Daniel Karaj

Transport Data Web Server in MirageOS with support for SQL queries

Computer Science Tripos - Part II

Christ's College

2016

Proforma

Name:	**Daniel Karaj**
College:	**Christ's College**
Project Title:	**Transport Data Web Server in MirageOS with support for SQL queries**
Examination:	**Computer Science Tripos - Part II, 2016**
Word Count:	**11905**
Project Originator:	Dr Richard Mortier
Supervisor:	Dr Richard Mortier

Original Aims of the Project

The primary aim of the project was to create a server, using MirageOS, to store a stream of bus route and location messages and provide access through the form of SQL queries. Observations made could then be used to evaluate the suitability of using MirageOS to the create the servers needed for Internet of Things devices, as well as the techniques that should be used when parsing information and storing data for good SQL query performance.

Work Completed

The aims of the project were generally achieved. A MirageOS web server was created that is able to accept web requests using the Cohttp OCaml library and store data using Irmin. A parser was written to convert transport data protobufs in their binary wire format into corresponding OCaml objects and then convert them to a CSV format for storage. A lexer and parser were created for an SQL subset (focusing on selection with conditionals) using Menhir and OCamllex allowing for queries to be evaluated against the data.

Special Difficulties

None.

Declaration

I, Daniel Karaj of Christ's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

Chapter 1

Introduction

1.1 Motivation

The advent of the Internet of Things will result in a vast increase in Internet communications, primarily focusing on the client server model where many devices report back to a single server. To provide good support for these devices, there is a large focus on improving server architecture to allow for easily scalable, more secure web applications.

Traditional web stacks carry large overheads, with a complete operating system required to run a single application. Many features of the operating system, such as support for external devices and separate user and kernel modes, are not required for the web server and instead a result of the many purposes that an operating system is used for. It has also become increasingly rare for a single web server to be run on a single machine – most now use a virtual machine run on a hypervisor. This introduces further redundancy where support for different underlying architectures is no longer required, resulting in even more code that serves no purpose for the application. This extra code slows the boot time of the server and increases the attack surface, leading to an increase in the number of exploitable security vulnerabilities. [1]

A unikernel is a web server that can be run directly on a hypervisor, without the need for an intermediate operating system. The functions generally performed by the operating system are instead provided through a set of libraries, with MirageOS being a particular set of these libraries. MirageOS allows OCaml web applications to be developed that can run directly on a hypervisor.

For the use of unikernels to become more widespread, web servers developed using them must be able to compete with those using a more traditional stack. The increased demand for servers in relations to the Internet of Things leads to an opportunity to display the advantages of unikernels in an important area that is set to expand.

1.2 Project Aims

The purpose of this project was to write a unikernel that fulfils many of the needs of an Internet of Things server, and evaluate the suitability of MirageOS unikernels as replacements for the traditional stack architecture. Transport data was used as an example of the kind of information that can be expected from Internet of Things devices, in particular time series data from multiple remote, mobile, distributed sensors reporting back to a centralised server to allow for holistic evaluation and processing on a larger scale. The server needs to efficiently process this data and make it available for further querying. SQL is used to provide familiarity and so that further evaluation can be performed to compare performance with a traditional web stack using a database.

1.3 Related Work

The functions that the server must perform are not uncommon and there are many well documented approaches using more traditional server architectures. The difficulty here arises from the use of MirageOS where prior work is more limited.

Examples of web applications developed using MirageOS have often been blogs – the MirageOS website one example of this [2] – with applications outside of this more limited. An example of a more complex application was developed by Thomas Leonard in the department, using MirageOS to create an action tracker that uses the JavaScript compilation offered by OCaml to run on the client side. [3]

Protobufs are a commonly used data format and such there exists a variety of implementations for different languages. In particular, Google offers language generators for Java, C/C++ and Python although the code generated by these is reliant on general libraries and difficult to read. [4] The Piqi project offers similar capabilities for OCaml, aiming to offer an alternative for the protobuf specification format that also supports other transmission formats such as JSON and XML. [5]

There is no database built for OCaml, generally language bindings for another database such as Sqlite or MySQL are used, however as MirageOS projects must be pure OCaml it was necessary to use an alternative storage format. As a widely used language, there are a variety of examples of implementing SQL query implementation over non-tradition data stores. Google's Tenzing project [6] implements SQL querying of data on the MapReduce Framework and Apache Kylin over Hadoop Hive tables. [7] Whilst these operate on structures vastly different to the ones used in this project they offer an insight into the approach that should be taken and the importance of SQL query support.

Chapter 2

Preparation

2.1 Requirements Analysis

As with any large project, it was important to break the system into pieces that could be evaluated and tested independently and then combined to produce the final product, understanding the impacts of MirageOS across the complete server implementation.

The web server:

- Must accept incoming requests containing transport information and make this information available for parsing.
- Must accept incoming requests to query the data and return a response given by a function in relation to the request.

The protobuf parser:

- Must parse valid GTFS-realtime protobuf messages into appropriate data structures to allow them to be to be stored and later read

The storage solution:

- Must accept parsed protobuf messages and store them
- Must accept requests for specific messages and return the appropriate information

The SQL implementation:

- Must parse some subset of SQL queries so that they can be performed
- Must read the necessary information for the query to be performed from the data store and manipulate it such that it can be returned as a response

2.2 Software Engineering Principles

The modular approach to the project identified in the requirements analysis provided a good structure for work to be performed. Each module could be developed and tested independently and so the stalling of one area need not lead to progress being entirely halted. To ensure that a complete end-to-end system was implemented, it was important to work on each section iteratively, preparing modules offering minimal functionality before further development led to a more feature complete project.

This, in coordination with Git version control and regular backups ensured that if any changes were made that reduced the functionality of the system or introduced bugs they could be easily reverted or compared to previous versions of the software.

Individual modules for components meant that unit testing could be used throughout the project, with sample protobuf parsing results compared to the Google implementations and sample query results compared to an Sqlite database.

2.3 Research

Whilst having some familiarity with more traditional web stacks, I had no experience using MirageOS or any other library operating system. To learn the capabilities of MirageOS and gain familiarity with the creation of unikernels, I first had to learn OCaml, a language I had no prior experience with. Once familiar with the language I was able to look into the other features that were needed for the project, to begin, the format of the bus data.

Whilst ordinarily one would generate the code needed to parse the protocol buffers, since I would be writing the deserialization code myself, it was necessary to research the encoding of the message in much more depth.

To access the data it was decided that SQL support provided the most useful interface for both evaluation and users. The process between receiving a query and responding needed to be broken down; in particular parsing the query into a format usable by the code constructing the response. This would involve lexing and then parsing the query string, a task that OCaml is well suited to due to its support for lexer and parser generators.

The final task was to focus on the type of queries that should be supported, heavily influencing how the transport data was stored. Concentrating on selection seemed to be the most valuable avenue due to the potential use of the data and the evaluation that could be performed.

2.3.1 OCaml

MirageOS unikernels are written in OCaml, a statically typed high-level language that supports both functional and imperative styles of programming. As a high level language,

OCaml offers type and memory safety guarantees important for unikernels where only a single address space exists, whilst still offering comparable performance to low level languages such as C. Whilst these features and demand for them have resulted in the increasing popularity of the language, it is still lacks the range of learning resources and examples that one can find with other more commonly used languages.

A useful resource for learning OCaml is the book *Real World OCaml* [8], an overview of the various features of the language and libraries and how they can be used to perform common programming tasks. As I had no previous experience with the language and much of my programming experience outside of the course has been using dynamic languages, working through the book was a key first step. Modules, heavily used by MirageOS to organise code and offer interchangeability of components; and records, a form of structured data in OCaml, with strict fields defined as a type; proved to be particularly important topics. With records being used to read the protobufs into whilst allowing for any necessary manipulation of the transport data before storage to be performed.

2.3.2 MirageOS

MirageOS is a set of libraries used to fulfil the functions generally provided by a traditional Operating System. These libraries can be included or excluded at will, allowing for much smaller web applications with numerous associated advantages, such as rapid start up and improved security. The faster boot time can be used to improve scalability and power consumption by starting instances of the server as a direct response to a web request [9] and the increased security is provided by the reduced attack surface of the web server – a smaller amount of code leads to fewer vulnerabilities in the system. Privacy of information stored on the server is of especial importance for the personal data that Internet of Things devices are likely to record and a secure architecture, free from vulnerabilities, is an important part of ensuring this.

The MirageOS website offers brief introductions to many of the libraries commonly required for web applications and example unikernels for common use cases, primarily focusing on the minimum configuration needed to serve static web content. [10] Whilst offering a good introduction to writing my own, this still left a lot of work to do looking into the more specific libraries that would be necessary for the project, such as Cohttp and Irmin.

Cohttp [11] and Irmin [12] are libraries that are used to receive and construct HTTP requests and provide a version controlled persistent data stores. Neither library is specific to MirageOS and as such documentation tended to be more general. With the development of MirageOS still moving quickly it could be difficult to find information that was up to date and provided examples relevant to what I was trying to achieve. A useful resource was the GitHub account of Thomas Leonard, a researcher in the department, where in addition to the aforementioned action list application there exist basic unikernels using Cohttp and Irmin which provided useful accompaniment to the library documentation. [13]

To prevent blocking functions from halting the operation of the server, MirageOS makes heavy use of the Light Weight Threads library [14] to construct user threads and bind the results of asynchronous operations to a subsequent function. Asynchronous operations return values of the type `'a Lwt.t` and as such cannot be directly passed to a function, instead a monad, the bind operator `>>=` is used to chain operations together. It does this by generating a thread that performs the subsequent operation as soon as `'a` becomes available. For example to delay the evaluation of the function `f` by 2 seconds, one can write `Time.sleep 2.0 >>= fun () -> f`. The Lwt library is used throughout MirageOS and Irmin and as such, understanding the syntax and the functions performed as a result is of great importance to writing unikernel code.

2.3.3 Transport Data

Data arriving to the server would be in the form of GTFS-realtime protocol buffers. Protocol buffers (protobufs) are a format used for efficient and compact data transportation, where both the client and server are fully aware of the details of the protocol and the messages that will be sent. As a result, field names and other similar information can be omitted, leading to more efficient data transfer than other formats such as JSON or XML. GTFS-realtime is a protobuf specification tailored specifically for live updates of a transportation network. [15]

For many languages commonly used in tradition web server development, generic libraries are provided for the processing of protobufs – a protocol specification file is used to generate parsing code. Attempts have been made to provide similar libraries for OCaml, in particular, the Piqi project, however for this project, it was decided that the code for parsing the GTFS-realtime protobufs should be written directly. To do this it was necessary to explore the documentation of the both the general protocol buffer format, where generic protobuf encoding is described, and that of the GTFS-realtime specification.

```
syntax = "proto2";
package transit_realtime;

message FeedHeader {
required string gtfs_realtime_version = 1;
enum Incrementality {
FULL_DATASET = 0;
}
optional Incrementality incrementality = 2 [default = FULL_DATASET];
optional uint64 timestamp = 3;
}
```

Code 2.1: A subsection of the GTFS-realtime specification file

Protobufs are formatted as a series of bytes – the different fields that the protobuf can contain first identified by a byte with the value of the field immediately following. In a

protobuf format file, the fields are given numbers and these are translated, in combination with the type of field, into the byte identifiers.

The first step in parsing the protobufs was to find the relationship between the byte identifiers and message fields. In the protobuf format file each field is numbered and has a type. To allow for unknown fields to be skipped, the types are grouped by how their length is formatted and then appended to the end of field number to form the byte identifier. For example, the field `vehicle` has field number `4` and stores an embedded message which is a length delimited-value with wire type `2`. Storing the field number as 5 bits (`00100`) and the wire type as `2` (`010`) results in `00100010` being used as the byte identifier for the `vehicle` field.

Type	Meaning	Used For
0	Varint	int32, int64, uint32, uint64, sint32, sint64, bool, enum
1	64-bit	fixed64, sfixed64, double
2	Length-delimited	string, bytes, embedded messages, packed repeated fields
3	Start	group groups (deprecated)
4	End	group groups (deprecated)
5	32-bit	fixed32, sfixed32, float

Table 2.1: Relationship between different fields and their associated wire type used to construct their byte key

The GTFS-realtime format supports a wide array of use cases and as a result has a wide array of different fields, many of which are never used in the bus data that was being processed for this project. To ease the implementation and to avoid masses of redundant code, I constructed a simplified protobuf specification containing only the fields used in the project. I have included this as well as a breakdown of the bytes in binary wire format and how they correspond to the different fields in the appendices.

Depending on the type of the field the value is stored in different ways. The different types of values can be grouped into categories organised by how their length is stored (the same types used to generate their byte identifier): fixed length values, such as floats; length-delimited values, such as strings; and varints, a variable length encoding of integers.

Length-Delimited Values

For length-delimited values, the field identifier is followed by the length of the value and then the bytes containing the value itself. For example, the string "`1.0`" is converted to the bytes `3`, `49`, `46` and `48`, where `3` is the length of the string (encoded as a varint) and `49`, `46` and `48` are the UTF8 character codes for `1`, `.`, and `0`.

Fixed Length Values

Fixed length values have no requirement to record the length and as such, simply the value as bytes. Floats for example follow the IEEE floating point number format in little-endian byte order.

Varints

Integers are used to store both values of the various integer types and the length of length-delimited values. Since the length of strings will often be short, using 4 bytes to store the length is wasteful and as such an alternative used in protobufs in the form of varints – variable length integers. This encoding means that the number of bytes used to store the integer varies with its magnitude. Bytes are read one at a time, with each byte containing 7 bits used to store the integer and a single bit to indicate whether the following byte should be included in the calculation of the value of the integer. For example, the decimal value `123456` is ordinarily encoded as the binary value `1 11100010 01000000`. To convert this into the binary wire format, it is split into groups of 7 bits, `111 1000100 1000000`, and a `1` prepended to all but first byte, which is instead padded with zeros to form `00000111 11000100 11000000`. When sent down the wire in little-endian byte order, they are received as `11000000 11000100 00000111` and the integer can read by reversing the manipulations made.

2.3.4 SQL Support

With the purpose of the server being to make the data from the protobufs easily accessible, to allow for further processing and analysis, a suitable API was needed to access the data. Before implementation, it was necessary to evaluate whether SQL querying was correct approach to take or an if an alternative, such as the API made available by the London transport network [16], would be better suited.

Whilst using an alternative API would have been easier to implement, since the aim of the project was to allow comparisons to be made to more traditional web application where the data would most commonly be stored in a database, it made sense to focus on SQL. It offers easier comparisons with traditional stacks and a standardised interface to analyse performance against. In addition to the evaluation benefits, the common use of SQL means that the capabilities of the system can be easily understood and offer familiarity to those used to working in more traditional languages.

MirageOS does not offer a database implementation and as such, a simpler key-value data store needed to be used. To translate SQL queries into results, the queries needed to be parsed and appropriate processing be performed on the stored files.

A simple solution for query parsing is to simply split the query by whitespace and try to identify the different parts using regular expressions against each element. This approach,

whilst initially easy, is flawed. Whenever any changes are to be made to the SQL subset supported it is difficult to understand the parts of the code that would need to be changed; in addition to fixing the many edge cases that may not be handled correctly in this naive approach. Whilst involving more work to begin with, using a parser generator to convert the SQL strings in a more useful data structure results in much more extensible and understandable code, less prone to unpredictable error cases.

Lexing and Parsing Queries

OCaml has generators for both lexing and parsing and so is well suited to this approach. OCamllex is used to generate lexers, taking an `.mll` file and generating functions that take a buffer and convert it into a string of tokens. The specification file includes a list of rules used to identify the various tokens present in the string in addition to helper functions for string and error handling, making it easy to identify and offer detailed feedback on syntax errors to the user.

Menhir [17], an enhanced version of Yacc, acts as a parser, taking the list of tokens generated by OCamllex and arranging it into a data structure that can be easily read. It reads a file similar to that of OCamllex with production rules that specify how sequences of tokens should be handled. Menhir offers the advantage that the resultant data structure can be set to match a defined type, providing the guarantee that the structure of the query is correct.

Query Type and Backus Nauer Form

To better define the type of queries that the system would accept it was necessary to define an appropriate type to the queries to be pass into. To ensure that the structure was maintainable and extensible, it was important that it followed conventional naming and structural formats. To aid with this I found a SQL specification in Backus Nauer Form [18] which offered a good base for both the query type to be used and the parsing rules, something that could not have been taken advantage of if the naïve approach had been taken.

SQL Subset Selection

Deciding on the subset of the SQL language that would be supported was an important step as it would affect the complexity of the lexing and parsing as well as the data structures that the data would be stored in.

With only two tables and a single one to one mapping between them I decided that it would be best to focus on selections. With the data in the form of protobufs, it was difficult to aggregate information across the different files. By focusing on selections the server would allow for easy sub-selection of information across all of the protobufs

that were delivered, greatly improving the ease with which further evaluation could be performed.

Following are a selection of queries to be supported by the server, and the implications that they result in.

```
SELECT * FROM entities
```

The simplest query is the bare minimum that can be used to test that the system is working, protobufs must be written to files, the query validated and parsed, the data read from files and returned in an appropriate format.

```
SELECT "hello world" FROM entities
```

The next step is to select literals for each row, at first, simply echoing user input.

```
SELECT timestamp FROM entities
```

Following this, values should be selected from the records, where manipulation of the data stored and handling fields with null values is required.

```
SELECT timestamp, trip_id, route_id, vehicle_id, vehicle_id FROM entities
```

Extending this, it become necessary to support the selection of multiple fields including the same fields multiple times.

Beyond this functionality, there is a large variety in the expressions that can be selected from the table including the results of expressions across multiple column and aggregate functions that operate across multiple rows of the table. Since the focus of the project was to make the data more easily available and improving the range of expressions supported would not have a large impact on how the data was stored, it was decided that it would be better to instead focus on the filtering of rows.

Rows can be filtered using a variety of methods, ranging from simple equality of a single field to chains to more complex expressions. As with selection, limiting the range of expressions reduces the amount of work without affecting the data storage options that can be evaluated.

```
SELECT * FROM entities WHERE vehicle_label = ...
```

Firstly a conditional on a single column should be supported.

```
SELECT * FROM entities WHERE longitude > ... AND longitude < ...
```

Followed by the expansion to any number of comparison predicates.

Performance gains here will arise from the ability to filter rows before they are read from storage and as such different options will be useful when performing an evaluation.

Chapter 3

Implementation

Work on the project was broken down into the groups identified in the preparation section, with the different components placed into OCaml modules to keep the code organised, aid in the evaluation of each part of system and allow for unit testing to be performed.

The web server uses different ports to receive transport data protobufs and query requests before passing them on to the different parts of the system. Protobufs are passed to the `Protobuf_parser` module, where the information is read into a GTFS-realtime message type that is used to store the data in an Irmin key-value data store after converting it to a CSV format. Query requests are handled in a similar manner, their received strings passed on to functions for lexing and parsing generated using OCamllex and Menhir. The resultant SQL type is then used to generate a list of files to be read, before the final response is constructed by filtering and manipulating the rows read from these files. This result, or alternatively any error that has been encountered, is returned in the request response.

3.1 Web Server

The web server acts as the beginning and end point of all operations of the system. It uses Cohttp to aid with the handling of requests and Irmin to perform the required reading and writing of data. MirageOS web servers are based on a configuration file and an OCaml module, `Main`, containing a `start` function and as a result fulfilling the unikernel signature definition.

```
module Main (Console:CONSOLE) (Stack:STACKV4) = struct
```

The server code all lies within this OCaml module, which is parameterized with a console and stack, allowing for different implementations of these components to be used depending on the environment, as specified in the configuration file. The configuration file is additionally used to specify the libraries that are required by the web server and, in

combination with the mirage command line tool, is used to generate the files necessary to compile the web server to either a UNIX application or to be run directly as a unikernel.

3.1.1 Accepting requests

Requests are handled with the aid of the Cohttp library. It makes the header and body of the HTTP request easily available and is used when constructing and returning a response. Callback functions are specified for requests containing protobufs and requests querying the data and, using the Cohttp library to parse the body of the request, the string containing either the protobuf or SQL query can be passed on to functions used for storage and query evaluation.

3.1.2 Responding to requests

When performing IO functions MirageOS uses light weight threads to prevent blocking from stalling the web server. As such the values returned to the callback functions must be captured using the bind syntax before they can be used. These values are then used to construct responses with appropriate status codes and provide logging information for the server.

General `try/with` blocks cannot be used in conjunction with these threads and so the Lwt alternative is used to handle any errors. For SQL queries some errors can be conveyed back to the user as they are expecting a response and syntax errors must be corrected, but for protobufs this data is logged locally so that it can be understood when the system is running.

3.1.3 Setting up Irmin

Irmin is used as the data store for the server. First its inclusion is specified in the `config.ml` file before being specifically configured for the server. Due to the library's flexibility, configuration primarily revolves around the use of functors with modules being provided for things ranging from the format in which keys should occur to the hashing function used for commits. Finally when the server is started a repository is created and a task generated, to be used when accessing the store.

```
module Mirage_git_memory =
    Irmin_mirage.Irmin_git.Memory(Context)(Git.Inflate.None)
module Store =
    Mirage_git_memory(Irmin.Contents.String)(Irmin.Ref.String)(Irmin.Hash.SHA1)
let config = Irmin_mem.config ()

let start console stack = Store.Repo.create config >>=
Store.master (fun s -> Irmin.Task.create ~date:0L ~owner:"S" s) >>= fun t
```

3.1.4 Storing and Accessing Data in Irmin

Irmin offers the ability to read and write data to a directory structure, in addition to a variety of features related to version control, merging and Git integration. Each action is performed with an associated task and uses the Lwt library to perform non blocking IO, with the value of the operation later accessible through binding. The specific ways in which Irmin is used for the web server are covered in the protobuf storage and accesses sections later in this chapter.

3.2 Parsing protobufs

With the server getting the protobuf string from the request, the modules in this section cover the parsing process from binary wire format to the structure used in the rest of the system.

3.2.1 Feed Message Type Definition

This type relates to directly to the message format defined in the GTFS-realtime protocol definition, replacing messages with records containing the same fields to hold the data contained in incoming protobufs in much the same way as the classes defined by the generators for C++ and Python.

```
module Vehicle_descriptor = struct
type t =
{
id : string option;
label : string option;
}
end
...
module Feed_entity = struct
type t =
{
id : string;
vehicle : Vehicle_position.t option;
}
end
```

Code 3.1: Elements of the protobuf message type definition

One complication arises from the type inference needed for OCaml's typing system. Different types defined in the GTFS-realtime specification file share fields and as a result, the type of {`record with id`} can be inferred to be either of type `Vehicle_descriptor.t` or of type `Feed_entity.t`. If the wrong type is selected, functions will be mislabelled and

the program will not compile. To avoid these issues, modules are used to separate the record definitions in a common OCaml pattern.

3.2.2 Protobuf Format Exception

Malformed protobufs are handled using the OCaml exception system. An appropriate exception type is defined, with the inclusion of a string for further details, allowing `try/with` blocks to catch exceptions and forward failure messages onto the user.

```
exception Protobuf_Format_Exception of string

let parse_message file_string =
raise (Protobuf_Format_Exception "Invalid Protobuf")

let () =
try parse_message "" with
| Protobuf_Format_Exception msg -> print_string msg
```

3.2.3 Parsing Bytes

The first and simplest access to be made to the message information is to read a single byte from the protobuf. This is used as the base for all other functions and sets the function signature that will be used throughout the process.

The protobufs are passed to the function as a string and so to read a section of the message the index of the location of the section that we wish to parse is also needed. Using the built in String library we can read the value of the string at the correct location and return a tuple containing the result and the index of the next byte to be processed, establishing the signature for all parsing functions.

```
val parse_... : bytes -> int -> ... * int = <fun>
```

Whilst the next location is easy to calculate for a byte, for other data structures it requires the parsing of the value itself. Following a regular format for all parsing functions makes the code easier to modify and easier to understand.

3.2.4 Parsing Varints

Used to store all integer fields and the lengths of any variable fields, variant integers, varints, are the logical first step from parsing individual bytes. The technique used involves recursively reading bytes until one is found where the first bit is not set to 1. Passing this value up the stack, left shifting it by 7 bits and adding on the next byte at each step, gives the overall value due to the little-endian encoding. The index of the next byte is passed

up in a similar fashion, allowing the original call to the function to return the value of the varint and the next index.

```
let rec parse_varint message_string index =
let current, next_index = parse_byte message_string index in
if current > 127 then
let sum_rest, next_index = parse_varint message_string next_index in
(current - 128) + sum_rest lsl 7, next_index
else
current, next_index
```

3.2.5 Parsing Strings

Parsing strings first involves finding the length of the string and then reading those bytes from the message. The length of the string is encoded as a varint to save on space (as strings are generally relatively short). In the case where a different data structure was being used repeated calls to `parse_byte` may have been useful, however since the parsing functions here all operate on strings it made most sense to use a sub-string function.

```
let parse_string message_string index =
let length, string_index = parse_varint message_string index in
Bytes.sub_string message_string string_index length, string_index + length
```

3.2.6 Parsing Floats

Floats are fixed in length and so all necessary bytes can be read immediately and then shifted to the correct bit positions using an integer to contain them. This integer is then treated as bits and converted to a float with some complication as the length of a standard integer in OCaml is set by the compiler.

```
let rec parse_float message_string index =
let first, second_index = parse_byte message_string index in
let second, third_index = parse_byte message_string second_index in
let third, fourth_index = parse_byte message_string third_index in
let fourth, next_index = parse_byte message_string fourth_index in
let bytes_as_int = first + second lsl 8 + third lsl 16 + fourth lsl 24 in
Int32.float_of_bits (Int32.of_int bytes_as_int), next_index
```

3.2.7 Parsing Records

Parsing records involves two steps, first the length of the record is found in much the same way is it is for the other variable length field, the string. Then, using this value, fields

can be read and appended to the record until the end of the encoded record is reached.

The protobuf specification puts no limit on the number of times a field can occur in a message, however only the final value read is to be used in the returned record (for non-repeated fields). The function is called using an empty record as an accumulator with fields appended to it until the end of the record is reached, the {record with field} construction proving to be particularly useful here. This tail recursive approach emulates the while loop that is used in the generated imperative code reading from a stream.

```
let rec parse_header_fields message_string index limit record =
if index >= limit then
record, index
else
let field_code, field_index = parse_byte message_string index in
if field_code == 10 then
let gtfs_realtime_version, next_index = parse_string message_string
    field_index in
parse_header_fields message_string next_index limit {record with
    gtfs_realtime_version}
else if field_code == 16 then
let incrementality, next_index = parse_varint message_string field_index in
if (incrementality != 0) then raise (Protobuf_Format_Exception "Incrementality
    must be FULL_DATASET");
parse_header_fields message_string next_index limit record
...
else
raise (Protobuf_Format_Exception "Header contained unknown field")

let parse_header message_string index =
let header_length, fields_index = parse_varint message_string index in
parse_header_fields message_string (fields_index) (fields_index +
    header_length) {gtfs_realtime_version = ""; incrementality = FULL_DATASET;
    timestamp = None}
```

There are a few notable points in the above function. To allow for protobuf specifications to add new fields without breaking compatibility with clients using older versions, the protobuf format specifies that unknown fields should be skipped, with the length of the field apparent from the field code. Since this system is making assumptions about the reduced field set being used in the transport data it instead seemed prudent to alert the user of the unknown code. The system takes a similar approach with the values of enumerated values where more are specified in the GTFS-realtime specification than are used in the Cambridgeshire transport network.

3.2.8 Parsing Repeated Fields

With a repeated field if a field is encountered for a second time replacing the value in the record is not appropriate, instead any values read multiple times are prepended to a list.

```
parse_message_fields message_string next_index limit {record with entity =
    entity::record.entity}
```

3.2.9 Parsing Complete Messages

Using the techniques described in this section, parsing functions can be written for all of the values present in the protobuf, culminating for such a function for the complete message. This approach also results in code that is well extensible. The process of adding new fields simply requires a parsing function to be composed, their field code added to the parent parsing function and their inclusion in the type definition.

3.3 Handling SQL Queries

SQL queries are handled in much the same way as protobuf messages. The server makes the SQL string available, from which the query must be extracted to allow for appropriate data to be read and returned.

3.3.1 Query Type definition

The first step in SQL query parsing is to define the type that the information will be parsed into. In doing so, a regular form for access is made available, and all of the parts of the query that are needed to construct a response are easily accessible. The type also serves as a definition of the SQL queries accepted, with commented record fields being used to illustrate the restrictions of the supported subset.

```
type query_specification =
{
(* set_quantifier *)
select_list: select_list;
table_expression: table_expression;
(* offset *)
limit: int option;
}
```

Code 3.2: Query type definition

```
<query specification> ::=
SELECT [ <set quantifier> ]
<select list>
<table expression>
```

Code 3.3: Corresponding definition in BNF

The record above specifies the top most structure of the queries. At its simplest, a query consists of a `select_list` and a `table_expression`. This is based on the Backus

Nauer Form of SQL and serves to illustrate what subset of the SQL syntax is supported. The `select_list` designates the fields to be output, either all or a subsection, and the `table_expression` the tables that these are to be selected from as well as restrictions that are placed on the rows of these tables. The `set_quantifier`, whether `ALL` or only `DISTINCT` records are selected is unsupported and the `limit` (an extension taken from PostgreSQL and MySQL) is optional.

By taking the structure from the BNF it is easy to extend the format to support an increased number of SQL statements without having to rebuild the entire format, for example simply adding the field `offset: int option` allows for offset to be included in query specifications. The full SQL type definition and the corresponding BNF definition are included in the appendices.

3.3.2 Parsing Queries

Parsing the information from SQL strings to the data type is a two step process. First the string is split into lexical tokens, sections of the string that should not be broken down any further and are needed semantically, and then these tokens parsed – the list of tokens read and the `Sql.query_specification` type containing the query constructed. As a human readable format, SQL tokens are much more distinct than the bytes of the protobuf and so the lexing step of parsing is more useful and, in general, the format lends itself better to traditional machine generating parsing.

3.3.3 Lexing of query tokens

The lexing step is performed using OCamllex, an OCaml implementation of the Lex lexical analyser generator. Lex reads a specification of the lexical analyser and generates the source code to convert an arbitrary buffer into a token list. Lex and Yacc (the parser) are built to work together and as such the tokens are only defined once - in the Yacc specification file. Tokens can be simple, containing only their own value or wrap a value such as an integer or a string.

```
%token <string> STRING
%token <int> INT
%token <float> FLOAT
%token SELECT FROM WHERE LIMIT
```

The Lex file itself is compiled of three parts, a section at the top for imports and definitions (although any arbitrary code can be placed here), a rules section, and a section of code at the bottom copied directly to the generated file.

For this SQL query lexer, the Lexing library and our generated parser, containing the lexeme types, are imported in addition to the definition of a syntax exception type – used to handle the lexing of invalid strings. Lexing is a OCaml library that contains functions

used in association with Lex as well as type definitions for the the structures used in the lexing process.

```
let white = [' ']+
let string = ['a'-'z' 'A'-'Z' '0'-'9' '_']+['a'-'z' 'A'-'Z' '0'-'9' '_' '-']*
...
rule read =
parse
| white     { read lexbuf }
| "select"  { SELECT }
| string    { STRING (Lexing.lexeme lexbuf) }
...
| _         { raise (SyntaxError ("Unexpected Char: " ^ Lexing.lexeme lexbuf))
   }
```

Following the definitions, rules defining the lexing procedure are defined, in a similar format to pattern matching. Regular Expressions are mapped to functions that handle the buffer, the return values of which are used to construct the list of lexemes forwarded to the parser. Some expressions such as those for white space or unknown characters are not mapped directly to return values but instead simple progress the lexing process or raise an exception. To match the current value read from the buffer with an expression each regular expression is evaluated in turn with the first longest match being used, giving `"select"` precedence over general strings.

Finally, in the last section of the file, helper functions are defined to create more useful error messages and functions to parse strings without having to construct a `lexbuf` first.

3.3.4 Parsing of queries

The lexing process simply results in a list of tokens, to map these tokens to the data structure, a parsing step is used. Menhir is an updated version of OCamlyacc and is used to construct a query of the specified type from the token list. In addition to the features offered by OCamlyacc, Menhir allows the type of the structure parsed to be defined as well as producing more useful error messages. Otherwise the parser construction follows much the same format – a Yacc specification file defines an analytic grammar written in a notation similar to BNF which is then used to generate the parser.

```
%start <Sql.query_specification> query
%%

query: SELECT; select_list =
    select_list; table_expression =
    table_expression; limit = limit;
    END;
{ {
Sql.select_list;
Sql.table_expression;
Sql.limit;
} }

select_list:
| ASTERISK; { Sql.Asterisk }
| select_list = separated_list(COMMA,
    select_sublist); { Sql.SelectList
    select_list }
```

Code 3.4: Menhir analytic grammar rules

```
<query specification> ::=
SELECT [ <set quantifier> ]
<select list>
<table expression>

<select list> ::=
<asterisk>
| <select sublist> [ { <comma>
    <select sublist> }... ]
```

Code 3.5: Corresponding BNF definitions

The grammar is constructed of rules followed by actions. Here `start` sets the first rule to be used as `query`. For a token expression to match with the query, it must begin with the `SELECT` token and then match with the `select_list` rule and subsequently the `table_expression` rule. At the end of the rule the value to be returned is defined, for `query`, this is a record containing the appropriate fields to match the type specification. Yacc has support for some built in rules, one example is used here to parse the list of identifiers, the tokens for for separators and values given as parameters.

3.4 Protobuf Storage and Access

With the receiving and parsing of transport data and SQL queries handled by other parts of the system, this section need only deal with the storage of information and the construction of query responses.

3.4.1 Transport Data Serialisation

Parsing the transport data from protobufs into the corresponding OCaml data type is an important step to access and validate the data. However to store the data, it must be first serialised to a string. Protobufs are made to save as much bandwidth as possible whilst on the server easy and efficient data access is the priority and so a CSV format was selected. This simplistic format is well suited due to the efficiency with which the values stored can be read in addition to the easy mapping that it allows to database tables. Common issues

with CSV, such as varying line delimiting and the needed to escape certain characters are easily avoided as all data is written by the system itself.

To convert the protobufs into CSV format, functions were written to translate value types into strings. These strings could then be concatenated together to form the strings that would be written to the data store.

3.4.2 Storing Messages in Files

Databases can use a variety of storage structures to implement their tables, most commonly using B+ trees, which allow for efficient access to data when selected using the key column. The key-value store implemented by Irmin offers similar advantages. One limitation on the primary key is that it must be unique; for entities there is no unique field. With more control over the structure than a tradition database, entities can instead by grouped in files by message, using the timestamp from the header, allowing for efficient selection by timestamp to be implemented.

```
let add_protobuf t protobuf_string =
let message = Protobuf_parser.parse_message protobuf_string in
let header_list, entity_lists = Protobuf.list_of_feed_message message in
let message_timestamp = (last header_list) in
Store.update (t ("Adding header " ^ message_timestamp))
["headers"; message_timestamp] (String.concat "," header_list) >>= fun () ->
Store.update (t ("Adding entities " ^ message_timestamp))
["entities"; message_timestamp] (String.concat "\n" (List.map (String.concat
    ",") entity_lists))
```

The data in the storage mechanism is now present under the `<table_name>/<header time_stamp>` hierarchy, with entities grouped together in files.

3.4.3 Selecting Rows from the Data Store

The first step in the SQL implementation is the `SELECT * FROM <table_name>` query. With operations being the same for any table, and `entities` having more depth than `headers` it will be the focus of the rest of this section.

To select all rows of a table, all files containing must be read and then concatenated together. This is fairly simple set of operations and involves getting the keys of the files, reading the contents and appending the resultant strings together to form the output.

```
Store.list t [query.Sql.table_expression.Sql.from_clause] >>= fun key_list ->
Lwt_list.map_p (Store.read_exn t key) key_list >>= fun file_string_list ->
Lwt.return (String.concat "\n" record_string_list)
```

The above code has been simplified to show the most basic query evaluation that can be performed. A list of keys is read from Irmin and then mapped over to read their contents. Since reading a file is an asynchronous task, the `Lwt_list.map_p`, light weight threads `map` alternative is used. This binds the results of each list element to form the type `a' list Lwt.t` instead of `a' Lwt.t list`. Reading of the files can be performed in parallel, so the `_p` variant is used, generating an independent user thread for each element of the list.

3.4.4 Selecting Fields from Rows

The next step in SQL query support is to construct the rows specified in the query input. The SQL data type supports fields in the form of *integers*, *floats* and *strings*, the last of which either mapping directly to a string or, more commonly, specifying the table column that the value should be taken from.

```
let get_field_value_for_value_expression query value_list = function
| Sql.NumericValueExpression x ->
(match x with
| Sql.Float x -> FloatValue x
| Sql.Int x -> IntValue x)
| Sql.StringValueExpression x ->
if String.get x 0 = "\""
then StringValue (String.sub x 1 ((String.length x) - 2))
else
try
List.nth value_list (index_of x (table_fields
    query.Sql.table_expression.Sql.from_clause))
with
| Not_found -> StringValue x
```

This function takes a `ValueExpression` from the SQL query and translates it to a value that can be used for row construction or comparison. Floating point and integer numbers are simply taken directly, whilst strings are looked up in the table fields and either taken from the list of record values, or treated literally.

Each row in the output must be constructed individually in a process that involves taking the file string read for each key and in turn mapping the row generation function onto each record in the file. The previous function is used to get the values for each field specified in the SQL query which are then concatenated together to form the row to be returned.

The first step in the process is to get the list of values from the record string and convert them to the correct type. The `select_list` is then matched against an `Asterisk` or a true list of fields. `Asterisk` cases allow the field_values to simply be converted to strings

and then concatenated, otherwise the value for each field in the selection is found, before returning the row in the same way.

The examples below illustrate the range of selection queries supported at this point, allowing any combination of fields to be used, in addition to support for fixed value columns.

```
SELECT * FROM entities
SELECT id FROM entities
SELECT id, timestamp FROM entities
SELECT id, id, id FROM entities
SELECT 1, 2, 3, "four", five FROM entities
```

3.4.5 Filtering Rows in the Selection

With functions for getting individual field values, query support could be extended to the filtering of results, each row checked against the rules described in the where clause. If the the row passes all tests, then it can be returned as in the previous subsection, otherwise the row should not appear in the output. By wrapping the return values in lists, a failure can return an empty list and the complete row list flattened to remove any skipped rows.

```
let perform_query_on_record_string query record_string =
...
if test_search_conditions query value_list then
match query.Sql.select_list with
| Sql.Asterisk ->
[String.concat "," (List.map string_of_field_value value_list)]
...
else []
```

The `test_search_conditions` function takes both the query and the list of values contained within a row and returns a boolean value indicating whether or not the value_list conforms to the boolean tests described in the query.

For the SQL subset supported, the where clause is built of a list of boolean terms, each of which must hold for the row to be accepted. The `List.for_all` function is used to test the value list against each term and the result returned to flag the row for inclusion or not.

```
let test_cp_field_values = function
| NullValue, _, _ -> false
| _, _, NullValue -> false
| v1, Sql.EqualsOperator, v2 -> v1 = v2
| v1, Sql.NotEqualsOperator, v2 -> v1 <> v2
| IntValue v1, Sql.LessThanOperator, IntValue v2 -> v1 < v2
| FloatValue v1, Sql.LessThanOperator, FloatValue v2 -> v1 < v2
```

```
| IntValue v1, Sql.GreaterThanOperator, IntValue v2 -> v1 > v2
...
| _ -> raise (QueryException "Invalid where clause")

let test_comparsion_predicate query value_list = function
| row_value_constructor_1, comp_op, row_value_constructor_2 ->
let v1 = get_field_value_for_rvc query value_list row_value_constructor_1 in
let v2 = get_field_value_for_rvc query value_list row_value_constructor_2 in
test_cp_field_values (v1, comp_op, v2)

let test_boolean_term query value_list = function
| negated, boolean_test ->
(match boolean_test with
| Sql.Predicate predicate ->
(match predicate with
| Sql.ComparisonPredicate comparision_predicate ->
negated <>
(test_comparsion_predicate query value_list comparision_predicate)))
```

To test the value list against a boolean term, first the term is passed through a series of pattern matches, not needed with the current SQL subset, but allowing for the code to be more easily expanded in the future, until the comparison predicate is reached. This is paired with a "negated" value which is XORed with the result of the comparison predicate testing.

To test the comparison predicate, first the field values are found – getting any fields from the value lists, and then compared using the operator.

By using field types, OCaml's pattern matching can be used to neatly implement the comparisons that can be made, as well as limit the set of valid comparisons and alerting the user when a comparison between different data types has been made.

The result of this function is a boolean value, which is passed back up the stack to mark whether or not the row should be included in the output.

The extensions from the functions above greatly increase the variety of SQL queries that are supported, any number of comparison predicates can be included in the `WHERE` clause, limiting the query result to as large or small a range as desired. A few useful examples are outlined below.

```
SELECT * FROM ENTITIES WHERE vehicle_label = "SCBD-36937"
SELECT * FROM ENTITIES WHERE longitude > -0.557 AND longitude < -0.506
SELECT * FROM ENTITIES WHERE longitude > -0.534 AND latitude > 52.391
SELECT * FROM ENTITIES WHERE 1 > 2
```

3.4.6 Efficient Filtering by Primary Key

At this stage focus in the project could be shifted onto performance. A useful subset of SQL was supported and for the system to be useful and to be evaluated against traditional databases, more efficient data access was important.

Primarily, the limitation on the speed of the system was in the fact that, to perform any query, every single file storing data needed to be read. Reducing the number of files to be accessed would offer a good chance to improve querying speed and as a result more accurately compare the storage format used to more traditional systems.

One feature that could be taken advantage of was that entities were being stored by the timestamp of their header. Whilst the timestamps of the individual entities varies from the value of the header timestamp, it can be guaranteed that it will not exceed this value, and in testing it was found that entity timestamps would not differ from their containing message timestamp by more than 1000 seconds.

As a result of these observations, it was possible to filter the key list of files to be read by using any comparison predicates present in the where clause.

The first step in filtering the list of keys is to find any predicates in the where clause where timestamps are compared to integer values, as these could be evaluated before the file is read. The comparison predicates are matched against useful forms and used to return a list of triples containing whether the predicate is negated, what the operator is and the value that the time stamp field should be compared against. This list can then be used to filter the key list, as for any entity contained with the file to be returned, it's file name must conform to all of these predicates.

```
let rec _filter_key_list key_list = function
| (false, Sql.EqualsOperator, int_val) :: tl
| (true, Sql.NotEqualsOperator, int_val) :: tl ->
_filter_key_list (List.filter (fun k -> (int_of_string (last k)) >= int_val &&
    (int_of_string (last k)) < (int_val + 1000)) key_list) tl
| (false, Sql.LessThanOperator, int_val) :: tl
| (true, Sql.GreaterThanOrEqualsOperator, int_val) :: tl ->
_filter_key_list (list_before (fun k -> (int_of_string (last k)) >= (int_val +
    1000)) key_list) tl
...
| _ -> key_list
```

With the list of relevant clauses, the key list can then be filtered by each comparison with each predicate in turn. The recursive function above matches a triple from the list with the restriction that should be placed on the key list. The key list is filtered before being passed forward in a recursive call with operators expanded to include any entities that do not directly match with the header time stamp.

If a query does not contain any predicates concerning timestamps no filtering can be performed, and instead the original key list returned.

```
let add_trip_ids t message_timestamp entity_lists =
let key = (String.sub message_timestamp 0 7) in
Store.read (t ("Reading trip ids " ^ key)) ["trip_ids"; key] >>= fun
    existing_trip_ids ->
let existing_trip_ids = (map_default (fun s -> s ^ ",") "" existing_trip_ids)
    in
Store.update (t ("Adding trip ids " ^ message_timestamp))
["trip_ids"; key] (String.concat "," (existing_trip_ids :: (List.map (fun e ->
    List.nth e 1) entity_lists)))
```

3.4.7 Improving Performance for Non-Primary Keys

Whilst the functions implemented in the previous sub-section increase the performance of queries restricting by timestamp, every file must still be read if the where clause is filtering by some other field. In finding a solution to this, the `trip_id` field was focused on, as specific values of the field where generally restricted to only a few hours of protobufs and the ability to trace the progress of a single trip provides useful functionality.

The first step in the process involves writing some additional files to the data store, unlike the timestamps, trip ids could only be read from the files themselves and so needed to become more easily available. Attempts to group files together by fields other than timestamp proved to be impractical, as any incoming file could result in a read and write for every single entity contained, conflating in storage times order of magnitudes slower than organising by timestamp. Instead, the technique chosen involves grouping the incoming protobufs and storing a list of trip ids contained within at least one of them.

The function above appends the trip ids contained within a message to a list that includes all trip ids contained within protobufs that share the most significant bits of their timestamps. The presence of a trip id in this file indicates that of the files beginning with the same set of most significant bits, at least one of them will contain an entity with that trip id.

One issue with the above function is the lack of support for concurrency, if another message comes in and the file is written to between reading and writing, the data will be overwritten, possibly resulting in the loss of trip ids. Irmin append only data stores do not include the ability to set custom keys and whilst provisions have been made for a `compare_and_set` function that can test whether or not a file has been changed, it is currently unsupported.

```
let rec where_clause_trip_id_values where_clause =
let open Sql in
match where_clause with
| (false, Predicate (ComparisonPredicate (ValueExpression
    (StringValueExpression string_val_1), EqualsOperator, ValueExpression
    (StringValueExpression string_val_2)))) :: tl
```

```
| (true, Predicate (ComparisonPredicate (ValueExpression
    (StringValueExpression string_val_1), NotEqualsOperator, ValueExpression
    (StringValueExpression string_val_2)))) :: tl ->
let value = (if string_val_1 = "trip_id" then string_val_2 else if
    string_val_2 = "trip_id" then string_val_1 else "trip_id") in
if List.mem value (table_fields "entities") then (where_clause_trip_id_values
    tl) else (strip value) :: (where_clause_trip_id_values tl)
| _ -> []
```

In a manner similar to the one used for timestamps, the above function returns the trip ids that the field value must be equal to as specified by the where clause. Due to the large number of trip ids and the unlikelihood that all trip ids contained within a file range should be excluded (selecting by timestamp is a much more sensible approach here) only equality is checked.

```
let rec trip_id_filter t key_list = function
| [trip_id] -> Store.list t ["trip_ids"] >>= fun trip_id_key_list ->
(Lwt_list.map_p
(fun key -> Store.read_exn t key >>= fun trip_id_list_string ->
Lwt.return (if (List.mem trip_id (string_split trip_id_list_string ',')) then
    [last key] else []))
trip_id_key_list) >>= fun filtered_ti_key_list -> Lwt.return (List.filter (fun
    key -> (List.mem (String.sub (last key) 0 7) (List.flatten
    filtered_ti_key_list))) key_list)
| [_::_] -> Lwt.return []
| _ -> Lwt.return key_list
```

With the list of trip ids that the field value must be equal to, the trip ids contained within each file range can be read and the values compared. In the case where the field must be equal to multiple values no valid rows exist. In the function the single trip id is selected and then its membership checked within the list of trip ids read from the trip id files. This process returns a list of key ranges that contain the trip id by taking the most significant bits of each key value and filtering the list of files to be read.

The functions in this subsection have illustrated the technique for just the trip id field. The same improvements can however be made to any field, with the performance improvements compared against the additional cost in storage and processing time when adding new protobufs.

Chapter 4

Evaluation

Evaluation of the project could be broken down in much the same way as the components that make up the web server. Whilst the complete web server allows for overall evaluation of the project, assessing each module in turn helps to understand the advantages and disadvantages that the unikernel design had over the traditional stack.

Various papers exist detailing the advantages of any unikernel system over a traditional stack. [19] [20] With the successful completion of the project, it is shown that these advantages can also be exploited for the kind of server that will be needed for Internet of Things type devices – in particular the small size of the compiled unikernel signifies associated performance and security benefits.

Beyond this, to show that unikernels make for a good replacement to a traditional stack it is important to show that similar performance can be achieved. For web servers used for Internet of Things devices, there are the two points that are particularly important: processing the data sent to the server and responding to requests the data. In this case, the first involves parsing the protobufs and the second reading the appropriate information from the data store.

4.1 Protobuf Parsing Performance

To ensure that the correct data was being read from the protobufs, a sample of the transport data protobufs in addition to various edge cases were read using a generated parser in Python and compared with the values read by the OCaml code. When parsing 64-bit unsigned integers using the built in OCaml type, integers with the first bit set are read as negative numbers. As the original value can still be found and timestamps will take some time to reach the most significant bit, it was decided that it was an acceptable limitation.

To evaluate the performance of the protobuf parsing code, three testing programs were generated, one in OCaml, using the protobuf parsing module from the project and two using the Google provided protobuf parser generators, one in C++ and one in Python.

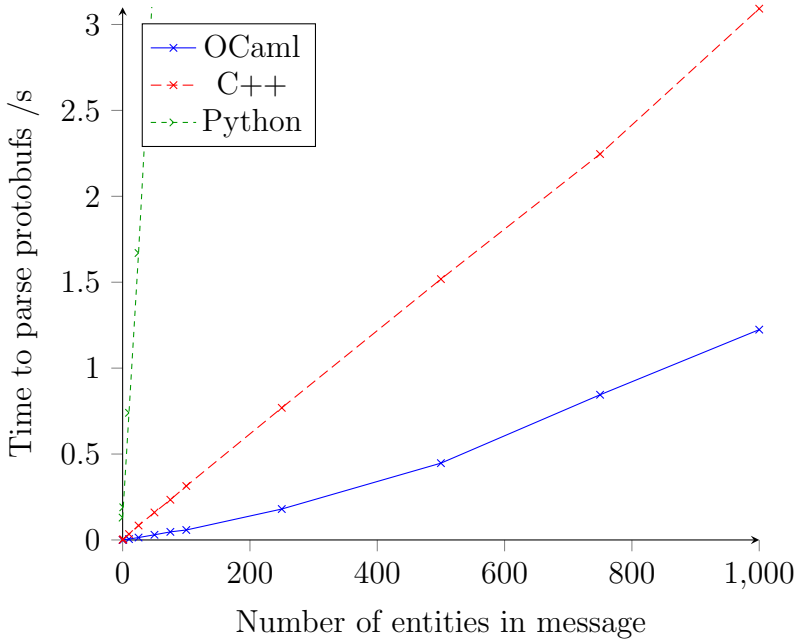
Each program was written to read a protobuf data file into a string and then parse the message contained within. The execution of the programs could then be measured using the `time` terminal tool with the highest performing program taking the least time to execute.

To reduce the influence of other factors, such as reading the protobuf file, from affecting the run time of the program and to allow small time measurements to be made, each program loaded the protobuf string into a variable once and then, using a loop, parsed the string 1000 times.

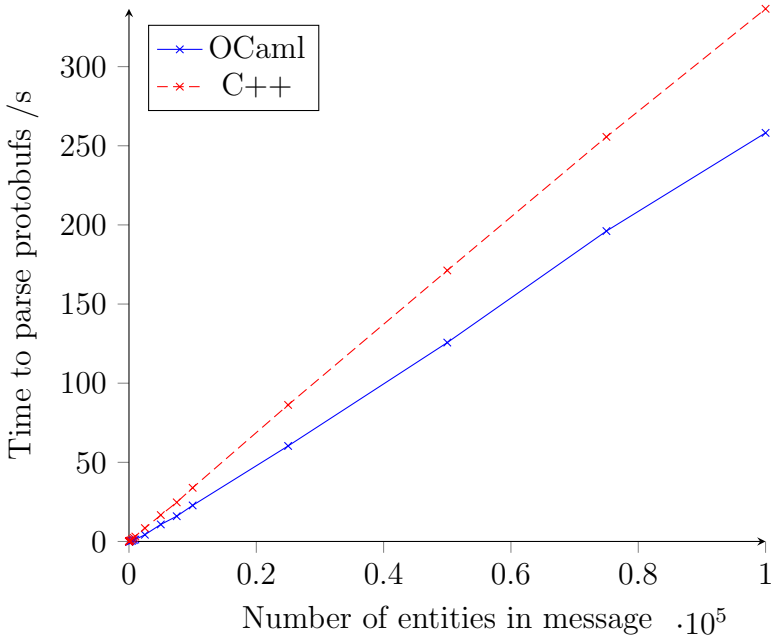
Running the program on a protobuf containing a transport update from the middle of the afternoon, chosen as regular, demanding value, gave an idea of the performance differences between the programs. The OCaml and C++ options proved to be an order of magnitude faster than the Python program, a not altogether unexpected result due to Python's interpreted nature against OCaml and C++'s static compilation. What was more encouraging was the performance advantage of the OCaml parser over the C++ program, showing that if performance of protobuf parsing is a priority, writing an OCaml parser in fact delivers better results that the C++ generator, the highest performing of all of the Google supported languages.

OCaml	**C++**	**Python**
0.53	2.02	45.85

To test the performance of the programs when parsing messages of different lengths, a Python program was created to write messages containing a specified number of entities which could then be read by the different programs and timed in the same way as before.



With the range of entities was chosen (encapsulating the messages received from the transportation network) OCaml performed better then C++ in all cases. However, to recommend the OCaml implementation style for general use, the growth must be linear. To assess further, testing with increased numbers of entities was performed.



With the growth of the OCaml implementation becoming increasingly linear it can be recommended for situations where large amounts of data is received and even at the 64MB default limit of the C++ implementation (a limit that is not present in the OCaml implementation), the OCaml parsing function still has an advantage.

With this slightly surprising performance result, it was important into the C code generated to identify the source of the performance difference. The C code uses a library to parse the messages, parameterised by the specification of the protocol. This library is more robust with error recovery than the OCaml code and is written to parse from buffers. As a result, the `ParseString` function converts the string into an array buffer before it can be read, an operation not needed for the OCaml code, designed to read the body strings returned by the Cohttp library.

4.2 SQL Performance

In situations where a lot of analysis is being performed it is important that SQL performance of the unikernel is comparable to that of the tradition stack.

The performance can be broken down into two sections, the time and space necessary to add the data and the time taken to query it. Python and Sqlite [21], a portable database written in C, were used to build programs to allow comparisons to be made to a tradition stack.

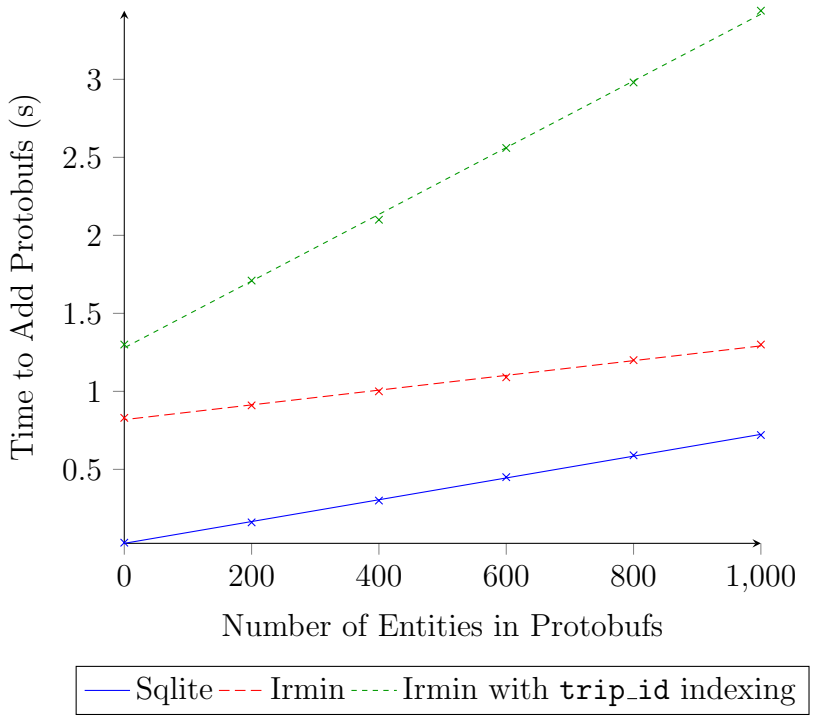
4.2.1 Insertion Performance

First to test was the time taken to add items to the data store. To do this, programs were written to add a protobuf message to the data store 120 times – the number of protobuf

messages that arrive in an hour. This process could be timed using the `time` tool; the repeated insertion aiding with the accuracy of the measurements made and the whole process repeated to reduce the effects of anomalies.

Due to the large disparity in the parsing time for OCaml and Python found in the previous section, a protobuf from the middle of the day was added to the code of the programs that could be added directly to the data store, giving a clearer indication of the actual storage time. The number of entities in this protobuf was varied to simulate different times of the day and how performance varied the number of buses on the roads.

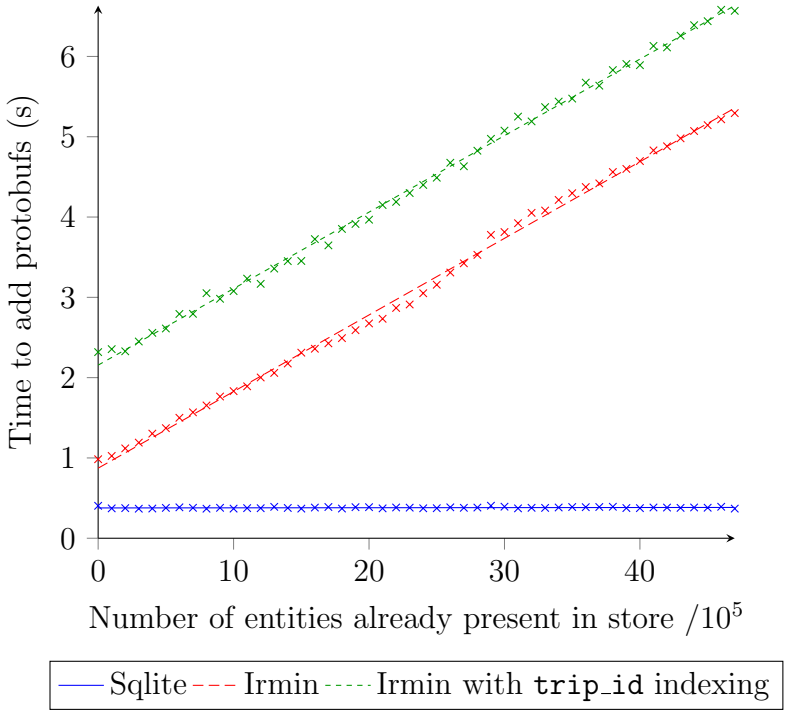
One important factor to consider is that the `trip_id` storage is based on timestamp, as a result, first attempts of evaluation which did not vary the data at all reflected very badly on it, as all protobufs fell into the same bucket. To prevent this, the timestamp in the header was incremented to simulate the data that would be received by the server.



As can be seen in the graph, even without any entities being added to the data store, due to the files being written, the Irmin implementation has a rather large overhead.

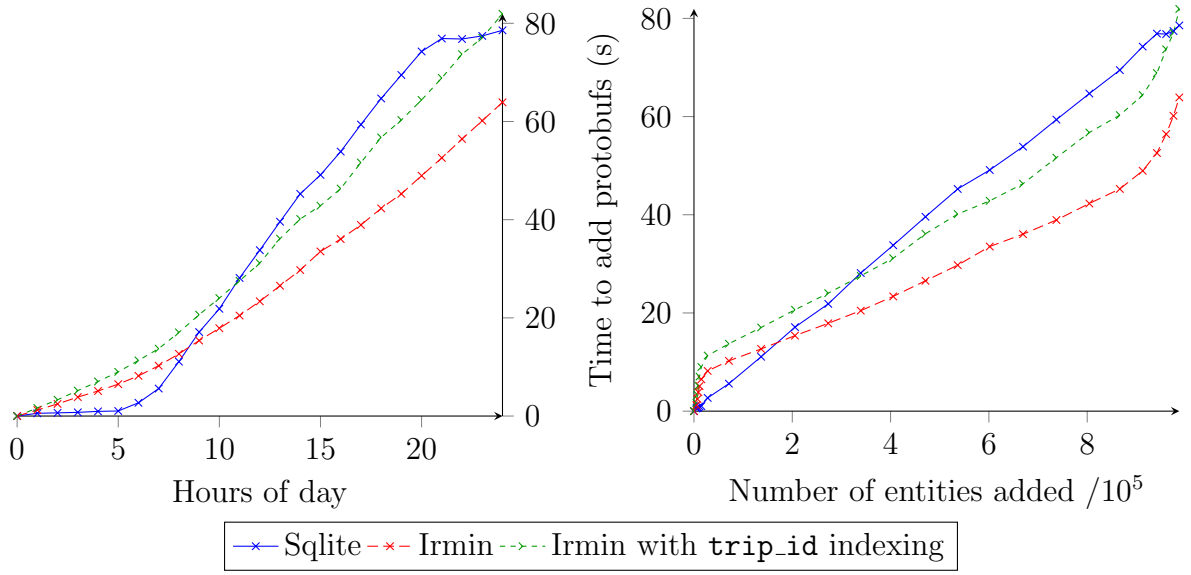
All options grow linearly, with the slow growth of the basic OCaml option making it suited to very large protobufs. For the protobufs received by this system, entities commonly number at around 500, Sqlite is the clear superior.

Between each run of the programs, the store was cleared to provide the same conditions for all tests. To test the effects of adding protobufs to a data store already containing information, the programs were extended such that an argument could be passed to specify the initial timestamp and then timed as 48 hours of data was added.



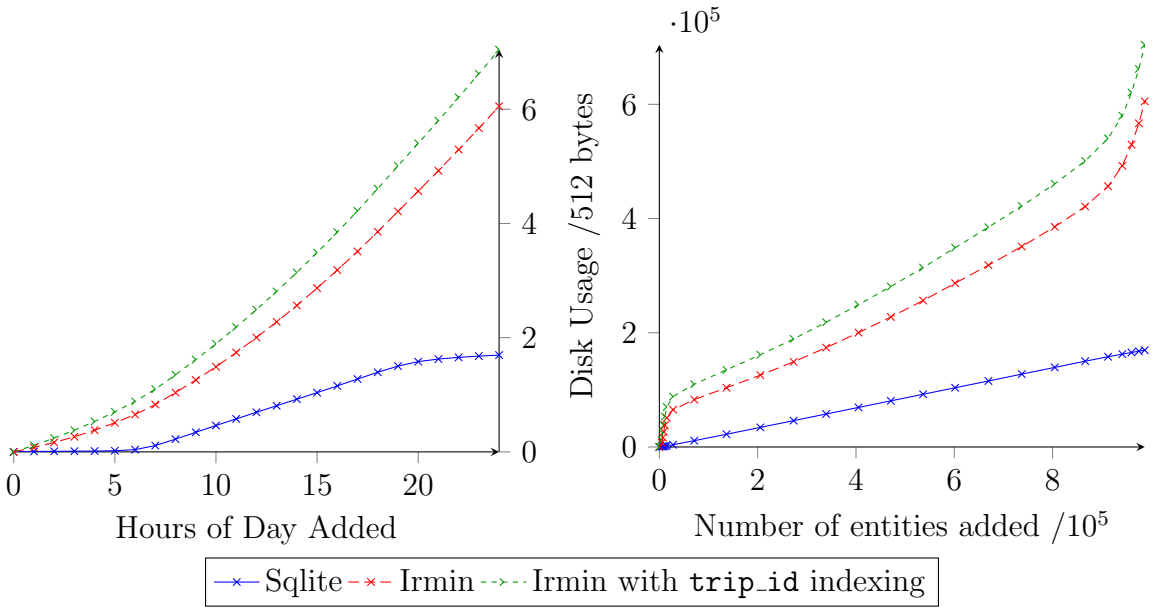
Whilst the time to add data to the Sqlite data store is independent of the quantity already present, the Irmin solutions take longer with each addition file. This is a severe disadvantage for the system produced – as the data added expands to weeks or years it will take an increasingly large amount of time to add them to the data store.

The following graphs display the time taken to both parse and store the protobufs at each hour across a full day. Whilst this reflects better on the OCaml system, comparison against a compiled language and a non-empty data store would put it at a significant disadvantage.



4.2.2 Storage Requirements

In addition to the time required to add new records to the data stores, the space required by the different options impacts how each solution can be used. Real data could be used as testing was time-independent so programs were written to add a specified range of files to the data store with the size of the data store measured for every hour in a complete day.



Two immediate observations can be made from the data, firstly, the storage required by Irmin is much greater than that required by Sqlite and, secondly, whilst the Irmin data stores appear to grow fairly independently from the number of entities in each protobuf, the size of the Sqlite database is directly proportional to the number of entities added. This can be attributed to the fact that the data stored in Irmin is recorded in a file, using the same logs, independent of the number of entities in the file. This evidently has a large overhead and the number of entities in the file does not cause large increases in the amount of space needed to store the file. The larger size of the stored data can be attributed both to the logging and version control that Irmin implements and to the CSV format used to store the data. Sqlite encodes its rows in bytes and so can take advantage of the formats of different columns, integers and enumerations in particular can be encoded much more efficiently.

4.3 Query Performance

To test the performance of query evaluation, the querying section of the server was broken out and programs written to accept queries from the command line. In addition to the OCaml programs, a similar program was written in Python to query the Sqlite database.

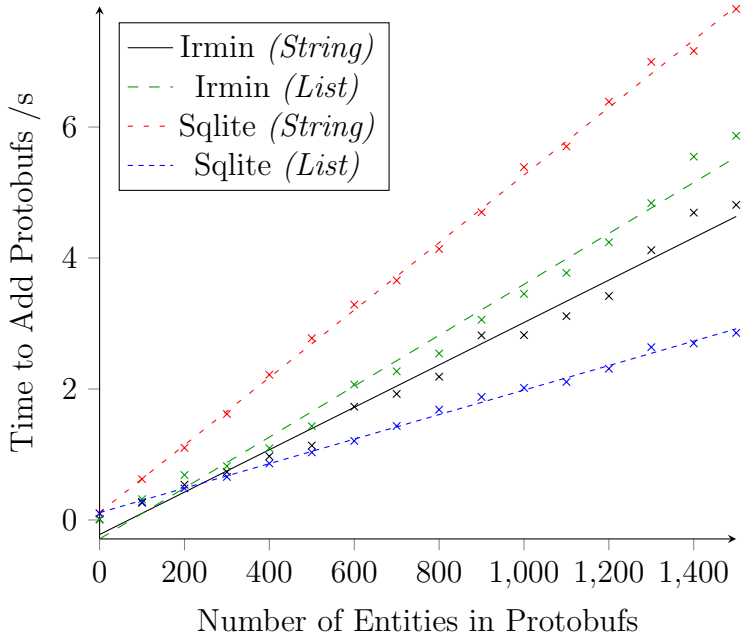
Protobuf messages needed to vary in contents for performance measurements to be realistic, however, the data received from the buses varies wildly, with protobufs ranging from 0

to more than 500 entities in size. To provide a workable dataset for reliable measurements, a python program was written to filter the protobufs to those containing more than 450 entities, discarding the extra entities present in any exceeding the limit. This provided a set of regularly sized protobufs, containing varying data over the course of a day.

The final factor to consider was the formatting of the output. The querying written for the server was focused on returning results as a string. To see how this affected the querying time, versions of both OCaml and Python programs were constructed with and without output formatting.

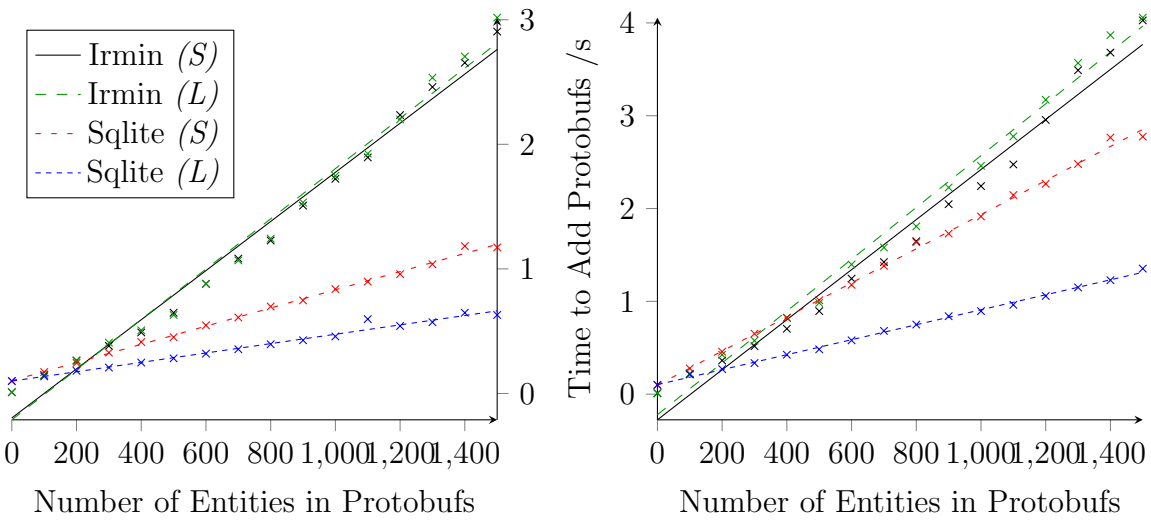
`SELECT * FROM ENTITIES`

The first query to evaluate was the simplest, getting all data from the storage. In practice, it was found that the timestamp and trip id filtering did not cause any slow down of unrelated queries and so have been omitted from the graph below.



There is a large disparity in the results depending on the result formatting. Ignoring the slight initial overhead with Python, the unformatted queries perform well whilst the conversion into an appropriate output format vastly increases the time required to complete the query. Performing somewhat worse regardless, the format produced by the Irmin programs has less impact on the performance, and curiously formatting the output as strings improves performance, a trait which can be attributed to the shorter lists that are produced.

The following graphs show the performance for `SELECT id FROM ENTITIES` and `SELECT timestamp, trip_id, route_id, vehicle_id FROM ENTITIES` respectively. Whilst the OCaml options tend not to vary considerably between queries, the sqlite database operates more quickly when fewer columns are produced for the output.

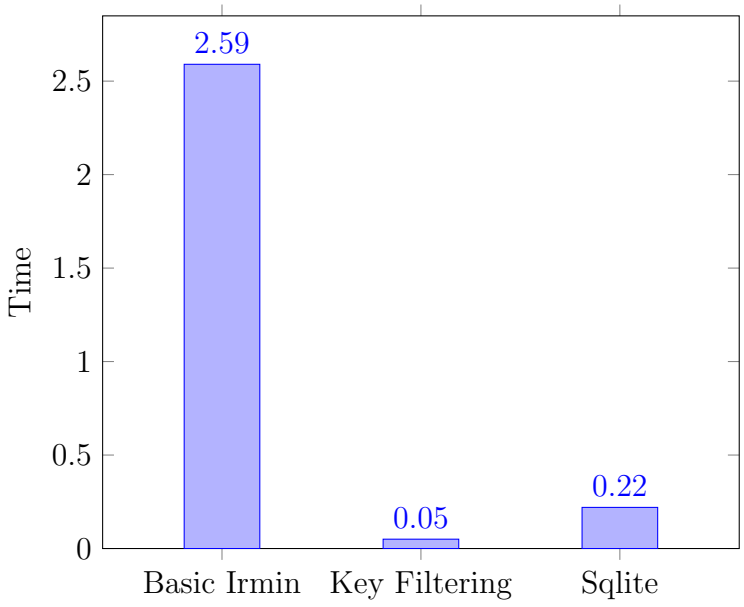


What is clear so far is that the size of the response constitutes a large contribution to the time taken to process a request, independent of the the amount of data the must be read. The following evaluation covers selection of a much smaller amount of rows using `WHERE` clauses, in addition selecting `1` reduces the effects of output construction and string production.

The primary concern here is to see the performance gains from the key filtering options implemented for `timestamp` and `trip_id`.

```
SELECT 1 FROM ENTITIES WHERE timestamp = 1435211397
```

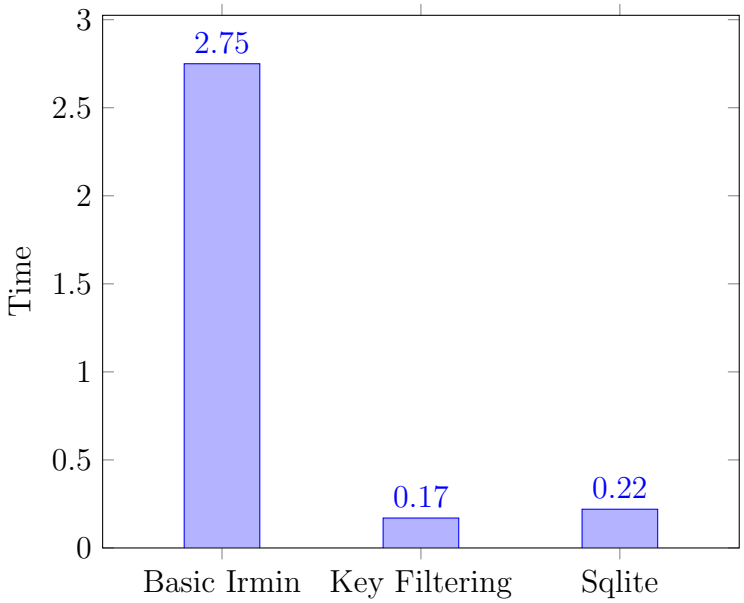
Specifying a specific timestamp displays the largest difference between the filtered and unfiltered programs with ranges moving linearly between the two values, depending on how many files are contained within the selection. Tests were run on the previously generated data – 12 hours of files each containing 450 entities. For Sqlite, timestamp values at the beginning middle and end of the data were tested and all resulted in similar times.



The performance advantage of the filtered file keys displays the need for improvement against the default which is dramatically slower than the Sqlite database. The speed advantage of the filtered Irmin store can be attributed to Sqlite's inability to key as the timestamp values are not unique. Performance comparable to the Irmin data store may be achievable with a NoSQL database since the same file structure as Irmin can be used, however no support for SQL would then be available.

The other field to test was `trip_id`. This would show whether performance comparable to `timestamp` querying could be achieved for non-keyed fields, using the methods implemented.

```
SELECT 1 FROM ENTITIES WHERE trip_id = "953628-20150223-20151231"
```



Positively, the additional data structures to improve the performance of selection by `trip_id` have led to a vast improvement in lookup time, bringing it much closer to the keyed timestamp and beyond the performance of Sqlite on this data.

Implementing the improvements for other fields can be expected to result in similar increases in performance, dependant on the number of results that is expected for any specific value. Filtering by a field such as vehicle id, which is included in many files and produces large responses, would not gain the same improvements.

Performance of the Irmin data store is generally comparable to the Sqlite database, with good query performance, albeit at the expense of storage, the size of which could be reduced in future by employing a more compact store style, `in_transit_to` for example is stored as a string for every entity despite being the only available value for that field. More experimentation could be performed to find the best file size to use and consolidate smaller protobufs to that size in addition to the query performance improvements extended to all fields.

Despite this, there is a clear disadvantage in using Irmin in this format for storage – the time needed to add extra entities increase with the size of the existing store. In systems

where large amount of data are expected, this is a severe limitation and would need to be resolved to use the system in production.

Chapter 5

Conclusion

5.1 Project Review

Over the course of the project the full array of features specified in the Project Proposal have been fulfilled, with a MirageOS web server created that is able to parse incoming transport data, place it into an Irmin data and recount information in response to a subset of SQL queries.

Performance of the system is mixed. Whilst performance of the protobuf parser exceeded the generated C++ code, the time needed to add protobufs to an existing data store limits the usage of Irmin, regardless of the good query performance.

In completing the project it has been shown that MirageOS can be used to implement the servers needed for Internet of Things devices, however further development is needed to offer a complete replacement for the traditional stack. Currently a combination of MirageOS and a tradition database system running separately provides a more practical solution.

In implementing the server I have learned a lot, both specifically to the implementation of servers using MirageOS and to the more widespread skills of using OCaml and the paradigms present in the language in addition to the construction of lexers and parsers, topics I had only covered theoretically in the course.

Were the project to be completed again I would have put more emphasis on producing working code earlier to allow more work to be produced. Much of the language features that I covered in preparation for the project, I was not able to fully understand before coming to rely on them. Starting work on the code earlier would have led to a quicker grasp of the concepts as well as prevented time being wasted on less important features.

5.2 Further Extensions

There is much room for further work to be done in relation to the project. To begin, direct measures to increase the usefulness of the server can be implemented. For example, providing a complete implementation of the GTFS-realtime protobuf which would allow for the system to be used with transport networks where a larger subset of the specification is used. Alternatively increasing the SQL subset available to approach the full range of a traditional database would allow for more analysis to before performed on the transport data.

The work on the project also allows for a variety of other avenues to be explored. For example, the success of the protobuf parser indicates that the approach used could have performance advantages over other implementations even if generated by a compiler. As a result it would be interesting to implement a complete protobuf parsing generator for the OCaml language and further compare performance to that of the other languages available.

Alternatively the SQL data store in the project is rather specific for the data in the protobufs. A useful addition to the libraries available for MirageOS would be a storage solution that more closely mirrors the database format in a traditional stack, extending the SQL subset to include to updating of table structures and the insertion of data.

Recently a paper was published evaluating the performance of various time-series stores for Internet of Things data [22]. By evaluating the performance of this project using the methods outlined in the paper, a more complete understanding of its performance could be achieved.

5.3 Final Comments

Whilst work on the project has often been difficult, it has offered many benefits outside of the traditional course structure. My understanding of functional programming and the advantages that OCaml, and other similar languages, can offer has vastly increased. I have become more comfortable with looking at the source code of libraries I wish to use and the usage of Lex and Yacc for the SQL parser proved to be particularly interesting. Overall it has been rewarding to have worked on a project that has covered such a wide variety of software engineering disciplines in addition to putting in to practice the theory offered in the course.

Bibliography

[1] A. Madhavapeddy and D. J. Scott, "Unikernels: Rise of the virtual library operating system," *Queue*, vol. 11, no. 11, p. 30, 2013.

[2] "Mirageos." `https://mirage.io`. [Online; accessed 11-May-2016].

[3] T. Leonard, "Cuekeeper." `https://github.com/talex5/cuekeeper`, 2015. [Online; accessed 11-May-2016].

[4] G. Developers, "Protocol buffers." `https://developers.google.com/protocol-buffers/`. [Online; accessed 11-May-2016].

[5] A. Lavrik, "The piqi project." `http://piqi.org`, 2014. [Online; accessed 11-May-2016].

[6] L. Lin, V. Lychagina, W. Liu, Y. Kwon, S. Mittal, and M. Wong, "Tenzing a sql implementation on the mapreduce framework," 2011.

[7] A. S. Foundation, "Apache kylin." `http://kylin.apache.org`. [Online; accessed 11-May-2016].

[8] Y. Minsky, A. Madhavapeddy, and J. Hickey, *Real World OCaml*. O'Reilly Media, 2013.

[9] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, D. Sheets, D. Scott, R. Mortier, A. Chaudhry, B. Singh, J. Ludlam, *et al.*, "Jitsu: Just-in-time summoning of unikernels," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pp. 559–573, 2015.

[10] "Hello mirageos world." `https://mirage.io/wiki/hello-world`. [Online; accessed 11-May-2016].

[11] "Cohttp." `https://github.com/mirage/ocaml-cohttp`. [Online; accessed 11-May-2016].

[12] "Irmin." `https://github.com/mirage/irmin`. [Online; accessed 11-May-2016].

[13] T. Leonard, "irmin-www." `https://github.com/talex5/irmin-www`, 2015. [Online; accessed 11-May-2016].

[14] Ocsigen, "Light weight threads." `https://ocsigen.org/lwt/`. [Online; accessed 11-May-2016].

[15] G. Developers, "Realtime transit." `https://developers.google.com/transit/gtfs-realtime/`. [Online; accessed 11-May-2016].

[16] T. for London, "Transport for london unified api." `https://api.tfl.gov.uk`. [Online; accessed 11-May-2016].

[17] F. P. Y. Régis-Gianas, "Menhir reference manual,"

[18] J. Leffler, "Bnf grammar for iso/iec 9075:1992 - database language sql (sql-92)." `http://savage.net.au/SQL/sql-92.bnf.html`. [Online; accessed 11-May-2016].

[19] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," in *ACM SIGPLAN Notices*, vol. 48, pp. 461–472, ACM, 2013.

[20] I. Briggs, M. Day, Y. Guo, P. Marheine, and E. Eide, "A performance evaluation of unikernels,"

[21] "Sqlite." `https://www.sqlite.org/index.html`. [Online; accessed 11-May-2016].

[22] D. G. Waddington and C. Lin, "A fast lightweight time-series store for iot data," *arXiv preprint arXiv:1605.01435*, 2016.

Appendix A

Minimal GTFS-Realtime Specification

Whilst the full GTFS-realtime protobuf specification [15] covers a wide array of messages relevant to a transport network, the Cambridge bus service only implements a small subset. To simplify the parsing code required, a minimal protobuf specification was created for the project. It is displayed below with the comments from the original specification omitted.

```
syntax = "proto2";
option java_package = "com.google.transit.realtime";
package transit_realtime;

message FeedMessage {
required FeedHeader header = 1;
repeated FeedEntity entity = 2;
}

message FeedHeader {
required string gtfs_realtime_version = 1;
enum Incrementality {
FULL_DATASET = 0;
}
optional Incrementality incrementality = 2 [default = FULL_DATASET];
optional uint64 timestamp = 3;
}

message FeedEntity {
required string id = 1;
optional VehiclePosition vehicle = 4;
}

message VehiclePosition {
```

```
optional TripDescriptor trip = 1;
optional VehicleDescriptor vehicle = 8;
optional Position position = 2;
optional uint32 current_stop_sequence = 3;
optional string stop_id = 7;
enum VehicleStopStatus {
IN_TRANSIT_TO = 2;
}
optional VehicleStopStatus current_status = 4 [default = IN_TRANSIT_TO];
optional uint64 timestamp = 5;
}

message Position {
required float latitude = 1;
required float longitude = 2;
optional float bearing = 3;
}

message TripDescriptor {
optional string trip_id = 1;
optional string route_id = 5;
}

message VehicleDescriptor {
optional string id = 1;
optional string label = 2;
}
```

Appendix B

Protobuf Byte Breakdown

With the structure of the protobuf messages defined before parsing, the only structural information contained within the file are field identifiers and lengths. Below is the breakdown of bytes present in a protobuf file containing a single entity with all fields. The byte values are displayed on the left and the corresponding label placed on the right. Indentation is used to group fields together.

```
10 : Header
: Length of Header
10 :   GTFS Realtime Version
3 :     Length of GTFS Realtime Version
49 :     "1" GTFS Realtime Version String (Always "1.0")
46 :     "."
48 :     "0"
16 :   Incrementality
0 :     Enum (Always 0)
24 :   Timestamp
:     Unsigned Integer 64
18 : Entity
:   Length of Entity
10 :   Id
:     Length of Id
:     Id String
34 :   Vehicle Position
:     Length of Vehicle Position
10 :     Trip
:       Length of Trip
10 :       Trip Id
:         Length of Trip Id
:         Trip Id String
42 :       Route Id
:         Length of Route Id
```

```
:         Route Id String
18 :     Position
:       Length of Position
13 :       Latitude
:         Latitude Float
21 :       Longitude
:         Longitude Float
29 :       Bearing
:         Bearing Float
24 :     Current Stop Sequence
:       Unsigned Integer 32
32 :     Current Status
2 :       Enum (Always 2)
40 :     Timestamp
:       Unsigned Integer 64
58 :     Stop Id
:       Stop Id String
66 :     Vehicle
:       Length of Vehicle
10 :       Vehicle Id
:         Length of Vehicle Id
:         Vehicle Id String
18 :       Label Id
:         Length of Label Id
:         Label Id String
```

Appendix C

Full SQL Type Definition

Only a subset of SQL is supported in the project, below is the type to which queries are parsed and the associated SQL in BNF. This shows the SQL subset supported in addition to extensibility offered by the close relationship to the BNF. Comments in the OCaml type definition are used to illustrate fields not supported with the structures of unsupported fields omitted completely.

```
type numeric_value_expression =
| Float of float
| Int of int
(*| Term of term *)
(*| Addition of
   numeric_value_expression * term *)
(*| Subtraction of
   numeric_value_expression * term *)
```

```
<numeric value expression> ::=
<term>
| <numeric value expression>
<plus sign> <term>
| <numeric value expression>
<minus sign> <term>
```

```
type value_expression =
| NumericValueExpression of
   numeric_value_expression
| StringValueExpression of string
(*| Datetime of datetime *)
(*| Interval of interval *)
```

```
<value expression> ::=
<numeric value expression>
| <string value expression>
| <datetime value expression>
| <interval value expression>
```

```
type derived_column =
   value_expression (* * as_clause *)
```

```
<derived column> ::=
<value expression> [ <as clause> ]
```

```
type select_sublist =
| DerivedColumn of derived_column
(*| Qualifier of qualifier *)
```

```
<select sublist> ::=
<derived column>
| <qualifier> <period> <asterisk>
```

```
type select_list = Asterisk
| SelectList of select_sublist list
```

```
<select list> ::= <asterisk>
| <select sublist> [ { <comma>
   <select sublist> }... ]
```

```
type table_reference = string
(*| TableName of table_name *
   correlation_specification option *)
(*| DerivedTable of derived_table *
   correlation_specification *)
(*| JoinedTable of joined_table *)

type from_clause = table_reference (*
   list *)

type row_value_constructor_element =
| ValueExpression of value_expression
(*| NullSpecification of
   null_specification *)
(*| DefaultSpecification of
   default_specification *)

type row_value_constructor =
   row_value_constructor_element (*
   list *)
(*| RowSubquery of row_subquery *)

type comp_op =
| EqualsOperator
| NotEqualsOperator
| LessThanOperator
| GreaterThanOperator
| LessThanOrEqualsOperator
| GreaterThanOrEqualsOperator

type comparison_predicate =
   row_value_constructor * comp_op *
   row_value_constructor

type predicate =
| ComparisonPredicate of
   comparison_predicate
(*| BetweenPredicate of
   between_predicate *)
(*| InPredicate of in_predicate *)
(*| LikePredicate of like_predicate *)
(*| NullPredicate of null_predicate *)
(*| QuantifiedPredicate of
   quantified_comparison_predicate *)
(*| ExistsPredicate of exists_pre...*)
(*| MatchPredicate of match_predi...*)
(*| OverlapsPredicate of overlaps...*)
```

```
<table reference> ::= <table name>
[ <correlation specification> ]
| <derived table>
<correlation specification>
| <joined table>

<from clause> ::= FROM <table
   reference> [ { <comma> <table
   reference> }... ]

<row value constructor element> ::=
<value expression>
| <null specification>
| <default specification>


<row value constructor> ::=
<row value constructor element>
| <left paren> <row value constructor
   list> <right paren>
| <row subquery>

<comp op> ::=
<equals operator>
| <not equals operator>
| <less than operator>
| <greater than operator>
| <less than or equals operator>
| <greater than or equals operator>

<comparison predicate> ::= <row value
   constructor> <comp op> <row value
   constructor>

<predicate> ::=
<comparison predicate>
| <between predicate>
| <in predicate>
| <like predicate>
| <null predicate>
| <quantified comparison predicate>
| <exists predicate>
| <match predicate>
| <overlaps predicate>
```

```
type boolean_primary = Predicate of
    predicate
(*| SearchCondition of
    search_condition *)

type boolean_test = boolean_primary
    (* * (bool * true_value) option *)


type boolean_factor = bool *
    boolean_test

type boolean_term = boolean_factor (*
    list *)



type search_condition = boolean_term
    list



type where_clause = search_condition


type table_expression =
{
from_clause: from_clause;
where_clause: where_clause option;
(*group_by_clause : group_by_clause
    option *)
(*having_clause : having_clause
    option *)
}

type query_specification =
{
(*set_quantifier : set_quantifier
    option *)
select_list: select_list;
table_expression: table_expression;
(*offset : offset option *)
limit: int option;
}
```

```
<boolean primary> ::= <predicate>
| <left paren> <search condition>
    <right paren>


<boolean test>  ::=  <boolean
    primary> [ IS [ NOT ] <truth
    value> ]

<boolean factor>  ::=  [ NOT ]
    <boolean test>

<boolean term>  ::=
<boolean factor>
| <boolean term> AND <boolean factor>

<search condition>  ::=
<boolean term>
| <search condition> OR <boolean term>

<where clause>  ::=  WHERE <search
    condition>

<table expression>  ::=
<from clause>
[ <where clause> ]
[ <group by clause> ]
[ <having clause> ]


<query specification> ::=
SELECT [ <set quantifier> ]
<select list>
<table expression>
```

Appendix D

Project Proposal

Daniel Karaj
Christ's College
dk484

Computer Science Part II Project Proposal

Transport Data Web Server in MirageOS with support for SQL queries

16th October 2015

Project Originator: Dr Richard Mortier

Project Supervisor: Dr Richard Mortier

Director of Studies: Dr Ian Leslie

Overseers: Dr Markus Kuhn and Dr Peter Sewell

Introduction and Description of the Work

This project involves constructing a web server using MirageOS to archive and process a stream of location data from buses in Cambridge and the surrounding areas. This data will then be made available through SQL querying.

The internet of things will vastly increase the amount of data reported to web servers for archival and processing. Due to the nature of the data, security and scalability are very important. Current web servers such as Apache and NodeJS must run on top of an operating system and hence have a large footprint making them slow to start up or scale and have a large attack surface for potential security breaches. MirageOS aims to solve these issues by combining the application and operating system. Web servers are written as unikernels which can run directly on top of a hypervisor, libraries are used to reduce operating system functions to only those that are needed, greatly reducing the amount of code that must and solving these issues.

In this project, transport data is used as an example of the kind of of data stream expected from internet of things devices and used to evaluate the performance of the MirageOS web server for this purpose in comparison to a preexisting Linux/Apache/Python solution.

In addition the data will be made available through an API accepting SQL queries allowing for easy access to the data so that it may be used by other services.

Starting Point

MirageOS is written in OCaml, I have no previous experience with the language beyond its brief coverage in the Part IB Compilers course and its commonality with the ML covered in Foundations for Computer Science. As such, before any work can be completed on the project, time must be dedicated to learning the language. Real World OCaml covers much of the relevant topics, with slight difference in the threading library used, and so initially time will be spent going through the book.

I have experience in writing more traditional web servers, having used NodeJS for the Part IB group project and worked on a Python web server for much of the summer. As such, whilst I have no experience with the particular libraries used with MirageOS should be familiar with many of the concepts.

Resources Required

No special resources will be required for the completion of the project. As an extension and for evaluation, the web server may be run on a XEN hypervisor, however this is not needed for the completion of the work.

Substance and Structure of the Project

Data from the bus network is received in General Transit Feed Specification Realtime format in Protocol Buffers as a POST request. First the web server must accept the request and route it for archival and processing.

Archival of the requests occurs without any processing, data is written to disk as it comes in, creating an accurate store of all data that has been received. Irmin is a library for persistent storage that operates in a similar manner to git allowing for branching and merging. Custom merge functions can be written to prevent collisions using knowledge about the data, caused by for example delays in the delivery of location data packets.

Routing and storage using Irmin to store data has been commonly covered using MirageOS and should provide no unexpected difficulties. More work is required, however, for processing the data received. Protobufs are a common format for delivering data across a network, in most situations where protobufs are used, code is automatically generated to handle the data and make it easily accessible. No such generators exist for OCaml and such the code to process the realtime transport data will need to be written explicitly.

Once the data is available, research must be performed to decide how it should be stored to allow for efficient access using SQL queries, and to decide upon appropriate tables to be made available. Irmin is not a database structure so an appropriate storage construction must be designed.

On top of this, a layer must be written to parse SQL queries and read the data to return the results specified. Implementing a complete SQL implementation would require vast amounts of work with most functions unnecessary and as such a subset will be chosen, initially covering selection, projection and aggregation of data with the possibility of extending to joins if time allows.

A basic API will be built to allow these queries to be requested correctly routed and the results returned over a HTTP connection.

Other extensions of the project can explore the construction of a more robust set of APIs, possibly performing scaling of the web server under heavy load using Jitsu for just in time booting of unikernels or creating data structures and the associated code for request throttling based on the use of API keys.

Other extensions include using the data made available by the API for example analysis and visualisations such those performed on the MBTA metro data set.

Success Criteria

The following should be achieved:

- Accept incoming POST requests containing bus location data and API requests and forward to appropriate parts of the system
- Archive data received to an Irmin database
- Create system to process protocol buffer and store processed data in a format suitable for SQL querying using Irmin
- Write a parser for a limited SQL subset that is then able to read data from Irmin and return the result over HTTP.
- Evaluate the performance of the unikernel in comparison to the currently existing system in terms of foot print (running and storage overhead), performance (throughput and latency), energy consumption, and scalability and hence assess the viability of MirageOS servers to be used for situations like this in the internet of things.

Timetable and Milestones

17th October - 6th November (Weeks 1 & 2)

Work through the first 10 chapters in the Real World OCaml book covering the majority of the language concepts and perform relevant exercises.

Set up a usable working environment and construct the "hello world" unikernel outlined in the MirageOS documentation, to make sure that work can be easily performed.

7th - 13th November (Weeks 3 & 4)

Cover any remaining relevant topics in the Real World OCaml book including the final two concepts chapters and those covering concurrent programming.

Build MirageOS unikernels to become familiar with the libraries needed and Irmin.

14st - 27th November (Weeks 5 & 6)

Build a web server capable of receiving the transport data and storing it using Irmin.

28th November - 11th December (Weeks 7 & 8)

Write the code needed to process the GTFS-realtime protobufs and research how the data should be stored to aid in being made available through SQL queries.

12th - 25th December (Weeks 9 & 10)

Integrate the protobuf processing with the existing archival system and use Irmin to store the data as decided.

26th December - 8th January (Weeks 11 & 12)

Construct basics of the API, allowing for requests to be made, but not yet returning data. Begin work on SQL query parsing, adjustments may have to be made to the data processing based on work here.

9th - 29th January (Weeks 13 - 15)

Refine code written and continue work on SQL query parsing, reading for the data store to return the correct results.

Review remainder of project plan in comparison to work completed remaining work. Write the Progress Report and prepare for the presentation.

30th January - 19th February (Weeks 16 - 18)

Complete work on the SQL parsing and connect with API to allow for queries to be performed on demand.

20th February - 4th March (Weeks 19 & 20)

Evaluate the performance of the system in comparison to the preexisting web server and perform any necessary refinements.

5th - 18th March (Weeks 21 & 22)

Outline the dissertation with respect to the result of the evaluation and work performed on the project.

19th March - 15th April (Weeks 23 - 26)

Complete a first draft of the dissertation, filling in the outline created using notes from the course of the project and evaluation

16th April - 6th May (Weeks 27 - 29)

Complete work on the dissertation addressing any feedback received and collate all work to be submitted, including a demonstration to my director of studies.

Resource Declaration

The development of the project will occur on my machine (Dual core i7, 8GB RAM, OS X). I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure. Development will occur under version control, using git for versioning and a private GitHub repository for remote backups. In addition files will be added to Dropbox, and the complete contents of my machine automatically backed up daily onto an external hard drive. In the case that my computer should fail, development can be moved to the MCS.