TorqueDB: Distributed Querying of Time-series Data from Edge-local Storage*

Dhruv Garg[1], Prathik Shirolkar[2], Anshu Shukla[1], and Yogesh Simmhan[2][0000−0003−4140−7774]

[1] Ericsson Research, Bangalore, India
[2] Indian Institute of Science, Bangalore, India, simmhan@iisc.ac.in

Abstract. The rapid growth in edge computing devices as part of Internet of Things (IoT) allows real-time access to time-series data from 1000's of sensors. Such observations are often queried to optimize the health of the infrastructure. Recently, edge storage systems allow us to retain data on the edge rather than moving them centrally to the cloud. However, such systems do not support flexible querying over the data spread across 10–100's of devices. There is also a lack of distributed time-series databases that can run on the edge devices. Here, we propose *TorqueDB*, a distributed query engine over time-series data that operates on edge and fog resources. TorqueDB leverages our prior work on ElfStore, a distributed edge-local file store, and InfluxDB, a time-series database, to enable temporal queries to be decomposed and executed across multiple fog and edge devices. Interestingly, we move data into InfluxDB on-demand while retaining the durable data within ElfStore for use by other applications. We also design a cost model that maximizes parallel movement and execution of the queries across resources, and utilizes caching. Our experiments on a real edge, fog and cloud deployment show that TorqueDB performs comparable to InfluxDB on a cloud VM for a smart city query workload, but without the associated monetary costs.

1 Introduction

Internet of Things (IoT) domains leverage the availability of affordable sensing and computing devices, along with pervasive communications and advances in analytics, to observe and manage cyber-physical systems to enhance their efficiency and resiliency. IoT domains span physical infrastructure such as Smart Cities, Smart Transportation and Industrial IoT, to consumer devices such as smart watches and smart appliances. A key characteristic of IoT applications is their closed-loop cycle where data about the system is analyzed and decisions are made, typically within seconds, to control the system [11,15]. E.g., in a manufacturing facility, sensors may monitor the temperature and pollution levels to ensure it is safe for the workers, and if not initiate cooling, scrubbing or other safety measures.

* Supported by the DST ICPS Program, Government of India

Edge devices comparable to Raspberry Pi and Arduino are widely deployed as part of such IoT applications to help gather and transmit observations from the sensors, and also to enact control decision onto their co-located actuators [14]. Traditionally, data collected from the field are sent to the Cloud for storage and analytics, and the control signals are sent back to the field. This introduces high network round-trip latency from the edge to the cloud, and additional network, compute and storage costs at the cloud data center.

Edge computing has gained prominence to make use of the captive computing and storage on edge devices, as well as to reduce the network latency between the edge and the cloud for decision making. Besides running tasks and analytics on such devices [2, 3, 11], recent works also propose their use for distributed data storage by offering file and block-based semantics for data update and access [5, 9, 10]. They also use workstation-class *fog resources* located near the edge devices, which help with management and as a gateway to the Internet [13].

Motivation IoT data tends to be time-series in nature since sensors continuously generate timestamped data. As a result, time-series querying and analytics is a key requirement for IoT applications [8, 15]. These operate on data collected over time to check if recent observations exceed historic averages, identify minimum and maximum outliers within time-windows, and to query and visualize data from specific sensor types and time ranges. This complements and is more flexible than Complex Event Processing (CEP) and publish-subscribe systems that operate on streaming data and limit the queries possible [3, 7, 12]. *Time-series databases (TSDB)* like InfluxDB and Apache Druid are popular for hosting of such IoT data and performing temporal queries, centrally on the cloud [8].

However, both the sensor data producers and the consuming applications for such TSDBs tend to reside on edge devices. Edge applications require *sub-second* query latency when responding to dynamic situations. Moving data from the edge to a TSDB on the cloud, and querying it back from the edge causes *unreliable performance* due to WAN variability. It also introduces additional *network and VM costs*, and *privacy concerns* when data is moved out of the private network to public clouds. There is also a lost *opportunity cost* in not utilizing the captive compute, storage and network capacity available on edge and fog resources.

Requirements and Gaps A natural progression is to host such time-series databases on edge and fog devices, co-locating the query clients near the data storage and also leveraging the available local compute and storage capacity on them [1]. However, individual edge or fog resources may not have the capacity to scale to workloads from many edge clients. This requires the use of a *distributed* TSDB operating across multiple edge and fog devices. However, existing systems are either proprietary, do not support distributed execution, or are not light enough to be hosted on edge and fog resources. Further, not all time-series data collected over time will be actively used all the time. Given the overheads of managing distributed databases, only recent or actively used data should be

stored in such TSDBs. Lastly, data stored in the TSDB will need to complement storing the data durably as files on the edge. This may required to support time-series analytics or machine learning models that operate outside the database and directly on files hosted on the edge devices [15]. We address these gaps.

Contributions We propose *TorqueDB (Temporal querying from edge storage Database)* which leverages the *ElfStore* distributed edge-local storage [9] along with *InfluxDB* TSDB to offer a distributed execution model for time-series queries over edge and fog devices. Here, ElfStore retains the persistent time-series data generated by sensors on the edge devices while InfluxDB instances running on the fog are used to host subsets of this data, on-demand, to support user queries. TorqueDB accepts queries defined using the *Flux* language used by InfluxDB, uses the basic search capabilities of ElfStore to identify blocks of interest, inserts and caches them into one or more local InfluxDB instances on the fog, executes subsets of the user query on each fog in parallel, and aggregates the results for returning to the user. This effectively offers a distributed TSDB with an edge-local backing store, and is the *first of its kind system* to offer distributed time-series querying on edge and fog devices.

Next, in § 2 we discuss background on ElfStore and InfluxDB, and related work on edge computing and querying; we introduce the TorqueDB design and query execution model in § 3; we present detailed performance results on a real-world edge and fog deployment in § 4; and lastly offer our conclusions in § 5.

2 Background and Related Work

2.1 ElfStore Distributed Edge-local Federated Storage

ElfStore [9] is a block-centric distributed storage system on edge and fog resources, for files that grow over time. Edges are connected to a *parent fog* that is present in their local network, and together form a *fog partition*. Many such fog partitions can exist, with fogs being able to talk directly to each other. These all form a peer-to-peer (P2P) network overlay, with edges serving as peers and fogs as super-peers, and its associated scaling characteristics to 1000's of devices.

Edges host data and metadata for a block. Fogs maintain a *mapping* from the block ID to the edge(s), and *indexes* over the block metadata, for blocks in their local partition. This allows fogs to perform basic value-based *searching* for blocks based on their metadata properties, and *lookups* of block replica locations using their block ID. Fogs also use Bloom Filters to maintain *approximate indexes* about contents in other fogs partitions to allow forwarding of metadata search and block retrieval requests across the overlay, within no more than 3 hops.

Since edge devices can have asymmetric reliability, ElfStore uses a block-specific replication level based on the required block reliability and the reliability of the edges chosen for placement. Statistics exchanged between the fogs about the reliability levels and storage capacities of edges in their partitions are used by the replication logic to guarantee a minimum resilience and load balancing of storage. It also recovers from failures by re-replicating blocks from failed edges.

2.2 Influx DB Time-series Database

InfluxDB is an open-source TSDB optimized for high read and write throughput. It stores data in *buckets* (databases) that contain *measurements* (tables). Each row in a table has a *timestamp* and columns that are either *tags*, which are indefaxed, or *fields*, which can be aggregated on. It has a native *Flux* query language that allows SQL-like queries over time-series data, with support for Select, Project, Aggregate, Window-aggregates and Joins. Besides network APIs provided for data insertion and querying, data can also be bulk-loaded into an InfluxDB table using a *line-protocol* CSV format.

2.3 Querying over edge devices

There have been recent works that examine the use of edge computing for *query processing over event streams*, though they do not support distributed time-series queries over a database or use an external edge-storage as the backend.

StreamSight [2] provides a declarative query model for matching complex patterns on data streams. The system compiles these queries into stream processing jobs for continuous execution on engines running on edge devices. The query plan is dynamically updated so that intermediate results are reused and not recomputed. It also supports approximate answers with error-bounds for latency-sensitive execution.

Periodic querying is essential in Industrial IoT. Here, contiguous queries can have overlapping input regions, and the sensor data retrieved by recent queries may be reused for answering the upcoming queries. Zhou, et al. [16] proposed a popularity-based caching strategy to leverage these patterns. They show significant reduction in the communication cost, when the number of queries is relatively large. Such caching strategies can also be incorporated into TorqueDB.

HERMES [7] enables query evaluation over data streams across cloud and fog nodes. They use reservoir sampling of incoming observational streams to reduce communication and memory consumption on fog nodes in resource-constrained environments. Similarly, our prior work [3] examines distributed analytics over event streams on edge and cloud using a CEP engine, rather than query over past data that we address in TorqueDB. Their key objective is to schedule a dataflow graph of dependent CEP queries on edge and cloud resources while minimizing the latency and conserving energy. Individual queries are not decomposed unlike TorqueDB does, and we use only edge and fog rather than the cloud.

Others have examined *query rewriting* in other contexts. Schultz, et al. [12] design a CEP system with operator placement decisions based on cost functions, and greedily selects a distributed deployment plan over machines in a cluster. They use query rewriting to increase the efficiency of operations by reusing common operators. TorqueDB's execution model operates on queries independently as these are one-off rather than standing CEP queries.

Grunert, et al. [4] use query rewriting and containment techniques from databases for efficient and privacy-aware processing of queries in an edge-cloud

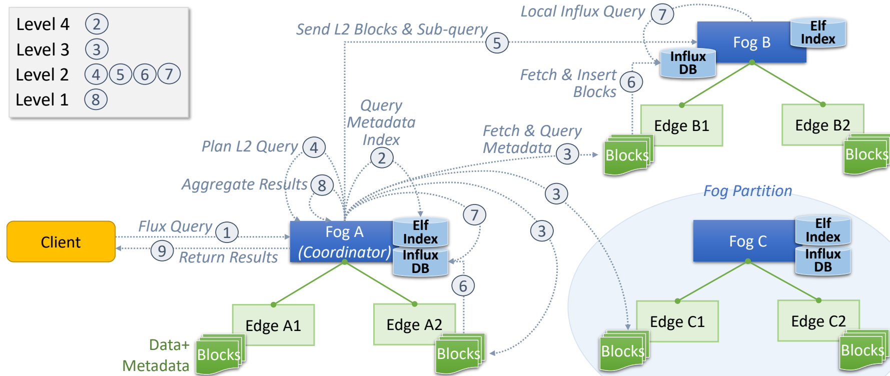


Fig. 1. Architecture and Query Execution Sequence of TorqueDB

setup. The input query is split into "fragment" and "remainder" queries. Fragment queries operates on resource constrained edge devices to filter and pre-aggregate data, while remainder queries execute the complex part of the query on fog devices. We do similar rewriting across different levels on fog devices.

There are other edge and fog storage systems that have been proposed as well, besides ElfStore. *DataFog* [5] is a data management platform at the edge on top of Apache Cassandra, for a geo-distributed and heterogeneous edge computing environment. They provided a locality aware distributed indexing mechanism and a replica placement approach to provide spatial proximity. Finally, they employed a TTL based data eviction policy to accommodate the constrained storage capacity at the edge. These can serve as alternative backends for TorqueDB.

3 TorqueDB Architecture

The architecture for TorqueDB is shown in Fig. 1. The *system model* contains edge and fog resources. Each edge is associated with one *parent fog*, which serves as a network gateway to other fogs and the Internet. All edges with the same parent fog form a *fog partition*, and devices in a partition are part of the same private network, with high bandwidth and low latency connectivity. All fogs can communicate with each other directly, either on the same private network or through the Internet. The network link between fogs may be slower than with the edge devices in their partition. We expect edge devices to have resources comparable to a Raspberry Pi with a 4-core low-power CPU, 1–2GB RAM and 128GB SD card storage, while the fog resources are comparable to a workstation or low-end server with 4–8-core CPU, 8–16GB RAM and 500GB–4TB HDD.

Edge devices host the input data accumulated from sensors in *blocks* managed by ElfStore. Each block contains rows of time-series data, typically from one or more sensors and for a specific time range. New blocks are added over time. Each block is identified by a unique *block ID*. ElfStore allows application-specific *metadata properties* to be stored for these blocks and searched upon. These

contain details such as the table name, sensor ID, sensor types, units, time range, location, etc. A subset of these properties match specific columns present in the time-series data, e.g., the location and the sensor ID column values may be common to all rows in the block, which are surfaced as a property for that block, while the minimum and maximum timestamps for the rows in the block will form the time-range property for that block. As an additional optimization, we also compute *aggregates* over the content in these blocks, such as the number of rows, minimum and maximum values for specific columns like temperature and humidity, etc. and store them as metadata properties for the block. ElfStore natively creates *replicas* of a block data and metadata, identified using the same block ID, on multiple edge devices to meet the reliability requirements specified.

Fog resources run ElfStore services to manage the edge devices, replication and block placement, as well as maintain indexes on the metadata for blocks stored in their partition. For TorqueDB, we also host an InfluxDB instance on each fog resource to execute Flux queries. The InfluxDB instance is primarily used as a *query engine* rather than for data management. It is a transient store (and optionally cache) for the time-series data on which complex Flux queries are executed, with the durable storage being the blocks in ElfStore.

This layered design, reusing ElfStore and InfluxDB, has several benefits over designing a distributed TSDB from the ground up. It avoids the complexity of distributed management and resilience of different instances of a TSDB, while leveraging the data reliability guarantees offered by ElfStore. It also allows edge applications that directly operate on the data blocks to be supported by ElfStore [15] while the queries are offloaded to TorqueDB. Lastly, it eliminates the need for redundant copies of data on both the edge-local file storage and the TSDB, instead using the TSDB just as a transient cache.

At this time, we limit our design to executing the Flux queries on InfluxDB instances running on the fog resources. This leverages their higher resource capacity relative to constrained edge devices and limits the coordination overheads. As future work, we propose to examine designs where the InfluxDB is hosted directly on the edge devices themselves to enhance the parallelism and limit data movement.

3.1 Query Lifecycle and Distributed Execution

TorqueDB supports a subset of the *Flux query language*, as illustrated in Fig. 2. Specifically, we support *range queries* over time-stamps (e.g., *time* BETWEEN start AND end), *filter queries* over column values (e.g., *dust* > 1000), *aggregation* functions such as sum, average, minimum and maximum over columns values (e.g., SUM *dust*), *aggregation windows* over time (e.g., WIN(SUM, *every hour*)), and *projection* of columns (e.g., SELECT *UV*) to the output. Support for join queries and complex nested queries is planned for future.

Users submit their Flux query to a TorqueDB service, which run on all fog resources. The fog receiving the query is called the *coordinator* for this query (Fig. 1 ①). The distributed execution plan for the query is decomposed into a query tree, with execution happening at *four levels* (Fig. 2). At *level 4 (L4)*,

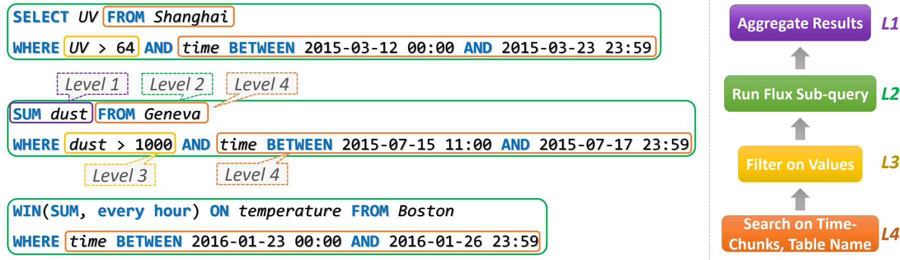


Fig. 2. Sample Flux queries and execution levels in TorqueDB

the coordinator attempts to identify the ElfStore blocks that contain the time-series data on which the query depends. For this, it extracts those parts of the query predicates that can be pushed down as a native ElfStore search over the block metadata index (②). Specifically, ElfStore can search for blocks with a given property value, and compose Boolean predicates using AND and OR. These include matching properties such as the table name, location, sensor ID, etc. which require a direct value comparison in the input query.

But ElfStore does not support range queries which are important for time-series data. To address this, we discretize the time-range for rows in a block into granular *time-chunk numbers* relative to an *epoch*, e.g., in *12-hour* increments starting from *2020-01-01 00:00*, and include the chunks numbers that the rows of a block overlap with in its metadata property. A similar discretization is done on the input time-range query into one or more chunk numbers, and composed as an OR on the time-chunk metadata matching any of these chunks numbers. E.g., if the user query has a time-range predicate from *2020-02-14 07:35* to *2020-02-14 20:15*, these overlap with the time-chunks 89 and 90. We search ElfStore for blocks that have time-chunk property with values of 89 or 90. Likewise, when storing blocks, we calculate the chunk numbers for their time-ranges, and store these as a multi-valued property for the time-chunk metadata.

The output of the L4 query is a filtered set of block IDs having the minimal data necessary for further query processing. These are passed as input to *level 3 (L3)*, where the coordinator optionally fetches the actual block metadata to further refine the search space (Fig. 1 ③). In particular, when we have value comparisons over non-timestamp columns present in the data, like "dust" and "UV", we use the minimum and maximum aggregate metadata values for these columns to decide if the block contains the relevant data or not for further querying. E.g., if the input query has a predicate that only retains rows with $dust \geq 1000ppm$, then we fetch and use the block metadata to eliminate those whose *maximum dust* is less than 1000. L3 is done only if value comparison predicates are present in the input query.

The output of L3 is again a set of block IDs that are a subset of the block IDs from L4. Now, the coordinator assigns these blocks to the available fog resources to load the block contents into their local InfluxDB instance and execute the

Flux query on it. The mapping of blocks to fogs is done by the *Query Planner* discussed in § 3.2 (Fig. 1 ④). The coordinator decomposes and rewrites the input query into *sub-queries* relevant to the blocks assigned to each fog, and sends them these block IDs and sub-query for execution in *level 2 (L2)* (⑤).

In L2, each fog receiving a sub-query and a list of blocks *fetches* the block contents from ElfStore, and *inserts* them into the local InfluxDB instance. We use a thread-pool for the fetch and insert of each block in parallel (Fig. 1 ⑥). All blocks are inserted into a single table, even across queries. This helps with caching, as we discuss later. During insertion, we add the block ID as a column in each row inserted into InfluxDB. These block IDs are also included as a value predicate in the sub-queries. This ensures that a sub-query only targets blocks relevant to the current query being executed on the fog and not other blocks inserted by previous or concurrent queries. This avoids duplicates results. E.g., if L3 returns block IDs $\langle 3, 5, 9\rangle$ for processing at L2, and $\langle 3, 5\rangle$ are assigned to Fog A and $\langle 9\rangle$ to *Fog B*. Say *Fog B* already had a copy of block 5 present in it. If we run the two sub-queries on *Fog A* and *Fog B*, we should not get duplicates for matching rows for the block 5 present both in *Fog A* and B. So the sub-query for *Fog A* will have a filter to limit the rows to those with the Block ID field as 3 or 5, while the sub-query for *Fog B* filters in only rows with Block ID 9.

Once all blocks assigned to a fog are inserted into the local InfluxDB, the sub-query is executed on the TSDB and the results returned to the coordinator (Fig. 1 ⑦). Multiple fogs having block assignments will operate in parallel. When the coordinator receives the L2 results from all fogs, in the absence of an aggregation operator, it just *appends* all the results and returns them to the client in *level 1 (L1)* (⑧, ⑨). However, if an *aggregation function* over a column is present, then the L2 query result from each fog will have the aggregation over the subset of rows in that fog. Here, we further aggregate across all these results to return a single result to the user. This aggregation is done inside the coordinator by code specific to each aggregation function. For functions like *mean*, L2 returns the *sum* and the *count*, which are used to compute the global mean.

3.2 Query Planning

In L2, we perform block fetch from ElfStore, insertion into the local InfluxDB and query execution, on one or more fogs. This is the most time-consuming level since it involves fetching the block from SD card on the edge and a network data transfer. The time taken to ingest data into InfluxDB is also significant. So we ensure the parallelism offered by multiple fog and edge devices is fully exploited. Given a set of blocks from L3, the edges (and parent fogs) that their replicas are present in and the available fogs, the goal of query planning is to partition these blocks to the fogs to reduce the execution time for L2. We propose two query planning strategies, *partition-local (QP1)* and *load-balancing (QP2)*.

The block transfer time is constrained by the I/O speed of the edge device ($\approx$ 100 *Mbps* seen for a Class 1 SD card), the network bandwidth from the edge to parent fog and from fog to fog ($\approx 100Mbps$–$1Gbps$), and the cumulative bandwidth into a fog ($\approx 1Gbps$). In both strategies, we first try and maximize the

cumulative disk and network bandwidth from different edge devices in parallel. From the available set of blocks, we maintain a *load count* for each edge, which is the number of blocks selected for reading from this edge and set to 0 at the start. We then sort the blocks in ascending order of the number of edges they are present on (replica count). For each block, in this order, we select one of its edge replicas such that this edge has the least load count among the replica edges, and increment the load count for that edge. This achieves *load balancing of the block-reads* from among the edges hosting the block replicas.

Next, in the *partition-local strategy (QP1)*, we simply assign a block replica to the parent fog for its edge. The intuition is that the bandwidth from the edge to its parent fog is high and one-hop, and the block is kept within this partition.

In the *load-balancing strategy (QP2)*, we prioritize balancing the number of blocks assigned to each fog. This maximizes the parallelism for the data inserts into InfluxDB and the query execution on the fogs. Here, we maintain a count of blocks assigned to a fog, initialized to 0. For each block replica, if the parent fog for the edge is the least loaded among all fogs, the block is assigned to this fog; if not, the block is assigned to the least loaded fog. The fog's load count is incremented, and this repeats for the next block replica.

3.3 Block Caching

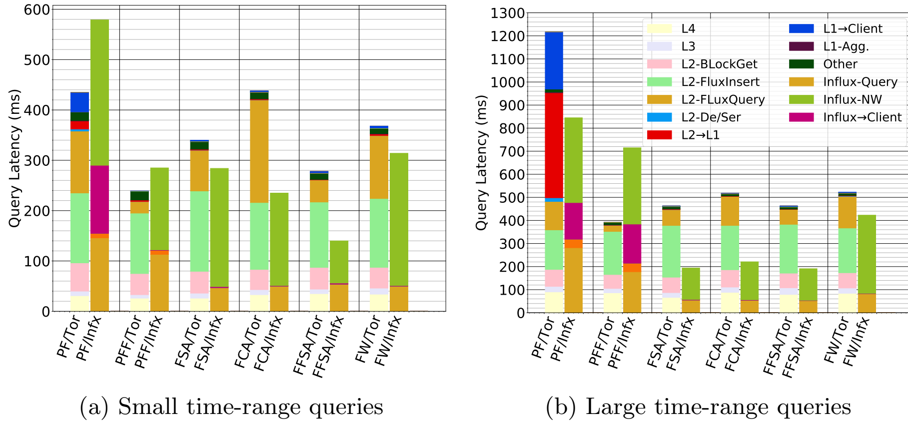
Much time in L2 is spent in fetching and inserting the blocks. We propose a caching mechanism where the coordinator maintains a local mapping from block IDs to the fog that has inserted that block into its local InfluxDB. This mapping is updated after the L2 of each query, and lazily propagated across all fogs. The query planner uses this knowledge to assign a block to the fog that it is cached in, and only triggers the QP2 mapping algorithm for blocks that are not cached.

The caching strategy will retain all blocks used in any query within the local InfluxDB of one of the fog resources. This ensures that blocks that are used once are available immediately on a fog for future queries, but unused blocks are not copied from ElfStore. In future, this can be combined with a cache replacement like least recently used (LRU) to more efficiently utilize the disk space, and may also load-balance the cached-block distribution across fogs.

4 Experiment Results

4.1 Setup

Our experiments use a 15-node IoT cluster with 12 Raspberry Pi 4B edge devices (ARMv8 4-core@1.5 GHz, 2 GB RAM, 64 GB UHS-1 SD card) and 3 fog resources (Intel Core i5 6-cores@2.1 GHz 8 GB RAM and 500 GB HDD). These 15-nodes form 3 fog partitions with 1 fog and 4 edges each, connected over hierarchical 1 Gbps switches with an average latency of 0.6 *ms*. As a baseline, we use a Microsoft Azure Standard D4 v3 VM (Central India) running Intel Xeon E5 4-cores@2.3 GHz, 16 GB RAM and 500 GB HDD. Its performance is comparable to the fog resource based on query benchmarks.



(a) Small time-range queries (b) Large time-range queries

Fig. 3. Stacked bar plot for median query on TorqueDB (QP2) vs. Cloud InfluxDB

ElfStore runs on the 15-node cluster with uniform replication factor of 3 and no edge failures. TorqueDB is implemented in Java v1.8 and runs on the fogs alongside InfluxDB v1.7.9 [3], which is hosted in container. By default, caching is disabled on TorqueDB and we use QP2.

We use data from *Sense your City* in our workload [4], which has ambient monitoring devices from 84 locations in 7 cities worldwide that sense dust, temperature, humidity, UV, etc. The devices report an observation every 3 mins over a 16 month period, to give ≈ 19.35 *million rows of time-series data*. Each 1 MB block in ElfStore holds 1 day of data per city with 5760 rows of data.

We use a query workload with 6 *predicate patterns*: Project+1 Value Filter (PF); Project+2 Value Filters (PFF); Filter+Simple Aggregate like sum/-count/min/max (FSA); Filter+Complex (mean) Aggregate (FCA); 2 Value Filters+Simple Aggregate (FFSA); and 1 Value Filter+Window Aggregate (FW). These queries are inspired by a prior IoT query benchmarking work [6]. They are also designed to cover the common query operators such as projection of specific columns from a tuple into the result set, filters defined on field values, simple and complex aggregation over field values, and moving windows over the time-series tuples. There is also a time-range filter in all cases, with a *small range* being over 3 days and *large range* being 12 days. We permute different values and time ranges to generate 30 instances of each pattern and range for a total of 360 queries. All queries are run from a client that is in the same local network as the fogs.

4.2 Analysis

Figs. 3a and 3b show the stack bar plots for the different components of the total execution times for one median query from each type for TorqueDB and for centralized InfluxDB on a cloud VM, for small and large time-range queries.

Performance of TorqueDB All query types, except PF with a large time-range, complete in under 600 *ms*, with smaller queries running under 400 *ms*. For the small time-range queries, the major fractions of the total execution time spent by TorqueDB are: 39% in inserting data into the InfluxDB in L2, 28% in the query execution at L2, and 14% in data transfer from ElfStore to L2. For the larger queries, the largest fractions are: 36% in inserting rows into InfluxDB in L2, 16% in query execution in L2, and $\approx$ 14% in L4 for locating matching blocks in ElfStore and the same in transferring results from L2 to L1. These L2 costs are due to on-demand copying and insertion of the relevant blocks from the edge to the InfluxDB on the fog, and these dominate the overall execution time. As we see later, it can be mitigated by caching.

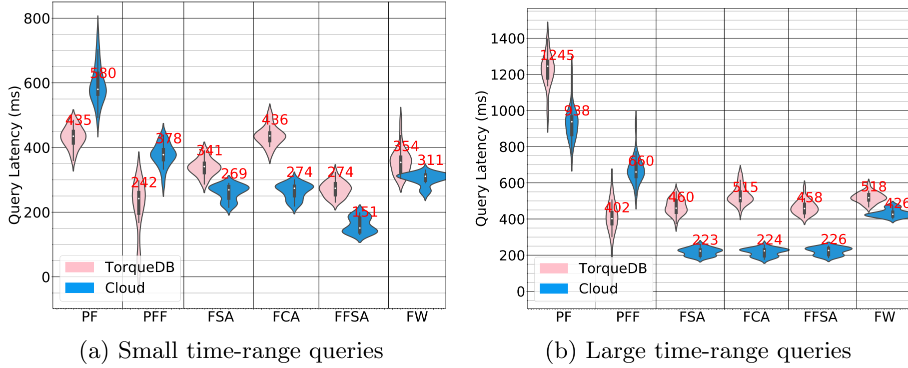
We also see that the block search, data transfer and insertion times are uniform 170-190 *ms* for all small time-range queries since the number of blocks transfered and rows inserted are the same at 3 blocks; and likewise the large time-range with 12 blocks inserted take 265-285 *ms*. The only exception is query type PFF where some blocks are filtered out at L3 and hence the data transfer and insertion costs are smaller.

The variability in the execution times across different query types arise from the actual query execution in L2. Among the query types, PF is the second slowest due to the large result set size returned by the query, though the query itself is not complex. The time spent in transferring data from L2 and L1, and returning the results to the user is higher. PFF is the fastest as its additional filter reduces this result set size substantially. FSA and FFSA perform an extra simple aggregation at L1, besides one and two filters. They are the fourth or third fastest depending on the small or large query range, though their aggregated result set size is only 1 row. FCA performs a complex aggregation for finding the mean by running two aggregation queries for sum and count, and hence is twice as slow as FSA; it is the slowest among all queries. Lastly, FW does a window aggregation within InfluxDB to return 10's of results and is the third slowest.

TorqueDB vs. Centralized InfluxDB on the Cloud Figs. 4a and 4b further show the violin plots for the total execution times for different query types and time-ranges, for TorqueDB and centralized InfluxDB on the cloud.

For the small time-range queries, the performance of TorqueDB and InfluxDB on the Cloud are similar, while for the larger time-range, the latter is mostly faster. These differences can be attributed to the query execution times, and to the other overheads. TorqueDB leverages parallel query execution across local

[3] https://www.influxdata.com/products/influxdb-overview/
[4] http://datacanvas.org/sense-your-city/



(a) Small time-range queries (b) Large time-range queries

Fig. 4. Violin plot of query latencies on TorqueDB (QP2) and Cloud InfluxDB

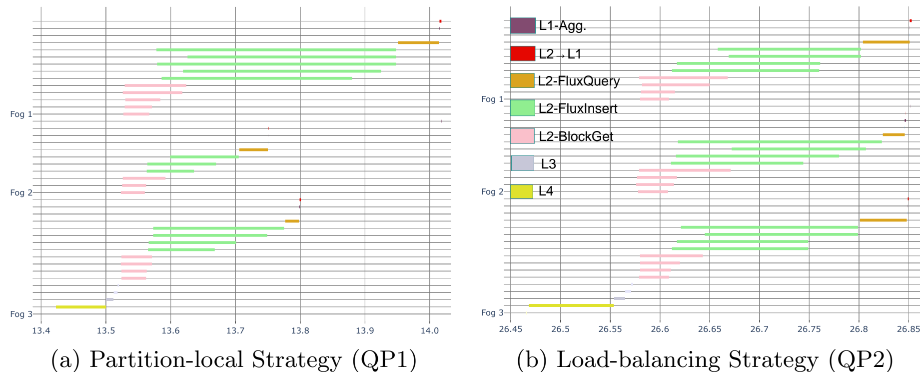
InfluxDBs on the fog, and this causes it to have a lower query execution time than the cloud for PF and PPF. But the cloud VM's CPU is faster in performing the aggregation operations, by 19%–147%, for FSA, FCA, FFSA and FW.

Besides this query execution, differences arise from the other components. Specifically, the network latency between the edge client and the cloud dominates for the small time-range queries on InfluxDB cloud, which have smaller query execution times. These overheads of $\approx 211\ ms$ take 64% of the total query time. But this absolute latency is about the same at $\approx 255\ ms$ but relatively smaller, at 57% of the total time, for the large time-range queries having longer query execution times. In addition, PF returns a large result set and this incurs costs to return the results to the client. However, for TorqueDB, the larger queries require more block fetches and insertions, and this increases its overall time.

In addition to these, the InfluxDB on the cloud took $\approx 18\ mins$ to transfer $3.28GB$ of data for the 7 cities from the edge to the cloud. This is amortized over a period of time in a real-world scenario. The WAN link between the edge and the cloud also shows more variability, ranging from 27.1–1048 ms latency and 21.2–536 $Mbps$ bandwidth, over a 24 hour period. In summary, while TorqueDB is slightly slower than queries on the cloud, the latter will have less deterministic execution times, and also incur additional VM and network costs.

Benefits of Query Planning The QP1 and QP2 query planning strategies in L2 pick the same set of edges to get the block replicas from, but select different fogs to assign them to; the former reduces cross-partition data transfers and the latter balances the load per fog. In our experiments, we report that QP2 is 0.2–7.6% faster than QP1, on average for the different query types. This is because the edge–fog and fog–fog bandwidths are comparable in our IoT cluster and hence the benefits of QP1 are not apparent.

However, the load balancing in QP2 does have benefits, in particular where many blocks are fetched and inserted in L2. Figs. 5 show the Gantt time-line plot for different time components of a FFSA query with large time-range, running



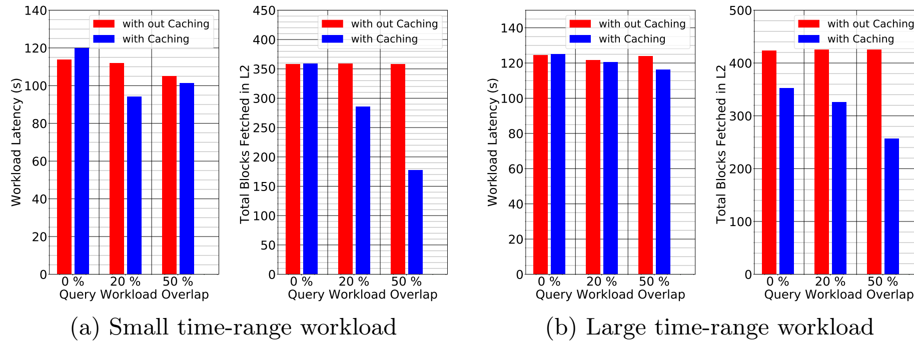
(a) Partition-local Strategy (QP1) (b) Load-balancing Strategy (QP2)

Fig. 5. Gantt plot of latency for a FFSA large query on TorqueDB using QP1 vs QP2

on different fogs, when using QP1 and QP2. The Y axis indicates threads in the 3 fogs and X axis is a relative time-line, in seconds. This fetches 12 blocks in L2. QP1 assigns 5 blocks to Fog 1, 3 blocks to Fog 2 and 4 blocks to Fog 3, since these are the parent fogs for the block replicas chosen. QP2 instead load-balances and assigns 4 blocks to each fog, even though they may cross partition boundaries and cause 2-hops for block transfer. As a result, QP2 achieves an ≈ 200 ms reduction in the L2 block fetch and InfluxDB insert.

Benefits of Caching Finally, we evaluate the benefits of caching in TorqueDB. Here, we use query workloads having a mix of 20 queries from each of the 6 types, to give 120 queries for the small and 120 queries for the large time-ranges. This has no (0%) overlaps in the query mix, i.e., all queries are unique. We use these to create two more workloads where 20% of the queries overlap, i.e., are duplicated, and 50% overlap. These 6 query workloads are run on TorqueDB, with and without caching enabled. Fig. 6 shows the total execution time for these workloads, and the total numbers of blocks fetched and inserted in L2. These are averaged over 3 runs.

For 0% overlap workload with small time-range, the total number of blocks fetched is the same at 359, both with and without caching. On the other hand, in the large time-range 0% workload, caching results in 17% fewer block fetches than without caching. This is because cached blocks can be reused across queries even without an exact duplication of the queries. Further, the number of blocks fetched proportionally reduces as the number of explicit query overlaps increase to 20% and 50%. However, the impact on the total latency is muted. Since we use four parallel threads per fog in L2, even having one block transfered in L2 can reduce the benefits of caching as that becomes the critical path.



(a) Small time-range workload (b) Large time-range workload

Fig. 6. Total workload latency and # of L2 blocks transfered, on TorqueDB with and without caching

5 Conclusions

In this paper, we have proposed TorqueDB, a novel platform for distributed execution of time-series queries on edge and fog devices, avoiding the need to keep a central TSDB in the cloud. This reduces monetary costs, keeps the data within the private network if needed, and also avoids the latency variability across a WAN to the cloud for edge applications. TorqueDB also leverages the persistence capabilities of ElfStore which allows non-query applications to use the same master data without creating duplicates within a TSDB. We also use the native TSDB querying of InfluxDB with its Flux query language, that is popular in IoT domains. Our optimizations on the query planning and caching show benefits, and mitigate the costs of on-demand block transfers in TorqueDB to give performance comparable to a central cloud VM.

As future work, we plan to extend the InfluxDB instances to run on the edge, besides the fog. This will avoid the data transfer penalty in L2, and also expose more parallelism for query execution. Support for joins and nested Flux queries is planned as well. It is also worthwhile to examine integrating TorqueDB with other distributed edge storage platforms, besides ElfStore, that may emerge over time. This is conceptually possible as we are only loosely-coupled with ElfStore, using just its public storage and lookup APIs which are likely to be offered by other systems as well. Larger scale experiments on 100's of devices with more heterogeneous compute and network capabilities will validate the scalability and performance further. Examining the impact of device unreliability on the query performance will also be examined, and contrasted against cloud TSDB.

Acknowledgment

The authors thank members of the DREAM:Lab, IISc, including Aakash Khochare, Shriram Ramesh and Sheshadri KR, for their assistance with the design, development and experiments of TorqueDB. This research was supported by grants from the DST ICPS program.

References

1. Abadi, D., et al.: The Seattle Report on Database Research. SIGMOD Record **48**(4) (2020). https://doi.org/10.1145/3385658.3385668
2. Georgiou, Z., Symeonides, M., Trihinas, D., Pallis, G., Dikaiakos, M.D.: Streamsight: A query-driven framework for streaming analytics in edge computing. In: IEEE International Conference on Utility and Cloud Computing (UCC) (2018). https://doi.org/10.1109/UCC.2018.00023
3. Ghosh, R., Simmhan, Y.: Distributed scheduling of event analytics across edge and cloud. ACM Transactions on Cyber-Physical Systems **2**(4) (2018). https://doi.org/10.1145/3140256
4. Grunert, H., Heuer, A.: Rewriting complex queries from cloud to fog under capability constraints to protect the users' privacy. Open Journal of Internet Of Things (OJIOT) **3**(1) (2017)
5. Gupta, H., Xu, Z., Ramachandran, U.: Datafog: Towards a holistic data management platform for the iot age at the network edge. In: USENIX HotEdge (2018)
6. Liu, R., Yuan, J.: Benchmarking time series databases with iotdb-benchmark for iot scenarios. Tech. Rep. arXiv:1901.08304, arXiv (2019)
7. Malensek, M., Pallickara, S.L., Pallickara, S.: Hermes: Federating fog and cloud domains to support query evaluations in continuous sensing environments. IEEE Cloud Computing **4**(2) (2017). https://doi.org/10.1109/MCC.2017.26
8. Martinviita, M.: Time series database in Industrial IoT and its testing tool. Master's thesis, University of Oulu (2018)
9. Monga, S.K., Ramachandra, S.K., Simmhan, Y.: Elfstore: A resilient data storage service for federated edge and fog resources. In: IEEE International Conference on Web Services (ICWS) (2019). https://doi.org/10.1109/ICWS.2019.00062
10. Nagato, T., Tsutano, T., Kamada, T., Takaki, Y., Ohta, C.: Distributed key-value storage for edge computing and its explicit data distribution method. IEICE Transactions on Communications (2019). https://doi.org/10.1587/transcom.2019CPP0007
11. Patel, P., Ali, M.I., Sheth, A.: On using the intelligent edge for iot analytics. IEEE Intelligent Systems **32**(5) (2017). https://doi.org/10.1109/MIS.2017.3711653
12. Schultz-Moller, N.P., Migliavacca, M., Pietzuch, P.: Distributed complex event processing with query rewriting. In: ACM International Conference on Distributed Event-Based Systems (DEBS) (2009). https://doi.org/10.1145/1619258.1619264
13. Simmhan, Y.: Big data and fog computing. In: Sakr, S., Zomaya, A.Y. (eds.) Encyclopedia of Big Data Technologies. Springer (2019). https://doi.org/10.1007/978-3-319-63962-8_41-1
14. Varshney, P., Simmhan, Y.: Characterizing application scheduling on edge, fog, and cloud computing resources. Software: Practice and Experience (2019). https://doi.org/10.1002/spe.2699
15. Zhang, W., Guo, W., Liu, X., Liu, Y., Zhou, J., Li, B., Lu, Q., Yang, S.: Lstm-based analysis of industrial iot equipment. IEEE Access **6** (2018). https://doi.org/10.1109/ACCESS.2018.2825538
16. Zhou, Z., Zhao, D., Xu, X., Du, C., Sun, H.: Periodic query optimization leveraging popularity-based caching in wireless sensor networks for industrial iot applications. Mobile Networks and Applications **20**(2) (2015). https://doi.org/10.1007/s11036-014-0545-4