[offset..(offset + size_of::<usize>())]
            .try_into()
            .expect("usize-sized slice is somehow not usize-sized"),
        )
    }
  }

  /// Gets the data of the `i`th field in the tuple, given that there are `field_count`
fields.
  fn field_bytes(&self, i: usize, field_count: usize) -> &[u8] {
    &self.0[self.field_start(i, field_count)..self.field_start(i + 1, field_count)]
  }

  pub fn decode(&self, types: &[Type]) -> Vec<Const> {
    types
      .iter()
      .enumerate()
      .map(|(i, field_type)| self.decode_field(i, field_type, types.len()))
      .collect()
  }
```

Listing 3: The new addressing scheme

```rust
  pub fn decode_field(&self, field: usize, data_type: &Type, field_count: usize) ->
Const {
```

This is essentially a byte buffer containing a serialized version of a record, along with field metadata. Although the current implementation takes a lot of space with each offset pointer being a `usize` (which is 32 or 64 bits depending on the platform [[6]]), This offers multiple benefits:

1. Data can be directly referenced from this byte buffer; There is no need for type information to access specific fields within the buffer.
2. There is scope now to optimize for space with the encoding. Something like delta encoding [[7]] can now be used to reduce the amount of the data stored in the record.
3. Most importantly, this does not serve the dual functionality of being an opaque pointer and also containing semantic meaning in different situations.

4. Implementing the RocksDB Storage Layer

Draupnir currently features 3 in-memory datastructures: a record array, a ring buffer and an BTree index. None of these are persistent, so a new storage backend had to be added. AARs as discussed above, are essentially key-value stores from a key tuple to an annotation. So it makes sense to look for key-value databases.

RocksDB stood out in particular here due to a number of useful features. First, it was write-optimized due to being based on an LSM tree and supported differential updates. Differential updates allows us to perform updates to records without having to perform a read. [[4]]

This however requires us to specify a merge operation. This merge operation is invoked during query time, and merges the initial value of a record along with all of the differential updates written to the database.

The following is the implementation of the merge function:

```
let mergefn = move |_: &[u8], existing_val: Option<&[u8]>, operands: &MergeOperands| ->
Option<Vec<u8>> {
  let mut lhs = match existing_val {
    Some(bytes) => encoded_tuple_to_record(
      &EncodedTuple(bytes.to_owned()),
      &[kt.clone(), vec![vt.clone()]].concat(),
    ),
    None => (vec![], zero.clone()),
  };

  // reduce loop
  for operand in operands {
    let rhs = encoded_tuple_to_record(
      &EncodedTuple(operand.to_owned()),
      &[kt.clone(), vec![vt.clone()]].concat(),
    );
    lhs = (
      rhs.0,
      eval_expr(
        &merge
          .apply(vec![rhs.1.into(), lhs.1.into()])
          .expect("Operands do not satisfy the type constraints."),
        &HashMap::new(),
        None,
      )
      .expect("Operands do not satisfy the type constraints."),
    );
  }

  Some(EncodedTuple::from([lhs.0.as_slice(), &[lhs.1.clone()]].concat().as_slice()).0)
};
```

Listing 4: Definition of the Merge function

The “existing value” is the initial value of the key. Differential updates are then merged into the existing value using a user-specified merge function.

A few more benefits to picking RocksDB here is that it offers snapshot reads, and that it stores data as raw bytes. This provides an opportunity to optimize for space if need be.

The full implementation of the storage is available in Appendix A.

5. Future Work

There is still a lot of room for optimizations in the system, with this being an non-exhaustive list:

1. The compiled bytecode operators currently read or write only a single row at a time. This leads to a lot of overhead in terms of function calls. Records can be fetched in batches, and stored in larger registers with the virtual machine.
2. The compilation to bytecode itself introduces a lot of overhead, which can be eliminated by defining operators natively.
3. In the same vein as number one, collections of records can be processed more quickly with the use of SIMD instructions for comparisons and other operations.

6. Conclusion

This work presented extensions to Draupnir, an on-disk Datalog engine, motivated by the high memory usage and startup costs of in-memory systems such as Souffle. A storage backend was implemented on top of RocksDB, and the benefits and drawbacks were discussed. An alternate addressing scheme was also proposed and implemented.

Bibliography

[1] H. Jordan, B. Scholz, and P. Subotić, "Soufflé: On synthesis of program analyzers," in *International Conference on Computer Aided Verification*, 2016, pp. 422–430.

[2] C. Koch, "Incremental query evaluation in a ring of databases," in *Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2010, pp. 87–98.

[3] C. Koch *et al.*, "DBToaster: higher-order delta processing for dynamic, frequently fresh views," *The VLDB Journal*, vol. 23, no. 2, pp. 253–278, 2014.

[4] S. Dong, A. Kryczka, Y. Jin, and M. Stumm, "Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications," *ACM Transactions on Storage (TOS)*, vol. 17, no. 4, pp. 1–32, 2021.

[5] T. J. Green, S. S. Huang, B. T. Loo, W. Zhou, and others, "Datalog and recursive query processing," *Foundations and Trends® in Databases*, vol. 5, no. 2, pp. 105–195, 2013.

[6] S. Klabnik and C. Nichols, *The Rust programming language*. No Starch Press, 2023.

[7] H. Tan, W. Xia, X. Zou, C. Deng, Q. Liao, and Z. Gu, "The design of fast delta encoding for delta compression based storage systems," *ACM Transactions on Storage*, vol. 20, no. 4, pp. 1–30, 2024.

7. Appendix A: Rocks Storage Implementation

```rust
use std::{cell::RefCell, collections::HashMap, sync::Arc};

use crate::{
  backend::Config,
  error::{DraupnirError, DraupnirResult},
  interpreter::{
    data_structure::{CursorConfiguration, EncodedTuple, ReadStatus, Writer},
    eval::eval_expr,
  },
  ir::{
    common::{Const, RelationSchemaDetail, Type},
    expr::LambdaFunction,
    pipeline::SinkConstraints,
  },
};
use rocksdb::{Direction, IteratorMode, MergeOperands, Options};

use super::{Cursor, Record, RecordBatch, Storage, WriteStatus};

#[derive(Debug)]
pub struct Rocks {
  value_type: Type,
  key_type: Vec<Type>,
  connection: Arc<rocksdb::DB>,
}

#[derive(Debug, PartialEq, Eq, PartialOrd, Ord, Clone)]
struct RocksKey(Vec<u8>);

impl RocksKey {
  /// Extracts the data part alone from the encoded tuple.
  fn from_encoded(encoded: &EncodedTuple, field_count: usize) -> Self {
    let mut key = Vec::<u8>::new();
    for field in 0..field_count {
      key.extend_from_slice(encoded.field_bytes(field, field_count));
    }
    Self(key)
  }
}
```

```rust
fn encoded_tuple_to_record(tuple: &EncodedTuple, types: &[Type]) -> Record {
  let flat_record = tuple.decode(types);
  let (value, key) = flat_record
    .split_last()
    .expect("Expected encoded tuple to have at least 2 fields.");
  (key.to_vec(), value.clone())
}

pub struct RocksCursor {
  value_type: Type,
  key_type: Vec<Type>,
  _current_key: Vec<Const>,
  _database: Arc<rocksdb::DB>,
  snapshot: rocksdb::Snapshot<'static>,
  iterator: Option<rocksdb::DBIterator<'static>>,
}

impl Cursor for RocksCursor {
  fn seek(&mut self, address: &EncodedTuple) -> DraupnirResult<bool> {
    let address = RocksKey::from_encoded(address, self.key_type.len());
    // println!("SEEKING TO {:?}", address);

    self.iterator = Some(
      self
        .snapshot
        .iterator(IteratorMode::From(&address.0, Direction::Forward)),
    );

    Ok(true)
  }

  fn seek_to_head(&mut self) -> DraupnirResult<()> {
    self.iterator = Some(self.snapshot.iterator(IteratorMode::Start));
    Ok(())
  }

  fn seek_to_tail(&mut self) -> DraupnirResult<()> {
    self.iterator = Some(self.snapshot.iterator(IteratorMode::End));
    Ok(())
  }
```

```
fn read_one(&mut self) -> DraupnirResult<ReadStatus<Record>> {
  let mut ret = ReadStatus::AtEnd;

  self.iterator.as_mut().map(|iter| {
    iter.next().map(|result| {
      result.map(|(_key, value)| {
        let record = encoded_tuple_to_record(
          &EncodedTuple(value.into_vec()),
          &[self.key_type.clone(), vec![self.value_type.clone()]].concat(),
        );

        ret = ReadStatus::Success(record);
      })
    })
  });

  Ok(ret)
}

fn read_batch(&mut self, max_size: usize) -> DraupnirResult<ReadStatus<RecordBatch>> {
  let mut ret = None;

  for _ in 0..max_size {
    if let Ok(status) = self.read_one() {
      match status {
        ReadStatus::Success(rec) => match ret {
          None => {
            let rb: RecordBatch = vec![rec].as_slice().into();
            ret = Some(rb);
          }
          Some(ref mut rb) => {
            rb.push(&rec);
          }
        },
        ReadStatus::Blocked => {}
        ReadStatus::AtEnd => break,
      }
    }
  }

  Ok(ret.map(ReadStatus::Success).unwrap_or(ReadStatus::AtEnd))
}
```

```
}

impl std::fmt::Debug for RocksCursor {
  fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
    write!(f, "Rocks Cursor")
  }
}

impl RocksCursor {
  pub fn new(rocks_backend: Arc<rocksdb::DB>, value_type: Type, key_type: Vec<Type>) ->
Self {
    let mut cursor = RocksCursor {
      _current_key: Vec::new(),
      value_type,
      key_type,
      _database: rocks_backend.clone(),

      // RocksCursor is supposed to independent in memory
      #[allow(clippy::missing_transmute_annotations)]
      snapshot: unsafe { std::mem::transmute(rocks_backend.snapshot()) },
      iterator: None,
    };

    cursor.iterator = Some(cursor.snapshot.iterator(IteratorMode::Start));

    cursor
  }
}

pub struct RocksWriter {
  connection: Arc<rocksdb::DB>,
  key_type: Vec<Type>,
}

impl Writer for RocksWriter {
  fn update_batch(
    &self,
    records: crate::interpreter::record::RecordBatch,
  ) -> DraupnirResult<WriteStatus> {
    for i in 0..records.len() {
      self.update_one(&records.get(i))?;
    }
    Ok(WriteStatus::Success)
  }

  fn update_one(&self, record: &crate::interpreter::record::Record) ->
DraupnirResult<WriteStatus> {
    let put_address = RocksKey::from_encoded(&EncodedTuple::from(&record.0),
self.key_type.len());
    let value = EncodedTuple::from(
      [record.0.as_slice(), &[record.1.clone()]]
        .concat()
        .as_slice(),
    );

    match self.connection.merge(put_address.0, value.0) {
      Ok(_) => Ok(WriteStatus::Success),
      Err(e) => Err(DraupnirError::Message(e.into_string())),
    }
  }

  fn seal(&self) -> DraupnirResult<()> {
    Ok(())
  }
}

impl Storage for Rocks {
  fn cursor(&self, configuration: &CursorConfiguration) -> DraupnirResult<Box<dyn
super::Cursor>> {
    let CursorConfiguration { key: _, version } = configuration;
    if version.is_versioned() {
      return Err(DraupnirError::Message(
        "RocksDB does not yet support versioned reads".into(),
      ));
    }
    Ok(Box::new(RocksCursor::new(
      self.connection.clone(),
      self.value_type.clone(),
      self.key_type.clone(),
    )))
  }

  fn writer(&self, version: usize) -> DraupnirResult<Box<dyn Writer>> {
    if version != 0 {
      return Err(DraupnirError::Message(
        "RocksDB does not yet support versioned writes".into(),
      ));
    }
    Ok(Box::new(RocksWriter {
      connection: self.connection.clone(),
      key_type: self.key_type.clone(),
    }))
  }

  fn supported_clusterings(&self) -> Vec<Vec<usize>> {
    todo!()
  }

  fn supported_sorts(&self) -> Vec<Vec<usize>> {
    todo!()
  }
}

impl Rocks {
  pub fn new(
    value_type: Type,
    key_type: Vec<Type>,
    path: String,
    merge: Arc<LambdaFunction>,
    zero: Const,
  ) -> DraupnirResult<Self> {
    let mut options = Options::default();

    let kt = key_type.clone();
    let vt = value_type.clone();

    let mergefn =
      move |_: &[u8], existing_val: Option<&[u8]>, operands: &MergeOperands| ->
Option<Vec<u8>> {
        let mut lhs = match existing_val {
          Some(bytes) => encoded_tuple_to_record(
            &EncodedTuple(bytes.to_owned()),
            &[kt.clone(), vec![vt.clone()]].concat(),
          ),
          None => (vec![], zero.clone()),
        };
        // reduce loop
        for operand in operands {
          let rhs = encoded_tuple_to_record(
            &EncodedTuple(operand.to_owned()),
            &[kt.clone(), vec![vt.clone()]].concat(),
          );
          lhs = (
            rhs.0,
            eval_expr(
              &merge
                .apply(vec![rhs.1.into(), lhs.1.into()])
                .expect("Operands do not satisfy the type constraints."),
              &HashMap::new(),
              None,
            )
            .expect("Operands do not satisfy the type constraints."),
          );
        }

        Some(EncodedTuple::from([lhs.0.as_slice(),
&[lhs.1.clone()]].concat().as_slice()).0)
      };

    options.set_merge_operator_associative("mergeop", mergefn);
    options.create_if_missing(true);

    rocksdb::DB::open(&options, path)
      .map(|connection| Rocks {
        value_type,
        key_type,
        connection: Arc::new(connection),
      })
      .map_err(|err| DraupnirError::Message(err.into_string()))
  }

  pub fn instantiate(
    schema: &RelationSchemaDetail,
    _: &SinkConstraints,
    config: &Config,
  ) -> DraupnirResult<Arc<RefCell<dyn Storage>>> {
    match Self::new(
      schema.annotation_type(),
      schema.attribute_types(),
      config
        .rocksdb_path
        .clone()
        .expect("expected a path for rocksdb from the config"),
      schema.annotation.plus.clone(),
      schema.annotation.zero.clone(),
    ) {
      Ok(rocks_instance) => Ok(Arc::new(RefCell::new(rocks_instance))),
      Err(err) => Err(err),
    }
  }

  pub fn snapshot(&'_ self) -> rocksdb::Snapshot<'_> {
    self.connection.snapshot()
  }
}

#[cfg(test)]
mod tests {
  // use std::iter::Successors;

  use super::*;
  use crate::interpreter::record::{Record, RecordBatch};
  use crate::ir::common::{AnnotationId, Const};
  use crate::library::{
    Library,
    stdlib::{ANNOTATION_EXISTS, ANNOTATION_SUM_INT},
  };
  use once_cell::sync::Lazy;

  /// Tests the Rocks module against a particular annotation and data.
  /// The db_url provided must be unique within tests as rocksdb is single-threaded.
  fn test_annotation(
    annotation_id: &Lazy<AnnotationId>,
    key_type: Vec<Type>,
    insert_data: Vec<Record>,
    expected_data: Vec<Record>,
    db_url: String,
  ) -> Result<(), Box<dyn std::error::Error>> {
    let library = Library::default();
```

```rust
let exists = library.get_annotation_err(annotation_id)?;

let rocks = Rocks::new(
  exists.base_type(),
  key_type,
  db_url,
  exists.plus.clone(),
  exists.zero.clone(),
)
.expect("Could not open the database");

let full_batch: RecordBatch = insert_data.as_slice().into();

// clean up db before executing test
for datum in &expected_data {
  rocks
    .connection
    .delete(EncodedTuple::from(&datum.0).0)
    .expect("deletion in rocksdb should go through.");
}

rocks.writer(0)?.update_batch(full_batch)?;

let mut cursor = rocks
  .cursor(&CursorConfiguration::default())
  .expect("Could not get a cursor.");

for datum in &expected_data {
  cursor
    .seek(&EncodedTuple::from(&datum.0))
    .expect("Couldn't seek to key.");
  let fetched_value = cursor.read_one();

  println!("{:?} {:?}", &datum.0, &fetched_value);
  // cannot compare Results as they do not implement PartialEq
  match fetched_value {
    Ok(ReadStatus::Success(tuple)) => {
      assert_eq!(datum.1, tuple.1)
    }
    _ => {
      panic!("fetched_value does not match {}", datum.1)
    }
    }
  }

  Ok(())
}

#[test]
fn test_rocks_annotate_exists() -> Result<(), Box<dyn std::error::Error>> {
  let data: Vec<Record> = vec![
    (vec![Const::Int(1), Const::Int(1)], Const::Bool(false)),
    (vec![Const::Int(2), Const::Int(1)], Const::Bool(true)),
    (vec![Const::Int(2), Const::Int(2)], Const::Bool(false)),
    (vec![Const::Int(3), Const::Int(2)], Const::Bool(false)),
    (vec![Const::Int(3), Const::Int(2)], Const::Bool(true)),
    (vec![Const::Int(1), Const::Int(1)], Const::Bool(true)),
    (vec![Const::Int(1), Const::Int(1)], Const::Bool(false)),
  ];

  let expected: Vec<Record> = vec![
    (vec![Const::Int(1), Const::Int(1)], Const::Bool(true)),
    (vec![Const::Int(2), Const::Int(1)], Const::Bool(true)),
    (vec![Const::Int(2), Const::Int(2)], Const::Bool(false)),
    (vec![Const::Int(3), Const::Int(2)], Const::Bool(true)),
  ];

  test_annotation(
    &ANNOTATION_EXISTS,
    vec![Type::Int, Type::Int],
    data,
    expected,
    "test/test0.rocks".to_owned(),
  )
}

#[test]
fn test_rocks_annotation_sum() -> Result<(), Box<dyn std::error::Error>> {
  let data: Vec<Record> = vec![
    (vec![Const::Int(5), Const::Int(1)], Const::Int(5)),
    (vec![Const::Int(6), Const::Int(1)], Const::Int(2)),
    (vec![Const::Int(7), Const::Int(2)], Const::Int(1)),
    (vec![Const::Int(8), Const::Int(2)], Const::Int(3)),
    (vec![Const::Int(8), Const::Int(2)], Const::Int(11)),
      (vec![Const::Int(5), Const::Int(1)], Const::Int(17)),
      (vec![Const::Int(5), Const::Int(1)], Const::Int(13)),
    ];

    let expected: Vec<Record> = vec![
      (vec![Const::Int(5), Const::Int(1)], Const::Int(35)),
      (vec![Const::Int(6), Const::Int(1)], Const::Int(2)),
      (vec![Const::Int(7), Const::Int(2)], Const::Int(1)),
      (vec![Const::Int(8), Const::Int(2)], Const::Int(14)),
    ];

    test_annotation(
      &ANNOTATION_SUM_INT,
      vec![Type::Int, Type::Int],
      data,
      expected,
      "test/test1.rocks".to_owned(),
    )
  }
}
```