Get Familiar with FDB Source Code

Download the FDB source code at https://github.com/apple/foundationdb and build it locally in your machine (either Linux or Mac; If you have Mac M1/M2, this sub-task is even more challenging). Prepare a document that lists the steps you did and the commands you executed.

- **SubTask Completion**: I couldn't complete this subtask, was having issue while setting up both in linux and mac os.

From the source code built at Sub-task 1, launch the FDB server and use the fdbcli to connect to it. Issue simple KV operations (get, set, getrange, delete) via the fdbcli tool.

- First I opened FDB command line tool using fdbcli.
- I tried out few of the below commands:

  ```
  writemode on
  set k1 v1
  set k2 v2
  set k3 v3
  writemode off
  get k1
  getrange k1 k4 5
  writemode on
  clear k3
  clearrange k1 k3
  begin
  set k1 v1
  set k2 v2
  commit
  getversion
  begin
  set k3 v4
  reset
  set k3 v3
  commit
  begin
  set k4 v4
  rollback
  exit
  ```

- **Accomplishment**: I successfully used the FDB command line tool (fdbcli) to connect to the FoundationDB cluster and tested various commands like writemode on, set, get, getrange, clear, clearrange, begin, commit, rollback, and exit within fdbcli.
- **SubTask Completion**: This task was completely by trying out various commands in fdbcli.
- **Obstacles**: There were no obstacles encountered during this process.

Use the FDB C library and write a C program (named the file basic_ops.c) to perform basic operations (get/set/getrange/delete) to the FDB server. The document for FDB C API could be found at: https://apple.github.io/foundationdb/api-c.html

- Created a C code and implemented basic operations (get, set, getRange, clear, clearRange).
- To run this go to build folder and run the following commands.

```
cmake ..
make
Run the executables which u want to
```

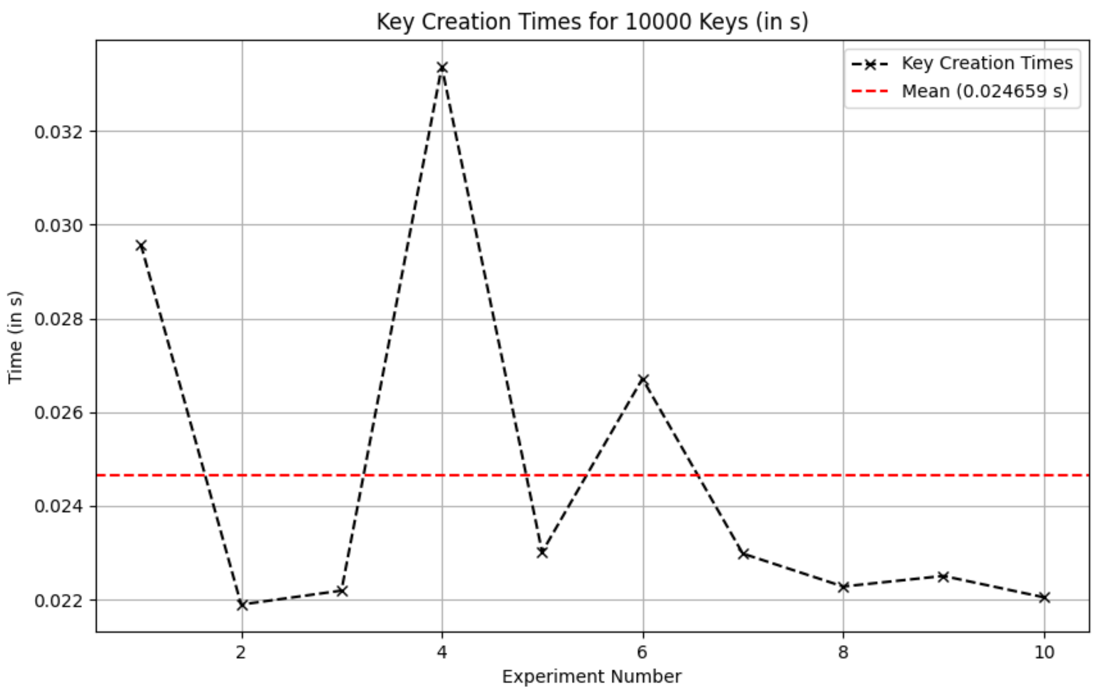
- **Accomplishment**: Successfully created a C code to interact with FoundationDB. The project implements basic operations like inserting, retrieving, and querying a range of keys in the database, using the FoundationDB Java client.
- **SubTask Completion**: The sub-task was fully completed. All required operations, including get, set, and getRange, were implemented as separate methods. In addition, other important methods such as clear and clearRange were also implemented for managing key-value pairs in the database
- **Obstacles**:
  - The program requires a properly installed and running instance of FoundationDB to function correctly. Without an active FDB server, the operations will not execute as expected.
  - For asynchronous handling, the getRange function required the proper use of CompletableFuture to handle results asynchronously.

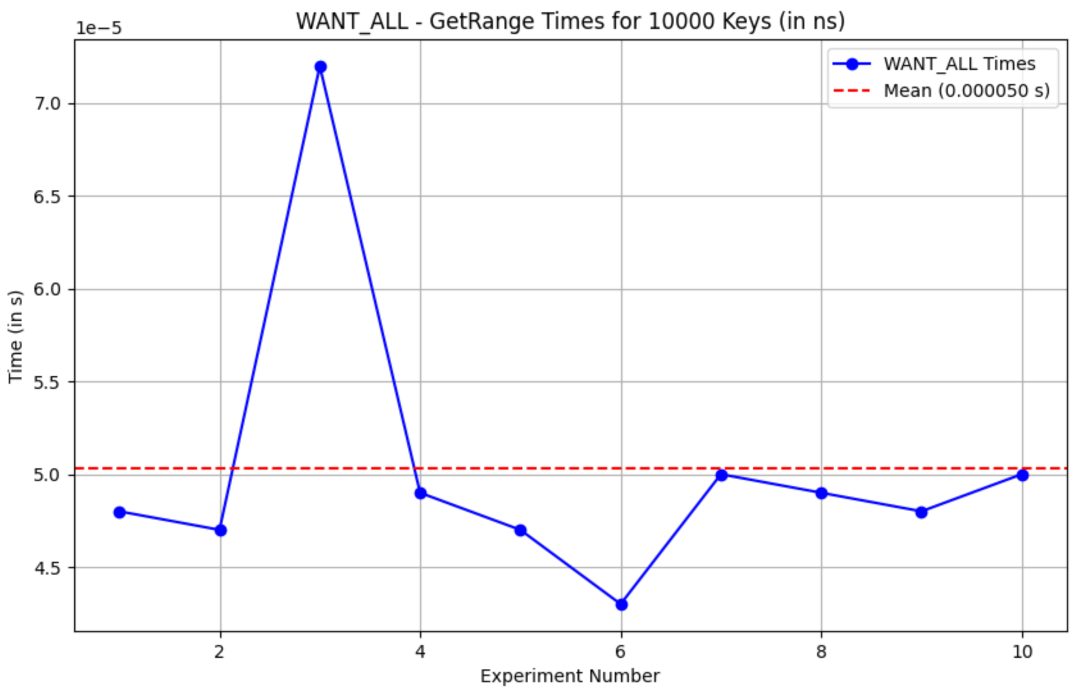
Measure single getrange performance:

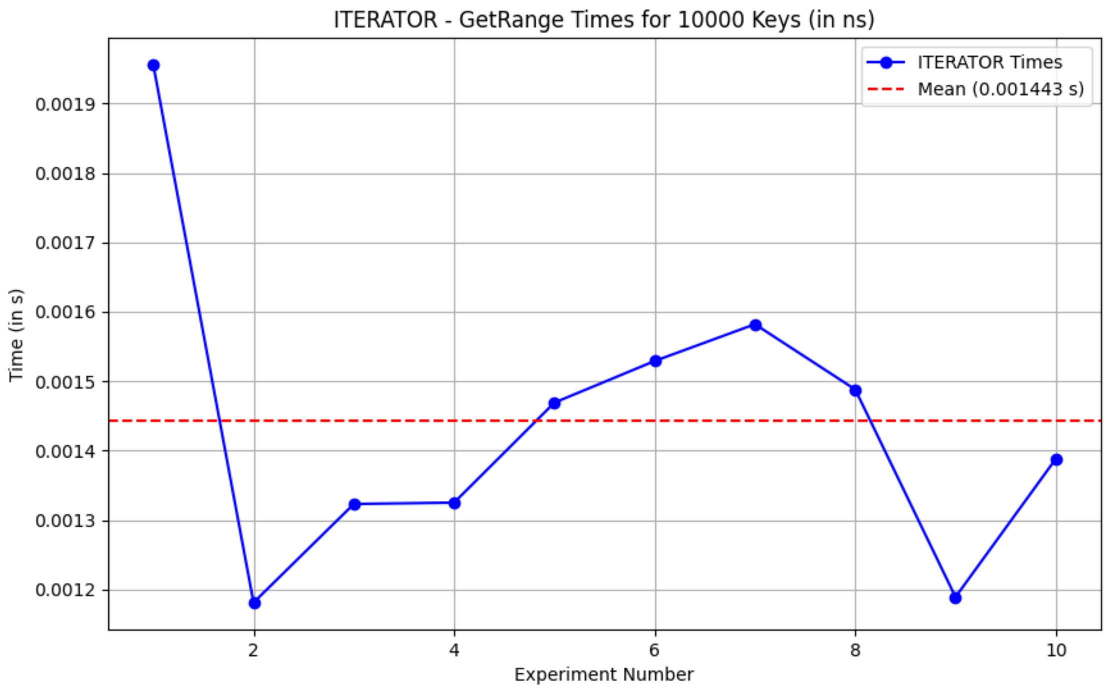
```
- Create a file single_get_range.c and use it for this sub-task
- Store 10k key-value pairs (key_i, val_i) in FDB
- Retrieve all 10k key-value pairs by executing a getrange on \x00 \xff
- Modify getrange to use different modes (WANT_ALL, EXACT, ITERATOR,
etc..) and report the response time of each execution
```

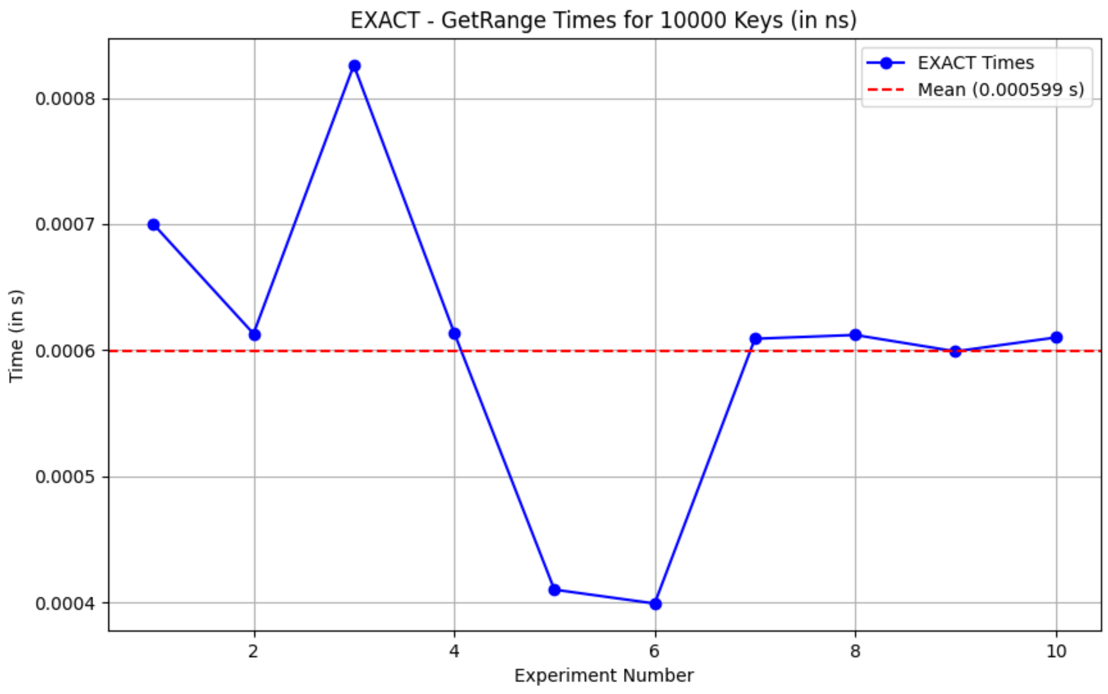
- For this task, I create 10k Key-Value pairs and stored it in FDB and retrieved it through different streaming modes, and deleted all the keys, and repeated the same experiment for a total of 10 times.

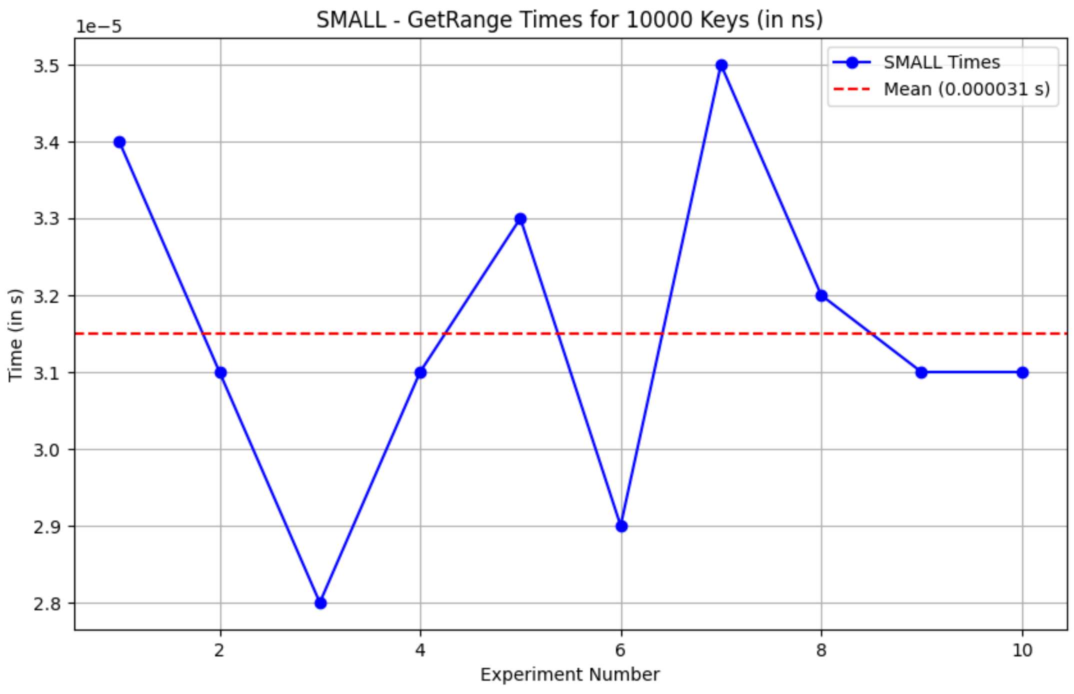
- Below are the results of the same :

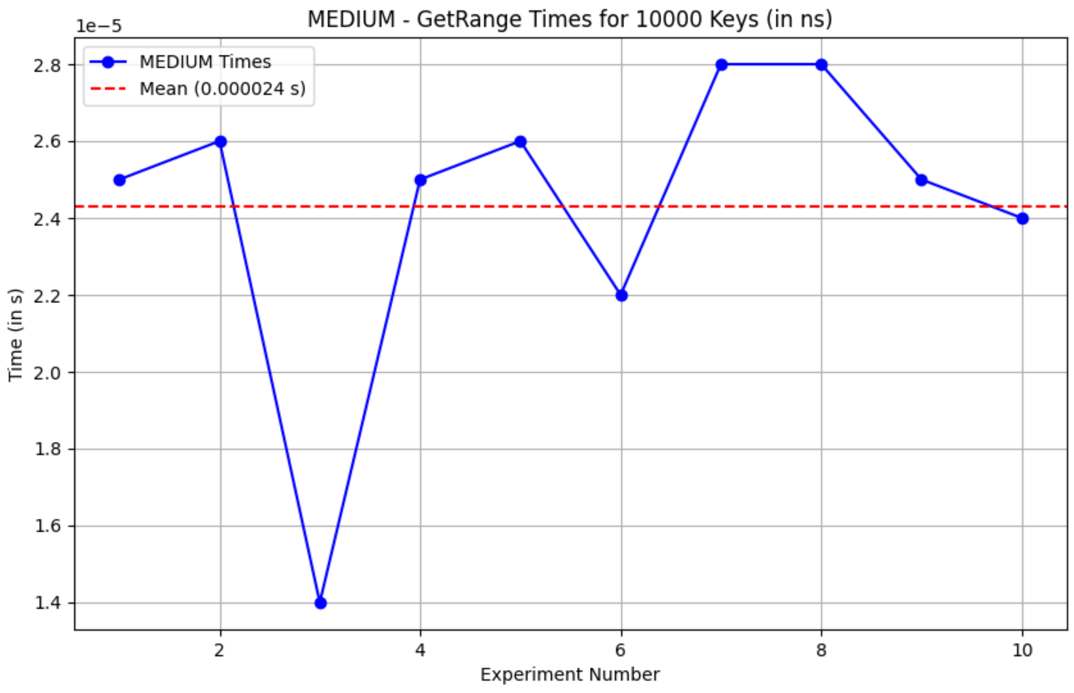


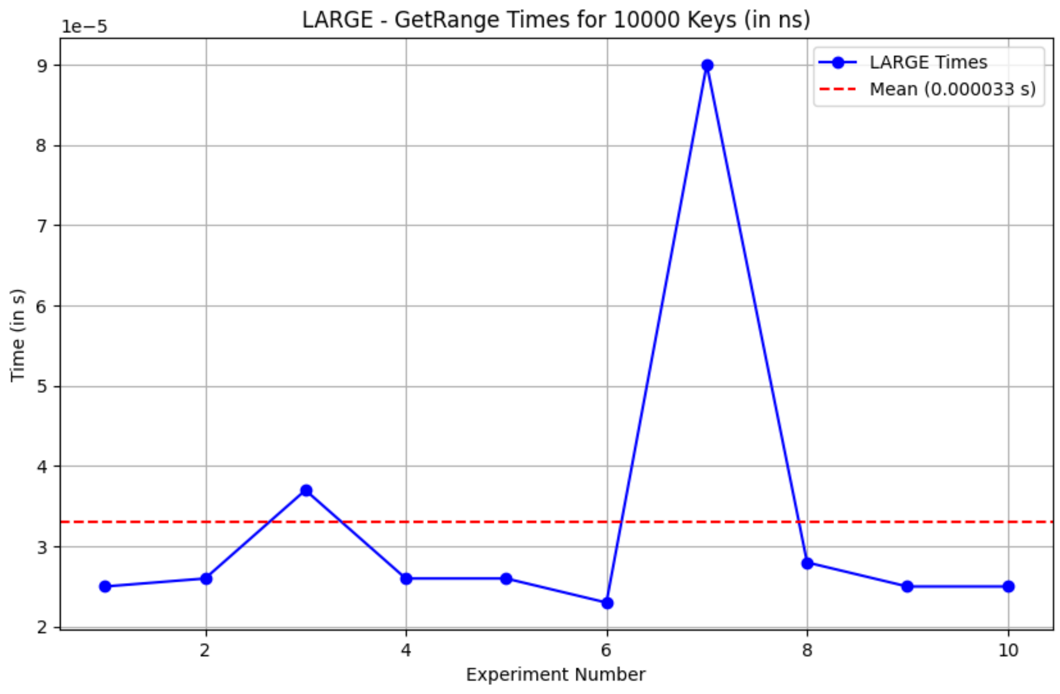


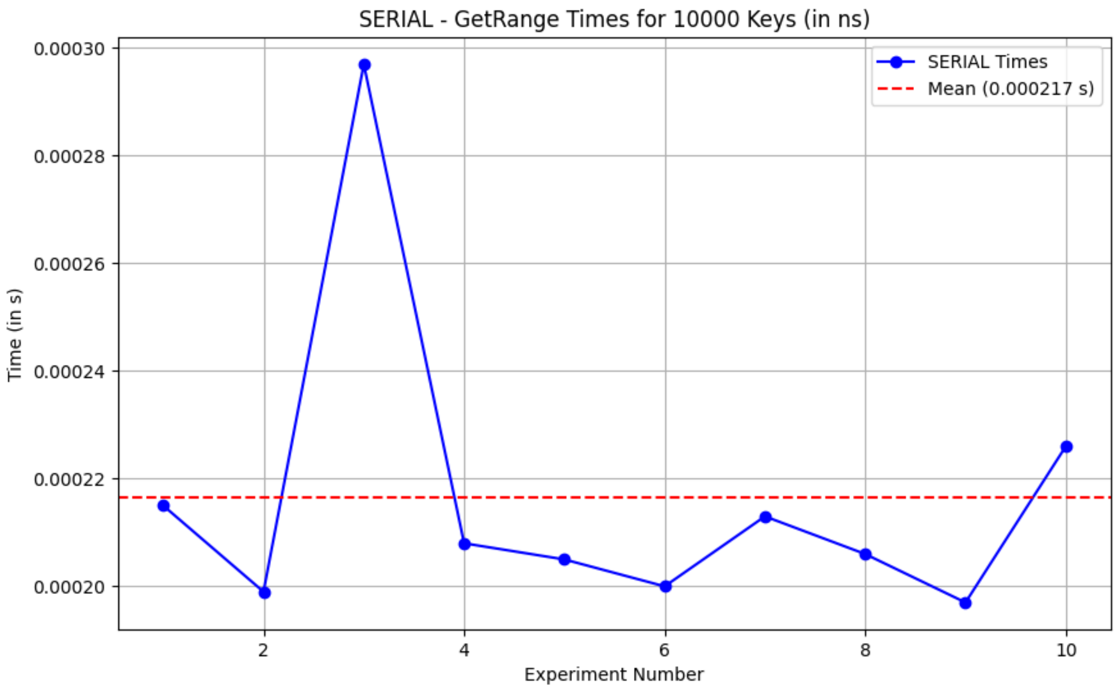

ITERATOR - GetRange Times for 10000 Keys (in ns)
ITERATOR Times
Mean (0.001443 s)
Time (in s)
0.0019
0.0018
0.0017
0.0016
0.0015
0.0014
0.0013
0.0012
2
4
6
8
10
Experiment Number


EXACT - GetRange Times for 10000 Keys (in ns)
EXACT Times
Mean (0.000599 s)
Time (in s)
0.0008
0.0007
0.0006
0.0005
0.0004
2
4
6
8
10
Experiment Number


SMALL - GetRange Times for 10000 Keys (in ns)
1e−5
Time (in s)
3.5
3.4
3.3
3.2
3.1
3.0
2.9
2.8
2
4
6
8
10
Experiment Number
SMALL Times
Mean (0.000031 s)


MEDIUM - GetRange Times for 10000 Keys (in ns)
1e−5
Time (in s)
2.8
2.6
2.4
2.2
2.0
1.8
1.6
1.4
2
4
6
8
10
Experiment Number
MEDIUM Times
Mean (0.000024 s)



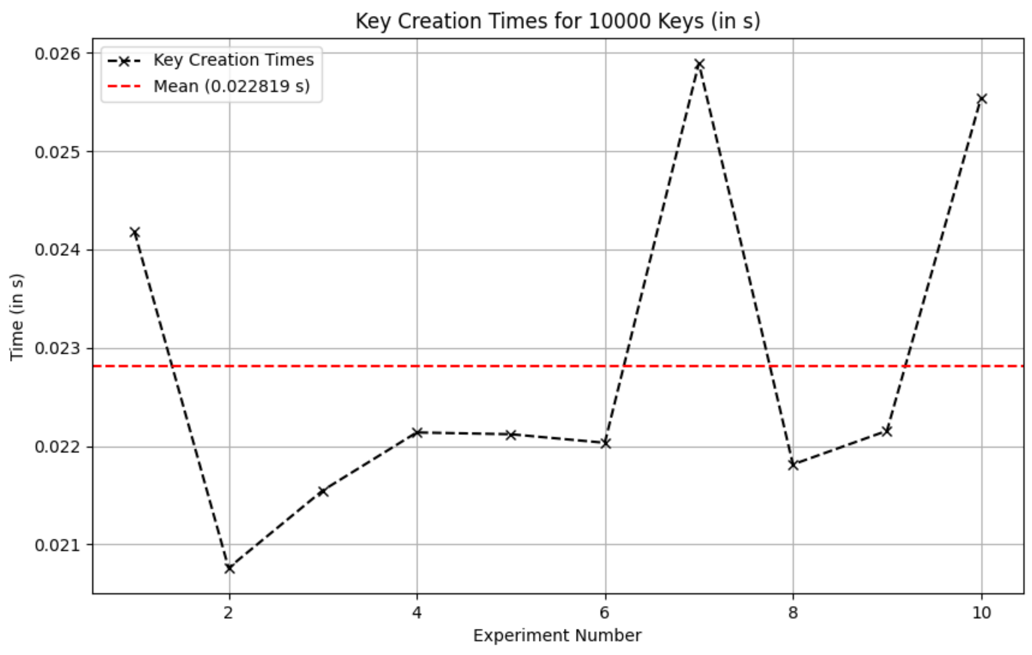


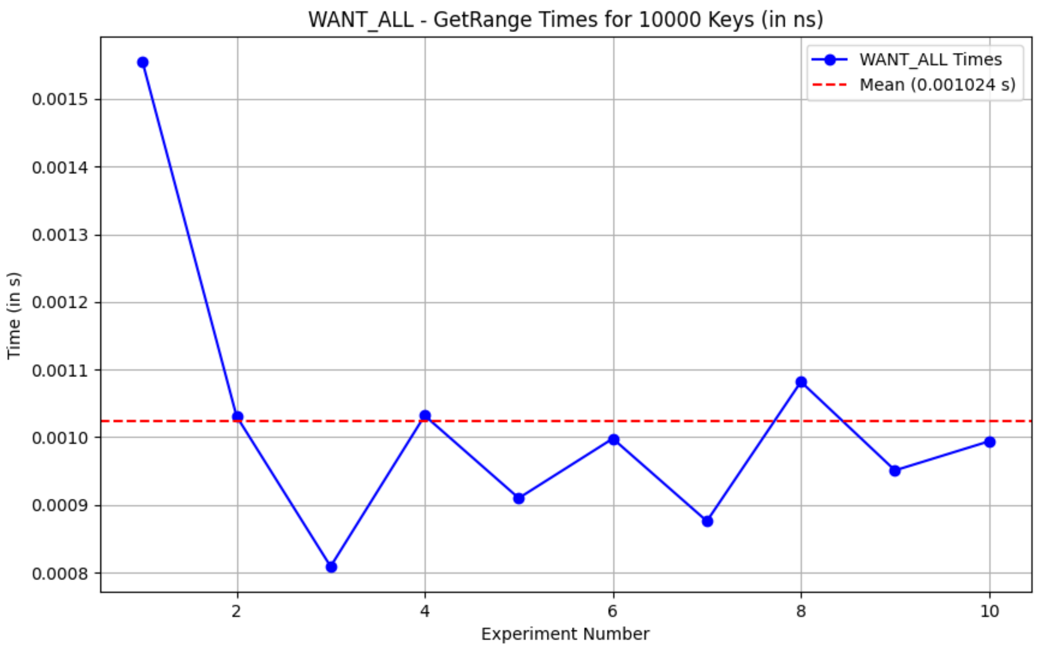
- **Accomplishment**: Successfully created a C code to interact with FoundationDB, and retrieve 10k Key-Value pairs using different streaming modes.
- **SubTask Completion**: The sub-task was fully completed.
- **Obstacles**:
  - The observations seems to be different to that of in JAVA, so I am bit confused on the behaviour, maybe I have made some mistake in my implementation.

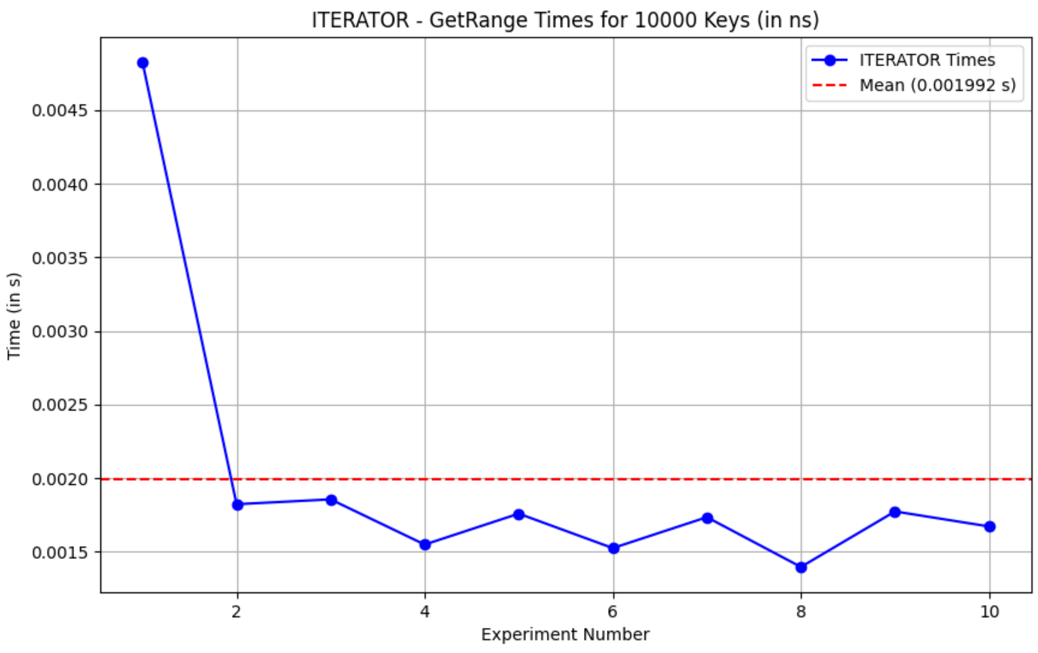
Compare single getrange vs multiple getranges sent in parallel:

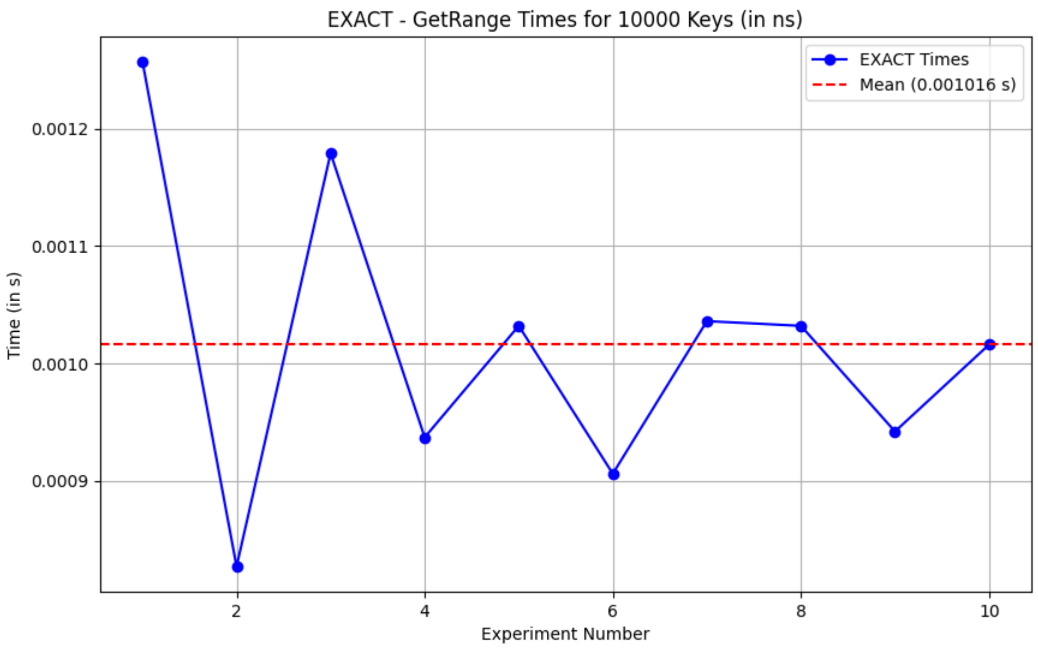
```
- Create a file single_vs_multi_ranges.c and use it for this sub-task
- With the 10k key-value pairs stored in sub-task 5, define 10 ranges R1,
R2, ..., R10 such that executing getrange on each of those 10 ranges
returns exactly 1k key-value pairs.
- Execute getRange() requests on these 10 ranges in parallel
- Repeat the execution with different streaming modes (WANT_ALL, EXACT,
ITORATOR, etc..) and report the response time of each execution. How are
the numbers compared with the case in sub-task 5 where we only issue one
getrange on \x00 \xff ?
```

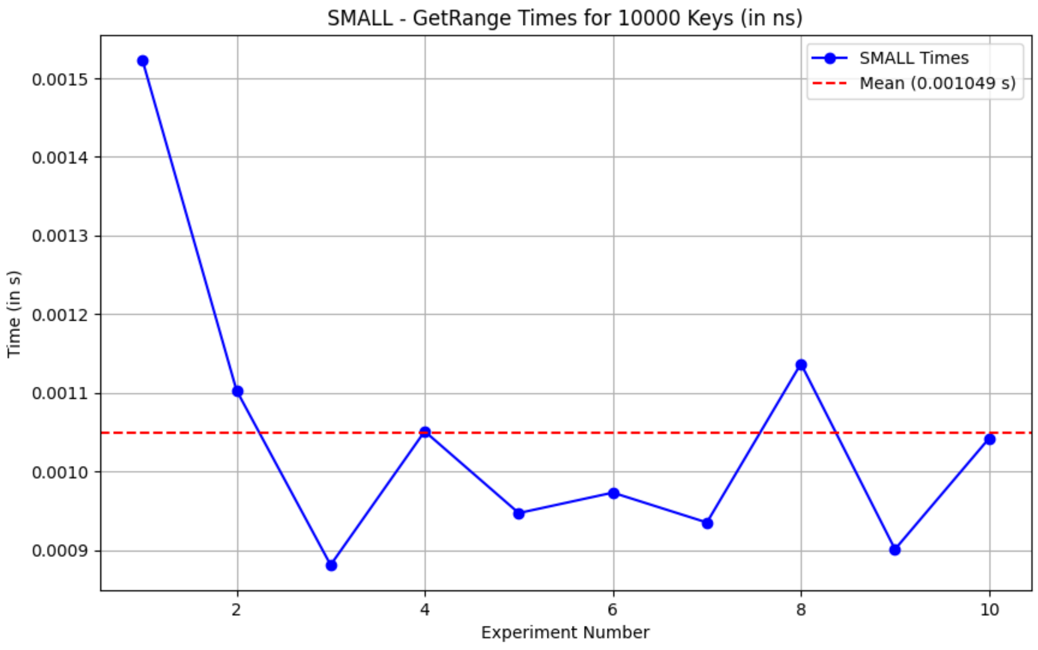
- I created 10k Key-Value pair in Fdb like key_i value_i.
- Divide the 10k into 10 equal ranges and call getrange in parallel 10 times, and wait for all the results to be populated, and do this in different streaming modes.
- Since keys were string to get 10 equal ranges I wrote script and found 10 equal splits from alphabetic order of keys.
- Repeat the whole experiment 10 times, and note timings for each iteration.
- **Observation**:
    - Below are the observation for the 10 experiments:

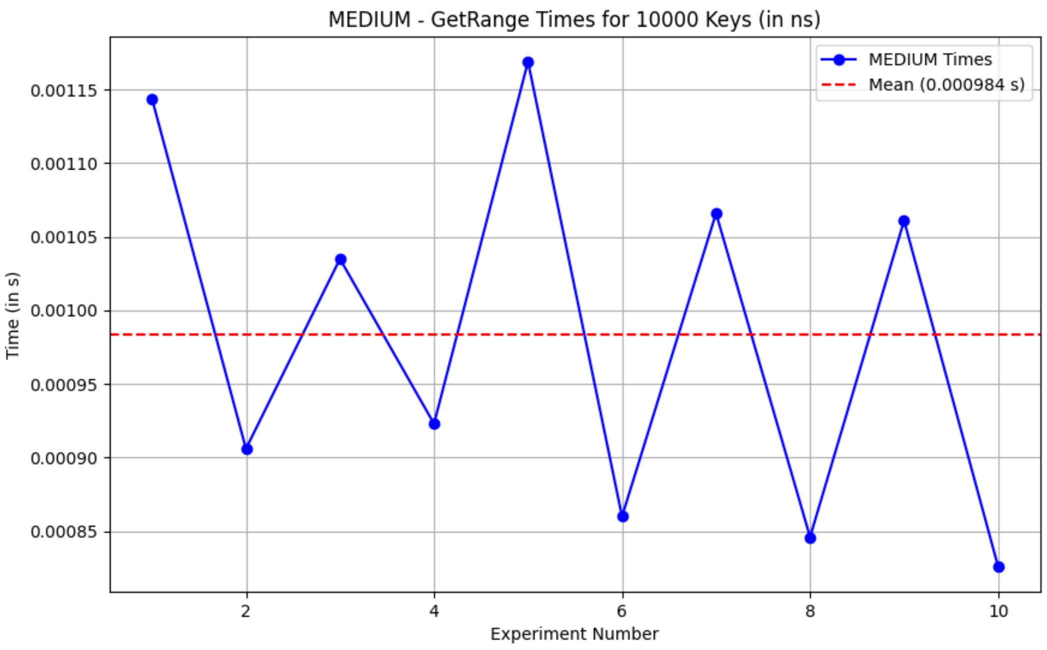


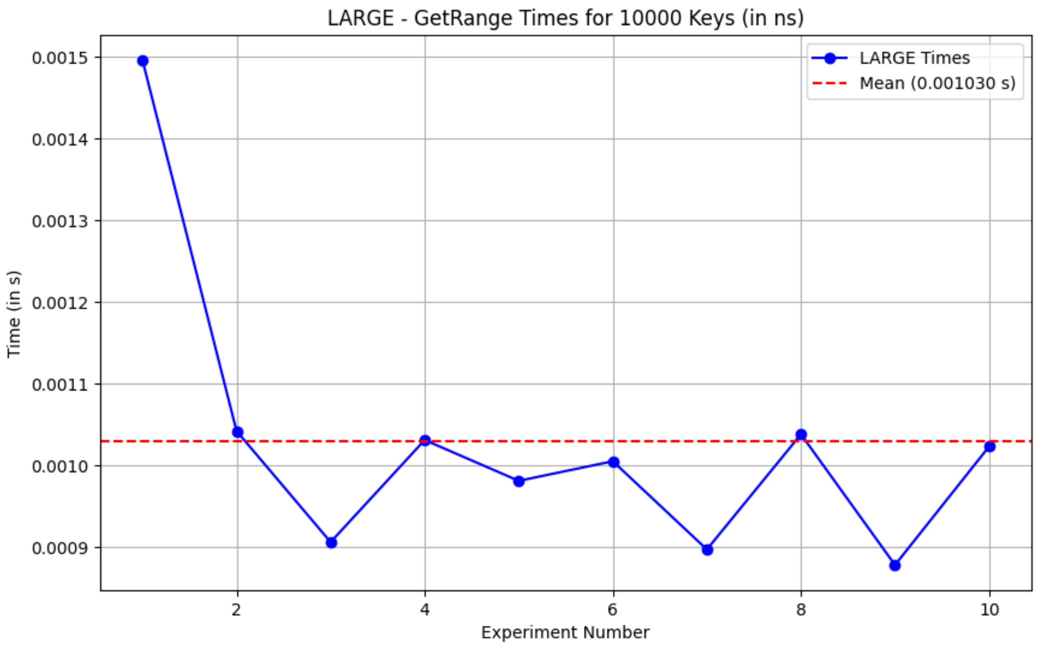

WANT_ALL - GetRange Times for 10000 Keys (in ns)
WANT_ALL Times
Mean (0.001024 s)
Time (in s)
Experiment Number
0.0015
0.0014
0.0013
0.0012
0.0011
0.0010
0.0009
0.0008
2
4
6
8
10

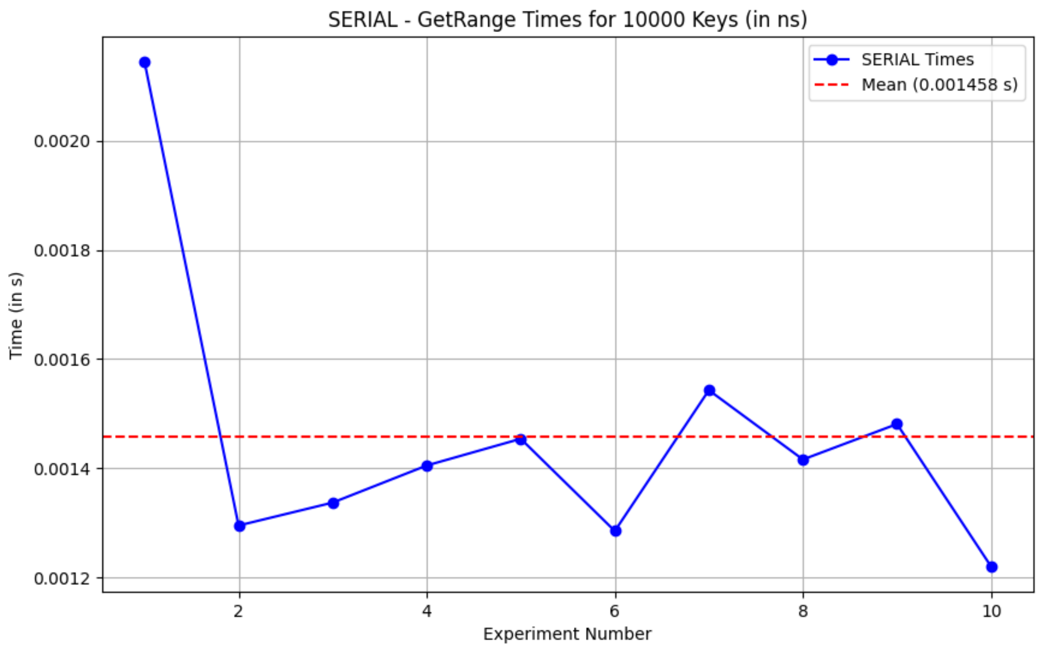

ITERATOR - GetRange Times for 10000 Keys (in ns)
ITERATOR Times
Mean (0.001992 s)
Time (in s)
Experiment Number
0.0045
0.0040
0.0035
0.0030
0.0025
0.0020
0.0015
2
4
6
8
10


EXACT - GetRange Times for 10000 Keys (in ns)
EXACT Times
Mean (0.001016 s)
Time (in s)
0.0012
0.0011
0.0010
0.0009
2
4
6
8
10
Experiment Number


SMALL - GetRange Times for 10000 Keys (in ns)
SMALL Times
Mean (0.001049 s)
Time (in s)
0.0015
0.0014
0.0013
0.0012
0.0011
0.0010
0.0009
2
4
6
8
10
Experiment Number


MEDIUM - GetRange Times for 10000 Keys (in ns)
MEDIUM Times
Mean (0.000984 s)
Time (in s)
Experiment Number


LARGE - GetRange Times for 10000 Keys (in ns)
LARGE Times
Mean (0.001030 s)
Time (in s)
Experiment Number



- **Table**:

	SingleGetRange	SingleVsMultiRanges
Key Creation	0.024659	0.022819
WANT_ALL	0.000050	0.001024
ITERATOR	0.001443	0.001992
EXACT	0.000599	0.001016
SMALL	0.000031	0.001049
MEDIUM	0.000024	0.000984
LARGE	0.000033	0.001030
SERIAL	0.000217	0.001458

- **Accomplishment**: Successfully created a C code to interact with FoundationDB, and retrieve 10k Key-Value pairs using different streaming modes parallely, and compare with previous task.
- **SubTask Completion**: The sub-task was fully completed.
- **Obstacles**:
  - I had few issues while comparing the time, and understanding the results.
  - I was having difficulty in understanding how exactly each streaming mode works.
  - I have doubt in the number of SMALL/MEDIUM/LARGE, as the trends seems to be different. I think I might have implemented it incorrectly.

Understand read snapshot:

```
- Create a class read_snapshot.c and use it for this sub-task
- Store any arbitrary 4 key-value pairs in the database. Denote the keys
as K1, K2, K3 and
K4
- Start a transaction T1 and read several keys (let's say K1, K2, K3)
- On another thread, start a transaction T4 that updates the values of K2,
K4
- Commit transaction T1. Would T1 be aborted? Explain why.
- Commit transaction T2. Would T2 be aborted? Explain why.
```

- **Analysis**:
  - Commit transaction T1. Would T1 be aborted? Explain why.
    - Read Snapshot: When T1 reads K1, K2, and K3, it takes a snapshot of the database at that point in time. This means that T1 is working with consistent data that will not be affected by other transactions (such as T4) until T1 commits.
    - When T4 updates K2 and K4, it modifies the database. However, T1 has already taken a snapshot of the data, so T1 sees the values of K2 and K4 at the time T1 started, and the changes made by T4 are not visible to T1.
    - T1 will not be aborted because it sees the snapshot data from the time it started, and the changes in T4 do not conflict with T1’s operations (since T1 is reading old values of K2 and K4). It can commit successfully without any issues. The changes made by T4 will only be visible to future transactions.
  - Commit transaction T4. Would T4 be aborted? Explain why.
    - T4 updates K2 and K4, but it does not conflict with any ongoing transaction because it is writing to K2 and K4, due to 5s MVCC, so T1 would read take the old snapshot.
    - T4 can commit successfully, as its updates will not interfere with T1’s snapshot. The transaction T4 will update the database, and its changes will be visible to future transactions, but not to T1, since T1 has already taken a snapshot of the data.
- **Accomplishment**: Successfully created a C code to interact with FoundationDB, and understand Read Snapshot and few basic transactions.
- **SubTask Completion**: The sub-task was fully completed.
- **Obstacles**: There were no obstacles.

Understand transaction conflict:

```
- Create a class tx_conflict.c and use it for this sub-task
- Store any arbitrary 2 key-value pairs in the database. Denote the keys
as K1 and K2.
- Start a transaction T1 to read K1 and update the value of K2
- Start a transaction T2 to read K2 and update the value of K1
- Commit T2. Would T2 be aborted? Explain why.
- Commit T1. Would T1 be aborted? Explain why.
```

- When T1 reads K1 and K2, it sees the initial values ("Value1" and "Value2"). However, before T1 commits, T2 reads K2 ("Value2") and updates K1 to "UpdatedByT2". T2 successfully commits because there are no conflicts at that point.
- When T1 tries to commit, FoundationDB detects that T1 has read stale data (the old value of K2) and is trying to update K2 based on that stale information. Since T2 has already committed and modified K1, committing T1 would lead to an inconsistent state. Therefore, FoundationDB aborts T1 to maintain data integrity and consistency.
- To resolve this conflict, T1 would need to retry its transaction, starting from the beginning, to ensure it has the latest data before making any modifications.
- In summary, T2 commits successfully, while T1 is aborted due to a transaction conflict caused by reading stale data and attempting to modify a key that has been updated by another transaction.
- **Accomplishment**: Successfully created a C code to interact with FoundationDB, and understand Transaction conflicts.
- **SubTask Completion**: The sub-task was fully completed.
- **Obstacles**: There were no obstacles.