2018 IEEE 11th International Conference on Cloud Computing

On Building Efficient Temporal Indexes on Hyperledger Fabric

Himanshu Gupta, Sandeep Hans, Sameep Mehta, Praveen Jayachandran

IBM Research, India

higupta8@in.ibm.com, shans001@in.ibm.com, sameepmehta@in.ibm.com, praveen.j@in.ibm.com

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

In a prior work [1], we presented two models for creating temporal indexes on Fabric which overcome these limitations and improve the performance of temporal queries on Fabric. The first model creates a copy of each event inserted and stores temporally close events together on Fabric. The second model keeps the event count intact but tags metadata to each event s.t. temporally close events share the same metadata.

In this paper, we present variants on these two models which are better able to handle the skew present in Fabric data. We discuss the details and show that these variants significantly outperform the approaches presented in [1] when Fabric data contains skew. We also discuss the performance tradeoffs among these variants across various dimensions - data storage, query performance, event insertion time etc.

I. Introduction

A blockchain is a distributed, shared ledger that records transactions between multiple and often mutually distrusting parties in a verifiable and permanent way. Popularity of this technology has led to the development of a number of blockchain platforms e.g., Hyperledger Fabric [2], Ethereum [3], Parity [4] etc. As more and more transactions happen, the data on the blockchain systems may grow 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.

The Hyperledger Fabric and many other blockchain systems 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. 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. Fabric houses the state-db data in a database while the history-db data is distributed across a set of blocks on file-system. Each block contains the details of a set of transactions and a link to the previous block.

This paper concerns with the temporal analytics of data stored on Hyperledger Fabric [2] history-db. Currently any temporal analytics on Fabric history-db data is costly as Fabric exposes this data using a limited API and does not support any temporal indexing. We discuss the Fabric architecture and the API it exposes in section II in detail. Efficient processing of temporal queries requires quick retrieval of the events within any duration $[t_1,t_2]$. However, this operation is costly on Fabric. 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 those blocks which satisfy the following two conditions - (a) the block contains a transaction which ingested a key-value pair with key k and (b) this pair describes an event which happened before or at t_2 i.e., within duration $[0,t_2]$. Since we can't get the relevant events by accessing only those blocks containing events ingested within duration $[t_1,t_2]$ and we need to access all blocks containing events within duration $[0,t_2]$, this operation incurs significant cost on Fabric.

In a previous paper [1], we proposed two indexing models - M1 and M2, to mitigate this problem. The first model presented a paradigm wherein we collect events within suitably chosen intervals and ingest them on Fabric as part of additional key-value pairs. This data is then used to accelerate the processing of temporal queries. The second model presented a paradigm wherein we don't ingest additional pairs but tag suitably chosen intervals as metadata to the events being ingested on Fabric. Using experiments on uniformly distributed data, we discussed how these two models simulate the construction of temporal indexes on Fabric , allow faster retrieval of Fabric history-db data in any query interval $[t_1,t_2]$ and hence allow efficient execution of temporal queries on Fabric.

In this paper, we build on this work and investigate the performance of models M1 and M2 when the data contains skew and is not uniformly distributed. We propose new variants which provide better performance on datasets containing skew. These variants concern with how to choose indexing intervals while building model M1 and M2 indexes. The contributions of this paper are hence as follows.

- We review and summarize the models M1 and M2 developed in our previous work [1]. We further benchmark the performance of the models M1 and M2 on data which contains skew and is not uniformly distributed. We also discuss the challenges, models M1 and M2 face while indexing the skewed data (Section V, VI, VII)

2159-6190/18/$31.00 ©2018 IEEE
DOI 10.1109/CLOUD.2018.00044

- In [1], we discuss a variant of model M1 and M2 wherein index intervals of equal length are created. In this paper, we present two new variants for models M1 and M2. The first variant creates indexing interval in a way so as the number of events in each index interval are more than or equal to a threshold. The second variant takes both interval-length and event-count into account. We discuss the details, how these variants construct indexes (Section VI, VIII-A).
- Using experiments on synthetically generated data, we show that the index performance is better when we take both index interval length and the event count in account. We also discuss trade-offs among the three variants across various dimensions - storage, query performance, data ingestion on Fabric, index construction time etc (Section VI, VIII-A).
- For model M2, the proposed variants take a long time to build indexes. We next discuss a probabilistic version of these variants which builds the indexes much faster on Fabric history-db data while providing only a little degradation in the temporal query execution times. We also discuss its performance and the trade-offs involved (Section IX).

II. The Hyperledger Fabric Architecture

The Hyperledger Fabric is a permissioned blockchain platform. The business logic that governs how different parties interact or transact with each other is encoded as a chaincode. A chaincode executes transactions which ingest events on Fabric in form of key-value pairs. A chaincode can insert any number of key-value pairs on Fabric 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 will not be recorded as part of history-db data.

The details of a set of transactions are stored together to form a block. Transaction details include the key-value pairs ingested, commit time, response status etc. Fabric enforces read-write conflicts on the transactions being part of the same block i.e., if a transaction modifies the state of key k, no other following transaction in the same block can read the state of key k as well as modify the state of key k. However a following transaction in the same block can modify the state of key k if it does not read the state of key k.

A. State-db and History-db

The ingested key-value pairs are stored on state-db and history-db. For each key k, the state-db stores its current state i.e., the latest key-value pair with key k. The history-db records all the states (i.e., all pairs ingested on Fabric) 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.

The state-db contains the current states which Fabric transactions potentially modify. Smart contracts also access these states to make business decisions. Past states are neither modified by transactions nor accessed to make business decisions. The Fabric hence keeps the current states separate from past states so that the current states can be efficiently accessed and modified. This separation of past and current states helps in improving Fabric latency and throughput.

B. Accessing Fabric States

Fabric exposes the following API to access the states.

- **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 key-value pair with key k. To process a GHFK call, the Fabric first retrieves the list of these block-ids. It then 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 certain point, the blocks with the remaining values are not deserialized. A GHFK call retrieves the historical states from the blocks which have been committed. A GHFK call does not retrieve past states ingested by transactions which are part of the current block which is being formed and hence is yet to be committed.

C. 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 those 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 data management community [5], [6], [7]. In this paper, we look at how we can improve the processing of temporal queries on-chain on Fabric. Blockchain analytics is an emerging area and only a few studies have looked at the issues concerning analytics of blockchain data. Recently the BLOCKBENCH system [8] benchmarked the popular blockchain implementations - Fabric [2], Ethereum [3] and Parity [4] against a set of database workloads. Similar efforts

include - benchmarking Fabric and Ethereum against transactional workloads [9], Bitcoin performance measurements [10] etc. The focus of this paper is on temporal analytics of Fabric history-db data.

An alternative paradigm "off-chain analytics" has also been investigated wherein blockchain data is taken out, stored and analyzed using a database. For example - blockchain data analytics using Memsql [11], Bitcoin data analytics using databases [12], [13]. The off-chain analytics allows for an efficient processing of the data but has privact issues and an additional overhead of transferring the data from blockchain platform to an external database. The focus of this paper is on-chain temporal analytics of Fabric history-db data.

IV. Experimental Evaluation Setup

A. The Use-Case

We adopt the blockchain supported supply chain scenario as discussed in [1] 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. The state-db here hence contains the current state of each shipment and container i.e., the container containing the shipment and the truck ferrying the container. The history-db data contains all the past states of the shipments and containers i.e, for each shipment, the set of containers it has been contained in and for each container, the set of trucks which ferried it at some point in time. This use-case models various abstractions needed to investigate various issues concerning temporal analytics on Fabric.

B. The Temporal Analytics Query

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 requires retrieving events of interest within a duration $[t_s, t_e]$ and also involves a join operation which is costly to execute.

C. Synthetic Workloads

We carry out our experimental evaluation using synthetically generated data. We write a synthetic data generator which generates a set of shipment load/unload events on/from containers and container load/unload events from trucks. The parameters are: (a) Number of shipments, containers and trucks, (b) Number of events for each shipment and container (c) Distribution of load events, (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 two synthetically generated datasets containing skew.

- **SD1:** (a) Number of shipments, containers and trucks are 400, 100 and 20 respectively. Total time length is 150K. Number of events for each key are 2K. Total number of events hence are 1M. Events for each keys are zipf distributed over this temporal range. For each key, the zipf parameter is chosen randomly between 0 and 1.
- **SD2:** (a) Same as SD1 but the number of shipments, containers and trucks are 15, 5 and 2 respectively. Number of events for each key are 2K. Total number of events hence are 40K.

Presence of zipf distribution implies that more events arrive in the start and the data gets more and more sparse as time progresses. This hence allows us to study how various models (and their variants) handle the skew. For both datasets, half of the events arrive within first 10K timestamps.

D. Fabric instance

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

E. The event insertion scheme

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

1) Single Event (SE): We ingest one event in one transaction. Dataset SD2 is ingested using this scheme.

2) Multiple Events (ME): We ingest multiple events in one transaction. Dataset SD1 is ingested using this scheme. 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 ingests 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. For dataset SD1, each transaction ingests an average of 38 events. Total number of transaction are $\sim$25K with 1193 transaction ingesting more than 100 events and 2154 transactions ingesting less than 10 events.

F. Metrics for Measuring the Performance of a Model

In this paper, we measure the performance of a model using the following three metrics - (a) Query execution times - time taken to execute the temporal query. (b) Ingestion times - time taken to ingest data on Fabric and (c) State access times - time taken to execute GetState and GHFK calls.

V. Temporal Join Performance on Fabric

In this section, we discuss how we execute temporal join query Q (section IV-B) on Fabric and discuss the performance on datasets SD1 and SD2. 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 only 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. We call this method TQF. Table I presents the performance numbers.

Consider the query interval τ=(0-10K]. This requires 500 GHFK calls as there are 400 shipments and 100 containers. The join time is 75s and the GHFK calls take up the majority of this time (71s). As mentioned in section IV-C, more than 50% of the events in dataset SD1 are within duration (0-10K] and it hence requires deserialization of lot of blocks to load all relevant events. These GHFK calls deserialize all blocks containing an event within duration (0-10K].

When the query interval is (10K-20K], we need 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 we do not know which blocks specifically contain the details of events ingested between (10K-20K]. The join time hence increases and we need 95.3s to compute the join. In general, to retrieve events 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 the performance. The join time hence steadily rises and the query interval moves right. When the query interval is (140K-150K], we require 120.8s to compute the join even though there are only ∼9K events within this interval. In prior work [1], we proposed two indexing models to mitigate these issues, which we discuss next.

VI. Model M1

In this section, we first summarize the model M1 proposed in [1] for creating temporal indexes on Fabric. We then benchmark its performance on skewed datasets SD1 and SD2, and present new variants of this model which are better able to handle the skew present in datasets SD1 and SD2.

A. Index Construction Process

Let $\mathcal{E}(k,\theta)$ represent the set of events which (1) belong to key k and (2) happen during time-interval θ. Let (0,t] be the time duration within which all events arrive. For each key k (i.e., each shipment and container), the indexing process divides the time-interval $(0,t]$ in a set of disjoint indexing intervals $\Theta(k)$=$\{\theta_1,\theta_2,...,\theta_m\}$. The number m and the intervals $\Theta(k)$ can be different for each key. For each key k, the indexing process executes a transaction which (a) first executes a GHFK call on key k, (b) constructs the set $\Theta(k)$ and (c) for each indexing interval θ in $\Theta(k)$, constructs the set $\mathcal{E}(k,\theta)$ and ingests the pair $\langle(k,\theta),\mathcal{E}(k,\theta)\rangle$ on Fabric.

The indexing process then executes a second transaction which ingests the pair $\langle(k,\theta),$ ""$\rangle$. This changes the current state of the "newly formed key" (k,θ) from $\mathcal{E}(k,\theta)$ to null. This operation hence removes the set $\mathcal{E}(k,\theta)$ from state-db. The set $\mathcal{E}(k,\theta)$ hence remains accessible only from history-db. Secondly, these two pairs are ingested only if the set $\mathcal{E}(k,\theta)$ is not empty. We remove the set $\mathcal{E}(k,\theta)$ from state-db so that state-db size remains minimal. Otherwise, the state-db will contain all the states and its size will hence blow up. We hence design our indexing models s.t., the content added on state-db for indexing purposes is minimal.

B. Temporal Analytics using M1 Indexes

Temporal queries can now be efficiently executed using these model M1 indexes. Consider we want to retrieve events within a query-interval τ for key k i.e., the set $\mathcal{E}(k,\tau)$. We first find out from state-db, 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. For each overlapping indexing interval θ, we execute a GHFK(k,θ) call, collect the events from all 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 $\mathcal{E}(k,\theta)$ together and stored them in a single key-value pair. In absence of model M1 indexes, the events in set $\mathcal{E}(k,\theta)$ are scattered across many blocks and hence construction of set $\mathcal{E}(k,\theta)$ requires deserialization of many blocks. This hence significantly speeds up the processing of temporal queries.

C. Variants for M1 Indexes

We next discuss three variants of model M1 indexes. These variants concern with different mechanisms for creating index intervals $\Theta(k)$ for key k. Out of these three, FL variant was discussed in [1]. In this section, we discuss the performance of FL variant on skewed datasets SD1 and SD2. We also propose two new variants - FC and ALC which improve on FL variant on skewed datasets.

- **Fixed-Interval-Length (FL)**: This partitions the range $(0,t]$ into disjoint index intervals of fixed length u. Each interval hence contains different number of events.
- **Fixed-Event-Count (FC)**: This scheme partitions the range $(0,t]$ into disjoint index intervals s.t., each interval contains a fixed number of events v. Each such index interval is hence of different length.
- **Adaptive-Length-Count (ALC)**: Given an interval length u and an event count v, this scheme partitions the range $(0,t]$ s.t. each interval satisfies one of the two conditions - (a) Index interval is of length u but the event count within this interval is greater than or equal to v and (b) The event count within this interval is v but the interval length is greater than or equal to u.

D. Experimental Evaluation

Table I benchmarks the performance of these indexing mechanisms and TQF on dataset SD1 (section IV-C). We ingest the data using ME scheme (section IV-E). We take a 10K sized query interval and vary its start and end points as shown in Table I. For FL and ALC scheme, we set the interval length parameter u to 2K. Assuming the event distribution was uniform, this would have implied 27 events within each index interval. For FC and ALC scheme, we therefore set the event count parameter v to 27. The indexes are built once when all the events have been ingested.

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

Dataset SD1, Ingestion with ME													Dataset SD2 with SE			
	Model M1 (u=2K, v=27)						Hyperledger Fabric		Model M2 (u=2K, 50K)				Model M1 (u=2K, v=27)			
Query Interval	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)			
	FL	FC	ALC	FL	FC	ALC	TQF	TQF	FL (2K)	FL (50K)	FL (2K)	FL (50K)	FL	FC	ALC	TQF
1-10K	6.4	15.4	6.3	3.6 (2500)	12.7 (10401)	3.7 (3000)	75.7	71 (500)	72.5	68.4	69.3 (2.5K)	65.3 (500)	0.32	1.11	0.33	1.93
10K-20K	4.6	12.0	4.9	3.5 (2500)	10.9 (7488)	3.9 (3440)	95.3	91 (500)	21.5	86.3	20.2 (2.5K)	85.2 (500)	0.18	0.4	0.25	2.63
20K-30K	4.2	5.3	4.1	3.5 (2500)	4.8 (3881)	3.6 (3180)	103.6	100 (500)	10.5	88.0	9.7 (2.5K)	87.5 (500)	0.14	0.22	0.23	2.70
60K-70K	3.8	2.2	2.0	3.4 (2499)	2.0 (1643)	1.9 (1639)	115.2	111 (500)	4.0	4.9	3.5 (2.5K)	4.8 (500)	0.15	0.09	0.12	3.04
70K-80K	3.8	2.0	1.9	3.4 (2481)	1.9 (1507)	1.8 (1500)	116.5	112 (500)	3.6	5.8	3.2 (2.5K)	5.6 (500)	0.12	0.08	0.08	3.13
80K-90K	3.6	1.9	1.7	3.3 (2450)	1.8 (1423)	1.6 (1416)	116.4	111 (500)	3.5	6.5	3.0 (2.5K)	6.3 (500)	0.11	0.75	0.10	3.25
120K-130K	3.3	1.5	1.4	3.0 (2150)	1.4 (1158)	1.3 (1160)	121.4	117 (500)	3.1	3.7	2.5 (2.5K)	3.5 (500)	0.11	0.06	0.05	3.39
130K-140K	3.0	1.4	1.3	2.8 (1998)	1.3 (1076)	1.2 (1060)	122.1	117 (500)	2.9	4.1	2.5 (2.5K)	3.9 (500)	0.1	0.07	0.08	3.20
140K-150K	2.7	1.1	1.0	2.5 (1731)	1.0 (833)	0.9 (838)	120.8	116 (500)	2.7	4.4	2.3 (2.5K)	4.3 (500)	0.12	0.04	0.05	3.43

1) FL Performance: We first look at the results for FL indexing scheme. The index length parameter u is 2K. For each key k, the FL variant hence ingests key-value pairs of form $\langle (k,\theta), \mathcal{E}(k,\theta) \rangle$ wherein θ takes values in {(0-2K], (2K-4K], ..., (148K-150K]}. For any query interval τ of length 10K, the FL method needs to hence make 5 GHFK calls over these indexes which result in deserialization of 5 blocks. This hence results in a significant improvement vis-a-vis TQF and we needed 6.4s to compute the temporal join for query inteval (0-10K].

For τ=(10K-20K], FL can get all relevant events by making 5 GHFK calls on indexes. As query interval τ shifts right, we get lesser and less events due to zipf distributed data. For this reason, the join time decreases. This is in contrast with TQF where we need full scan to retrieve the relevant events and the join time hence steadily increases. The performance gap between FL and TQF hence widens as the query interval moves right. This shows the efficacy of FL approach.

2) FC Performance: We next look at the results for FC indexing scheme. The FC variant keeps the event count within each indexing interval constant. As there are more events at the start due to the zipfian nature of the data, FC creates more indexing intervals at the start. As we move right, lesser events are encountered and lesser indexing intervals are created. The length of the index intervals hence increases as we move right. An analysis of the index-intervals created revealed that the FC scheme created 10401 index intervals (for all keys combined) within (0-10K] duration while it created 833 intervals within (140K-150K]. The join time hence decreases steadily from $\sim$15s for τ=(0-10K] to $\sim$1.1s for τ=(140K-150K]. In comparison, FL scheme created 2500 and 1731 index intervals within (0-10K] and (140K-150K] ranges respectively. Due to this reason, FC performs better than FL when the data is sparse (i.e., when τ is at the right end) and is worse then FL when the data is bursty (i.e., when τ is at the start of the temporal range). Table I also presents the number of index intervals created for each variant (numbers within braces)

3) ALC Performance: Finally we look at the ALC performance. The ALC scheme creates an index interval when both the FL and FC criteria are first met. This hence takes the best of both the FL and FC scheme. In the beginning, when the number of events are high, it creates indexes based on interval length and its performance is hence FL like. Towards the end, when the number of events are very small, it creates the indexes based on event count and its performance is hence FC like. Overall, its performance is hence better than FL and FC at all times as shown in Table I.

These results show that the ALC variant handles the data skew better then FL and FC variants. We obtain similar trends on dataset SD2 however the individual query times are lower as the data volume is smaller in SD2. If we use uniform distribution instead of zipf distribution in SD1 and SD2, the three variants perform equally well. We do not discuss these results due to lack of space.

E. The cost of Model M1 Indexes

Broadly, there are two costs associated with model M1 indexes. First, the number of key-value pairs on state-db increase and hence the storage cost increases. However ALC creates least number of index intervals and hence the ALC storage cost is smaller than FL and FC variants. On dataset SD1, FL. FC and ALC variants created $\sim$35K, $\sim$36K, $\sim$14K number of additional states respectively. On dataset SD2, these numbers are $\sim$14K, $\sim$14K, $\sim$6K respectively.

The second cost is index construction time. As the model M1 includes a separate indexing phase, this increases the overall data ingestion time. Each invocation of indexing process executes one GHFK call for each key k. For each key, it further executes two transactions to ingest the index. On dataset SD1, FL, FC and ALC index construction took $\sim$10, $\sim$13 and $\sim$12 mins respectively. The data ingestion time excepting index construction was $\sim$109mins. The index construction time for the three variants is similar as they execute same number of transactions as well as GHFK calls to construct the indexes.

Note that unlike our experiments, the data may be continuously streaming on Fabric and there is hence no one time when we can build the indexes in one go. The indexing process hence needs to happen periodically. For example, we can choose to build model M1 indexes after each 25K timestamps and each time, we index the data that arrives in previous 25K timestamps. Note that, each invocation of index construction process is costlier than the previous one. This is

because for each invocation we need to issue a full GHFK scan for each key and the GHFK cost keeps increasing with time. We built ALC indexes for dataset SD1 after each 25K timestamps. There were hence a total of 6 invocations and these 6 invocations took a total of 38mins. This is a big bottleneck. We next discuss the second model M2 proposed in [1] which does not have a seperate indexing phase and hence improves on this aspect.

VII. Model M2

In this section, we first review model M2 proposed in [1], for creating temporal indexes on Fabric. We then discuss its performance on skewed datasets SD1 and SD2.

A. Index Construction Process

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 event ingestion itself. 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) FL Approach: Similar to model M1 - FL approach, we choose intervals of fixed size u. For an incoming 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.

B. Temporal Analytics using Model M2 Indexes

We use the following approach to retrieve the events for key k, which lie within a query interval τ i.e., the set $\mathcal{E}(k, \tau)$. From the state-db, we find out all indexing intervals for key k which overlap with τ i.e., all intervals θ s.t. there is a state for key (k,θ). 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 query interval τ.

C. Experimental Evaluation

Table I presents the performance of model M2. We try the FL scheme with two different index interval lengths (u) - 2K and 50K. We first discuss the results with u=2K. When query interval τ is (0-10K], the model 2 takes ∼72 s which is the same time taken by TQF. This is because both approaches deserialize similar number of blocks. When the query interval is (10-20K], TQF needs to scan all events within duration (0-20K]. While in model M2, we need to deserialize only blocks containing events within (10K-20K]. The model M2 hence deserializes fewer blocks then TQF and this manifests itself 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 call. This effect becomes more severe as the index interval τ moves right. Due to zipfian data, the model M2 join time steadily decreases however the TQF time steadily goes up. For τ=(140K-150K], model M2 takes only 2.7s to compute the join while TQF requires 120s.

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], ..., (40K-50K]. 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 significant difference in the performance.

VIII. Model M2 - MinCount and Adaptive Variants

Similar to model M1, we outline the Min-Count (MC) and Adaptive-Length-Count (ALC) variants for model M2 which are better able to handle skew in the data.

1) MinCount(MC): This variant creates a new index-interval when the event-count within the current index-interval excluding events which are part of the current block being formed, is greater than or equal to a threshold v.

2) Adaptive-Length-Count(ALC): Given an interval length u and an event count v, this variant creates a new index-interval when one of the following two conditions is satisfied - (a) The interval-length is greater or equal to u , and (b) The event-count within the current indexing interval excluding events which are part of the current block being formed, is greater than or equal to v.

A. Implementation of M2 Variants

Note that to count the number of events within an interval, we need to issue a GHFK call. As discussed in section II-B, a GHFK call returns the events which are part of the committed blocks. Due to this reason, the MC and ALC variants are designed so as not to take into account the events which are part of the block currently being formed. We implement the MC variant as follows.

For each key k, we create a new state (k,"$index$"). This state tracks, how many index intervals have been created for key k and the start-point of the current index interval. Let IC(k,t) represent the index-counter for key k at time t. The index-counter IC(k,t) starts with 0 for each key k and keeps getting incremented by 1 with the creation of each new index-interval for key k. Let S(k,t) represent the start-point of the index-interval at timestamp t for key k i.e., the interval represented by IC(k,t). Whenever a new index-interval

is created for key k at time t, the variant MC ingests a key-value pair ((k,"$index$"),(IC(k,t),S(k,t)).

The variant MC transforms each incoming pair (k,(v,t)) to the form ((k, IC(k,t)), (v,t)). All events in the same index-interval hence share the same index-counter. For each incoming pair (k,(v,t)), MC executes the following sequence of operations.

1) MC executes a GHFK call on (k,"$index$") and obtains the current index-counter for key k. This is obtained from the last state of the result returned by GHFK call. Let this counter be ic.
2) MC then executes a GHFK call on (k, ic) and counts the number of events in the current index-interval.
3) If this count is less than v, MC ingests the pair ((k, ic), (v,t)). Note that this event shares the same index-counter i.e., ic.
4) If this count is greater or equal to v, MC creates a new index-interval, increments the value of index-counter and ingests two pairs - ((k,"$index$"),(ic+1,t)) and ((k, ic+1), (v,t)). Note that a new index-interval starts at time-stamp t and hence S(k,t)=t and IC(k,t)=ic+1.

Following points need to be noted so as the above details are clear.

- Unlike FL scheme, we can't know apriori the end point of an index interval when it is formed. We therefore represent an index-interval using a counter and store the start-point as separate states (i.e., (k,"$index$")). The end point of an index interval is specified as the start point of the next index-interval minus 1.
- In step 1 above, we do not execute a *GetState* call to retrieve the index-counter for key k. This is because as mentioned in section II-B, Fabric enforces read-write conflict among the transactions in a block. A Fabric transaction hence is not allowed to read the current state for a key k if this state has been earlier modified by a transaction in the same block.

1) ALC Variant: The ALC variant is identical to the MC variant except in step 4 above, the ALC conditions (section VIII-2) are checked.

B. Temporal Analytics using MC and ALC variants

We use the following approach to retrieve the events for key k which lie within a query interval τ i.e., the set $\mathcal{E}(k,\tau)$. We execute a GHFK call on (k,"$index$") and obtain the index-counter of all index-intervals which overlap with the query interval τ. For each such counter ic, we execute a GHFK call for (k,ic). We collect the events from these GHFK calls and remove the events which do not fall within the interval τ. Note that each such GHFK call takes much smaller time vis-a-vis GHFK call for key k in TQF.

C. Experimental Evaluation

We next evaluate the performance of MC and ALC variants and compare the performance with FL variant. Table II presents the results. We again see the same trend that the ALC variant handles the data skew much better than the variants FL and MC. Consider the query interval τ=(10K-20K]. The FL variant has three index-intervals for all keys, all of length 50K, first of which is (0-50K]. FL variant hence needs to deserialize all blocks with events in range (0-20K]. The variant MC creates index-intervals based on event-count. As this dataset contains half the events within (0-10K], mostly the first index-interval created by MC for each key ends before the timestamp 10K. The variant MC hence does not need to deserialize many blocks with events in range (0-10K].

TABLE II
PERFORMANCE COMPARISON FOR MODEL M2 VARIANTS - FL VS MC VS ALC VS P-ALC

Index interval length u=50K, Event Count v=667, Time in seconds										
	Dataset SD1, Ingestion with ME					Dataset SD2, Ingestion with SE				
Query Interval	FL	MC	ALC	p-ALC p=0.1	p-ALC p=0.01	FL	MC	ALC	p-ALC p=0.1	p-ALC p=0.01
1-10K	68.4	73.3	76.7	69.8	66.7	2.1	2.4	2.8	2.0	2.1
10K-20K	86.7	49.0	57.5	54.4	49.3	2.5	3.6	3.3	1.6	1.4
20K-30K	88.1	26.3	36.9	30.3	32.7	2.8	1.6	1.9	0.9	1.1
60K-70K	4.9	25.9	27.7	29.9	25.0	0.2	1.0	1.3	1.0	0.7
70K-80K	5.8	26.2	9.9	16.5	22.1	0.2	0.9	0.30	0.7	0.6
80K-90K	6.5	25.9	7.5	11.5	19.7	0.3	0.9	0.3	0.4	0.6
120K-130K	3.6	25.9	5.5	6.9	14.2	0.17	0.87	0.25	0.2	0.5
130K-140K	4.0	24.7	4.8	6.3	12.6	0.18	0.97	0.22	0.2	0.5
140K-150K	4.4	22.3	4.2	4.8	9.9	0.24	0.99	0.17	0.2	0.4

We analyzed the index-intervals created for each key, found out the first index-interval for each key which overlaps with the range (10K-20K] and computed the average of the start-point of all such index-intervals. This turned out to be $\sim$5K. Hence on average MC needed to deserialize all blocks with events in range (5K-20K]. This manifests itself in much improved performance of variant MC. Variant MC hence handles the bursts in data better then FL.

Consider next the query interval τ=(130K-140K]. Due to zipfian nature of the data, there are lesser number of events in this range. The FL variant works better then MC variant for this case. When the data is sparse, MC creates lengthier index-intervals vis-a-vis FL. As a result, MC needs to deserialize larger number of blocks. This manifests in the better performance of FL. Here, FL and MC needed to deserialize all blocks with events in range (100K-140K] and (22K-140K] respectively. For MC, on average, the first index-interval overlapping with range (130K-140K] started at $\sim$22K.

The ALC variant takes the best of both the variants. When the data arrives in burst, it creates index-intervals based on fixed event-count and hence index-intervals of shorter length are created. When the data is sparse, index-intervals of fixed length are created and hence containing smaller number of events. As a result, its performance is competitive throughout. Table II presents the results for dataset SD2 as well and this presents similar trends.

D. The Cost of MC and ALC Variants

Similar to model M1, one of the cost is state-db storage. The three variants FL, MC and ALC created $\sim$1.5K, $\sim$1.5K and $\sim$2.5K states for dataset SD1. For dataset SD2, these numbers

are 56, 55 and 92 respectively. ALC in model 2, creates larger number of states vis-a-vis FL and MC variants. The second cost associated with MC and ALC variants is that the data ingestion becomes slow. This is because for each incoming pair, the MC and ALC variants issue 2 GHFK calls - one to retrieve the current index-interval for each key and one to get the count of events in the current index-interval for each key. These two GHFK calls incur significantly additional cost.

TABLE III
INGESTION TIME (MINS)

DataSet SD1						DataSet SD2					
FL	MC	ALC	TQF	p-ALC p=0.1	p-ALC p=0.01	FL	MC	ALC	TQF	p-ALC p=0.1	p-ALC p=0.01
105	664	616	121	183	129	170	191	191	156	172	171

Table III presents the ingestion times. As mentioned in section IV-C, for dataset SD1, an average of 39 events are ingested in each transaction. Each transaction on average hence executes 78 GHFK calls which adds up to a significant overhead. As a result, MC and ALC variant take much larger time to ingest the data on Fabric. For dataset SD2 run, we ingest only one event per transaction and hence each transaction executes 2 GHFK calls. The relative overhead is ALC variant is hence much smaller ($\sim$21mins) vis-a-vis FL.

These observations suggest that ALC variant is a good option for the case when each transaction ingests one or small number of events but its performance degrades as the number of events ingested per transaction increase. We next present a probabilistic version of ALC variant which significantly improves the ingestion time at the cost of a little degradation in the query performance.

IX. PROBABILISTIC ALC VARIANT (P-ALC)

The p-ALC variant checks the ALC condition only with a probability p. Whenever a key-value pair (k,(v,t)) arrives, the p-ALC variant generates a random number. If this number is greater than p, p-ALC finds out the current index-counter ic for key k and ingests the pair ((k,ic),(v,t)). If the random number is less than p, only then it checks the ALC condition (section VIII-2) and creates a new index-interval if the ALC condition is satisfied. This way, p-ALC avoids executing costly GHFK calls to find out the event count in the current index-interval for the instances when the generated random number is greater than p.

Table III presents the data ingestion times. We tried p-ALC variants with p=0.1 and 0.01. On dataset SD1, the data ingestion times for these two variants are 183 and 129 mins which are significantly improved on ALC time of 616 mins. On dataset SD2, these times are 172 and 171 mins which are equal to time taken by FL variant.

We next discuss the query performance of p-ALC variant. Table II presents the results. For dataset SD1, p-ALC variants perform better than MC variant for most query intervals. The p-ALC variants perform better than FL when the data is bursty and perform worse when the data is sparse. In general, the performance will degrade as the value of p goes down. We obtain similar trends for dataset SD2 as well.

X. CONCLUSIONS

In this paper, we discussed multiple approaches to index Fabric history-db data. We benchmarked these variants and analyzed the performance tradeoffs involved. The obtained insights can be summarized as follows.

1) For both models M1 and M2, the ALC variant provides best query performance. For model M1, FL works better than FC when the data is bursty while FC works better than FL when the data is sparse. For model M2, the MC variant works better than FL when the data is bursty while FL works better when the data is sparse. All variants however significantly outperform the TQF case, where no indexes are built.
2) The data ingestion times are minimum for model M2-FL variant. Model M2 - MC and ALC variants have large index construction overheads but these overheads can be significantly reduced by using their probabilistic variants. The performance of the probabilistic variant degrades as the value of parameter p goes down.
3) If the data volume on Fabric is small or the data arrives at a low enough rate, we can prefer model M1 indexes. If the data volume is large, model M2 indexes should be used. For model M2, FL or p-ALC variant can be chosen, depending on the data volume and arrival rate.

We next plan to study how temporal analytics can be efficiently carried out on other blockchain platforms e.g., Ethereum, TenderMint etc.

REFERENCES

[1] H. Gupta and et al., "Efficiently processing temporal queries on hyperledger fabric," in *ICDE*, 2018.
[2] "HyperLedger Fabric. https://www.hyperledger.org/projects/fabric."
[3] "Ethereum Blockchain App Platform. https://www.ethereum.org/."
[4] "Ethcore. Parity: next generation ethereum browse. https://ethcore.io/parity.html."
[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] P. Suporn and et al., "Performance analysis of private blockchain platforms in varying workloads," in *ICDCN*, 07 2017, pp. 1–6.
[10] "Satoshi Nakamoto. bitcoin: A peer-to-peer electronic cash system."
[11] "Coinalytics https://www.crunchbase.com/organization/ coinalytics-co."
[12] "BlockParser. https://github.com/mcdee/blockparser."
[13] D. Ron and et al., "Quantitative analysis of the full bitcoin transaction graph," in *Financial Cryptography and Data Security*, 2013, pp. 6–24.