Efficiently Processing Temporal Queries on Hyperledger Fabric

Himanshu Gupta #1, Sandeep Hans #2, Kushagra Aggarwal *3, Sameep Mehta #4, Bapi Chatterjee #5, Praveen J. #6

# *IBM India Research Lab*, * *IIT Kharagpur*

1 higupta8@in.ibm.com, 2 shans001@in.ibm.com, 3 kushagra.iitkgp@iitkgp.ac.in
4 sameepmehta@in.ibm.com, 5 bapchatt@in.ibm.com, 6 praveen.j@in.ibm.com

***Abstract*—In this paper, we discuss the problem of efficiently handling temporal queries on Hyperledger Fabric, a popular implementation of Blockchain technology. The temporal nature of the data inserted by the Hyperledger Fabric transactions can be leveraged to support various use-cases. This requires that the temporal queries be processed efficiently on this data. Currently this presents significant challenges as this data is organized on file-system, is exposed to users via a limited API and does not support any temporal indexes.**

We present two models for overcoming these limitations and improving the performance of temporal queries on Fabric. The first model creates a copy of each event inserted by a Fabric transaction and stores temporally close events together on Fabric. The second model keeps the event count intact but tags some metadata to each event being inserted on Fabric s.t. temporally close events share the same metadata. We discuss these two models in detail and show that these two models significantly outperform the naive ways of handling temporal queries on Fabric. We also discuss the performance trade-offs for these two models across various dimensions - data storage, query performance, data ingestion time etc.

I. Introduction

Blockchain technologies have become very popular and are being used to provide trust and security guarantees across more and more commercial use-cases. This has led to the development of a number of blockchain platforms e.g., Hyperledger Fabric [1], Ethereum [2], Parity [3] etc. A blockchain is a distributed, shared ledger that records transactions between multiple and often mutually distrusting parties in a verifiable and permanent way. These transactions are maintained in a continuously growing list of ordered records called "blocks", which are tamper-proof and support non-repudiation.

Unlike Bitcoin [4], many blockchain systems (e.g., Fabric, Ethereum, Parity) make a distinction between current and historical states of the data. Data pertaining to various events is ingested on these systems in form of key-value pairs. Usually there are many pairs with the same key. For a key k, the latest pair is called the current state of the key k while all the pairs including the latest pair form the historical states of key k. The collection of current states for all keys is termed as state-db while the collection of the historical states is termed as history-db. The state-db data hence is a snapshot of the history-db data at the latest timestamp.

As more and more transactions happen, the size of history-db data steadily increases and overtime this data grows to a large volume. This data is inherently temporal in nature and analytics of this data can generate valuable business insights and support various use-cases e.g., lineage, visualization, reporting, compliance etc. For example, consider a blockchain supported supply chain use-case wherein a set of shipments are placed in containers and the containers are ferried by trucks. Whenever a shipment s is placed in a container c at time t, a key-value pair $\langle s, (c, t, \text{“}l\text{”})\rangle$ is inserted on blockchain. When shipment s is taken out of container c at time t, the pair $\langle s, (c, t, \text{“}ul\text{”})\rangle$ is inserted. The symbols l and ul denote load and unload events. Similarly the events $\langle c, (tr, t, \text{“}l\text{”})\rangle$ and $\langle c, (tr, t, \text{“}ul\text{”})\rangle$ denote the events when a container c is loaded on/unloaded from a truck tr at time t. Given these load and unload events, we may be interested in finding the trucks which have ferried a shipment within t_1 and t_2 or the set of shipments, a truck has ferried within t_1 and t_2. This requires temporal analytics over the history-db data.

In Hyperledger Fabric, the state-db data is housed on a database while the history-db data is distributed across a set of blocks on file-system. This makes the analytics of history-db data cumbersome and inefficient. Each block contains the details of a set of transactions and a link to the previous block. Secondly, efficient processing of temporal queries requires quick retrieval of the events of interest between any duration (t_1,t_2]. However, this operation is costly on Fabric. Retrieval of key-value pairs which belong to key k and which describe events within duration (t_1,t_2] requires deserialization of many blocks and this incurs a large disk IO cost. Specifically we need to deserialize all blocks which satisfy the two conditions - (a) the block contains a transaction which ingested a key-value pair with key k, and (b) this pair describes an event happening within duration (0,t_2].

Contributions: We first discuss TQF, a naive way of processing temporal queries on Hyperledger Fabric history-db data. We discuss why this poses significant challenges (Section V). We then propose two models to efficiently process temporal queries on Fabric (Section VI and VII). We show that both models easily outperform the method TQF. We discuss the performance trade-off among TQF, model M1 and model M2 across various dimensions - storage, query performance, blockchain performance etc. To the best of our knowledge, ours is the first paper to investigate efficient handling of temporal queries on Hyperledger Fabric as well as on any blockchain platform.

II. The Hyperledger Fabric Architecture

The Fabric is a permissioned blockchain platform. The business logic that governs how different parties interact with each other is encoded as a chaincode. The chaincode executes transactions wherein a transaction can insert any number of key-value pairs as well as read the current states of any number of keys. For a key, a Fabric transaction persists only one state on the ledger. If a transaction ingests two pairs with the same key, only the latest state is persisted on the ledger and the earlier state is ignored.

The ingested key-value pairs are stored on state-db and history-db. For each key k, state-db stores its current state i.e., the latest pair with key k. The history-db records all states (i.e., all key-value pairs) including the current states. The state-db is located on a database (LevelDB or CouchDB) but the history-db data is distributed across a set of blocks on file-system. Each block contains the details of some transactions e.g., the key-value pairs inserted, transaction commit time, a link to previous block etc. We can access the states on Fabric as follows.

- **GetState(k)**: This call returns the current state of key k.
- **GetHistoryForKey(k) (GHFK)**: For key k, this call returns all the past states of key k in the history.
- **GetStateByRange(k_1,k_2)**: The state-db data is sorted on keys. Given a range, we can retrieve the list of keys in state-db which are within this range and their current states. We also call this a range scan query.

For each key k, the Fabric maintains a mapping of the blocks which contain the details of at least one transaction which ingested a key-value pair with key k. Thus given a key k, the Fabric knows the set of block-ids which together contain the details of all transactions which ingested a pair with key k. To process a GHFK call, the Fabric retrieves the list of these block-ids, deserializes the content of these blocks and extracts out the values inserted. These blocks are deserialized lazily i.e., a GHFK call returns an iterator and as more and more values are accessed through this iterator, more and more blocks are deserialized. If we stop accessing the iterator at a point, the blocks with the remaining values are not deserialized.

A. Why temporal analytics is inefficient on Fabric history-db?

The Fabric does not provide any indexing capability on the history-db data. There is hence no generic call to retrieve those historical values for a key which were ingested between two timestamps t_1 and t_2. To retrieve such values, one needs to execute a GHFK call which as discussed above requires scanning all states for key k between timestamps 0 and t_2. We need to then filter all the states which were ingested between t_1 and t_2. Larger the value of t_1, the more inefficient this operation gets as more and more redundant states in range (0,t_1] need to be retrieved.

III. Related Work

Efficient handling of temporal queries has been a very well researched topic in database community [5], [6], [7]. In this paper, we look at how we can improve the processing of temporal queries on-chain on Fabric. The BLOCKBENCH system [8] benchmarks Fabric [1], Ethereum [2] and Parity [3] against a set of database workloads. In this paper, we look at benchmarking Fabric against temporal workloads - an aspect not investigated previously. Some studies [9], [10] have also looked at performance studies of public blockchain systems e.g., Bitcoin [4]. As the analytics over blockchain data is not convenient, many systems have looked at taking blockchain data out and analyzing the data using a database e.g., [11], [12], [13] etc. Our goal is not to take the Fabric history-db data out. Rather we look at developing techniques for efficiently analyzing Fabric history-db data on-chain.

IV. Experimental Evaluation Setup

1) The Temporal Analytics Query: We consider the supply chain scenario discussed in section I involving shipments, containers and trucks. The key-value pairs describe the shipments load/unload events on/from containers and the container load/unload events on/from trucks. The state-db here contains the current state of each shipment and container. The history-db contains all the past states of the shipments and containers. This is a simple use-case however models various abstractions which we require to investigate how to improve the performance of temporal queries on Fabric.

We consider the following temporal join query Q. Given a duration $\tau : (t_s, t_e]$, for each shipment s, we want to find out the trucks which have ferried the shipment s during this duration and the associated time-intervals. This query is a good candidate to test a temporal analytics algorithm. The query involves analytics over all three entities, requires retrieving events of interest within a duration $(t_s, t_e]$ and also involves a join operation which is hard to optimize.

2) Synthetic Workloads and Data Ingestion: We write a synthetic data generator with following parameters (a) Number of shipments, containers and trucks (nS, nC, nTr), (b) Number of events for each key (nEv), (c) Distribution of load events (dEv), (d) Total time length within which all events lie (t_{max}). Given a load event ev, the corresponding unload event is randomly chosen at any point before the start of the next load event. In this paper, we use the following three synthetically generated datasets.

- **DS1:** Number of shipments, containers and trucks are 400, 100 and 20 respectively. Number of events for each key are 2K. Event distribution is uniform. Total time length is 150K. Total number of events hence are 1M.
- **DS2:** Same as DS1 except the event distribution is zipf. For each key, the zipf parameter is chosen randomly between 0 and 1.
- **DS3:** Same as DS1 but the number of shipments, containers and trucks are 15, 5 and 2 respectively. Total number of events hence are 40K.

We ingest these datasets on Fabric in two following ways. We first sort all the load/unload events on time and then execute transactions to sequentially insert these events.

- **Single Event (SE)**: We ingest one event in one transaction.

- **Multiple Events (ME)**: We ingest multiple events in one transaction. For each transaction, we choose a batch of events to ingest with each batch being a maximal set of consecutive events s.t. in this set no two events share the same key. Different transactions hence insert different number of events but for each shipment and container, one transaction can hence ingest at most one event. We do not include two events with the same key because as discussed in section II, one transaction on Fabric only persists one state for a key.

3) Fabric instance: We use Fabric release v1.0, single peer setup running on a Lenovo T430 machine with 4GB RAM, dual core Intel i5 processor. We use a single peer but we keep the consensus mechanism turned on. We use all default configuration settings to run our experiments.

V. Temporal Queries on Fabric (TQF)

In this section, we describe the method TQF to execute query Q (section IV-1) and demonstrate the issues encountered while executing any temporal analytics on Fabric history-db data. We first look at the state-db and retrieve the list of all shipments and containers using a range-scan query. For each shipment and container, we issue a GHFK call on the history-db. We scan each iterator returned by the GHFK calls and retain those states (i.e., pairs/events) which fall within duration (t_s,t_e]. We load this data into memory and compute the temporal join. Table I presents the performance numbers of TQF on datasets DS1, DS2 and DS3 (section IV-2). We move the query interval and note down the time taken by TQF to compute the temporal join as well as the time taken by GHFK calls while computing temporal joins. We first discuss the results for dataset DS1.

Consider the query interval τ=(0-10K]. There are 500 entities in play - 400 shipments and 100 containers and there are hence 500 GHFK calls. The join time is 12.2s and the majority of this time goes in executing 500 GHFK calls. This is along the expected lines. As discussed in section II, the GHFK calls require the deserialization of the blocks on file-system and hence this is a costly operation. These GHFK calls deserialize all blocks which contain the details of any transaction ingesting an event within duration (1-10K].

When the query interval is (10K-20K], TQF needs to deserialize all blocks which contain the details of any transaction ingesting an event within duration (0-20K]. As discussed in section II, Fabric does not support any temporal indexes and hence TQF does not know which blocks specifically contain the details of events ingested between (10K-20K]. TQF hence deserializes all blocks with events ingested within (0-20K] and loads those events in memory which are ingested within interval (10K-20K]. TQF hence approximately deserializes twice the blocks vis-a-vis when the query interval is (0-10K]. The join time hence increases as well.

As the query interval moves right, TQF needs to deserialize more and more blocks. The join time hence also increases steadily. The same trend is also present for dataset DS3. The dataset DS3 is smaller in size and the query times hence are smaller. Events in dataset DS2 are zipf distributed and more than half the events occur within interval (0-10K]. The join time hence is large and it takes 75s to compute the join. As the query interval moves right, the join time increases as expected. However the relative increase is smaller as compared to DS1. This is because events in dataset DS2 are zipf distributed and the number of events decrease as the query interval moves right. The bottleneck with TQF is that to retrieve events happening within query interval (t_s,t_e], we need to deserialize blocks containing events within duration (0,t_e]. Larger the value of t_s, worse is hence TQF's performance. We next present two models to overcome this bottleneck.

VI. Model M1

In this section, we present our first model for creating temporal indexes on Fabric. Let $\mathcal{EV}(k,\theta)$ represent the set of events which (1) belong to key k and (2) happen during the time-interval θ. In this model, we ingest additional key-value pairs of the form $\langle(k,\theta),\mathcal{EV}(k,\theta)\rangle$ on Fabric. These pairs are not part of the business logic and are generated only to accelerate the processing of temporal queries. In this section, we describe the details of how exactly we choose the intervals θ and ingest these pairs on Fabric. We call this activity as indexing process. We call these pairs *indexes* as they allow fast retrieval of events within any query interval and call the associated intervals θ as *index-intervals*.

1) Index Construction Process: Model M1 runs this indexing process periodically. Let t_1 be the timestamp when the indexing process last ran and let t_2 be the timestamp when the indexing process runs next. For each key k (i.e., each shipment and container), the indexing process divides the time-interval $(t_1,t_2]$ in a set of disjoint indexing intervals $\Theta(k)$={θ_1,θ_2,...,θ_m}. The number m and the intervals $\Theta(k)$ can be different for each key. For each key k and for each indexing interval θ in $\Theta(k)$, the indexing process executes a transaction which constructs the set $\mathcal{EV}(k,\theta)$ and ingests the pair $\langle(k,\theta),\mathcal{EV}(k,\theta)\rangle$ on Fabric. The indexing process then executes a second transaction which ingests the pair $\langle(k,\theta),\text{“”}\rangle$. This changes the current state of the "newly formed key" (k,θ) from $\mathcal{EV}(k,\theta)$ to null. This operation hence removes the set $\mathcal{EV}(k,\theta)$ from state-db, replacing it by null. The set $\mathcal{EV}(k,\theta)$ hence remains accessible only from history-db. Secondly, these two pairs are ingested only if the set $\mathcal{EV}(k,\theta)$ is not empty. We remove the set $\mathcal{EV}(k,\theta)$ from state-db so that state-db size remains minimal. We design our indexing models s.t. the content added on state-db for indexing purposes is minimal.

2) Temporal Analytics using M1 Indexes: We can efficiently execute temporal queries using these indexes. Consider we want to retrieve events within a query-interval τ for key k i.e., the set $\mathcal{EV}(k,\tau)$. We first find out, the index intervals created for key k i.e., $\Theta(k)$. We next find out index intervals in $\Theta(k)$ which overlap with query-interval τ. Let $\mathcal{O}(\Theta(k),\tau)$ be this set of overlapping intervals. For each indexing interval θ in $\mathcal{O}(\Theta(k),\tau)$, we execute a GHFK($k$,$\theta$) call, collect the

TABLE I
PERFORMANCE COMPARISON - MODEL M1 VS TQF VS MODEL M2

Query Interval	Dataset DS1, Ingestion with ME								Dataset DS2 with ME			Dataset DS3 with SE		
	Model M1		TQF		Model M2				Model M1	TQF	Model M2	Model M1	TQF	Model M2
	Join Time (s)	GHFK Time (s) and Calls	Join Time (s)	GHFK Time (s) and Calls	Join Time(s)		GHFK Time (s) and Calls		Join Time (s)			Join Time (s)		
	u=2K	u=2K	-	-	u=2K	u=50K	u=2K	u=50K	u=2K	-	u=2K	u=2K	-	u=2K
1-10K	3.8	3.0 (2500)	12.2	10.0 (500)	12.9	16.7	11.7 (2500)	16.0 (500)	6.4	75.7	72.5	0.25	0.39	0.42
10K-20K	3.7	3.0 (2500)	19.4	17.2 (500)	12.6	23.6	11.5 (2500)	22.9 (500)	4.6	95.3	21.5	0.27	0.61	0.39
20K-30K	3.9	3.2 (2500)	29.8	27.3 (500)	12.5	34.2	11.3 (2500)	33.4 (500)	4.2	103.6	10.5	0.27	0.84	0.41
60K-70K	3.8	3.2 (2500)	59.5	57.1 (500)	13.7	20.9	12.6 (2500)	20.3 (500)	3.8	115.2	4.0	0.24	1.73	0.42
70K-80K	3.7	3.0 (2500)	71.4	68.1 (500)	12.9	32.2	11.8 (2500)	31.4 (500)	3.8	116.5	3.6	0.24	2.23	0.46
80K-90K	3.8	3.1 (2500)	79.7	76.2 (500)	13.0	39.9	11.9 (2500)	39.1 (500)	3.6	116.4	3.5	0.26	2.33	0.40
120K-130K	3.8	3.1 (2500)	108.3	104.6 (500)	12.1	31.1	10.9 (2500)	30.4 (500)	3.3	121.4	3.1	0.24	3.26	0.38
130K-140K	3.5	2.8 (2500)	113.8	109.5 (500)	12.9	37.6	11.7 (2500)	36.5 (500)	3.0	122.1	2.9	0.26	3.34	0.37
140K-150K	3.7	3.0 (2500)	120.8	113.7 (500)	12.2	43.6	10.9 (2500)	42.3 (500)	2.7	120.7	2.7	0.22	3.38	0.33

events from these GHFK calls and remove those events which do not fall within the query-interval τ.

Note that each GHFK(k,θ) call deserializes only one block as the indexing process has collected all events in set $\mathcal{EV}(k,\theta)$ together and stored them in a single key-value pair. In absence of model M1 indexes, the events in set $\mathcal{EV}(k,\theta)$ are scattered across many blocks and hence construction of set $\mathcal{EV}(k,\theta)$ requires deserialization of many blocks. Construction of the set $\mathcal{EV}(k,\tau)$ hence requires deserialization of $|\mathcal{O}(\Theta(k),\tau)|$ blocks which is much smaller than the number of blocks deserialized in TQF. This provides improvement vis-a-vis TQF.

3) Creating Index Intervals: In this paper, we adopt a simple strategy to create indexing intervals $\Theta(k)$ for a key k. Say $(t_1, t_2]$ is the range that needs to be partitioned. We partition the range $(t_1, t_2]$ into disjoint index intervals of fixed length u. Each interval hence contains different number of events. Many other ways of creating indexing intervals are possible and we plan to explore them as part of future work.

A. Experimental Evaluation

Table I benchmarks the performance of model M1 indexes on datasets DS1, DS2 and DS3. We build model M1 indexes after all events have been ingested. There is hence only one invocation of the indexing process. We take the index interval length u to be 2K. We first discuss the results on dataset DS1. For each key k, the model M1 ingests key-value pairs of form $\langle(k,\theta),\mathcal{EV}(k,\theta)\rangle$ wherein θ takes values in {(0-2K], (2K-4K], ..., (148K-150K]}. For any query interval τ of length 10K, model M1 needs to make 5 GHFK calls over these indexes which results in deserialization of 5 blocks. This results in a significant improvement vis-a-vis TQF.

Consider the case when query interval τ is (0-10K]. The temporal join time is 3.8s. For each key k, model M1 collects events in intervals of length 2K and stores them together. Events in these indexing intervals of length 2K hence can be retrieved by accessing one block only. Model M1 makes a total of 2500 GHFK calls but each call deserializes only one block. In comparison, in TQF, these events are scattered across multiple blocks and all these blocks needs to be accessed to collect the relevant events. TQF makes 500 GHFK calls but each call requires deserialization of multiple blocks. Model M1 hence deserializes less number of blocks vis-a-vis TQF and hence provides better performance (3.8s vs 12.2s).

This performance gap widens as the query interval τ moves right. Consider the case when query interval τ is (10K-20K]. TQF needs to deserialize all blocks containing events within (0-20K] but model M1 still needs to access 5 blocks for each key. TQF hence takes 19.4s to execute the join while model M1 takes 3.7s. As events in dataset DS1 are uniformly distributed, model M1 approximately takes the same time for each query interval of length 10K but the TQF time keeps increasing as the query interval moves right. When query interval is (140K-150K], the performance gap is maximum (120.8s for TQF vs 3.7s for model M1).

We obtain the same trends for datasets DS2 and DS3. Dataset DS3 is smaller and the query times are smaller as well. Events in dataset DS2 are zipf distributed. As the query interval moves right, we encounter lesser and lesser events. Model M1 join time hence decreases as the query interval moves right. However, the TQF time still increases as discussed in section V.

1) Impact of Varying u on Performance: We next look at how model M1 performs if indexing parameters u is changed. We repeat the table I experiment for dataset DS1 and vary the parameter u. We take two query intervals - τ = (0-40K] and τ = (20-90K] and note the model M1 join time. Table II presents the results. The trends are along expected lines. As u increases, the indexes store a larger set of events together. We need to issue lesser number of GHFK calls which implies we need to deserialize a lesser number of blocks. The join times hence steadily decrease.

TABLE II
MODEL M1 PERFORMANCE ON VARYING INDEXING PARAMETER u

Dataset DS1, Ingestion using ME		
u	τ=(20-90K]	τ=(0-40K]
2K	26.9	15.5s
10K	9.2s	5.1s
50K	8.5s	4.0s

2) The cost of Model M1 Indexes: As the model M1 includes a separate indexing phase, this increases the overall data ingestion time. The indexing process involves executing costly GHFK calls as well as two transactions to ingest each index. For dataset DS1 in table I, the model M1 index construction process took ∼8.5mins. The data ingestion time for TQF was ∼134mins. So the index construction process increased the overall ingestion time by ∼6%.

This effect gets accentuated when there are multiple invocations of the index construction process. Note that unlike our experiments, the data may be continuously streaming in and there is hence no one time when we can build the indexes in one go. The indexing process hence needs to happen periodically. Each successive invocation is costlier than the previous one. To illustrate this, we build model M1 indexes on DS1 at a frequency of 25K timestamps. We measure the index construction time for each invocation, time taken to ingest events since last invocation and the total time elapsed at the end of each invocation. Table III presents the results.

TABLE III
IMPACT OF INDEX CONSTRUCTION ON DATA INGESTION TIME

Dataset DS1, Model M1 Indexes, Ingestion using ME, u=2K			
Timestamp	Index Construction Time	Data Ingestion Time since last index	Total Data Ingestion Time
25K	6m48s	21m46s	28m34s
50K	7m13s	22m16s	58m3s
75K	7m25s	23m6s	88m34s
100K	8m4s	22m54s	119m32s
125K	8m30s	23m55s	151m57s
150K	8m28s	20m25s	180m50s

Note that each invocation builds the indexes for past 25K timestamps but needs to process all the data ingested. The index construction time hence increases at each invocation. After 6 invocations, the total index construction time is ∼46 mins while the total time taken to ingest the events is ∼134min. Index construction time is hence ∼34% of the data ingestion time. This approach is clearly not scalable. We next discuss indexing model M2 which does not have a separate indexing phase and hence does not have these short-comings.

VII. Model M2

In this model, we store the indexing interval information along-with each key-value pair being ingested on Fabric. An incoming key-value pair $\langle k,(v,t)\rangle$ is transformed to $\langle (k,\theta),(v,t)\rangle$ wherein θ is an indexing interval s.t. the timestamp t lies within index interval θ. We discard the incoming pair $\langle k,(v,t)\rangle$ and only store the transformed pair. This way, the indexing information gets stored on Fabric history-db and unlike model M1 we do not need to create additional key-value pairs for creating temporal indexes. Model M2 hence does not require a separate indexing phase and rather embeds the indexing information during the event ingestion time itself.

To create indexing intervals θ, we adopt the same strategy as model M1 i.e., we choose intervals of fixed size u. For a key-value pair $\langle k,(v,t)\rangle$, we associate the index interval $(\lfloor \frac{t}{u} \rfloor * u$, $\lceil \frac{t}{u} \rceil * u]$. Each indexing interval of length u will hence have different number of events. Note than unlike model M1, we do not store events within an index interval together as part of a single key-value pair. Events in model M2 are still distributed across multiple blocks. However model M2 introduces a mechanism whereby to retrieve events within a query interval, we do not need to scan all events from t=0.

1) Temporal Analytics using Model M2 Indexes: To retrieve the events for key k which lie within a query interval τ i.e., the set $\mathcal{EV}(k,\tau)$ we use the following approach. From state-db, we find out all indexing intervals for key k which overlap with τ. This is done using a range-scan query on state-db. For each such interval θ, we execute a GHFK call for (k,θ). We collect the events from these GHFK calls and remove the events which do not fall within the interval τ. Note that each GHFK call for (k,θ) in model M2 is precisely going to access those blocks on flie-system which contain info regarding original key-value pairs with key k which were ingested at a timestamp within interval θ. As a result, GHFK call (k,θ) in model M2 takes smaller time vis-a-vis GHFK call for a key k in TQF.

A. Experimental Evaluation

Table I presents the performance numbers for model M2. We discuss results on dataset DS1 first. We try model M2 indexes with two different index interval lengths - 2K and 50K. We first discuss the results with u=2K. When the query interval τ is (0-10K], the model M2 takes 12.9s which is similar to the time taken by TQF. This is because both approaches deserialize similar number of blocks. When the query interval is (10K-20K], TQF needs to scan all events within duration (0-20K]. In model M2, we need to deserialize only blocks containing events within (10K-20K]. The model M2 deserializes fewer blocks then TQF and this manifests in significantly improved timings of model M2 vis-a-vis TQF.

We are able to achieve this improvement because with model M2 indexes, we exactly know which set of blocks are going to contain events within interval (10K-20K]. Specifically we retrieve the history of keys (k,(10K-12K]), (k,(12K-14K]), (k,(14K-16K]), (k,(16K-18K]) and (k,(18K-20K]). With TQF, we do not have this information and we need to resort to a full GHFK scan. This effect becomes more severe as the index interval τ moves right. However model M2 takes more time than model M1. This is because unlike model M1, model M2 does not store events within an indexing interval together in a single key-value pair. The distribution of events across blocks in model M2 is statistically same as TQF. Unlike model M1, a GHFK call in model M2 hence needs to access multiple blocks.

We next look at the results with u=50K. For τ=(0-10K], the TQF and model M2 times are similar. Same is the case for query intervals (10K-20K] and (20K-30K]. However when the query interval is (60K-70K], the model M2 needs to deserialize blocks with events within (60K-70K] only while TQF needs to deserialize blocks with events within (0-70K]. There is hence a huge difference in the performance. We observe similar trends for datasets DS2 and DS3 as well.

B. The cost and trade-off with Model M2 Indexes

From the above discussion, it is clear that smaller the index interval length u, better is the performance. If the index intervals are large, the query performance when the query interval τ is small vis-a-vis the indexing interval θ may not be efficient. However the associated cost is the increased state-db size. Smaller the value of u, larger is the size of state-db. If for a key k, n indexing intervals are created, the number of states on state-db increase by n-1.

The second cost of model M2 is that it does not allow the chaincode applications to run seamlessly with Fabric. This is because the model M2 has transformed the structure of the key-value pairs being ingested. If an application tries to query the state of key k, it will not find it on state-db. This was not an issue with the model M1 because model M1 created a set of new key-value pairs for indexing purposes and it did not modify the original data being ingested on Fabric. The blockchain applications hence need to use the API exposed by model M2. We next discuss how the GetState(k) and GHFK(k) calls on the base data can be simulated in terms of GetState(k, θ) and GHFK(k, θ) calls on the transformed data.

1) Accessing Original States: Let $\Theta(k)$ be the set of indexing intervals created for key k, such that for each interval θ in Θ, a key-value pair has been ingested on Fabric with key (k,θ). Let θ_{max} be the latest indexing interval in $\Theta(k)$. The current state of key k on base-data is then same as the current state of key (k,θ_{max}) on the transformed data.

The challenge hence is how to get the set $\Theta(k)$ for a key k and the interval θ_{max} in $\Theta(k)$. One option is to get the set $\Theta(k)$ using a range-scan query and then find out the interval θ_{max}. The second option is to start with the current indexing interval θ=($\lfloor\frac{t}{u}\rfloor$*$u$, $\lceil\frac{t}{u}\rceil * u$] and issue a GetState call on (k,θ). If we find a state for this key, we have found θ_{max}. If not, we compute the previous indexing interval θ=($\lfloor\frac{t-u}{u}\rfloor$*$u$, $\lceil\frac{t-u}{u}\rceil$*$u$] and repeat the process. We hence keep computing the previous indexing interval and stop as soon as we find a state on state-db. We take the second approach.

To get the historical states of key k, we need to issue a GHFK(k,θ) call for each indexing interval θ in $\Theta(k)$ and take a union of the results returned. To do this, we start with the current indexing interval θ and issue a GHFK call on (k,θ). We then find the previous indexing interval and issue a GHFK call. We repeat this process all the way to the first indexing interval (0,u] and take a union of all the results returned.

2) Cost of Accessing Original States: We term these calls as GetState-Base and GHFK-Base and study their performance. We take the dataset DS1 and build model M2 indexes with different values of u. We measure the time taken to execute 100K GetState-Base and 2K GHFK-Base calls. For each call, the key k is chosen randomly. We compare these numbers with the time taken to execute 100K GetState and 2K GHFK calls on base data. Table IV presents the results.

The GetState-Base time decreases as u increases. When u is small, for each key k, we need to query the state-db more times before we find the last indexing interval θ_{max} in $\Theta(k)$. As u increases, the number of required GetState calls decrease. Table IV also presents the number of GetState calls made. These numbers are mentioned within braces. For u=2K, a total of 329K GetState calls were made across 100K GetState-Base calls. For u=10K, this number decreases to 164K GetState calls. The GetState-Base time hence drops significantly. For u=50K and 75K, 100K GetState-Base calls needed exactly 100K GetState calls and the time is hence similar to the time taken for TQF. The GHFK-Base call times are similar across all values of u. This is because the distribution of events across blocks remains the same irrespective of the value of u used.

TABLE IV
IMPACT OF M2 INDEXES ON GETSTATE-BASE AND GHFK-BASE CALLS

Dataset DS1, Ingestion Using ME # GetState-Base Calls=100K, # GHFK-Base Calls=2K		
Index Interval Length (u)	GetState-Base Time	GHFK-Base Time
2K	172s (329K)	271s
10K	79s (164K)	250s
50K	50s (100K)	268s
75K	50s (100K)	243s
Base Data - GetState : 53s, GHFK : 249s		

3) Data Ingestion: The data ingestion time of model M2 is similar to the TQF time. This is because, model M2 neither executes any additional costly GHFK calls to build the indexes nor executes any additional transactions.

VIII. CONCLUSIONS

We presented two models for efficiently processing temporal queries on Hyperledger Fabric. We benchmarked these two models and comprehensively analyzed the performance tradeoffs involved. Both the models easily outperform the naive ways of handling temporal queries on Hyperledger Fabric.

In this paper, we created indexing intervals of identical length. We plan to explore more ways of creating model M1 and M2 indexes. We also plan to benchmark the performance of model M1 and M2 against workloads wherein each transaction also reads the current state of various keys. The approaches presented in this paper can also be generalized to other analytical queries e.g., spatial queries [14].

REFERENCES

[1] "HyperLedger Fabric. https://www.hyperledger.org/projects/fabric."
[2] "Ethereum Blockchain App Platform. https://www.ethereum.org/."
[3] "Ethcore. Parity: next generation ethereum browse. https://ethcore.io/parity.html."
[4] "Satoshi Nakamoto. bitcoin: A peer-to-peer electronic cash system."
[5] D. Gao and et. al., "Join operations in temporal databases," *VLDB J.*, vol. 14, no. 1, pp. 2–29, 2005.
[6] M. F. Mokbel and et al., "Spatio-temporal access methods," *IEEE Data Eng. Bull.*, vol. 26, no. 2, pp. 40–49, 2003.
[7] B. Chawda and et al., "Processing interval joins on map-reduce," in *EDBT*, 2014, pp. 463–474.
[8] T. T. A. Dinh and et al., "BLOCKBENCH: A framework for analyzing private blockchains," in *SIGMOD 2017*, 2017, pp. 1085–1100.
[9] C. Decker and et al., "Information propagation in the bitcoin network," in *P2P*, 2013, pp. 1–10.
[10] K. Croman and et al., "On scaling decentralized blockchains," in *Financial Cryptography and Data Security Workshop*, 2016.
[11] "Coinalytics https://www.crunchbase.com/organization/ coinalytics-co."
[12] D. Ron and et al., "Quantitative analysis of the full bitcoin transaction graph," in *Financial Cryptography and Data Security*, 2013, pp. 6–24.
[13] "BlockParser. https://github.com/mcdee/blockparser."
[14] H. Gupta and et al., "Processing multi-way spatial joins on map-reduce," in *EDBT*, 2013, pp. 113–124.