Schema Transactions Using FlatBuffers

Overview

This experiment compares access, update, and schema change time of an entire row of a table as well as using the FlatBuffer and FlexBuffer data serialization format. We are interested in these two formats due to their ability to access serialized data without first copying it into a separate part of memory. This allows for efficient access to data without parsing, copying, or dynamically allocating objects which would require additional processing. For our serialization data, we used sample data from the TPC-H benchmark because of its broad industry-wide relevance.

Experiment Setup

1. Machine Configuration
    - OS Ubuntu 16.04 (LTS)
    - CPU 8-core Intel Core i7 930 at 2.8 GHz
    - Kingston 256GB SSD
    - RAM 12GB
2. Generate LINEITEM table data using TPC-H benchmark
    Resource: http://www.tpc.org/tpch/
3. Create a FlatBuffer and FlexBuffer consisting of LINEITEM records
    Source code for FlatBuffers: https://github.com/google/flatbuffers

A tutorial with additional code for creating FlatBuffers and FlexBuffers can be found in the document "Creating FlatBuffers and FlexBuffers" in the CROSS repository found below.

The code and files used for our experiments can be found in the git repo:
https://github.com/billyinthe510/CROSS
https://github.com/uccross/skyhookdb-ceph/tree/skyhook-kraken/src/progly/flatbuffers

For each of our data type accesses, the *gettimeofday()* method provided by the <*sys/time.h*> library was used to time the following:

- Accessing/storing data using FlatBuffers vs. FlexBuffers
- Updating a Field In-Place vs. Rewriting FlatBuffer
- Alter Table Transactions Requiring a FlatBuffer Rewrite

Data Access Timing: FlatBuffers vs. FlexBuffers

This experiment timed the following:

- Getting the pointer to the root object
  Note: This assumes we already have the buffer pointer and want the root.
- Accessing/storing data using the pointer

- To read FlatBuffer data, we use the generated API from *flatc* to assign fields to local variables
- To read FlexBuffer data, we use a version-controlled record descriptor to access the corresponding field in a FlexBuffer vector

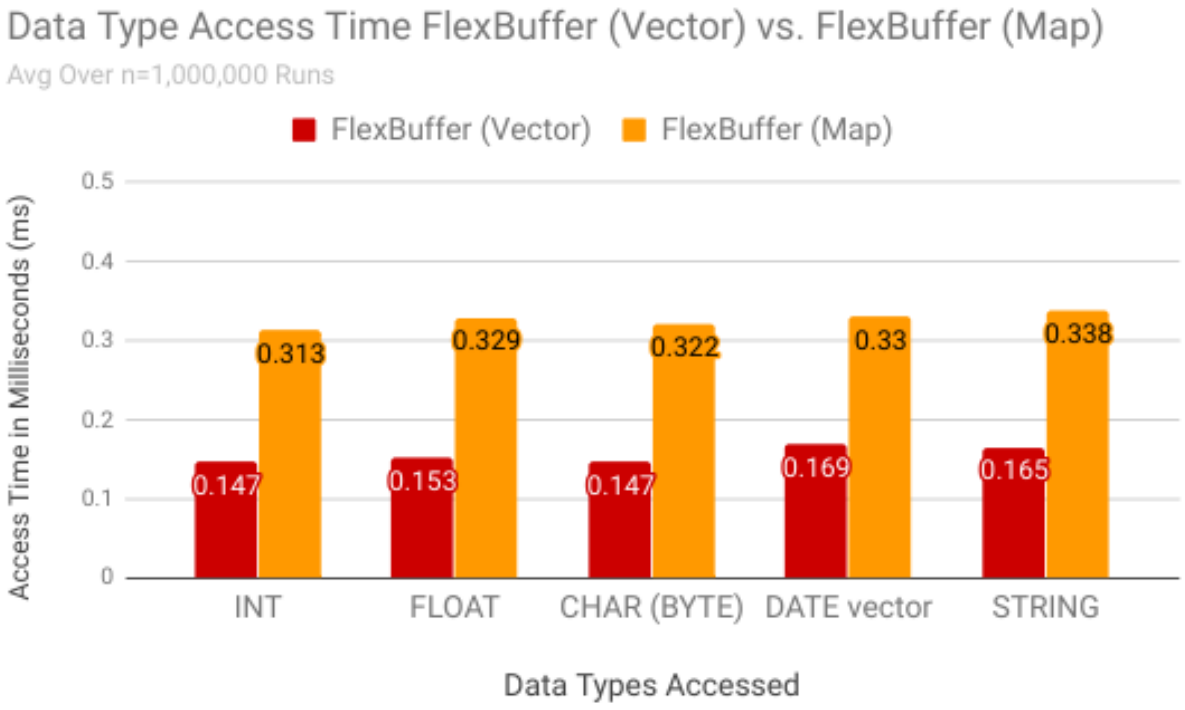
The resulting access times are averaged over 1,000,000 runs and appear to be consistent for each order of magnitude from 100 - 100,000 runs.

The source code for this test was written in C++ and the code can be found in:

- *rowAccessTime.cpp* *(FlatBuffer)*
- *flexBuffersAccessTime.cpp* *(FlexBuffers)*
- *flatflex.cpp* *(FlatBuffer of FlexBuffers::Vector)*
- *multirowFlatBuffer.cpp* *(Multiple Row Flatbuffer)*

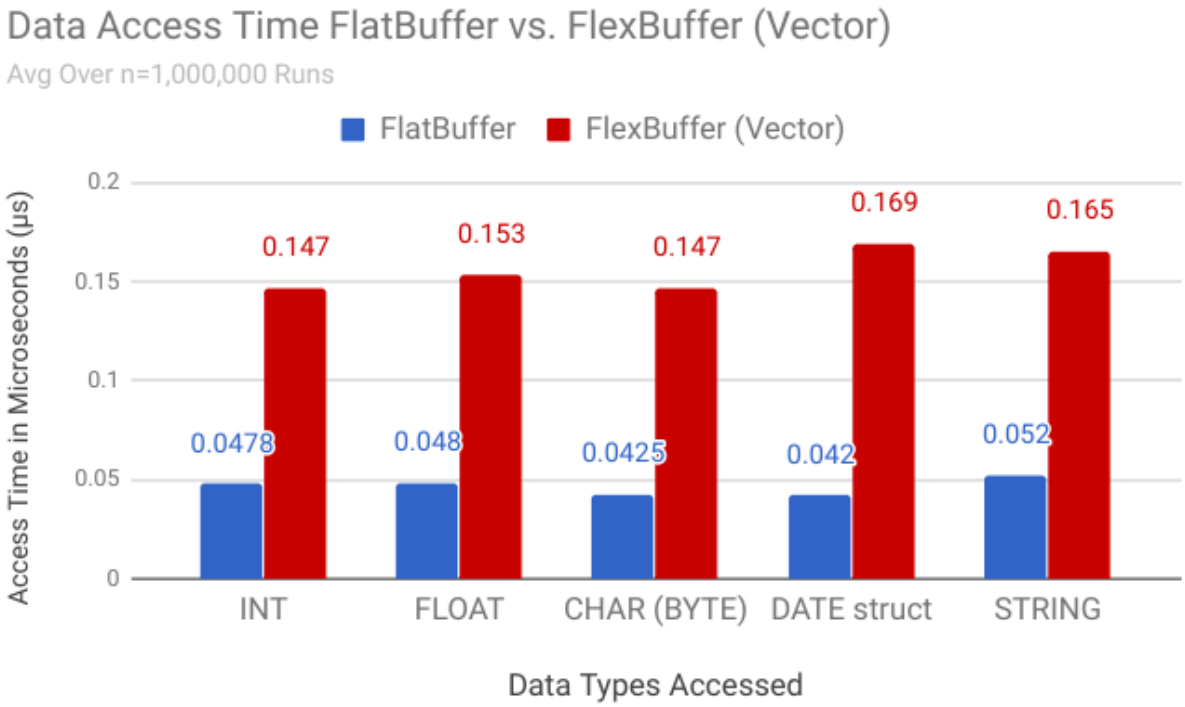
The following two charts shows the average access time of a single row, single field of a LINEITEM FlatBuffer vs. FlexBuffer for various data types. The first chart uses a FlexBuffer(Vector) format while the second chart is a comparison between a vector-based FlexBuffer and map-based Flexbuffer.

Below is included the same data access experiment on FlexBufers using two different data structures to store our row. Vectors are used instead of maps as they are generally a lot more efficient.



The results show that vector-based FlexBuffer has a considerably faster access time for each tested data type over the map-based FlexBuffer (52% on average). However, the use of vectors would require some mechanism for tracking whether any fields are missing/null within the FlexBuffer. We can track null

fields by including a null-bit indicator vector. This can be used to reserve memory for scalar fields which hold a null value but may be updated later.



The chart shows FlatBuffers have a faster access time in all of the tested cases. What isn't shown in the chart above is the time to initially compile a schema file and then include the header as overhead for using FlatBuffers. Although FlatBuffers allow faster data access, they incur these overheads which will make propagating schema changes in a distributed file system challenging. This is where FlexBuffers has an advantage for our use case due to its flexible nature.

We also note the following buffer sizes required to store a sample row using both maps and vectors:

```
Buffer Size (map): 355
Buffer Size (vector): 175
```

FlexBuffer Size For the Same Row (bytes)

In comparison, a FlatBuffer used 172 bytes to hold the same sample row.

```
Buffer Size (FlatBuffer): 172
```

FlatBuffer Size For the Same Sample Row (bytes)

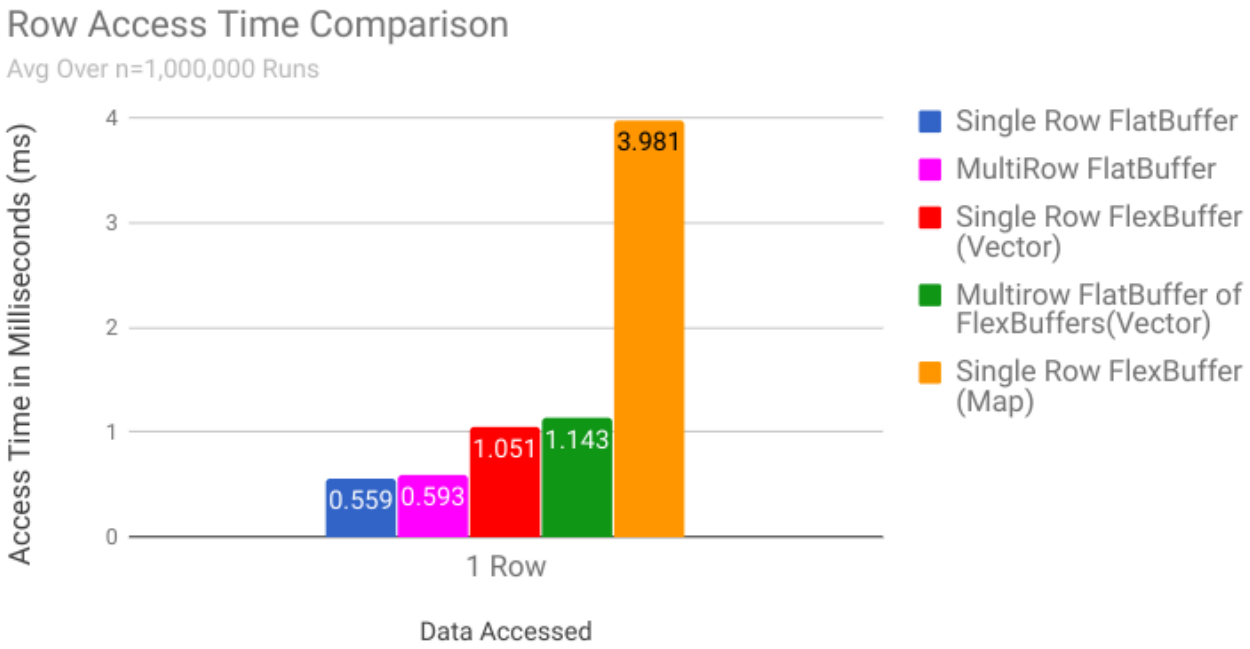
Although FlexBuffers appear to take up a larger amount of memory, we need to consider the overhead included in FlatBuffers. To use FlatBuffers, our sample schema file 'lineitem_generated.h', uses an extra *9,869 bytes*.

Data Access Timing: FlatBuffers vs. FlexBuffers (Single Row vs. Multiple Rows)

This experiment measures the time to read an entire row from a single row buffer vs. a mulirow buffer. We read an entire row of LINEITEM data using the formats mentioned above (FlatBuffer, vector-based FlexBuffer, map-based FlexBuffer) and for the multirow cases we examined two formats:

1. Multirow FlatBuffers
2. FlatBuffer of Multiple FlexBuffers(vector), each FlexBuffer represents a row

With the second format, the FlatBuffer specifies a fix number of rows where the rows themselves are FlexBuffers, which allow schema changes. However, with both formats, appending a row into an existing buffer is not possible because FlatBuffers were not designed to be altered after they're packed.



The access time for an entire row of LINEITEM data is relatively consistent with our previous results. Additionally, the multirow FlatBuffer of FlexBuffers resulted in a slightly slower access time compared to the single row vector-based FlexBuffer, which can be attributed to the overhead of offset indirection through the parent FlatBuffer's root table and vtable to locate the data. The vector-based FlexBuffer created from the same sample row as before now incurs a 53 byte overhead using the FlatBuffer schema.

```
Single FlexBuffer Size: 175 bytes
Multirow FlatBuffer Size (FlatBuffer of FlexBuffers): 228 bytes
Row Count: 1
Overhead for 1 row: 53 bytes per row
```

FlatBuffer of FlexBuffer:vector Overhead (1 Row)

Regardless of how many FlexBuffers(rows) are stored in our FlatBuffer of FlexBuffers, the average row access time did not change. To verify this, we timed row access for a FlatBuffer consisting of a single FlexBuffer as well as 20,900 FlexBuffers (creating a FlatBuffer of size 4MB); both resulting in the same average access time for a read operation of one row. Although average access times are similar, the additional overhead in bytes can vary according to the number of rows present. We observed the average overhead in bytes varied from 53 for a single row down to 21 for the multirow case.

Update Timing: In-Place Mutate Buffer vs. Rewrite Buffer

Here we measured the data update times of a FlatBuffer containing multiple FlexBuffers rows by mutating a single scalar field of FlexBuffer in-place compared to creating a new FlatBuffer then rewriting the updated FlexBuffers and unchanged FlexBuffers into the new FlatBuffer. For this experiment, we timed the following:

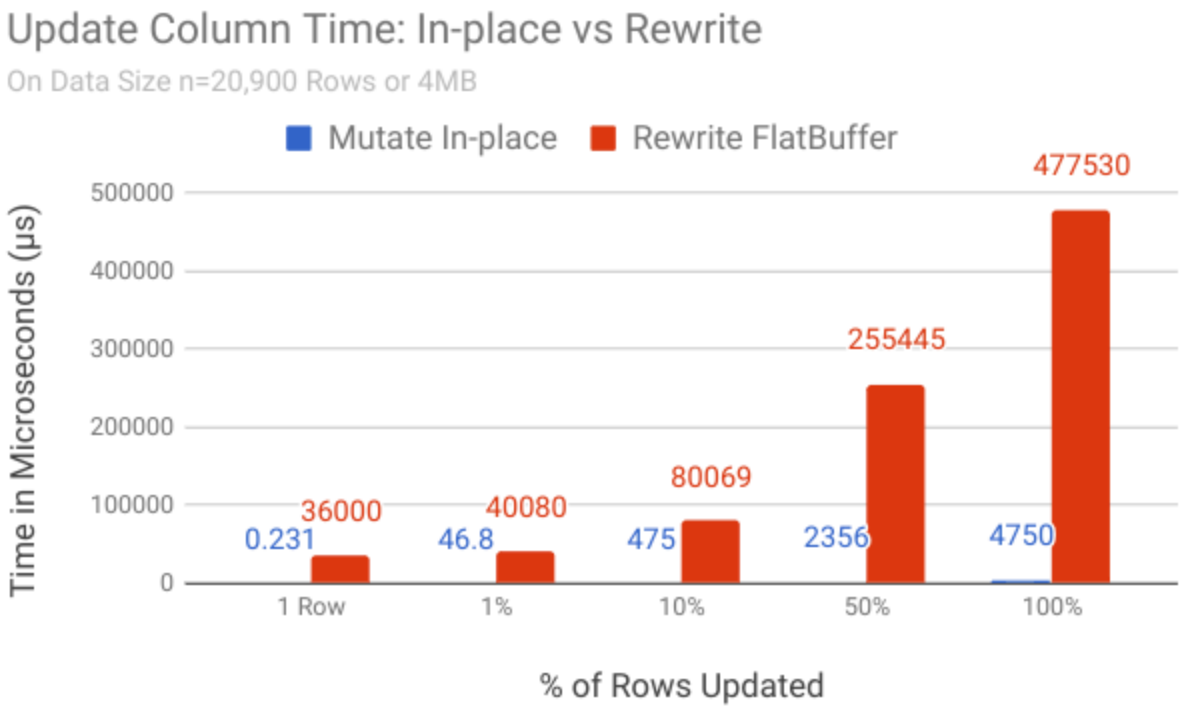
- Getting the pointer to the root object
  Note: This assumes we already have the buffer pointer and want the root.
- Getting $x \subset \{1, 0.01n, 0.1n, 0.5n, n\}$ rows to update
- Mutate an INT;
  - Update In-place
  - Update Rewrite: build a new FlexBuffer with the updated INT and push onto the new FlatBuffer's *rows_vector*, then:
    i. Copy the remaining unchanged rows by using memcpy and pushing into *rows_vector*
    ii. Serialize *rows_vector* and build new FlatBuffer

Below is the test we timed for updating a single INT field within *rowNum* rows. Here we mutate field index 1 by setting it to 556. Our timing code for updating a field by rewriting the entire FlatBuffer can be found in *updateRewriteTest.cpp*.

```
// In-Place Mutate
table = GetMutableTable(buf);
const flatbuffers::Vector<flatbuffers::Offset<Rows>>* recs = table->data();
for(int k=0;k<rowNum;k++) {
        auto flxRoot = recs->Get(k)->rows_flexbuffer_root();
        auto mutatedCheck = flxRoot.AsVector()[1].MutateUInt(556);
}
auto _version = table->version();
table->mutate_version(_version+1);
```

In-Place Mutate *(updateTest.cpp)*

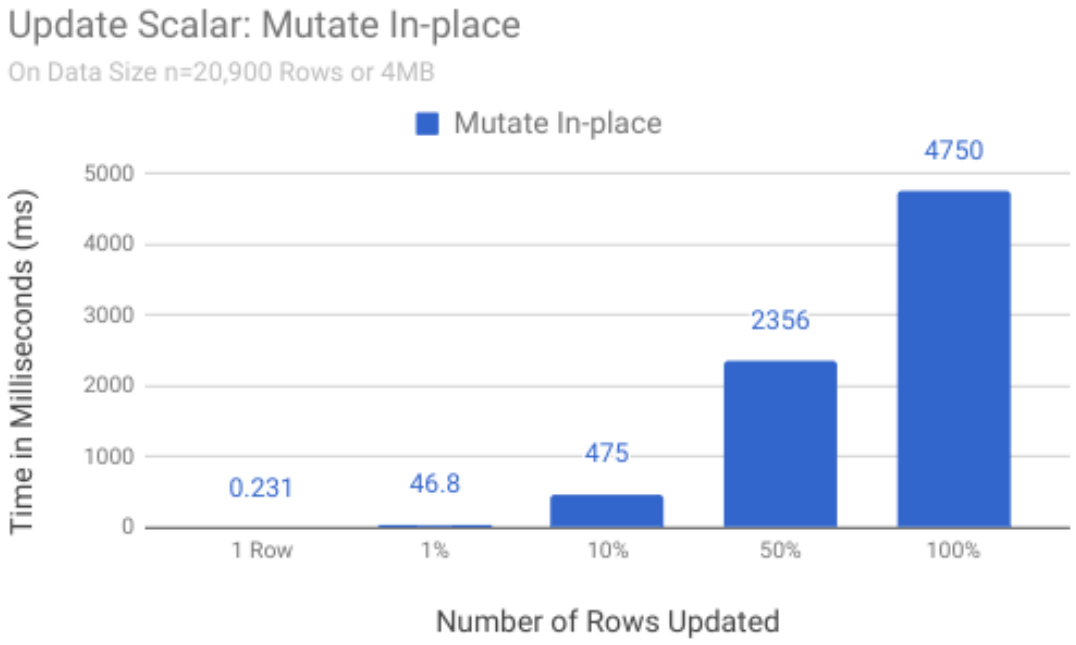
The following chart shows the average update time of *rowNum* rows within a FlatBuffer of FlexBuffers using in-place mutation (as shown in the code above) as well as rewriting the FlatBuffer. The chart reports access times averaged over 1,000 runs (this appeared consistent for each order of magnitude from 100 - 100,000 runs that we tested).



Update Scalar In-Place vs. Rewrite FlatBuffer

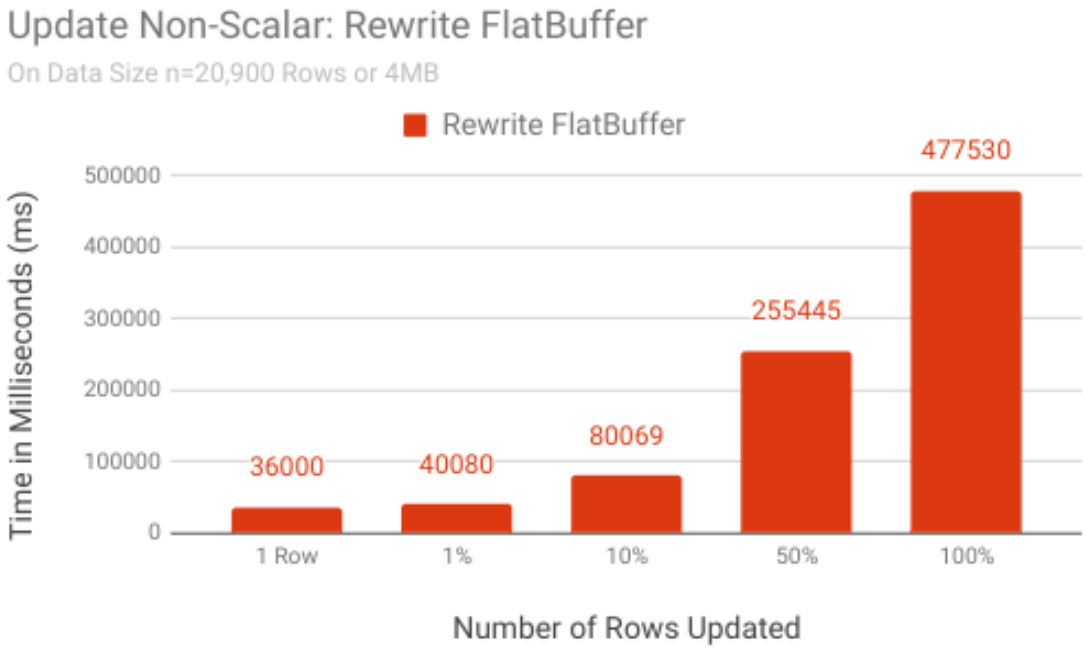
Update by mutation is much faster than rewriting the entire FlatBuffer. This generally only works for scalars and would not work for non-scalar fields. An update by mutation is only possible if the updated value can safely fit within the old value's bit-width. For example, an update on an int from a 32-bit integer to an int that requires 64-bit may fail. Other cases that may cause a mutation to fail include a mutate on default values that were never written or NULL values. This is logical as default values are stored inline and cannot be updated unless flatbuffers::ForceDefaults was used to force all fields to be written.

Updating Scalars: Mutate in-place if possible, otherwise rewrite the FlatBuffer



Updating Non-scalars: Rewrite the FlatBuffer due to possible change in container size
Load Table: Time to build a FlatBuffer of *n* rows
Insert Row: Rewrite the FlatBuffer with new row appended to the end of our vector of rows
Delete Row: Mark this row as dead in a "delete vector" for rows.

Over time, compact will allow for holes in our memory to be removed



Deleting a row may not necessarily require rewriting the entire FlatBuffer; at least not immediately. For deletions, we can simply use a “delete vector” stored in the parent FlatBuffer to denote “alive” and “dead” rows. This will allow for a *compact* method to later clean up holes in our FlatBuffer by rewriting the buffer. The benefit of this “dead” vector is to reduce the number of times we need to rewrite all of the data.

Schema Change Timing: ALTER TABLE

This experiment timed the following:

- Getting the pointer to the root object
  Note: This assumes we already have the buffer pointer and want the root.
- Adding or deleting a column by rewriting each FlexBuffer within the FlatBuffer

Adding or deleting a column within a FlatBuffer of FlexBuffers is currently not possible without rewriting the entire FlatBuffer. Since timing data on alter table transactions are similar to updating by rewrite, please refer to the above graph for “Update Non-Scalar: Rewrite FlatBuffer” for timing measurements.

Conclusions

Single row FlatBuffers will consistently have the fastest access time for each data type tested but a disadvantage of FlatBuffers is challenges involving schema changes. This is why FlexBuffers may perform better for our use case; for its flexibility.

By merging FlatBuffers and FlexBuffers, we have a multirow FlatBuffer of FlexBuffer rows. This will allow for flexibility but will incur additional overhead. The size of this larger overhead will vary depending on the number of rows stored within the parent FlatBuffer as each FlexBuffer row will need to be serialized into the parent buffer.

Updating a scalar field takes the least time if we are able to mutate the value in-place. When updating a non-scalar, we have no choice but to rewrite the entire parent FlatBuffer as the updated FlexBuffer size may grow/shrink.

Inserting and deleting a row would also require a rewrite of the entire buffer. Appending to an existing FlatBuffer would not be possible even if enough spaces are allocated or “reserved” because once a buffer is packed it is not designed to be altered. Since the cost of rewriting an entire FlatBuffer is high, we need to create a method to insert/delete that would not need a rewrite every time. One such method is the “*delete vector*” that tracks deleted rows and can later be used to compact our buffer and free unused memory.

Our resulting data serialization format consists of a FlatBuffer of FlexBuffers with a version field in the parent FlatBuffer. This will allow for version-control among FlatBuffers distributed across multiple nodes. A “delete vector” is also included in the parent vector to keep track of deleted rows. The benefit of having this extra overhead is to reduce the number of times we would need to rewrite our FlatBuffer on

deletion. We also include a null-bit field indicator within each FlexBuffer row. This allows space to be reserved for null values which may be later updated.