A project report on

TorqueDB: DISTRIBUTED QUERYING OF TIME SERIES DATA FROM EDGE-LOCAL STORAGE

Submitted in partial fulfilment for the award of the degree of

Bachelor of Technology

By

DHRUV SHEKHAR GARG

(16 BCE 1190)

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

May, 2020

VIT®
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

School of Computer Science and Engineering

DECLARATION BY THE CANDIDATE

I hereby declare that the Capstone Project Report titled "***TorqueDB*: DISTRIBUTED QUERYING OF TIME SERIES DATA FROM EDGE-LOCAL STORAGE"** submitted by me to VIT University, Chennai in partial fulfillment of the requirement for the award of the degree of **Bachelor of Technology** in **Computer Science and Engineering,** is a record of bonafide Non-PAT industrial internship undertaken by me under the supervision of the guide **Dr. Shyamala L (Asso. Prof VIT Chennai)**, and two external guides **Anshu Shukla (Ericsson Research, Bangalore)** and **Prof. Yogesh Simmhan (Indian Institute of Sciences (IISc), Bangalore).**

I further declare that the work reported in this report has not been submitted and will not be submitted, either in part or in full, for the award of any other degree or diploma in this institute or any other institute or university.

Chennai **Signature of the Candidate**

Date:

VIT®
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

School of Computer Science and Engineering

CERTIFICATE

This is to certify that the report titled **"*TorqueDB*: DISTRIBUTED QUERYING OF TIME SERIES DATA FROM EDGE-LOCAL STORAGE"** is prepared and submitted by **DHRUV SHEKHAR GARG** (**16BCE1190**) to VIT Chennai, in partial fulfilment of the requirement for the award of the degree of **B. Tech. CSE** programme. This is a bonafide record carried out under my guidance.

The project fulfils the requirements as per the regulations of this University and in my opinion meets the necessary standards for submission. The contents of this report have not been submitted and will not be submitted either in part or in full, for the award of any other degree or diploma and the same is certified.

Signature of the Guide:

Name: **Dr. Shyamala L**

Date:

Signature of the Internal Examiner

Name:

Date:

Signature of the External Examiner

Name:

Date:

Approved by the Head of Department, **B. Tech CSE**

Name: **Dr. Justus S**

Date:

(Seal of SCOPE)

Ericsson Confidential
INTERNSHIP LETTER

Date	Reference
2019-11-22	EGIL/HR-19:4716 Uen
Your Date	Your Reference

Attending to this matter
NO/EGI/H Rajat Bajaj/VG

Mr. Dhruv Shekhar Garg

Dear Mr. Garg,

This has reference to your request letter for Internship in our organization and subsequent discussion regarding the same. We are pleased to allow you for this training in our organization at **Bangalore** office, starting from 02-Dec-2019 to 15-Jun-2020.

You will be assigned a project upon joining by Chandramouli Sargor

Please contact Chandramouli Sargor on the day of joining.

With best wishes
Yours sincerely,
For ERICSSON INDIA GLOBAL SERVICES PRIVATE LIMITED

Rajat Bajaj
Talent Acquisition
Human Resources

Ericsson India Global Services Private Limited
Knowledge Boulevard,
A-8A, Sector 62A. NOIDA
INDIA - 201 309
www.ericsson.co.in / www.ericsson.com

Tel: + 91 120 3029200
Tel: + 91 120 4256000
Fax: + 91 120 3029135

Registered Office
4th Floor, Dakha House
18/17, W.E.A., Pusa Lane,
Karol Bagh,
New Delhi 110 005 INDIA

ACKNOWLEDGEMENT

I would like to express my profound gratitude to **Dr. Shyamala L**, Associate Professor, SCOPE, Vellore Institute of Technology, for her continual guidance, encouragement, and understanding throughout the 6 months of the Capstone project. It was a splendid opportunity to work with an intellectual and expert in the field of Distributed Computing, and her timely interventions definitely enabled me to fulfil the objectives of the project.

I would also like to express my thanks to the Chancellor, VPs, VC, and Pro-VC for providing an environment to work during the tenure of the entire project.

Further, I express my wholehearted thanks to **Dr. Justus S** (Head of the Department), **Dr. Kumar R**, and **Dr. Prabhakar Rao** (Project Coordinator) SCOPE, VIT Chennai, for their longstanding support. I would also like to thank **Dr. Jagadeesh Kannan R**, Dean, and **Dr. Geetha S**, Associate Dean, SCOPE, VIT Chennai, for helping the students in every aspect.

Moreover, I would like to express my profound appreciation for **Anshu Shukla (Ericsson Research, Bangalore)** and **Prof. Yogesh Simmhan (Indian Institute of Sciences (IISc), Bangalore)** for their regular inputs and feedback throughout my internship. It was an enriching experience to have worked with some of the brightest minds in Systems and Distributed Computing. I would also like to thank **all the team members of Ericsson Research, Bangalore, and DREAM: Lab, IISc Bangalore**, for sharing their insights and experiences with me. Sincere thanks to **Chandramouli Sargor (Former Team Lead- Ericsson Research, Bangalore)** and the Ericsson management for facilitating the project and for the knowledge that I gained during my internship.

Last but not least, I express my gratitude and appreciation to my **parents,** my **friends**, and all those who have helped me directly or indirectly toward the successful completion of this project.

Place: Chennai Dhruv Shekhar Garg

Date: (16 BCE 1190)

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

S. No.	Abbreviation	Expansion
1	AP	Access Points
2	API	Application Programming Interface
3	ARM	Advanced RISC Machine
4	CAT6/6E	Category 6, Category 6 Enhanced cables
5	CEP	Complex Event Processing
6	CPS	Cyber Physical Systems
7	CPU	Central Processing Unit
8	CSV	Comma-Separated Values
9	2D	2-Dimensional
10	DB	Database
11	FSA	Filter + Simple Aggregate
12	FCA	Filter + Complex Aggregate
13	FFSA	2 Filters + Simple Aggregate
14	FW	Filter Window Aggregate
15	2G/3G/4G/5G	2/3/4/5 Generations of wireless technology
16	GB	Gigabyte
17	Gbps	Gigabits per second
18	GHz	Gigahertz
19	HDD	Hard Disk Drive
20	ICT	Information and Communication Technology
21	IO	Input Output
22	IOT	Internet of Things
23	ID	Identity
24	IP	Internet Protocol

S. No.	Abbreviation	Expansion
25	L1/L2/L3/L4	Levels 1,2,3,4
26	LAN	Local Area Network
27	LSM	Log-Structured Merge Tree
28	LRU	Least Recently Used
29	MB	Megabyte
30	Mbps	Megabits per second
31	ms	Milliseconds
32	P2P	Peer-to-peer
33	PF	Project + Filter query
34	PFF	Project + 2 Filters query
35	QP	Query Planning strategy
36	RAM	Random Access Memory
37	SD Card	Secure Digital Card
38	SSTable	Sorted Strings Table
39	SQL	Structured Query Language
40	UHS-1	Ultra High Speed
41	TB	Terabyte
42	TSDB	Time-series Databases
43	TSM	Time Structured Merge Tree
44	TTL	Time-to-live
45	VM	Virtual Machine
46	WAL	Write Ahead Log
47	WAN	Wide Area Network
48	WLAN	Wireless Local Area Network
49	Wi-Fi	Wireless Fidelity

EXECUTIVE SUMMARY

Numerous real-time applications necessitate extremely short response times for queries posed to the system. As a consequence, edge and fog based Internet of Things deployments are gaining traction. While a lot of work has been done on asynchronous data upload to the cloud, data retention, compression, and context-aware placement of data, a *gap exists* in a query-able, distributed data storage layer at the edge and fog level. Not only is there a need to store data reliably at the edge to increase locality and minimize latency, but also to make sure that the storage layer supports a high-volume of concurrent writes and reads on the data. This is particularly well suited to Industrial IoT applications, where many sensors are continuously pushing time-stamped data to the storage layer and the monitoring tools are querying the data.

This project presents ***TorqueDB***, distributed service to store and query time-series data. It makes use of *ElfStore*, which reliably stores blocks of data on transient edge devices and offers federated indexing and Bloom filters to locate data. In our implementation, the high read and write throughput of *InfluxDB* is leveraged to ingest and query streaming data. For a query on time-series data made by a client on the TorqueDB, there are three locations of execution- the edge, the fog, and the client itself. In the current deployment, TorqueDB can respond to a client's queries by performing the computation on the fog node.

TorqueDB utilizes *query optimization* and *query re-writing* techniques to decide at run-time, the device(s) from where the block(s) should be retrieved. Our experiments show that TorqueDB determines the optimal path of execution of queries and returns accurate query-responses with minimal latency.

ABSTRACT

The rapid growth in *edge computing* devices as part of the *Internet of Things (IoT) applications* allows real-time access to time-series data from thousands of sensors. Such sensor data is often queried upon to serve the purpose of an application or to optimize the health of the infrastructure.

Recently, with the *increased computing* and *storage* capabilities of the edge devices, edge storage systems are being used to retain data on the edge rather than moving them to a central database/VM on the cloud. However, such systems do not support flexible querying over the data spread across tens to hundreds of devices, and certainly not for geo-distributed edge devices. Since edge computing is still evolving, there is only limited work done on distributed time-series databases that can run on the resource-constrained edge devices.

This research proposes ***TorqueDB***, a distributed *query engine over time-series data* that operates on the distributed edge and fog resources. TorqueDB leverages prior work on ElfStore, a distributed edge-local file store, and InfluxDB, an open-source time-series database, to enable temporal IoT queries to be re-written and executed in parallel, across multiple fog and edge devices. Interestingly, the data required for a query is moved into InfluxDB *on-demand*, while retaining the durable data within ElfStore for use by other applications. As an optimization, TorqueDB also uses a *cost model* that maximizes parallel movement and execution of queries across resources and utilizes *caching*. To emulate real-time scenarios, our experiments on a real edge, fog, and cloud deployment. The results show that TorqueDB performs comparably to InfluxDB on a cloud VM for a Smart City query workload but without the associated monetary costs.

1. INTRODUCTION

1.1 About the project

Recent research work in IoT and distributed computing evidently show, that to meet the short response times of Internet of Things applications, the deployment *architectures* are becoming more and *more edge-centric*. This is understandable since the higher one goes up in the hierarchy, the greater is the communication latency. Given the proliferation in edge computing and IoT, *time-series databases* (TSDBs) have become hugely popular[1], as seen in Fig. 1.1. A lot of work has already been done in de-congesting the network, data aggregation, retention policies, and locality of data. Although these works provide a solution to various problems faced in IoT applications today, they fail to look at the problem holistically. This means that future architectures need to be modified on a case-by-case basis, which is not a long-standing solution.

TorqueDB presents a holistic *distributed querying service over time-series data* at the edge. In TorqueDB, data is stored at the edge, with replicas created to enhance reliability and spatial proximity. A major contribution of our design is on *query optimization* and *re-writing* to enable the system to respond to client queries even in *tight-time constraints*. The project makes use of InfluxDB as the engine to handle concurrent reads and writes on time-series data. The results of the experiments illustrate the efficient querying ability of TorqueDB that utilizes query optimization models to *intelligently plan* and *execute queries* at different edge and fog devices.

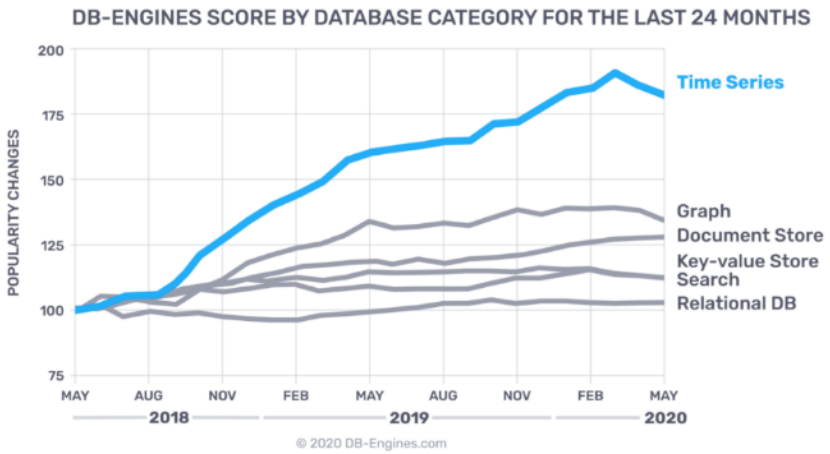


Figure 1.1: Growth of time-series databases (Source: influxdata.com)

[1] https://www.influxdata.com/time-series-database/

1.2 About the company (Ericsson India Global Services)

The telecommunication industry is one of those industries that touch billions of people, everywhere daily. Ericsson is one of the frontrunners in providing *wireless network infrastructure* and Information and Communication Technology (ICT). It has operations across 180 countries.

Ericsson aspires[2] to *"empower an intelligent, sustainable, and connected world"* by making communication available for all. The company has been developing smart tools and intelligent technologies for many decades together. Not just that, the company's vision is to make their solutions available across various sectors of the society, leading to impactful positive change.

As of 2018, Ericsson had a significant market share (approximately *27%*) in the 2nd, 3rd, and 4th generation wireless network infrastructure.

The main offerings of the company lie in the following areas:

- *services, software and infrastructure in communication technology* for telecom operators
- *traditional telecom solutions*
- *Internet Protocol (IP) networking*
- *wired and wireless broadband*
- *business support services and operations*

Research is of prime importance for an evolving technology company like Ericsson[3]. Sharing research results with a wider audience – research communities, industry analysts, and the interested public, in addition to customers and partners – is also vital for the company's *visibility* and *recognition*.

The company has also been the driving force behind numerous powerful technologies known to mankind. Ericsson invests nearly *15% of its annual revenue* into research, and it is not surprising that it holds more than **54,000 granted patents**[4]. It is also credited with being the inventor of Bluetooth technology. They strive to use these

[2] https://www.ericsson.com/en/about-us/our-purpose
[3] https://www.ericsson.com/en/future-technologies
[4] https://www.ericsson.com/en/patents

cutting-edge technologies to solve real-life problems and business needs, which leads to profound changes in society.

Ericsson is also one of the *frontrunners in 5G deployment* and has launched live commercial 5G networks on four continents. In early 2017, Ericsson, SK Telecom, and BMW Korea ran a demonstration for data transmission speeds on a 5G network. The run used an advanced 5G network to follow a connected vehicle moving with speeds of up to 170 km/hour. Using advanced technologies like beam tracking and beamforming, the high-performance network connection was able to support point-to-point data transmission from a connected vehicle with downlink speeds of approximately *3.6 Gbps*[5]. This opened the avenues for cutting-edge 5G services and delivered yet another example supporting seamless mobility.

Today, their solutions and technology are used to connect *2.5 billion subscribers* to 2G, 3G, 4G, and 5G, i.e. nearly one-third of the global population.

1.3 About the research group (DREAM:Lab, IISc Bangalore)

The DREAM:Lab[6] (Distributed Research on Emerging Applications and Machines Lab) works on research in *distributed systems* for the efficient use of computing systems and paradigms. For the growing large-scale data, scalable software and architectures need to be designed. This involves the development of innovative programming methods, data abstractions, and algorithms. *Systems utilizing distributed computing* to support data and compute-intensive applications (be it scientific or engineering), could lead to huge advances in society.

The lab is housed at the Department of Computational and Data Sciences (CDS) at the Indian Institute of Science (IISc), Bangalore. Prof. Yogesh Simmhan[7] heads the group and the lab explores various facets of the data science stack, from *Big Data platforms* to *data-driven applications*, and to *emerging distributed infrastructure*.

[5] https://www.ericsson.com/en/news/2017/2/ericsson-sk-telecom-and-bmw-group-korea-reach-new-world-record-speed-with-5g
[6] http://dream-lab.cds.iisc.ac.in/about/overview/
[7] http://cds.iisc.ac.in/faculty/simmhan/

The lab explores the following broad topics:

- *Data-Intensive Applications*
    - CPS, Internet of Things, Smart Cities
    - Deep learning, Social networks
- *Abstractions and Algorithms*
    - Programming Linked & Fast Data
    - Metadata, Provenance, Semantics
    - Distributed Graph Algorithms
- *Big Data Platforms*
    - Stream Processing Platforms
    - Graph Analytics Platforms
    - Optimization, Heuristics
- *Distributed Machines*
    - Cloud & Commodity Clusters
    - Heterogeneous Accelerators
    - Low Power Edge & Fog Devices

The research at DREAM:Lab endeavours to advance solutions to problems by effectively *scaling data-intensive applications* on both *emerging* and *contemporary* distributed computing infrastructure. Research in the lab emphasizes on taking an *integrated view* across the deployment stack, from the system to the application. This reduces the chances of researching under idealized conditions which might be impractical and detached from reality. This is especially important for research in the computing systems area where changes in computing technology and hardware are fast-paced. However, it is also kept in mind that the projects undertaken should not "re-invent the wheel" by going into building systems or software from scratch.

This practical grounding is aimed at demonstrating the interdisciplinary research, and to train collaborators for future advancements in technology.

2. PROJECT DESCRIPTION AND GOALS

2.1 Background

Internet of Things (IoT) applications leverage the availability of large-scale affordable sensing and computing devices. Along with pervasive communications and advances in analytics, the IoT applications can observe and manage cyber-physical systems to *enhance* their *efficiency* and *resilience*. IoT applications span across multiple domains: from large physical infrastructures such as Smart Cities, Smart Transportation, and Industrial IoT, to consumer devices such as smartwatches and smart appliances. A key characteristic of IoT applications is their *closed-loop cycle*, where the data captured about the system is analysed and decisions are made to control the system, typically *within seconds* [10, 13]. E.g., in a manufacturing facility, sensors may monitor temperature and pollution levels to ensure they are safe for workers, and if not initiate cooling, scrubbing, or other safety measures.

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 decisions* onto their co-located actuators [12]. Traditionally, data collected from the field is sent to the Cloud for storage and analytics, and the control signals are sent back to the field. This introduces *performance variability* and a *high* network round-trip *latency* between the edge and the cloud. There is also an additional network, compute, and storage costs that the user pays for the services used at the Cloud data centre.

Edge computing has gained prominence to utilize the *idle* compute power and storage on edge devices, as well as to reduce the end-to-end decision-making time caused by the network latency between the edge and the cloud. Besides running tasks and analytics on such devices [2, 3, 10], recent works also propose their use for distributed data storage by offering file and block-based semantics for data update and access [5, 8, 9]. They also employ *workstation-class fog* resources located close to the edge devices, which help with management and additionally as a gateway to the Internet.

2.2 Motivation

IoT data is usually time-series in nature since sensors continuously generate timestamped data, and the application requirements determine the granularity of the sensed data. As a result, *querying* and *analytics on* this *time-series data* is a key requirement for IoT applications [7, 13]. These operate on data collected over time to check if recent observations exceed historic averages, identify the minimum and maximum outliers within time-windows, query, and visualize data from specific sensor types and time ranges. This set of queries and analytics complements and is more flexible than Complex Event Processing (CEP) and publish-subscribe systems. The latter two usually operate only on real-time data and limit the queries possible [3, 6, 11].

Time-series databases (TSDB) such as InfluxDB and Apache Druid are popular for hosting historical and real-time IoT data and performing *temporal queries*, centrally on the cloud [7]. However, in such cases, the data generated at the edge is moved to the cloud which adds network variability when edge applications send queries to the cloud databases. Also, for IoT applications that utilize *privacy-sensitive data*, retaining the data *within the private network* of the source (such as edge devices), and querying over it may be necessary.

Fig. 2.1 demonstrates the steps for event processing at the edge and the cloud. It motivates the need for applications to utilize the edge resources to perform non-complex computations. This way of *pushing tasks towards the edge* could also save on the latency and network transmission time involved in processing at the cloud.

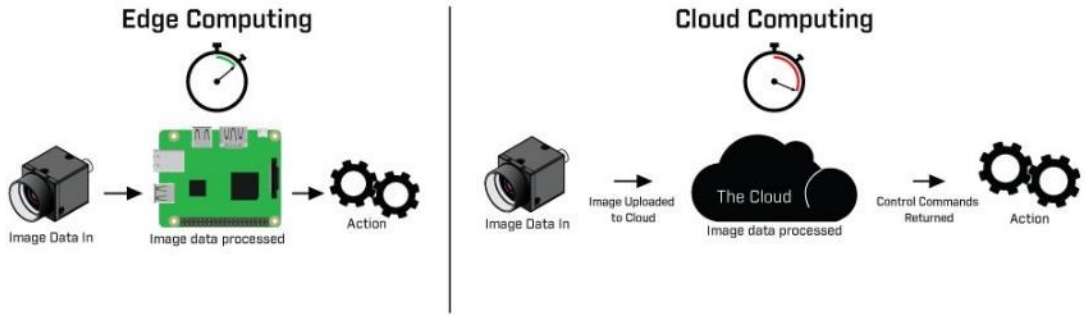


Figure 2.1: Event processing and response in the edge and the cloud

2.3 Gaps in the existing work

As the number of applications trying to *move towards more localized compute-resources* increase, a natural progression is to host time-series databases on edge and fog devices. This enables us to *co-locate the query clients* near the *data storage* and also leverages the available local compute and storage capacity on them [1].

However, individual edge or fog resources may *not have the capabilities* to scale to workloads that require the use of multiple edge clients. This motivates the use of a *distributed TSDB* operating across multiple edge and fog devices, which are currently unavailable. They are *either proprietary* or are *not light enough* for the edge to simultaneously run the distributed TSDB and other necessary applications.

Further, it is also true that not all time-series data collected over time will be actively used, and only recent or actively used datasets need to be stored in such TSDBs. Lastly, storing such sensor data in the TSDB will need to be complemented by storing it durably as files on the edge, say, to support other time-series analytics or machine learning models that operate outside the database on the edge devices [13].

All of the above-mentioned gaps are addressed in this project.

2.4 Contributions and objectives

This project proposes ***TorqueDB*** (***T***emp***or***al ***qu***erying from ***e***dge storage ***D***ata***b***ase) which leverages the ElfStore distributed edge-local storage [8] along with InfluxDB TSDB. This offers a *distributed execution model* for time-series queries over multiple edge and fog devices in a cluster.

Here, *ElfStore* stores and manages 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, ingested into the TSDB on-demand, to support user queries. TorqueDB accepts queries defined using the *Flux language* used by InfluxDB, and further uses the metadata and search capabilities of ElfStore to identify blocks of interest.

The data of interest is inserted and cached into one or more local InfluxDB instances on the fog, that *executes subsets* of the *user query* on each fog in *parallel*. Finally, the results are aggregated and returned to the user.

This effectively offers a *distributed TSDB* utilizing the captive storage and compute resources, with an edge-centric data store. ***TorqueDB*** is *one of the first* systems to offer ***distributed time-series querying over numerous edge and fog devices.***

Next, in § 3 ElfStore, InfluxDB, and related work on edge computing and querying are discussed; the TorqueDB architectural design and distributed query execution model is introduced in § 4; some key optimizations for improved query performance are described in § 5; detailed performance results on a real-world edge and fog deployment, in comparison with a Cloud VM are presented in § 6; conclusion and future work are discussed in § 7; references in § 8; and lastly an appendix section to include some code snippets in § 9.

3. BACKGROUND AND LITERATURE SURVEY

From the recent work, there is an observable trend in storage and locality of data, latency constraints in applications, TSDBs, and query mechanisms for IoT time-series data.

3.1 Latency critical IoT applications

With the ongoing digital revolution, increased scalability and efficiency across operations have become a constant endeavour. The manufacturing industry is now on the cusp of *Industry 4.0* and the *digital evolution* of the industry empowered by growing Internet of Things (IoT) adoption allows many new avenues for enterprises to create value [15]. *Connectivity* and *analytics* are seen as key facilitators for personalized goods recommendations, faster delivery chains, and shorter machine downtimes [16]. To achieve these goals, data collected from the connected assembly line must be leveraged. While there are several opportunities for analytics in manufacturing, the *voluminous data* generated must be *stored* and *queried optimally*.

Ravindran, et al. [17] stated that latency criticality in challenging operating conditions at the edge can only be achieved through application specific guidelines. They maintained the latency criticality of one class of data by designing *two latency tuning knobs* that tuned the latency associated with channel interference and bufferbloat. To reduce bufferbloat, they used key-frame similarity to discard matching key-frames-thus, *trading off accuracy for latency*.

Bharde, Madhumita, et al. [18] proposed the streaming of IoT data such that the data is persisted in changed log files and replicated at regular intervals using an asynchronous replication protocol. For two applications (video streams and time-series data), they developed a plugin to *detect semantic similarity* at the edge, *dropping redundant data* by matching with previously generated representative coefficients. Here again, there is a trade-off between accuracy and latency.

3.2 Data management for IoT

Psaras, Ioannis, et al. [19] analysed a major problem with the IoT paradigm: voluminous data transfer from the edge to the core. They state that this causes *network congestion* and sensors *mobility isn't handled well* by IP networks. In their paper, they provide a data-centric communication approach to enhance network connectivity using *local storage resources*. They leverage edge storage (base station, Wi-Fi APs, or Access Points) to process and temporarily store data. The data is then asynchronously synced with the cloud following the best strategy depending on data/application requirements.

Aral, et al. [20] described a *decentralized* approach for *replication* and *placement of data* on the edge devices in a network. They focused on performance benefits of replication: reducing latency and bandwidth consumption through replica client proximity. The storage nodes act as local optimizers by evaluating the *cost of storing* replicas as well as the latency *improvement on migration* or duplication of replicas to one of their neighbours.

In 2013, Abu-Elkheir, et al. [21] stated that the data received from IoT sensors had several distinctive features that made *relational-based database* management systems *incapable* of effectively handling the data. Their paper elucidated the design primitives essential in an IoT data management solution. Based on the 8 stage IoT data life-cycle described by them, they envisioned a framework that would efficiently manage IoT data.

3.3 Distributed Edge-local storage

ElfStore [8] is a *block-centric distributed storage* system on distributed edge and fog resources, for files that grow over time. They note that the data captured or generated near the edge and fog devices is only *transiently available* before it is moved to the cloud. Thus the applications making use of such transient data, are either forced to run on the cloud or retrieve it to edge for computation. ElfStore attempts to address two key gaps: (i) The lack of a holistic *data access service* from which applications can consume the data at the edge and cloud, and (ii) *avoiding data relocation* for computation by managing data at the edge and fog level.

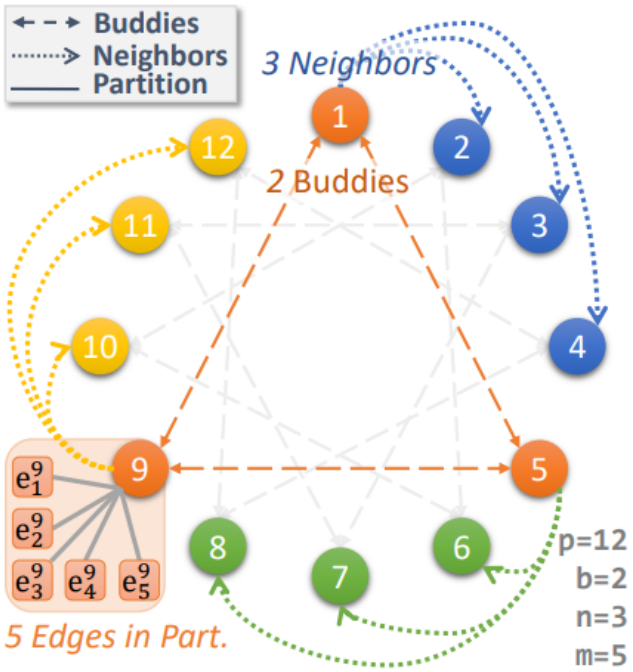


Figure 3.1: Network overlay of buddy and neighbour fogs (Source: ElfStore [8])

Multiple edges of varying reliability are connected to a parent fog that is present in their local network, and the edges and parent fog together form a single *fog partition*. The *edges* serve as primary locations for storing data, and enhance the data locality; while *fogs* are used for data management and discovery. Many such fog partitions can exist, and the fogs can talk directly to each other. All these different fog partitions form a *peer-to-peer (P2P) network overlay*, with the edges serving as peers and fogs as super-peers, and its associated scaling characteristics to 1000's of devices. The store can then scale horizontally with edges that join or leave the system. Fig. 3.1 shows a fog super-peer overlay with b=2 buddies and n=3 neighbours. It also shows fog device 9, which maintains the details of two buddies <1,5>, three neighbours <10,11,12>, and five edges <1-5>.

Edge devices host *data replicas* and *metadata* for the blocks. The number of replicas for each block is determined by the replication factor of the stream, and the locations of the replica are decided based on the global statistics available at the fogs. Replicas are distributed across fog partitions to maintain *reliability* and have *sufficient availability*.

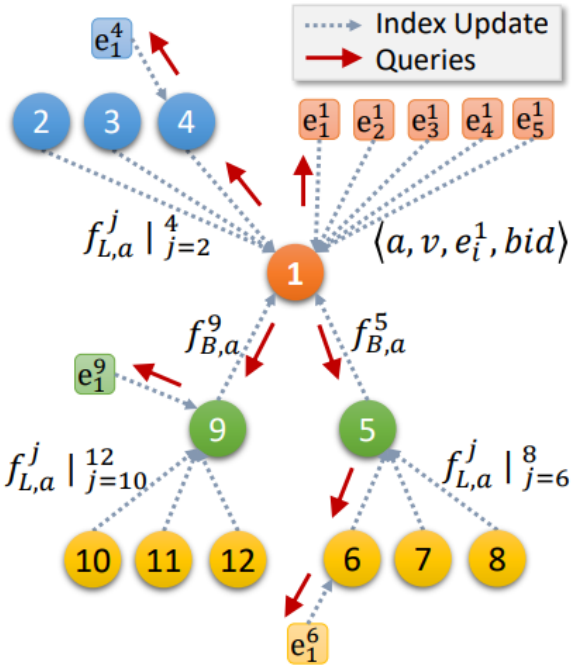
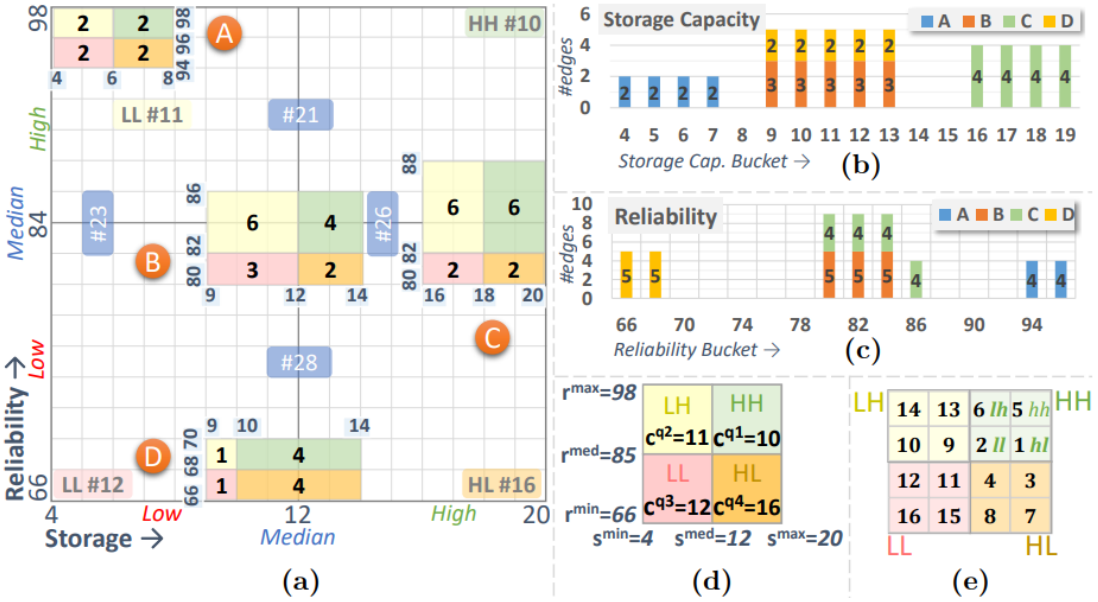


Figure 3.2: Index updates and search query from a fog (Source: ElfStore [8])

As ElfStore is made for IoT scenarios and can be run on the commodity hardware cluster, there exists a *heartbeat* mechanism for the health monitoring of devices. Heartbeat messages *piggybacked* with *metadata* and *statistics* are sent from edges to parent fog. A prolonged duration of missed heartbeat messages indicates a loss of edge node, which triggers the re-replication of blocks present on the missing edge.

Users search over the stream and block stream metadata to identify the block IDs that need to be accessed. Fog devices maintain a mapping from block ID to edge(s) and *indexes* over the block metadata, for all the blocks present in their local partition. This allows fog devices to perform *value-based search operations* for blocks based on their metadata properties, and lookups of block replica locations using the block ID associated with each block. Fogs also use *Bloom Filters* to maintain approximate indexes of content in other fog partitions to facilitate forwarding of metadata search and block retrieval requests across the overlay, within *no more than 3 hops* per block search. However, it must be noted here that bloom filters could also return false positives in some situations. Fig. 3.2 gives an overview of the index update and query process through communication between the fogs.



(a) Global Mapping across fog partitions, (b) storage and (c) reliability of edges across partitions, (d) count of edges and (e) test sequence

Figure 3.3: Populated reliability and storage Global Matrix (Source: ElfStore [8])

Since edge devices can have asymmetric reliability and are typically run on commodity hardware, ElfStore uses block specific replication level. Each edge periodically reports its reliability storage to the parent fog, as heartbeats. The parent fog then determines the median, minimum, and maximum storage and reliabilities for the edges, along with the count. The *counts* correspond to high or low storage capacity and reliability- `HL`, `LH`, `LL`, and `HH`. A local matrix is thus obtained, having the count of edges in the respective four quadrants. Periodically, the *statistics are exchanged* between the buddy and neighbour fogs as heartbeats. Using the tuples obtained from the different fog devices, each fog can *independently* and *consistently* construct a *global* storage and reliability distribution *matrix*, shown in Fig. 3.3. The statistics are then used by the *replication logic* to guarantee a minimum resilience and load balancing of storage.

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.

3.4 Time-series databases

Time-series databases (TSDBs) are optimized for time-stamped data. Such data consists of measurements that are *monitored*, *down-sampled*, and *aggregated* over time. The data lifecycle management, summarization, and large range scans in time-series data make it *different from other data* workloads. The choice of the database is extremely crucial for the applications deployed on the resource-constrained hardware. Several popular TSDBs were evaluated before deciding to use InfluxDB, for this implementation.

Joshi, et al. in [22], provide an analysis of InfluxDB, MariaDB, TimescaleDB, and Redis *databases for edge-centric architectures*. When they conducted experiments for the write workload, they found that while MariaDB gave the highest throughput for large batch sizes, InfluxDB outperformed all others on increasing the number of threads concurrently writing to the database. In the read workloads, they observed that with InfluxDB, the latency increases as the result size increases. InfluxDB, an open-source TSDB, shines when faced with a mixed workload of reads and writes: it provides a good read throughput while still maintaining its write throughput.

Besides the observations from the above paper, experiments were run to verify the write throughput of InfluxDB. Its excellent performance in concurrent write workloads is in line with Industrial IoT use-cases which have numerous sensors on the assembly line, requiring *concurrent writes* from multiple devices. Additionally, there will also be many *concurrent queries* on the InfluxDB instance from the monitoring systems deployed. Given that InfluxDB can is optimized for high read and write throughput while maintaining its write throughput, it was a good fit for our implementation. The *InfluxDB storage engine*[8] draws inspiration from the *LSM Tree* and maintains a Write Ahead Log (WAL) alongside read-only data files, similar to SSTables in an LSM Tree.

The storage engine ties multiple components together, each of which serves a particular role:

- **Write-Ahead Log (WAL):** It is a storage format optimized for *fast writes*. The new data points that arrive are compressed and stored here. However, WAL files are *cannot be easily queried*.

[8] https://docs.influxdata.com/influxdb/v1.8/concepts/storage_engine/

- **Cache:** It is the *in-memory representation* of the *data* stored in the Write-Ahead Log files. To respond to user queries, the data from the cache is merged with the data from the TSM files.
- **Time Structured Merge (TSM) Tree files:** These files are *read optimized* and store *compressed series* data.
- **Compression:** The blocks are compressed to minimize the storage space and Disk IO while querying. The *compression techniques* for the timestamps and values depend on the encodings given for different data types and shapes.
- **Compaction:** It is a *recurring process* that migrates recent data (currently in the write-optimized format) into a more read optimized format.

InfluxDB has been built from scratch to store and run queries on time-series data. The above features in the storage engine make it possible for InfluxDB to have excellent ingestion and querying throughput. The TSDB stores data in *buckets* (databases) which in turn contain *measurements* (tables). Each row in a table has a timestamp and columns that are either *tags*, which are indexed or *fields*, which can be aggregated on. Since tags are *indexed*, queries utilizing them benefit in *shorter execution times*. InfluxDB 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.

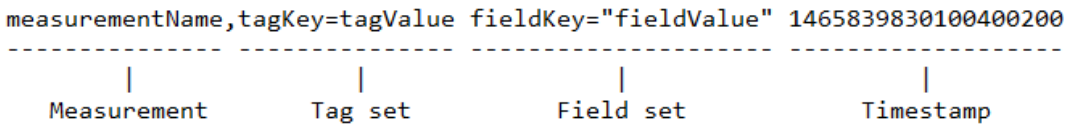


Figure 3.4: Native Line Protocol format of InfluxDB

When a Line Protocol line in the format shown in Fig. 3.4 is ingested into InfluxDB, the group of fields given in the line protocol are *decomposed into multiple lines*, each having one fieldKey and one fieldValue. E.g. a line-protocol line $tag_1=v_1,tag_2=v_2$ $field_1=v_3$, $field_2=v_4$ $time_i$ is decomposed into *two* separate *records*. The first record being $tag_1=v_1,tag_2=v_2$ $field_1=v_3$ $time_i$ and the second record being $tag_1=v_1,tag_2=v_2$ $field_2=v_4$ $time_i$. This enables InfluxDB to provide a *flexible schema* for time-series data.

3.5 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 modelling* technique to match complex patterns on data streams. The system compiles the 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 *not recomputed but reused*. 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 for recent queries could probably be *reused* to answer upcoming queries. Zhou, et al. [14] presented a *popularity-based caching strategy* to leverage these patterns. They show a significant *reduction in the communication cost* when a large number of queries need to be answered. Such caching strategies can also be incorporated into TorqueDB.

HERMES [6] enables query evaluation over data streams across cloud and fog nodes. They use *reservoir sampling* of incoming observational streams to the memory consumption and communication overheads 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 is addressed 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 only edge and fog devices are used, rather than the cloud.

Others have examined query rewriting in other contexts. Schultz, et al. [11] 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 containment and *query rewriting* techniques from databases for privacy-aware and *efficient computation* of queries in an *edge-cloud setup*. The input query is split to form “fragment” and “remainder” queries. The “fragment” queries operate on resource-constrained edge devices to pre-aggregate and filter data, while “remainder” queries operate upon the complex part of the query and execute them on fog devices. Similar rewriting is done across different levels on fog devices.

4. *TorqueDB* ARCHITECTURE AND DESIGN

4.1 System model

The system model for TorqueDB is shown in Fig. 4.1, and it 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. *Edge devices* are expected to have resources comparable to a Raspberry Pi with a 4-core low-power CPU, 1–2GB RAM, and 128GB SD card storage. The *fog devices* 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 the form of *blocks*, which are *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 and each block is identified by a unique *block ID*. ElfStore allows application specific *metadata properties* to be *stored* for these blocks and *searched upon*.

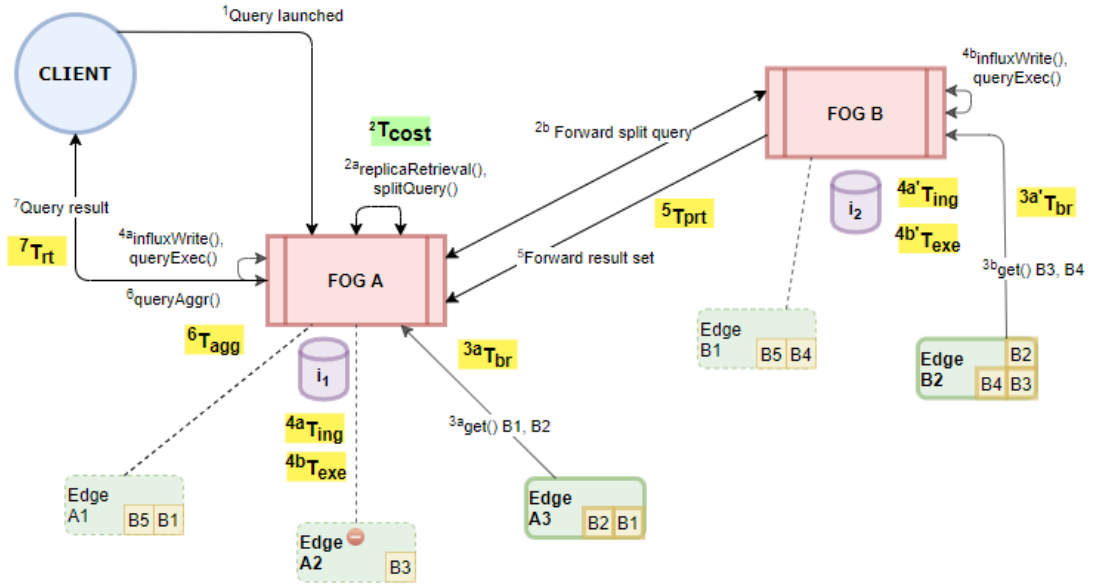


Figure 4.1: System model showing the execution sequence and cost incurred

The application requirements determine the metadata fields for ElfStore. Once the blocks are inserted into ElfStore, *TorqueDB* will be able to *leverage* the *metadata* in *answering the queries*. The metadata contains 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, *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. are also computed and stored as properties for the block. ElfStore 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, an *InfluxDB* instance is *hosted 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.

4.2 Design benefits

The TorqueDB design has several benefits. It *avoids the complexity* of distributed management and resilience of different instances of a TSDB while leveraging the data reliability guarantees offered by ElfStore. ElfStore is designed to be *lightweight* and *data* with its associated *metadata* is *stored on the edges*. On the other hand, TSDBs are unable to run optimally at the edge level and would have to have all the data at one or more fogs. Moreover, since the number of fogs is much lesser than the number of edges, *managing failures*, and *load balancing* is a lot *more difficult in TSDB*. This division of tasks between ElfStore and TorqueDB also allows for edge applications that directly operate on the data blocks to be supported by ElfStore [8] while the queries are offloaded to TorqueDB.

In the current scenario, there are *no distributed TSDBs designed* specifically *for* the *edge*. This also means that there is *no tiered TSDB model* that utilizes both edge and fog resources, contrary to what TorqueDB is able to achieve. Lastly, it *eliminates* the need for *redundant copies* of data on both the edge-local file storage and the TSDB. Although some data is stored in the TSDB as and when required, data redundancy is low given that *less frequently accessed data* is *evicted* periodically.

4.3 Flux query execution

At this time, execution of *Flux queries* is only done on the *InfluxDB* instances running *on fog resources* and InfluxDB instances are not run on the edge devices. This leverages their *higher resource capacity* relative to constrained edge devices and *limits* the *coordination overheads*. For future work, the project aspires to examine designs where the InfluxDB is hosted directly on the edge devices themselves to *enhance parallelism* and *limit data movement*. Even when TorqueDB utilizes InfluxDB at the edge in the future, ElfStore will be helpful for two reasons. Firstly, it is responsible for distributed data storage and management. Since all the data cannot be stored on all edge devices, ElfStore will maintain the mapping for blocks and their replicas present at different edges. Secondly, it maintains the data-files which can be consumed by applications outside the TSDB.

The *execution of* the client's *Flux query* happens in a *hierarchal fashion* where *Levels 4 and 3* (L4, L3) utilise the metadata of blocks from *ElfStore*, while *Level 2* is done by the service, in *parallel across fogs*. Execution at *Level 1* occurs at the *coordinator fog* alone. The splitting up of the original query and the information utilized at the different levels is shown in Fig. 4.2.

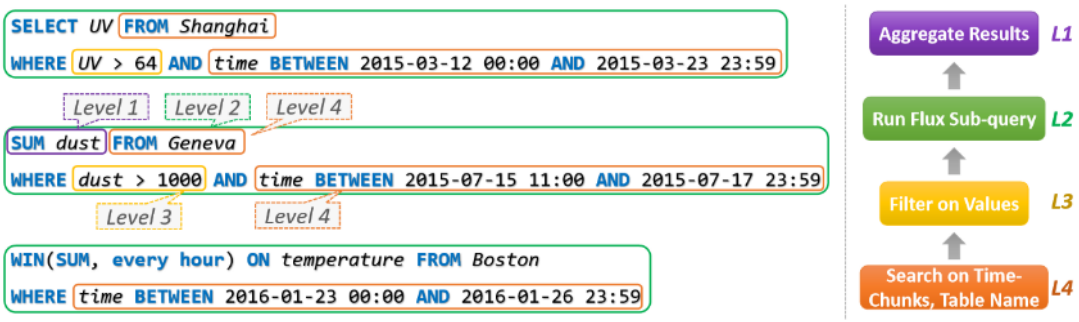


Figure 4.2: Sample Flux queries and execution levels in TorqueDB

4.4 Query lifecycle and distributed execution

TorqueDB supports a *subset of the Flux query language*, as illustrated in Fig. 4.2. Specifically, the set of queries supported are *range queries over time-stamps* (e.g., time BETWEEN start AND end, where the start time is inclusive and end time is exclusive), *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 the future.

The entire query lifecycle and distributed execution sequence are depicted in Fig. 4.3. Users *submit* their *Flux query* to a TorqueDB service, which runs on all fog resources. The fog receiving the query is called the coordinator for this query (Fig. 4.3 ①). The *distributed execution plan* for the query is *decomposed into* a *query tree*, with execution happening at four levels as shown in Fig. 4.2. At level 4 (L4), 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 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.

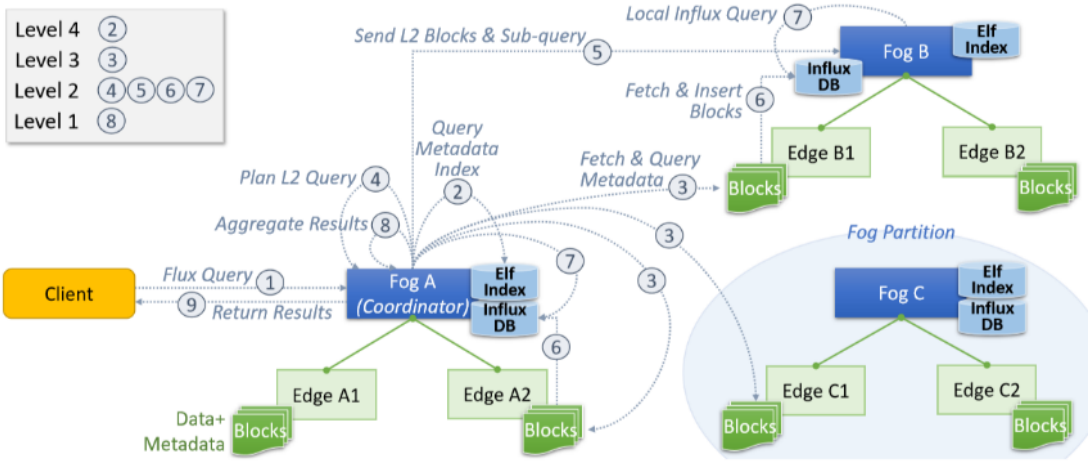


Figure 4.3: Query execution sequence in TorqueDB

However, ElfStore does not support range queries which are important for time-series data. To address this, the time-range for rows in a block is discretized 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 within 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 2020-0214 20:15, these overlap with the time-chunks 89 and 90. *ElfStore is searched* for blocks that have a *time-chunk property* with values of 89 or 90. Likewise, when storing blocks, the chunk numbers for their time-ranges are calculated and are stored 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. 4.3 ③). In particular, when *value comparisons* are to be done over non-timestamp columns present in the data, like "dust" and "UV", the minimum and maximum aggregate metadata values for these columns are used 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 ≥ 1000ppm, then the block and its associated metadata are fetched to eliminate those blocks whose maximum dust is less than 1000. L3 is done only if value comparison predicates are present in the input query. Although queries like min/max can be done at L3 as well, it is thought of only as an idea that might provide *additional optimization* in the future. Taking the min/max from the metadata itself removes the need to fetch a block, ingest it into InfluxDB and then execute the query on InfluxDB. But the optimization will work only in cases where the time range spans the entire block's data. However, for queries that utilize a subset of rows of the block, metadata cannot be directly used to respond to the 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 § 5.2 (Fig. 4.3 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. A *thread-pool* is utilized for the fetch and insert each block in parallel (Fig. 4.3 ⑥). All blocks are inserted into a single table, even across queries. This helps with caching, as is discussed later. During insertion, the block ID is added as a column in each row while it is being 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 duplicate* and *erroneous results*. E.g., if L3 returns block IDs <3,5,9> for processing at L2, and <3,5> are assigned to Fog A and <9> to Fog B. Say Fog B already had a copy of block 5 present in it. If the two sub-queries are run on Fog A and Fog B, the result should not include duplicates for matching rows since 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. It is important to note that even though some blocks might get invalidated due to deletion/compaction, *ElfStore* will *identify* the *missing replicas* and *re-replicate*. This maintains the reliability of all blocks by having cross-partition replicas of the same block. Block read failures for TorqueDB should not ideally occur due to ElfStore.

Once all blocks assigned to fog are inserted into the local InfluxDB, the *sub-query* is *executed* on the TSDB and the *results returned* to the *coordinator* (Fig. 4.3 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, further *aggregation* across all these results is *done 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.

5. OPTIMIZATIONS IN *TorqueDB*

5.1 Need for optimization and the Cost Model

The distributed query execution of TorqueDB gives us a broad overview of the steps involved from the time a query is launched by a client to the time that the client gets the response back. However, there is scope for *optimization* since any given *query* can be *executed through various paths*. In our quest to respond to queries in the shortest amount of time, *pruning of the search space* must be done to *eliminate* the execution *plans* that are *not optimal*.

Some factors that enable us to prune the numerous query plans, for query execution at the fog are stated below.

- **Maximizing data localization**
  Selecting fog partition(s) having the maximum number of blocks, irrespective of whether they have been cached at the fog device.
- **Maximizing parallelism**
  This involves *load balancing across fog* devices that may or may not already have the blocks cached. While this would likely increase the aggregation operations, overall execution time might decrease with the parallel ingestion of blocks and execution of sub-queries.
- **Utilizing caching**
  Priority could be given to the fog devices that already have some blocks inserted into their InfluxDB instances.
- Listed next are a *few additional optimizations* that are not a part of the current TorqueDB implementation, but *could be incorporated* in the future if the *benefits* are found to be *substantial*.
    - An important factor for the selection of edge and fog devices could be the *compute availability* at these devices. Query plans involving edges and fogs with *less CPU compute* available *can be pruned* since they will affect data ingestion into InfluxDB and the query execution performance.
    - Another optimization for the selection of the edge(s). The edges can be chosen based on the *network bandwidth* available between the fog, and the edge.

- Also, for the selection of fog device performing the *final aggregation*, *available compute* could be an important factor. In this case, the estimated *network transfer time* incurred for the transmission of *query results* from different fogs to the chosen fog is also considered.

The overall cost for query execution can be derived from the sub-tasks implemented at different stages.

- `L4` and `L3` time

  In these steps the coordinator fog gets the *block IDs and replica information*, and utilizes the current state of the system to *determine the optimal plan* for query execution.
- `L2-BlockGet` time

  This captures the time taken for the block(s) to be *read from the SD card* on the edge, followed by *transmission* of that block data to the fog over the network.
- `L2-FluxInsert` time

  Depending on the parallel threads spawned at each fog, *pipelined block fetch* and *InfluxDB insert* takes place.
- `Influx-Query` time

  Variations in query execution times are seen based on the *operators* and *query predicates* used. To maximise the performance, the query predicates are kept as lean as possible, and the indexed tags are utilized.
- `L2→L1` time

  The result aggregation time depends on the *transmission* of *sub-query results* among the devices, which in turn depends on the length of the sub-query result and the available bandwidth between the devices.
- `L1-agg` time

  The final aggregation time depends on whether the *operation* being done is trivial (concatenation of results) or non-trivial (mathematical aggregation).
- `L1→Client` time

  The transmission for the final result will depend upon the *result set size* and available network bandwidth between the coordinator fog and the client.

Thus, the cost estimation equation for query execution can be formulated as given in Fig. 5.1 with the explanations for the symbols given in Table 1.

Table 1: Expansion for symbols used in the Cost Estimation equation

Symbol	Represents
b	Total number of blocks to be fetched
h_i	Number of hops required for block to reach i-th node
d_{block}	Data in each block
B_i^j	Bandwidth between the i-th node and the j-th node
d_{node}^{ing}	Amount of data to be ingested at a node
r_{c_i}	Rate of ingestion at i-th node for given available compute
Q_{exe}	Time of query execution
t_{agg_f}	Time taken for aggregation at fogs
t_{res}	Time taken to send result to client

$$
\begin{aligned}
Cost =& T_{\text{L4,L3}} + T_{\text{L2-BlockGet}} + T_{\text{L2-FluxInsert}} \\
&+ T_{\text{Influx-Query}} + T_{\text{L1-Agg}} + T_{\text{L1-Client}} \\
=& \sum_{1}^{\mathbf{b}} \sum_{1}^{\mathbf{h_i}} \frac{\mathbf{d_{block}}}{\mathbf{B_i^j}} + \sum_{1}^{\mathbf{b}} \frac{\mathbf{d_{node}^{ing}}}{\mathbf{r_{c_i}}} + \mathbf{Q_{exe}} \\
&+ \sum \mathbf{t_{agg_f}} + \mathbf{t_{res}}
\end{aligned}
$$

Figure 5.1: Equation for Cost Estimation of query execution

To make TorqueDB more performant, *optimizations are formulated* in the query execution based on the cost estimation equation given in Fig. 5.1. Through a series of *micro-benchmarks* shown in Fig. 5.2, Fig. 5.3, Fig. 5.4 and Fig. 5.5 some factors were identified based on the inferences given below.

- **Inferences from the ingestion benchmark**

  Based on preliminary runs, it was found that *block ingestion* into InfluxDB or $T_{\texttt{L2-FluxInsert}}$ was a *bottleneck*. To check if parallelism was being fully leveraged, *2MB blocks* were *split* into *smaller chunks* by varying the number

of threads doing parallel ingestion. E.g. *2 * 1MB* blocks were ingested in parallel using *2 threads* or *4 * 0.5MB* blocks were ingested using *4 threads*. The observed results as shown in Fig. 5.2 were pleasantly surprising. They showed that the CPU was able to *ingest*, *in lesser time*, the same amount of *data* when *given in smaller chunks* and *more parallel threads*. There was a near *2.71 times speedup* achieved when the number of threads varied from one to eight threads.

- **Inferences from SD card read benchmark**

  As seen in Fig. 5.3, the random read speed on the Raspberry Pi Model 4B devices were in the range of *7.5 MB/s to 10.1 MB/s*. While this disk-read speed is a property of the device, and cannot be altered, it showed us that *multiple concurrent reads* from a single edge device could be *detrimental* to the *block fetch* operation and increase $T_{L2\text{-}BlockGet}$. This called for a *Query Planning* strategy which *avoided selection* of the *same edge for more than one replica* at once, if possible. In case it was not possible to allot only one block ID per edge, the planner must *load balance* among the *edges*, for better performance.

- **Inferences from network latency and bandwidth benchmark**

  From the results of the benchmark shown in Fig. 5.4 and Fig. 5.5, the *bandwidth* between any *edge-fog pair* was around *930 Mbits/s*, while the latency was around *0.6 ms*. This meant that the time to *transfer blocks* from one *edge to* the *parent* fog *or remote fog* would be mostly *the same*. It opened up the possibility of performing *load-balancing across fog partitions* in case the replicas were identified asymmetrically.

- **Improving query execution performance**

  In the case of too *many predicates or* use of too *many filter operations*, it was noticed that the *query execution* time was *significantly longer*. The query performance also deteriorated by *2-3 times* when the *number of series* in the bucket (database) *exceeded 1 million*. To overcome this, the *data was re-organized* slightly and the *re-written queries* remained *as lean as possible*. Queries also now utilized *indexed tags* for faster access to relevant data in the bucket.

Thus, these optimizations in a few stages of the execution could potentially bring about substantial performance gains in TorqueDB.

The *identified stages* to optimize were: `L2-BlockGet` (requires block read from SD card), `L2-FluxInsert` (multiple blocks of data being inserted into an InfluxDB bucket in parallel), and `Influx-Query` (depends on the query operations and predicates).

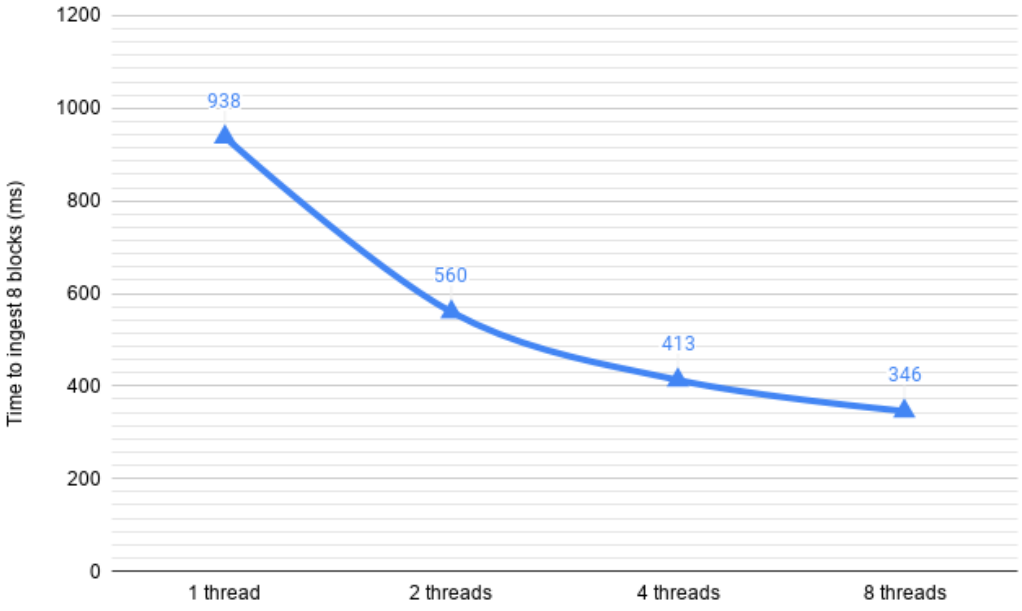


Figure 5.2: InfluxDB ingestion time benchmark

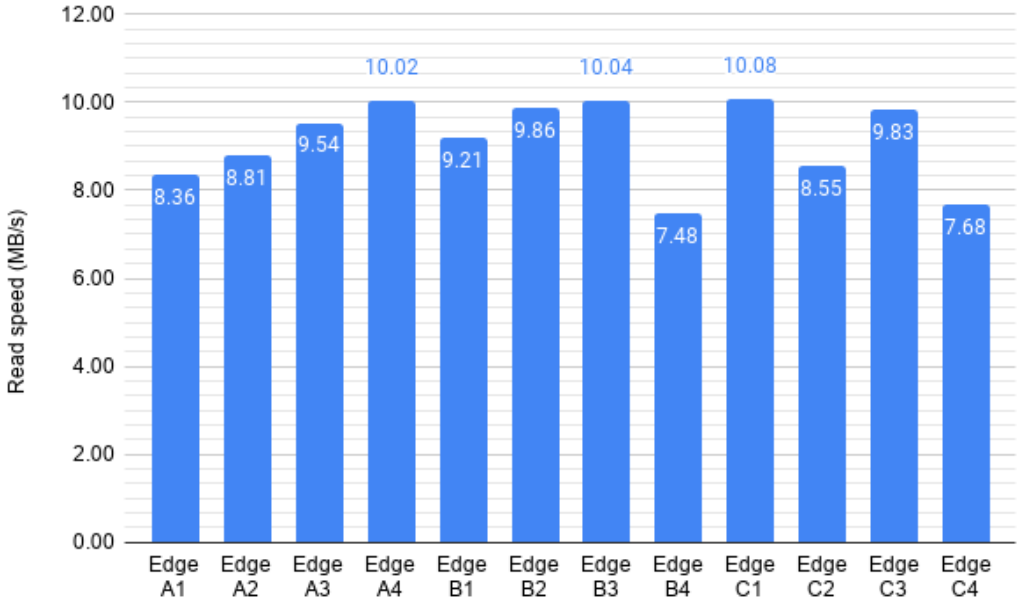


Figure 5.3: Raspberry Pi Model 4B SD Card read speed benchmark

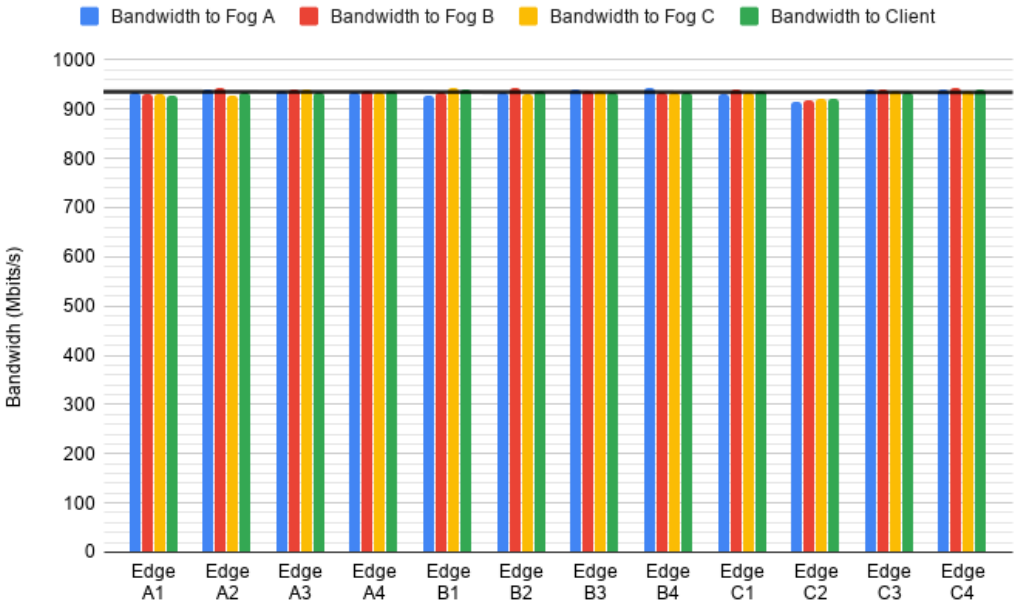


Figure 5.4: Network bandwidth benchmark between edges and fogs

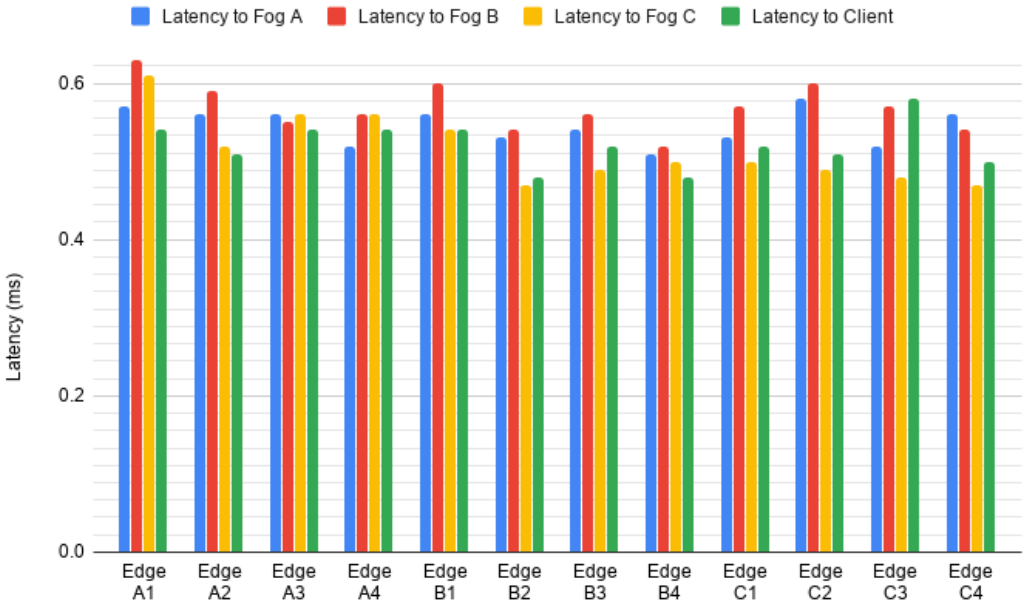


Figure 5.5: Point-to-point latency benchmark between edges and fogs

The next two subsections § 5.2 and § 5.3 describe the changes made for reduced turnaround time for query response.

5.2 Query planning and placement

In L2, *parallel block fetches* from ElfStore are performed, followed by *insertion into* the local *InfluxDB* and *query execution*, at one or more fogs. This is the *most time-consuming* level as it involves fetching the data by *reading files* from the *SD card* on the edge, followed by *network transfer* of data. The *time* taken to *ingest data* into InfluxDB is also *significant*. The aim is to ensure that the parallelism offered by multiple fogs 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. Thus, two query planning strategies are proposed, ***partition-local (QP1)*** and ***load-balancing (QP2).***

The *block transfer* time is *constrained by* the *I/O speed* of the edge device (≈ *100 Mbps* seen for a Class 1 SD card), the *network bandwidth* from the *edge to parent fog* and from *fog to fog (≈ 100Mbps–1Gbps)*, and the *cumulative bandwidth into a fog* (≈ *1Gbps*). In both strategies, the first *goal* is to try and *maximize* the cumulative *disk and network bandwidth* from different edge devices *in parallel*. For this, a *count* of the *blocks* selected for reading *from an edge* is maintained and the blocks assigned to each edge is *initialized to 0*. To prioritize blocks with fewer edges having their replicas, the blocks are *sorted in ascending order of* the number of *replicas* they have. In this order of blocks, one of the edge replicas which currently has the least number of blocks assigned to it, is selected. Before starting the next block-edge replica assignment, the *blocks-assigned count* for that edge is *incremented*. This achieves *load balancing* of the *block-reads* from *among* the *edges* hosting the block replicas.

Next, in the **partition-local strategy (QP1)**, a block replica is assigned 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)**, *balancing* the number of *blocks* assigned *to each fog* is prioritized. This *maximizes* the *parallelism for* the *data inserts into InfluxDB* and the query execution on the fogs. Here, a count of blocks assigned to fog is maintained, which is 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.

5.3 Block caching

Much time in L2 is spent in fetching and inserting the blocks. A *caching* mechanism is thus proposed, 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 fog *for future queries*, but unused blocks are not copied from ElfStore. In the 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.

6. EXPERIMENTAL RESULTS

6.1 Experimental Setup

Our experiments use a *15-device 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 devices* (Intel Core i5 6-cores@2.1 GHz 8 GB RAM and 500 GB HDD). The *edge reliability is uniform* and set to 85%, and *failures are not simulated* in our experiments. Further, to ensure stable bandwidth across all endpoints in the cluster, the use of *WLAN is avoided*. All fog and edge devices are connected within a *LAN network* using *CAT 6/6E Ethernet* cables. These 15-devices form *3 fog partitions* with *1 fog and 4 edges* each, connected over 8-Port TP-Link Gigabit (TL-SG1008D) *switches* with an average *latency of 0.6 ms*. The experiment setup is shown in Fig. 6.1.

As a *baseline*, 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 is used. Its performance is comparable to the fog resource based on query benchmarks.

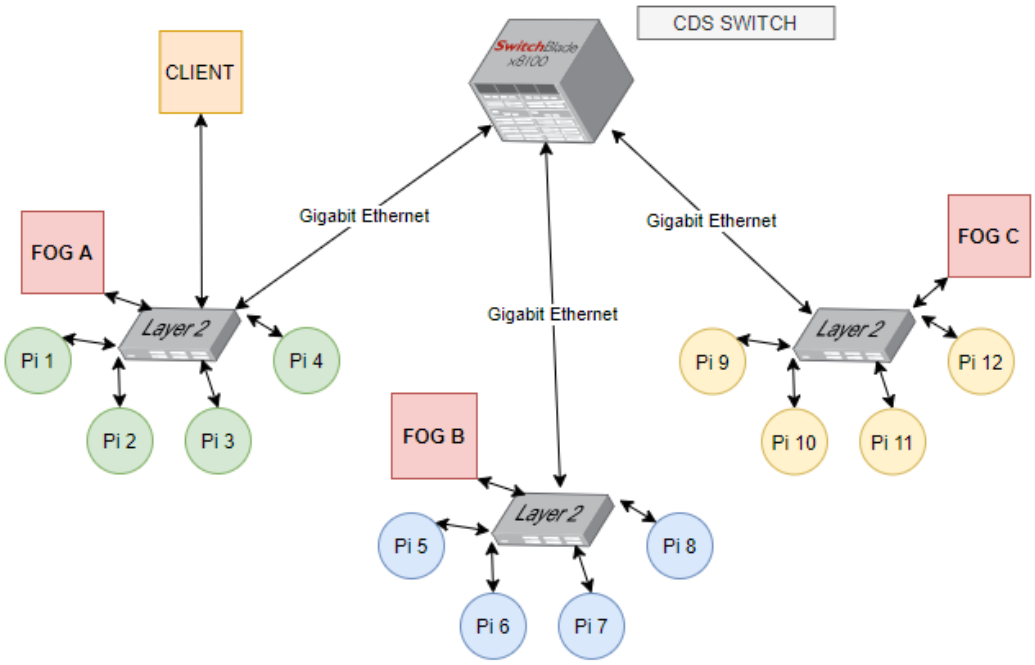


Figure 6.1: Fog partitions and network layout in the experiments

ElfStore runs on the *15-device cluster* with a uniform *replication factor of 3* with no edge or fog failures. *TorqueDB* is implemented in *Java v1.8* and runs on the fogs alongside *InfluxDB v1.7.9*, which is hosted in a container. By *default*, *caching is disabled* on TorqueDB and *QP2 is used*.

6.2 IoT Dataset and ElfStore schema

The *Sense your City dataset* by Datacanvas is used for our experiments. It is a crowd-sourced city *environmental monitoring* dataset with sensors deployed across *7 cities* and *3 continents*. There are about *12 sensors per city*, emitting *5 timestamped observations* of temperature, humidity, light, dust, and UV *every 3 minutes*.

The original dataset included ~14 million rows and was 1.4 GB in size. It covered data for the 7 cities for approximately 2 months. Since the amount of *data per city varied significantly* in the original dataset, the *data* for cities having lesser data *was extrapolated* to ensure that *all cities have* a *uniform number of rows i.e. 2.76 million* rows. This was done by appending previously recorded data to the next timestamps. Also, to simulate *large time range queries* as is often the case in IoT applications, the original sampling rate of 1 row per sensor per minute was increased to *1 row per sensor every 3 minutes*. Thus, the total data now spanned across *16 months*, and this could be used to showcase query results for large time-range queries.

The *modified dataset* for 7 cities now came up to *19.35 million rows* and *~3.28 GB*. As *ElfStore* is utilized as our *data storage and management layer*, the data for each city is chunked into *1MB blocks* before it is ingested into ElfStore. This results in 7 (cities) * 480 (blocks per city) = *3360 blocks of 1MB size*, containing *5760 line-protocol rows each*. *Each block* represents the *24-hour* worth of *data* for one city. The calculation for the total rows in the modified dataset is as follows: 7 (cities) * 12 (devices) * 16 (months) * 24 (hours) * 20 (entries/hr) = 19.35 million rows.

All the *blocks ingested* into ElfStore *have* a corresponding *metadata file* that *forms* the *static metadata* of the blocks. This metadata is of *vital importance for identifying* the *block IDs later* and querying the distributed data. The metadata files for each block for this dataset contains: bucket name, measurement name, block ID, start timestamp, end

timestamp (used for L4 filtering), and minimum and maximum field values for all the 5 environment parameters (used for L3 filtering).

The *stream reliability* used is *0.99*, which *ensures* that there will be *at least 3 replicas* of each block. These *replicas* are expected to be *distributed across fog partitions* as *ElfStore's data placement algorithm* tries to *place replicas in distinct fog partitions* while trying to also ensure that there is *at least one copy* in the local *partition where the block ingestion is initiated*. To ensure uniform distribution of a city's data across the 3 fog partitions of our setup, `Put-block` requests *of each city's data* are *submitted* only *from a single fog*. This means that there is *one fog* which has a *single city's entire data* it's local partition *and* the replica placement algorithm ensures that the *other two replicas* go into the *other fog partitions*.

It is also important to point out here that the *total* amount of *data* stored across the 12 distributed edge devices is *not* equal to *3.28GB* or the size of the dataset. This is because *ElfStore* maintains *3 replicas* of every block. This means that there is *3.28GB * 3 = 9.84GB of data in the setup*. By design, it is also the responsibility of *ElfStore* to *balance the storage utilization across edges and* different *fog partitions*.

6.3 Query workload

The query workload used in our experiments is a *superset* of the proposed 10 classes of IoT queries used in for IoT TSDB benchmarking [23]. This IoTDB-Benchmark aims to establish a *standard for comparing different TSDB systems* and *evaluates* InfluxDB, OpenTSDB, KairosDB, and TimescaleDB. The benchmark takes into account *10 basic IoT query types* and some special data ingestion scenarios. This work not only *measures* the *performance metrics of the TSDBs* but *also* considers the system *resource utilization*.

The query templates used to generate the workload are shown in Table 2. There are 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`). There is also a time-range filter in all cases, with a *small range* being *3 days* and a *large range* being *12 days*.

Table 2: Query templates used to generate the workload

Mnemonic	Query Template
PF	SEL <OBS> <CTY> between <D_S:T_S> and <D_E:T_E> <S/L>
PFF	SEL <OBS> <CTY> where <OBS> <OP> <VAL: 99%ile> between <D_S:T_S> and <D_E:T_E> <S/L>
FSA	<FN: SA> <OBS> <CTY> between <D_S:T_S> and <D_E:T_E> <S/L>
FCA	<FN: CA> <OBS> <CTY> between <D_S:T_S> and <D_E:T_E> <S/L>
FFSA	<FN: SA> <OBS> <CTY> where <OBS> <OP> <VAL> between <D_S:T_S> and <D_E:T_E> <S/L>
FW	<FN: W> <OBS> <CTY>, AGG <INTVL> between <D_S:T_S> and <D_E:T_E> <S/L>

The above *query templates have placeholders*, which are expanded as: SEL (select), OBS (observation– temperature, light, sound, dust, UV), CTY (San Francisco, Singapore, Shanghai, Boston, Bangalore, Geneva, Rio de Janeiro), FN (function), SA (simple aggregate- min, max, sum, count), CA (complex aggregate- mean), D_S:T_S (start date:time), D_E:T_E (end date:time), S/L (small or large time-range), OP (lesser-than or greater-than operator), VAL (value for filter), AGG (aggregate window) and INTVL (interval for aggregate window).

There are *different permutations* for the *values and time ranges in* the *templates* shown above, 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.

6.4 Analysis – Performance of TorqueDB

Fig. 6.2(a) and Fig. 6.2(b) show the *stack bar plots* for the *different components* of the *total execution times* for one *median query* from *each query type* for TorqueDB and centralized InfluxDB on a cloud VM, for small and large time-range queries.

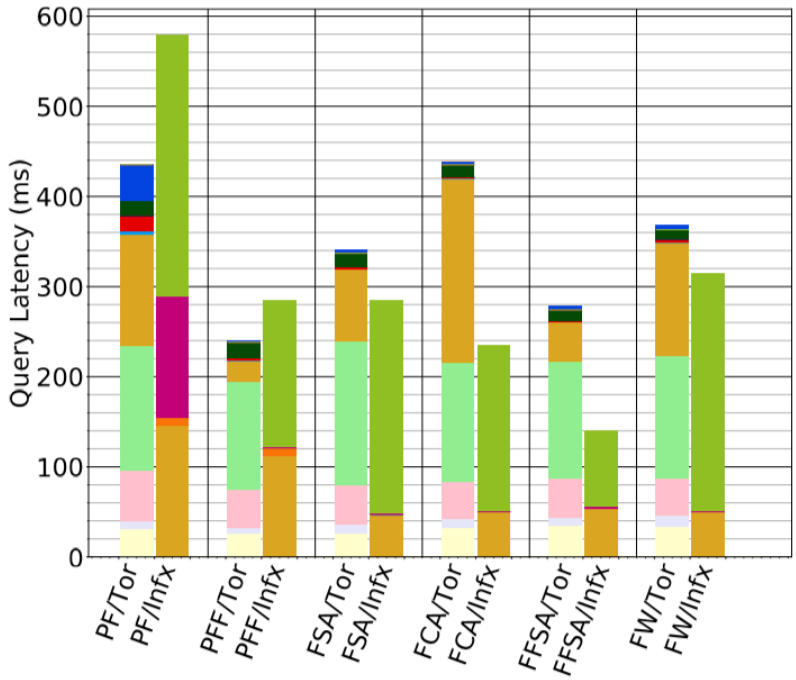


Figure 6.2(a): Small time-range queries

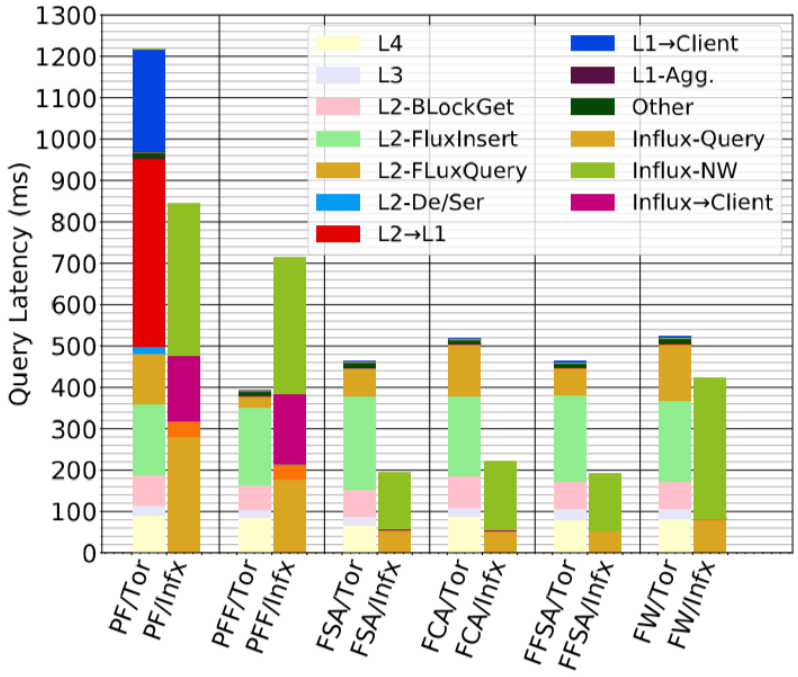


Figure 6.2(b): Large time-range queries

Figure 6.2: Stacked bar plot for median query on TorqueDB (QP2) vs Cloud InfluxDB

All query types, except PF with a large time-range, complete in under *600ms*, with smaller queries running under *400ms*.

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 data* into InfluxDB in L2, *16% in query execution* in L2, and ≈ *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 seen later, these *costs might be mitigated* in certain scenarios *by caching*.

It is seen 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 transferred and rows inserted are the same at *3 blocks*; 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* arises 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*.

6.5 Analysis – Performance of TorqueDB vs Centralized InfluxDB on the Cloud

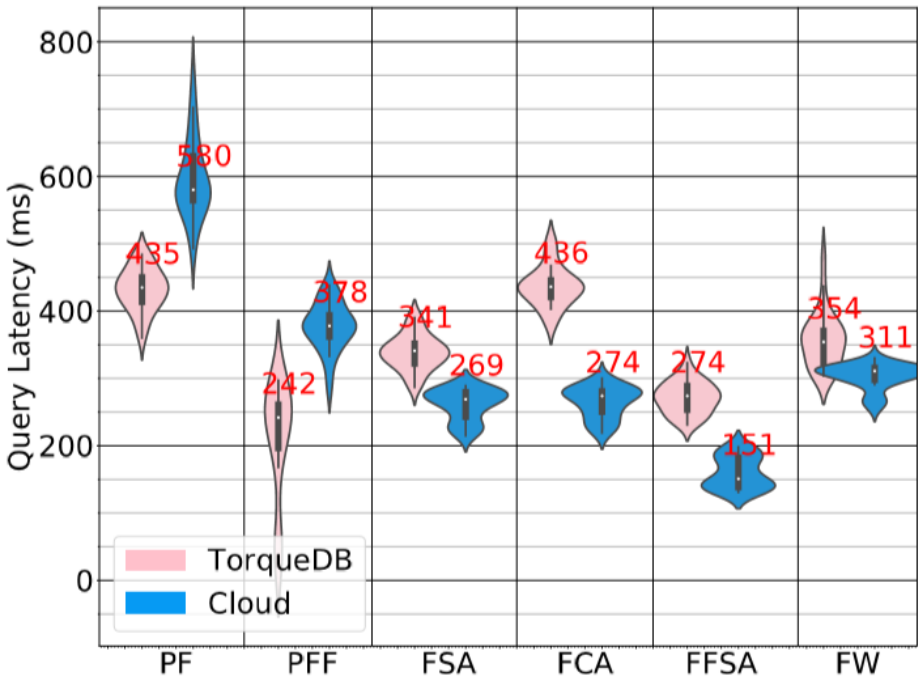


Figure 6.3(a): Small time-range queries

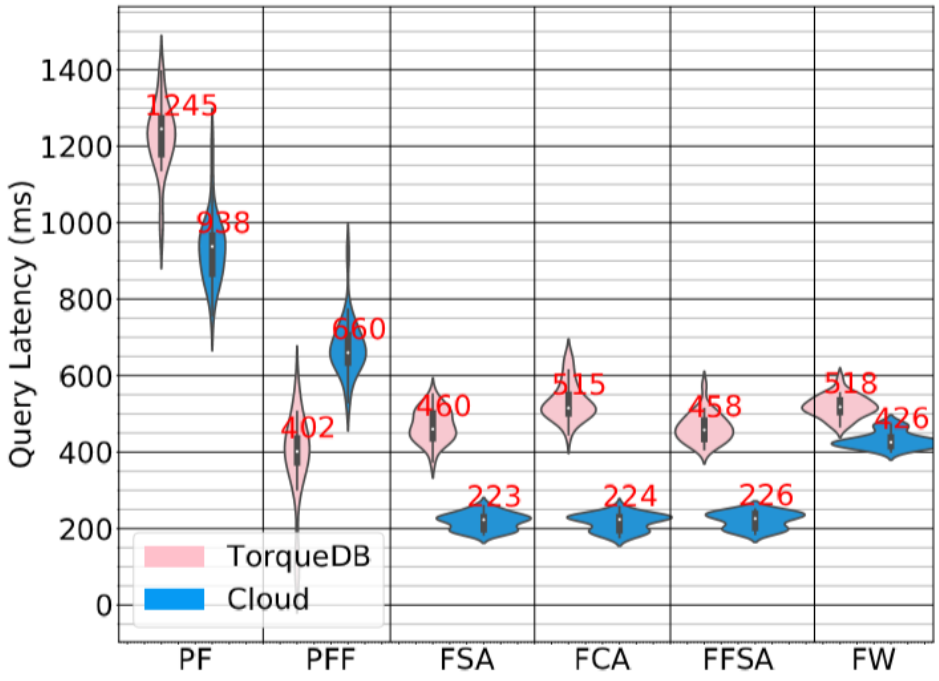


Figure 6.3(b): Large time-range queries

Figure 6.3: Violin plot of end-to-end query latencies on TorqueDB (QP2) and Cloud InfluxDB

Fig. 6.3(a) and Fig. 6.3(b) 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 the other overheads. TorqueDB leverages parallel query execution across local 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 the InfluxDB cloud, which have *smaller query execution times*. These overheads of ≈ *211 ms* take *64% of the total query time*. But this absolute *latency is about the same at* ≈ *255 ms* but relatively smaller, at *57% of the total time*, for the *large time-range queries* having *longer query execution times*. Besides, `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 ≈ *18 mins to transfer 3.28GB of data* for the 7 cities from the edge to the cloud. This is amortized over a long period 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 *24-hours*. This will cause the cloud query latencies to have *less deterministic execution times*, compared to TorqueDB on the edge and fog resources, besides the additional VM and network costs.

Thus, a major benefit of using *TorqueDB* is that *queries are responded to within consistent time bounds* and at *speeds only slightly slower* than the cloud. This is complemented with the *operational savings* since the *costs* associated with using *cloud services is avoided.*

6.6 Analysis – Benefits of Query Planning

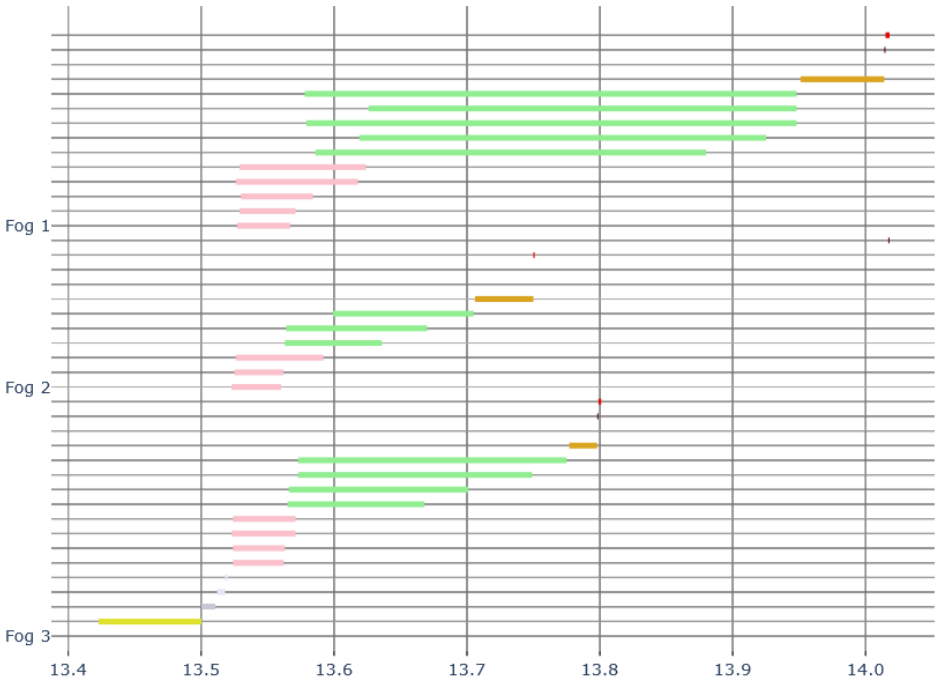


Figure 6.4(a): Partition-local Strategy (QP1)

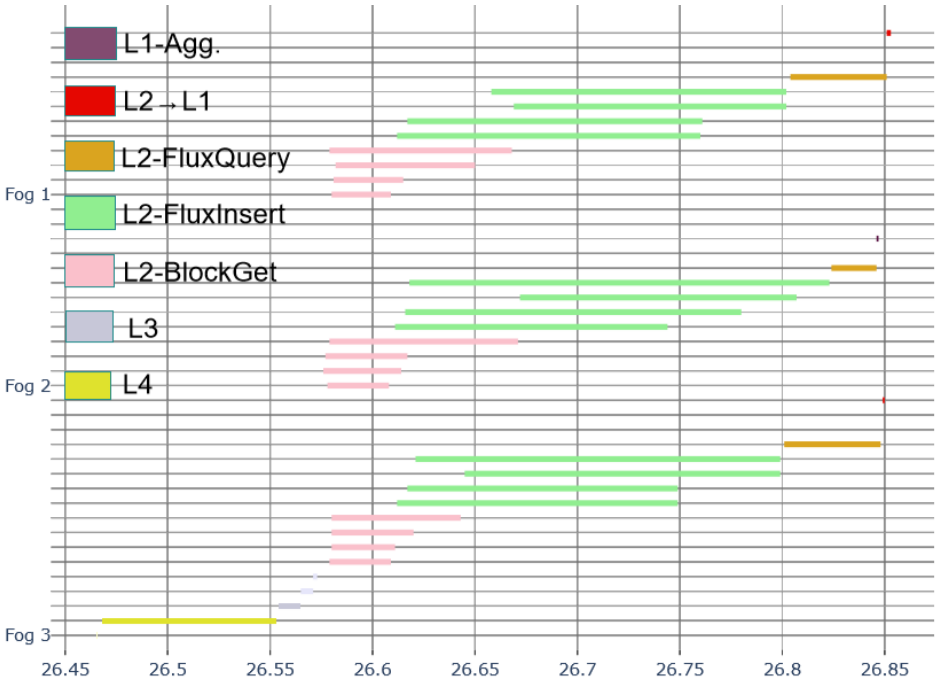


Figure 6.4(b): Load-balancing Strategy (QP2)

Figure 6.4: Gantt plot of latency for a FFSA large time-range query on TorqueDB using QP1 vs QP2

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, it is found 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 (Fig. 5.4) 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. 6.4(a) and 6.4(b) shows the Gantt time-line plot for different time components of an `FFSA` query with a large time-range, running on different fogs when using QP1 and QP2.

The *Y-axis indicates threads* in the 3 fogs and the X-axis is a relative timeline, in seconds. Since this is a large time-range query, 12 blocks in L2 are identified to be fetched. In *QP1, preference is given to send blocks from edges to parent fogs*. Thus, the query planner 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. As a consequence, there is a *mismatch in the blocks to be fetched, ingested into InfluxDB at the fog and queried*. Due to *additional load* on Fog 1 (assigned 5 blocks), the *block fetch and ingestion time increase*. Since, the end-to-end time is determined by the slowest part of the pipeline, the *query latency increases*.

On the other hand, *QP2 instead load balances* and *assigns 4 blocks to each fog*, even though they may cross partition boundaries and *cause 2-hops for the block transfer*. This load-balancing approach can leverage two benefits from the system. First, although there is a block transfer across fog partitions, the *network transfer time is not hit* since the edge-fog bandwidth and latency remains the same across partitions (Fig. 5.4). Second, the *pipelined steps* of parallel `L2-BlockGet` and `L2-FluxInsert` are *not slowed down due to increased blocks*. As a result, *QP2 achieves a ≈ 200 ms reduction* in the L2 block fetch and InfluxDB insert.

6.7 Analysis – Benefits of Caching

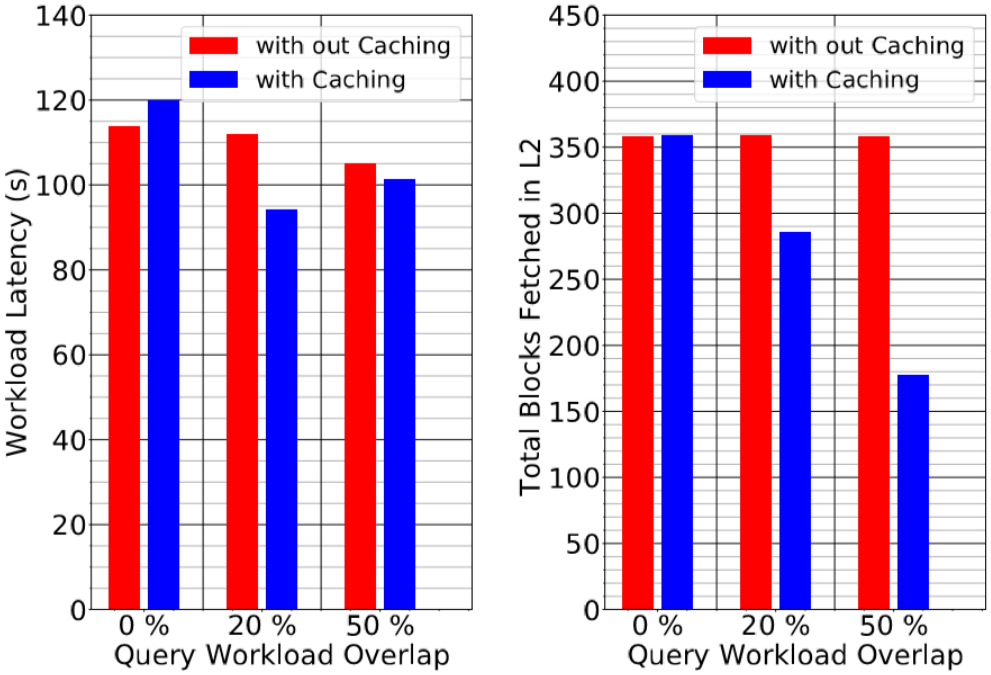


Figure 6.5(a): Small time-range workload

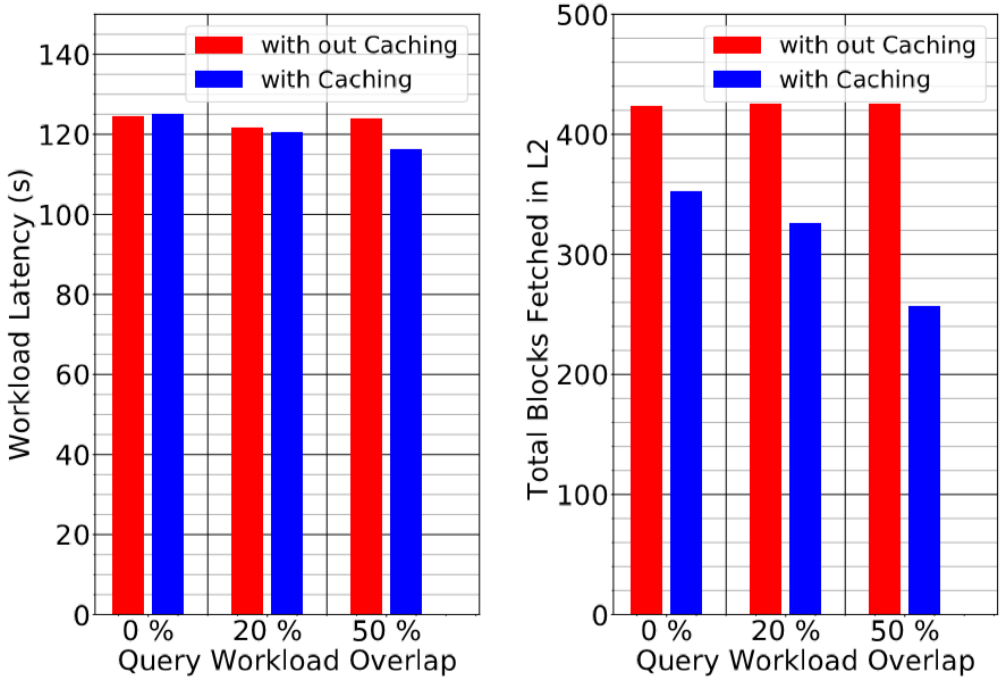


Figure 6.5(b): Large time-range workload

Figure 6.5: Total workload latency and number of L2 blocks transferred, on TorqueDB with and without caching

Finally, the benefits of caching in TorqueDB are evaluated. Here, *query workloads having a mix of 20 queries* from each of the *6 types* are used, to give *120 queries* for the *small* and *120 queries for the large* time-ranges. It is noted here that due to the restricted time range of 16 months in the dataset, and each large query corresponding to 12 days, there are overlaps in time-range among the 120 large queries. However, this is not the case with the small time-range workload, since each small time-range query only requires 3 blocks and the workload will be able to utilize 360 out of total 480 blocks of a city.

Using the 120 queries for small and large time-range, *three query workloads were created*. The first query mix has *no (0%) overlaps in the queries*, i.e., all queries are unique. These set of queries were used 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. 6.5(a) and Fig. 6.5(b) shows the total execution time for these workloads, and the total number of blocks fetched and inserted in L2. These are averaged over 3 runs.

For 0% overlap workload with a *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 queries *overlaps increases to 20% and 50%.*

However, the *impact of the caching* of blocks on the *end-to-end query latency* is *not significant*. This is because it is *unlikely* that the workload contains a *query* where *all 3 blocks or all 12 blocks of the query are already cached*. Since four parallel threads per fog are used in L2, even having one block transferred in L2 can *reduce the benefits of caching* as that becomes the *critical path*. The query latency will the include the time taken for `L2-BlockGet` and `L2-FluxInsert` as well. However, *if all 3 blocks or all 12 blocks* required for the query *are already cached*, *significant benefits* in query execution time might be observed. This is because the execution path will *skip through* the `L2-BlockGet` and `L2-FluxInsert` stages, which are the *most time-consuming* steps, as stated before.

7. CONCLUSION AND FUTURE WORK

In this project, *TorqueDB* has been proposed and described, which is 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 allow non-query applications to use the same master data without creating duplicates within a TSDB. The *native TSDB querying of InfluxDB with its Flux query language is used*, which 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 perform comparably to a central cloud VM.

As future work, the plan is 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. Larger scale experiments on 100's of devices will *validate the scalability further.*

8. REFERENCES

1. Abadi, D., Ailamaki, A., Andersen, D., Bailis, P., Balazinska, M., Bernstein, P., ... & Dong, L. (2020). The Seattle Report on Database Research. ACM SIGMOD Record, 48(4), 44-53.

2. Georgiou, Z., Symeonides, M., Trihinas, D., Pallis, G., & Dikaiakos, M. D. (2018, December). Streamsight: A query-driven framework for streaming analytics in edge computing. In 2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC) (pp. 143-152). IEEE.

3. Ghosh, R., & Simmhan, Y. (2018). Distributed scheduling of event analytics across edge and cloud. ACM Transactions on Cyber-Physical Systems, 2(4), 1-28.

4. Grunert, H., & Heuer, A. (2017). Rewriting Complex Queries from Cloud to Fog under Capability Constraints to Protect the Users' Privacy. Open Journal of Internet Of Things (OJIOT), 3(1), 31-45.

5. Gupta, H., Xu, Z., & Ramachandran, U. (2018). Datafog: Towards a holistic data management platform for the iot age at the network edge. In {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 18).

6. Malensek, M., Pallickara, S. L., & Pallickara, S. (2017). Hermes: Federating fog and cloud domains to support query evaluations in continuous sensing environments. IEEE Cloud Computing, 4(2), 54-62.

7. Martinviita, M. (2018). Time series database in Industrial IoT and its testing tool.

8. Monga, S. K., Ramachandra, S. K., & Simmhan, Y. (2019, July). ElfStore: A resilient data storage service for federated edge and fog resources. In 2019 IEEE International Conference on Web Services (ICWS) (pp. 336-345). IEEE.

9. Nagato, T., Tsutano, T., Kamada, T., Takaki, Y., & Ohta, C. (2019). Distributed key-value storage for edge computing and its explicit data distribution method. IEICE Transactions on Communications.

10. Patel, P., Ali, M. I., & Sheth, A. (2017). On using the intelligent edge for IoT analytics. IEEE Intelligent Systems, 32(5), 64-69.

11. Schultz-Møller, N. P., Migliavacca, M., & Pietzuch, P. (2009, July). Distributed complex event processing with query rewriting. In Proceedings of the Third ACM International Conference on Distributed Event-Based Systems (pp. 1-12).

12. Varshney, P., & Simmhan, Y. (2020). Characterizing application scheduling on edge, fog, and cloud computing resources. Software: Practice and Experience, 50(5), 558-595.

13. Zhang, W., Guo, W., Liu, X., Liu, Y., Zhou, J., Li, B., ... & Yang, S. (2018). LSTM-based analysis of industrial IoT equipment. IEEE Access, 6, 23551-23560.

14. Zhou, Z., Zhao, D., Xu, X., Du, C., & Sun, H. (2015). Periodic query optimization leveraging popularity-based caching in wireless sensor networks for industrial iot applications. Mobile Networks and Applications, 20(2), 124-136.

15. Mourtzis, D., Vlachou, E., & Milas, N. J. P. C. (2016). Industrial Big Data as a result of IoT adoption in manufacturing. Procedia cirp, 55, 290-295.

16. Lade, P., Ghosh, R., & Srinivasan, S. (2017). Manufacturing analytics and industrial internet of things. IEEE Intelligent Systems, 32(3), 74-79.

17. Ravindran, A., & George, A. (2018). An edge datastore architecture for latency-critical distributed machine vision applications. In {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 18).

18. Bharde, M., Bhattacharya, S., & Shree, D. D. (2018). Store-Edge RippleStream: Versatile Infrastructure for IoT Data Transfer. In {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 18).

19. Psaras, I., Ascigil, O., Rene, S., Pavlou, G., Afanasyev, A., & Zhang, L. (2018). Mobile data repositories at the edge. In {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 18).

20. Aral, A., & Ovatman, T. (2018). A decentralized replica placement algorithm for edge computing. IEEE transactions on network and service management, 15(2), 516-529.

21. Abu-Elkheir, M., Hayajneh, M., & Ali, N. A. (2013). Data management for the internet of things: Design primitives and solution. Sensors, 13(11), 15582-15612.

22. Joshi, J., Chodisetty, L. S., & Raveendran, V. (2019, May). A Quality Attribute-based Evaluation of Time-series Databases for Edge-centric Architectures. In Proceedings of the International Conference on Omni-Layer Intelligent Systems (pp. 98-103).

23. Liu, R., & Yuan, J. (2019). Benchmark Time Series Database with IoTDB-Benchmark for IoT Scenarios. arXiv preprint arXiv:1901.08304.

9. APPENDIX: CLUSTER, CODE SNIPPETS

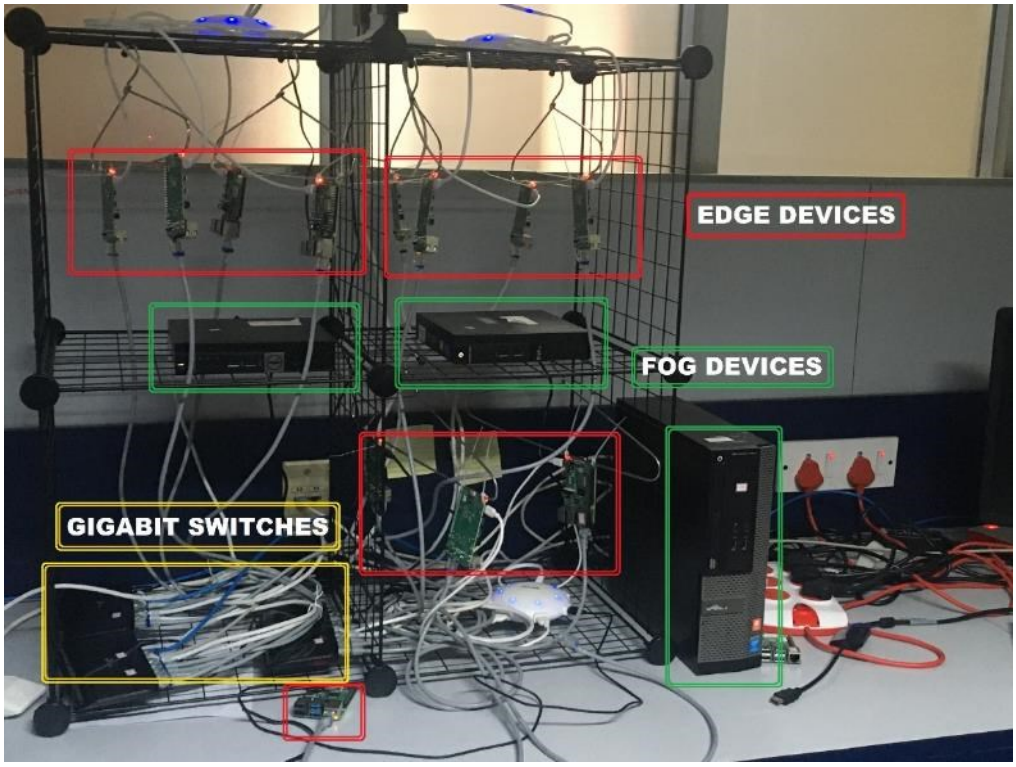


Figure Ap-1: Physical hardware setup used for the experiments

Fig. Ap-1 shows 12 Raspberry Pi devices, 3 fogs and 3 8-Port Gigabit switches

```
public class AggregateWindow implements IAggregate,Cloneable,Serializable{
        public String every;
        public ITransform fn;
        public String column = "_value" ;
        public String timeColumn = "_stop";
        public String timeDst = "_time";
        public boolean createEmpty = true;

        public AggregateWindow() {}

        public Object clone() {
                try {
                return (AggregateWindow) super.clone();
            } catch (CloneNotSupportedException e) {
                return new AggregateWindow(every,fn,column,timeColumn,timeDst,createEmpty);
            }
        }

        public AggregateWindow(String every, ITransform fn, String column, String timeColumn, String timeDst, boolean createEmpty) {
                assert every.matches(".*[smhd].*") : "Invalid <every> for AggregateWindow";
                this.every = every;
                this.fn = fn;
                this.column = column;
                this.timeColumn = timeColumn;
                this.timeDst = timeDst;
                this.createEmpty = createEmpty;
        }
```

Figure Ap-2: Code snippet for implementing the Aggregate window operator

Similar to Fig. Ap-2, other operators like Keep, Max, Count, etc. were implemented.

```
public class QueryUtility implements Cloneable,Serializable {
        public Query q;
        public Join j = null;
        public String experiment = "default";

        @Override
        public QueryUtility clone() {

                QueryUtility temp = new QueryUtility();
                temp.addBucketName(q.bucket);
                for(IBuiltIn i: q.operations) {
                        switch(i.getClass().getName()) {
                                case "com.dreamlab.LatestElfFlux.QFunctions.Range":
                                                Range r = (Range)i;
                                                temp.q.operations.add((Range)r.clone());
                                                break;
                                case "com.dreamlab.LatestElfFlux.QFunctions.Filter":
                                        Filter f = (Filter)i;
                                        temp.q.operations.add((Filter)f.clone());
                                        break;
                                case "com.dreamlab.LatestElfFlux.QFunctions.Mean":
                                        Mean m = (Mean)i;
                                        temp.q.operations.add((Mean)m.clone());
                                        break;
                                case "com.dreamlab.LatestElfFlux.QFunctions.AggregateWindow":
                                        AggregateWindow aw = (AggregateWindow)i;
                                        temp.q.operations.add((AggregateWindow)aw.clone());
                                        break;
                                case "com.dreamlab.LatestElfFlux.QFunctions.Keep":
                                        Keep k = (Keep)i;
                                        temp.q.operations.add((Keep)k.clone());
                                        break;
```

Figure Ap-3: Code snippet for the Query Utility class

Query Utility class creates the query object using each operators interface in IBuiltIn

```
public FindBlockQueryResponse level4(QueryUtility q) throws TException, InterruptedException{
        String bucket = q.getBucket();
        String measurement = q.getMeasurement();
        LocalDateTime globalStartTS = q.getStartTime();
        LocalDateTime globalStopTS = q.getStopTime();


        LocalDateTime start =LocalDateTime.of(2015, 2, 1, 0, 0);
        LocalDateTime stop =LocalDateTime.of(2016, 5, 27, 0, 0);

        LocalDateTime prev = null;
        int start_found = -1;
        for(LocalDateTime i=start; i.isBefore(stop)||i.isEqual(stop); i=i.plusHours(DELTA)){
                if(i.isEqual(globalStartTS)) {
                        start_found = 1;
                        startTS = i;
                }
        }

        if(start_found == -1) {
                for(LocalDateTime i=start; i.isBefore(stop)||i.isEqual(stop); i=i.plusHours(DELTA)){
                        if(!i.isBefore(globalStartTS)) {
                                startTS = prev;
                                break;
                        }
                        prev = i;
                }
        }

        int stop_found = -1;
        for(LocalDateTime i=start; i.isBefore(stop)||i.isEqual(stop); i=i.plusHours(DELTA)){
```

Figure Ap-4: Code snippet for Query Decomposition into L1, L2, L3 and L4

The Query Decomposition class extracts information and performs query re-writing

```java
public MetadataResponse getMetadataByBlockid(long mbid, String fogip, int fogport, EdgeInfoData edge, List<String> keys) throws TE>
        TFramedTransport transport;

        TTransport t_transport = new TSocket(fogip, fogport);
        transport = new TFramedTransport(t_transport);
        transport.open();

        TProtocol protocol = new  TBinaryProtocol(transport);

        FogService.Client client = new FogService.Client(protocol);

        //System.out.println("WOrkinng");
        //int product = client.multiply(3,5);
        //MatchPreference matchPref = new MatchPreference();
    long start = System.currentTimeMillis();
        MetadataResponse result = client.getMetadataByBlockid(mbid,fogip,fogport,edge,keys);
        long end = System.currentTimeMillis();
        System.out.println("@@@@@@@@@@@@@@@@@@@@@@ Pure GET_META ElfStore Time: " + (end - start) + "ms");
        System.out.println("&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&Pure ElfStore Call");
        transport.close();
        return result;
}

public ReadReplica readFog(long microbatchId, boolean fetchMetadata, String fogIP, int port) throws TException{
        TFramedTransport transport;

        TTransport t_transport = new TSocket(fogIP, port);
        transport = new TFramedTransport(t_transport);
        transport.open();

        TProtocol protocol = new  TBinaryProtocol(transport);

        FogService.Client client = new FogService.Client(protocol);
```

Figure Ap-5: Code snippet for interfacing with ElfStore

L4 and L3 information is passed and metadata and block fetch is performed by ElfStore

```java
public void putToPerFogMbids(java.lang.String key, java.util.List<java.lang.Long> val) {
  if (this.perFogMbids == null) {
    this.perFogMbids = new java.util.HashMap<java.lang.String,java.util.List<java.lang.Long>>();
  }
  this.perFogMbids.put(key, val);
}

public java.util.Map<java.lang.String,java.util.List<java.lang.Long>> getPerFogMbids() {
  return this.perFogMbids;
}

public CostModelOutput setPerFogMbids(java.util.Map<java.lang.String,java.util.List<java.lang.Long>> perFogMbids) {
  this.perFogMbids = perFogMbids;
  return this;
}

public void unsetPerFogMbids() {
  this.perFogMbids = null;
}

/** Returns true if field perFogMbids is set (has been assigned a value) and false otherwise */
public boolean isSetPerFogMbids() {
  return this.perFogMbids != null;
}

public void setPerFogMbidsIsSet(boolean value) {
  if (!value) {
    this.perFogMbids = null;
  }
}
```

Figure Ap-6: Thrift Auto-generated code for RPC calls

TorqueDB and ElfStore services communicate with each other using Thrift calls

```
public class DataMigrationServer {
	public static DataMigrationHandler handler;

	public static DataMigration.Processor processor;

	public static void main(String[] args) {
		try {
			String ip = args[0];
			String port = args[1];
			List<String> fogs = Arrays.asList("10.24.24.214", "10.24.24.154", "10.24.24.153");

			InfluxdbConnections connections = new InfluxdbConnections();
			connections.writeConnections = getAllWriteConnections(fogs);
			connections.readConnections = getAllReadConnections(fogs);

			handler = new DataMigrationHandler(connections);
			processor = new DataMigration.Processor(handler);

			ExecutorService pool = Executors.newFixedThreadPool(1);
			Runnable temp = new HeartBeat(ip, handler, fogs);
			pool.execute(temp);
			pool.shutdown();

			Runnable simple = new Runnable() {
				public void run() {
					simple(processor, port);
				}
			};

			new Thread(simple).start();
		} catch (Exception x) {
			x.printStackTrace();
		}
```

Figure Ap-7: Code snippet for the Data Migration Server class

This server class waits for query requests and handles communication between fogs

```
public String getPlan(ByteBuffer queryUtility) throws InterruptedException, TException {
	String res = "";
	try {
		QueryDecomposition queryDecompose = new QueryDecomposition();
		long start = System.currentTimeMillis();
		byte[] byts = new byte[queryUtility.remaining()];
		queryUtility.get(byts);
		ObjectInputStream istream = null;
		istream = new ObjectInputStream(new ByteArrayInputStream(byts));
		Object obj = istream.readObject();
		QueryUtility query = ((QueryUtility) obj);
		long end = System.currentTimeMillis();
		System.out.println("@@@@@@@@@@@@@@@@@@@@@@@@ Deserialization Time " + (end - start) + "ms");

		experiment = query.experiment;
		queryDecompose.experiment = experiment;
		PropertyConfigurator.configure(path);
		logger.info("Experiment=>" + experiment + ", DeserializationTime=>start=>" + (start));
		logger.info("Experiment=>" + experiment + ", DeserializationTime=>stop=>" + (end));

		start = System.currentTimeMillis();
		CostModelOutput t_result = queryDecompose.decompose(query, cache);
		end = System.currentTimeMillis();
		logger.info("Experiment=>" + experiment + ", TotalDecompositionTime=>start=>" + (start));
		logger.info("Experiment=>" + experiment + ", TotalDecompositionTime=>stop=>" + (end));

		shortCircuit = queryDecompose.shortCirtuit;
		aggwin_shortCircuit = queryDecompose.aggwin_shortCirtuit;
		System.out.println("#######CostModelOutput perFogMbids= >" + t_result.perFogMbids);

		// TTransport transport;
		System.out.println("#######Connecting to L1 FogIp:" + t_result.Level1Fog + " port:8050");
```

Figure Ap-8: Code snippet for the Data Migration Handler class

Handler interfaces with ElfStore and obtains the query execution plan across 4 levels

```
public List<String> sendingResultToL1(List<FluxRecord> result, String query_received) {

	if (result.isEmpty()) {
		System.out.println("Result is empty");
		return Arrays.asList(new String("Empty"));
	}

	long start = System.currentTimeMillis();

	List<String> resultList = new ArrayList<String>();
	int resultSetSize = result.size();

	System.out.println("#####Number of rows in the result: ####" + resultSetSize);
	String originalQuery = query_received;

	if ((!originalQuery.contains("|>mean()")) && (!originalQuery.contains("|>sum()"))
			&& (!originalQuery.contains("|>aggregateWindow")) && (!originalQuery.contains("|>keep("))
			&& (!originalQuery.contains("|>count()")) && (!originalQuery.contains("join("))
					&& (!originalQuery.contains("|>min()")) && (!originalQuery.contains("|>max()"))) {
		for (FluxRecord record : result) {
			StringBuilder singleRecordResult = new StringBuilder("");
			String value = "";
			try {
				value = record.getValue().toString();
			} catch (NullPointerException e) {
				continue;
			}
			String time = record.getTime().toString();
			singleRecordResult.append(value).append(" ").append(time);
			resultList.add(singleRecordResult.toString());
		}
	}
```

Figure Ap-9: Code snippet for the Data Migration Helper class

Helper class uses InfluxDB APIs to write, query and manage data stored in the TSDB

```
  private static String perform(DataMigration.Client client,QueryUtility query) throws TException
  {
		try {
		//System.out.println(client.writeToInfluxWithPath("climate", 164, "localhost", 8087));
			ObjectOutputStream ostream;
			ByteArrayOutputStream bstream = new ByteArrayOutputStream();
			ostream = new ObjectOutputStream(bstream);
		    ostream.writeObject(query);
		    ByteBuffer buffer = ByteBuffer.allocate(bstream.size());
		    buffer.put(bstream.toByteArray());
		    buffer.flip();
			return client.getPlan(buffer);
		//System.out.println(client.queryOnInflux("climate",query, "localhost", 808));
		}
		catch(Exception e) {
			e.printStackTrace();
			return "Client Side Error!!!!!!!!!";
		}
}

  public static Map<String,InfluxDB> getAllWriteConnections(List<String> fogs){
		Map<String,InfluxDB> connectionMap = new HashMap<>();
		for(String fog: fogs) {
			connectionMap.put(fog+":8086", InfluxDBFactory.connect("http://"+fog+":8086"));
			System.out.println(connectionMap.get(fog+":8086"));
		}
		connectionMap.put("localhost:8086", InfluxDBFactory.connect("http://localhost:8086"));
		return connectionMap;
	}
```

Figure Ap-10: Code snippet for the Data Migration Client class

Client class creates queries of QueryUtility type and sends them to the coordinator fog

```
public class Test3 {
	static Map<String, InfluxDB> WriteConnectionMap;
	String experiment;

	public Test3(Map<String,InfluxDB> connectionMap, String experiment) {

		this.WriteConnectionMap = connectionMap;
		this.experiment = experiment;
	}

	public String writeToInfluxAtLevelN(String dbname, String data_sent, String fogIp, int influxPort) {
		String status = "Unsuccessful";
		StringBuilder url = new StringBuilder(fogIp);
		url.append(":").append(influxPort);
		if (data_sent.equals("Empty")) {
			return "No data to write";
		}

		long start = System.currentTimeMillis();
		InfluxDB influxDB = WriteConnectionMap.get(url.toString());
		long stop = System.currentTimeMillis();
		start = System.currentTimeMillis();
		influxDB.setDatabase(dbname);
		//influxDB.write(data_sent);
		String rp = TestUtils1.defaultRetentionPolicy(influxDB.version().toString());
		influxDB.write(dbname, rp, InfluxDB.ConsistencyLevel.ONE, TimeUnit.SECONDS, data_sent);
		stop = System.currentTimeMillis();
		//influxDB.close();
		return "Write successful";
	}
```

Figure Ap-11: Code snippet for the InfluxDB ingestion benchmark

The benchmark utilized data stored in varying sized chunks and varied the number of parallel inserts through a thread-pool.