Lindorm-UWC: An Ultra-Wide-Column Database for Internet of Vehicles

Qianyu Ouyang[†‡] Peng Yu[‡] Qilu Zhong[‡] Zhicheng Ji[†‡] Dan Pei[†]
Chunhui Shen[‡§] Qiang Xiao[‡] Xiang Wang[‡] Wei Meng[‡] Wei Zhang[‡]
Wenlong Yang[‡] Jianhui Lei[‡] Yong Lin[‡] Cen Zheng[‡] Feifei Li[‡]
Yadong Chen[‡] Qingyi Meng[‡] Sheng Wang[‡] Jingren Zhou[‡]
Tsinghua University[†] Zhejiang University[§] Alibaba Cloud[‡]
{ouyangqianyu.oyqy,tianwu.sch,zhengyan.ywl,tianyu.yp,xiaoqiang.xiao,leijianhui.ljh,yadong.cyd,qilu.zql,wangxiang}
{linyong.ly,qingyi.mqy,jizhicheng.jzc,mw371030,mingyan.zc,sh.wang,zwei,lifeifei,jingren.zhou}@alibaba-inc.com

ABSTRACT

In the Internet of Vehicle (IoV) systems, intelligent vehicles generate huge amounts of data that supports diverse services and applications. In practice, database systems are deployed in the cloud to manage data uploaded from the vehicle side and provide real-time query capacities. However, existing database systems are ill-suited because IoV data contains a large number of metrics and is written at an extremely high throughput. To better understand IoV data and corresponding challenges to underlying database systems, we conduct the first extensive empirical study of real-world IoV workloads. According to our findings from the study, we design Lindorm-UWC as a superior database for IoV systems. It implements a distributed architecture and a cold/hot data separation mechanism to accommodate massive amounts of IoV data. In each data partition, it deploys an ultra-wide-column storage engine to efficiently handle the query and ingestion of multi-metric data. We evaluate Lindorm-UWC under different data scales and various types of query. Our experimental results show that it can always achieve higher write throughput (over 79% increase) and competitive query performance compared to various alternative solutions. Lindorm-UWC has been serving IoV enterprise customers on Alibaba Cloud since 2019, managing tens of petabytes of IoV data.

PVLDB Reference Format:
Qianyu Ouyang, Chunhui Shen, Wenlong Yang, Peng Yu, Qiang Xiao, Jianhui Lei, Yadong Chen, Qilu Zhong, Xiang Wang, Yong Lin, Qingyi Meng, Zhicheng Ji, Wei Meng, Cen Zheng, Sheng Wang, Dan Pei, Wei Zhang, Feifei Li, Jingren Zhou. Lindorm-UWC: An Ultra-Wide-Column Database for Internet of Vehicles. PVLDB, 17(12): 4117 - 4129, 2024.
doi:10.14778/3685800.3685831

1 INTRODUCTION

With the development of information and communication technology (ICT) as well as in-vehicle sensing technology, the automotive industry is undergoing a significant transformation—intelligence and digitalization have become the new standards for modern automobiles [32, 37], where the number of intelligent vehicles is rapidly increasing [1]. Vehicles now are equipped with numerous sensors that continuously collect data on vehicle operations, driving behaviors, and road conditions [20, 21, 23]. Through the network, vehicles can timely upload data to the gateways in the cloud, providing data-driven services and applications and connecting the vehicles to the external world.

The above trend facilitates the emerging system of the so-called Internet of Vehicles (IoV) [47]. Within the IoV system, the data is collected and utilized by multiple stakeholders, including vehicle manufacturers, telematics service providers (TSP), autonomous driving vendors, government regulation platforms, *etc.* They extract valuable information and insights from vast amounts of data to provide various services. For instance, when a vehicle encounters a malfunction, a TSP can remotely diagnose the issue by examining the vehicle's operational data [21]. As the scale of an IoV system (*i.e.*, the number of vehicles, sensors, services) expands, the efficiency of managing and utilizing IoV data is of great importance.

In practice, the IoV data is naturally time-series data in a form that is consistent with the Internet of Things (IoT) data and DevOps metric data [26]. The data is generated on a per-vehicle basis. A single vehicle is equipped with a multitude of sensors, constantly generating a vast number of metrics that form multi-dimensional time series. There are many time-series database systems in the market that are good at handling IoT and DevOps scenarios, such as IoTDB [44], InfluxDB [13] and Prometheus [15]. These database systems are employed to support high-rate data ingestion and low-latency real-time queries. However, due to the uniqueness of IoV scenario, these time-series databases are ill-suited for handling IoV data. One major difference arises from the scale of the metrics. In DevOps systems, a single server or compute node typically generates no more than dozens or hundreds of monitoring metrics [16, 19] (*e.g.*, CPU usage, disk IO time, Free Memory); in IoT systems, a single device contains no more than several hundreds of sensors [25, 40, 44]. In contrast, in an IoV system, a vehicle can easily generate thousands of metrics [31, 44], since it consists of many complex subcomponents, *e.g.*, Advanced Driver Assistance Systems (ADAS), Battery Management System (BMS), Domain Control Unit (DCU). Note that such a large number of metrics are collected and retrieved on a per-vehicle basis, which poses significant challenges for existing time-series databases that manage data on a per-metric basis. This inspires us to explore a comprehensive understanding

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685831

of IoV workload patterns, which can guide us to design well-suited database systems for IoV scenarios.

To the best of our knowledge, no prior work extensively studies IoV workloads. In this paper, we provide the first empirical study of IoV workloads to investigate their unique requirements to underlying database systems. We explore both data read and write patterns from three major automakers and TSPs in China (detailed in §2), and summarize three key challenges for IoV data management:

- **C1. Extremely high data ingestion rate.** IoV systems are write-intensive, where the data traffic can easily reach over 1GB per second and 80TB per day. This demands the underlying database to sustain extremely high write throughput. In addition, the huge volumes of data resulting from the high ingestion rate further put enormous pressure on storage costs.
- **C2. A huge number of metrics per vehicle with update-style writing.** A single vehicle can generate over 2500 distinct time-series metrics simultaneously, which is nearly an order of magnitude more than what a traditional time-series database can handle. Moreover, different components in a vehicle upload their own metrics data independently, which means that a write request to the database will not involve all metrics of the vehicle. Hence, the underlying database has to handle many small writes with an incomplete metric format.
- **C3. Diverse patterns for querying a small or large number of metrics.** To support numerous downstream services, the database has to be capable of handling different query types with high concurrency and low latency. Different queries may involve different metrics from the massive dataset. Some of them retrieve a large number of metrics (*e.g.*, applications that fetch data for comprehensive analysis), while others need only a small subset (*e.g.*, engineers that query related metrics for remote failure diagnosis).

We note that existing time-series databases and any other databases are unable to fully address above three IoV workload challenges. Wide-column databases, *e.g.*, HBase [8], and column-oriented time-series databases, *e.g.*, InfluxDB [13], have to consume unaffordable amounts of computation and space resources to index massive metrics, resulting in unacceptably poor write throughputs (challenge C1&C2). When querying massive metrics, time-series databases need to perform independent retrievals for each metric, introducing a vast amount of I/Os; HBase, which treats each data point as a key-value entry, has to spend a lot of CPU time on performing key comparisons (challenge C3). To address the issues of writing and querying massive metrics, document-oriented databases like MongoDB [14], which models all metric values from a vehicle generated at the same timestamp as a flexible schema-free document, seem to be a suitable solution. However, due to their document-oriented storage layout, for those queries involving a small subset of metrics, they have to fetch all metrics (*i.e.*, the entire document), leading to prohibitive read amplification (challenge C3). Moreover, they also struggle when metric values in one document arrive in multiple rounds (challenge C2), as each round is treated as an extra update that has to read the target document out first.

After spotting the gap between IoV workload challenges and existing database solutions, we propose Lindorm-UWC (ultra-wide-column) for data management in IoV systems. To accommodate massive amounts of IoV data, Lindorm-UWC has a distributed architecture that partitions data by vehicles and time and supports automated load balancing. In order to efficiently handle multi-metric IoV data, we design an *ultra-wide-column* storage engine based on the Log Structured Merge tree (LSM-tree) [38], where each partition in Lindorm-UWC employs an independent one. The storage engine implements two mechanisms to accelerate the ingesting of multi-metric data: first, multiple metric values contained in one write request are consolidated into a single column to eliminate indexing and grouping of different metrics on the write path; second, each write request is processed in an append-only way instead of in-place updating on existing data of that vehicle. To efficiently handle various query patterns, Lindorm-UWC organizes on-disk data in both row-oriented and column-oriented storage formats, allowing it to choose and read from a suitable file format that can reduce read amplification. To lower storage cost, we employ a tiered cold-hot data storage layout, offloading less-frequently accessed cold data to poor-performing but cheap storage media. Lindorm-UWC has been serving IoV enterprise customers on Alibaba Cloud since 2019. It manages tens of petabytes of IoV data and handles more than 10 million requests per second.

Our major contributions are summarized as follows:

- We conduct the first empirical study to highlight the workload characteristics in real-world IoV systems. By analyzing data ingestion and query patterns, we obtain a series of valuable findings that can drive the database design for IoV workloads.
- We propose Lindorm-UWC, a database designed for IoV systems. To manage the vast amount of IoV data, Lindorm-UWC employs a tailored distributed architecture and a cold-hot data separation mechanism. To efficiently handle the writing and querying of multi-metric data, we innovatively design an ultra-wide-column storage engine, which supports high write throughput and provides efficient queries of diverse patterns.
- To evaluate the effectiveness of Lindorm-UWC's design, we develop a benchmark suite based on the characteristics of real-world IoV workloads, and then conduct comprehensive experiments with it. We compare Lindorm-UWC with three typical databases as baselines: MongoDB, HBase, and InfluxDB. The experimental results indicate that Lindorm-UWC significantly outperforms our baselines. In a variety of workloads, Lindorm-UWC's write performance is from 79% to an order of magnitude higher. For query efficiency, Lindorm-UWC can always sustain high concurrency for any queries retrieving either a small or a large number of metrics.

2 EMPIRICAL STUDY

2.1 IoV System Background

We first introduce how vehicles generate data in the IoV system, and how the data is used by different application services. The IoV system is typically a three-tier structure [20, 32], consisting of the physical layer, the connectivity layer, and the cloud layer.

The physical layer refers to vehicles equipped with network devices and a large number of sensors. The running vehicles continuously collect and process information about the road environment, vehicle components, and vehicle running status through the sensors, and then upload them to the connectivity layer through the

network devices at regular intervals. The connectivity layer contains various network elements (gateways, base stations, switches, *etc.*), different levels of networks, and the links between them. With the help of wireless access technologies, such as 5G, vehicles are able to establish secure, fast, and highly-available network connections with central systems in the cloud. The cloud layer is the central area where various business applications and underlying data management systems are hosted. Specifically, after the data are uploaded to the cloud layer through the connectivity layer, it first enters queueing systems, such as Kafka. Different applications will pull the required data from the queue into their own data management systems. Database [8, 13, 14, 42] plays a main role in these data management systems, responsible for storing massive amounts of IoV data and supporting low-latency queries.

Here we mainly focus on the database part in the cloud layer. In the following, we will study the workloads of data write and query from several real-world IoV systems. We aim to figure out the characteristics and challenges of storing IoV data in the database.

2.2 Real-world IoV Workload Analysis

IoV data format. A running vehicle in the IoV system constantly uploads data collected by sensors to the cloud layer. These data are naturally in the form of multi-dimensional time-series data, where a dimension (denoted later by **column** or **metric**) refers to a metric collected on a certain sensor. The data sent to the database contains the following three parts: a vehicle ID, a timestamp, and multiple metrics and corresponding values. Table 1 shows three sampled rows of multi-metric IoV data. It is worth noting that the metric set to be collected is dynamically determined on the vehicle side, which is difficult to synchronize on the database side. Thus it is impractical to preset the metric-related settings in the database.

Table 1: IoV data samples.

Vehicle ID	Timestamp	Metrics
v001	1699255800	BMS_volt=11.5, BMS_batt=60, BMS_svdb=10, BMS_ncell=130, ...
v002	1699255800	STATE_bcm=3, STATE_run=2, STATE_abs=0, STATE_epb=1, ...
v003	1699255800	DCU_temp=65, DCU_cool=10, DCU_rots=4000, DCU_idc=1, ...

Recall that each write operation issued from a vehicle only contains a subset of metrics. This is because that each sub-component from the vehicle has its own sensors to collect and upload data, where data from different sub-components may have unaligned timestamps. When data from different requests are written to the database, their timestamps will be aligned to a certain precision (*e.g.*, a second or a minute) as required by the specific application. From the database's logical perspective, when the aligned timestamp already exists in previous write operations, the current request does not insert a new row, but instead "update" an existing row by adding new metrics. We refer to this as "update-style writing", which will be discussed in detail in the following analysis.

Data source. Our study is based on month-long sets of data writing records and week-long sets of data query logs provided by three IoV companies, which are referred to as A, B, and C in this paper. Specifically, A and B are vehicle manufacturers who not only produce the vehicles but also manage vehicle-generated data and offer related online services. C is a third-party telematics service provider (TSP), providing data management services for some old-model, small-scale, or technologically constrained vehicle manufacturers. From public information, A manages about 40,000 vehicles, while B has more than 200,000 vehicles. Since C provides services for other manufacturers, we are unable to obtain the precise number of vehicles under their management.

Table 2: Metric scales from different companies.

Company	#Total metrics	#Metrics per write
A	2999	223
B	2083	2083
C	260	43

Write workload. Table 2 lists the metric scales of data from the three companies. It is clear that a vehicle is equipped with numerous sensors that collect and upload a wide range of metrics. As shown in the second column of Table 2, A and B have more than 2,000 metrics, while the vehicles managed by C are older models and therefore the number of metrics is significantly lower, totaling 260. The third column of the table shows the average number of metrics included in each write operation. In A, a write operation contains an average of 223 metrics, which is 7.4% of the total metrics (*i.e.*, 2999) meaning that one row is written into the database via 14 write operations on average. For C, a vehicle divides a row into six parts for uploading on average. It is noteworthy that data in B does not exhibit update-style writing. Therefore, the number of metrics written each time is equal to the total number of metrics. This is because writes belonging to the same row are merged before entering the queue for ease of subsequent processing. However, this preprocessing requires additional computational resources and places external development demands on its system, which will be discussed in §5.

The remarkably high data write throughput in IoV systems is attributed to two major factors: 1) a large number of vehicles constantly upload data; 2) each single vehicle generates numerous metrics values. Figure 1 shows the write operation and data throughput for the three companies. Over the month, the system in B averages 21K/s to 27K/s write operation throughput per day (Figure 1b), where a single write operation contains all metrics from a vehicle. Compared to B, A is different in that it performs update-style writing, so that its write operation throughput is an order of magnitude higher, averaging over 110K/s for a month (Figure 1a). The data in C is uploaded in update-style writing way as well. Its average write operation throughput is relatively lower and below 20K/s on most days (Figure 1c), probably because of fewer vehicles. From Figure 1, we can observe that the data throughput in IoV systems is immensely large. A has very high data throughput at 560MB/s to 950MB/s, averaging 721MB/s over a month (Figure 1a). In B, the data throughput exceeds 800MB/s almost every day with an average of 907MB/s (Figure 1b), and can sometimes reach 1GB/s. Data in C has fewer metrics, so its data throughput is not particularly high

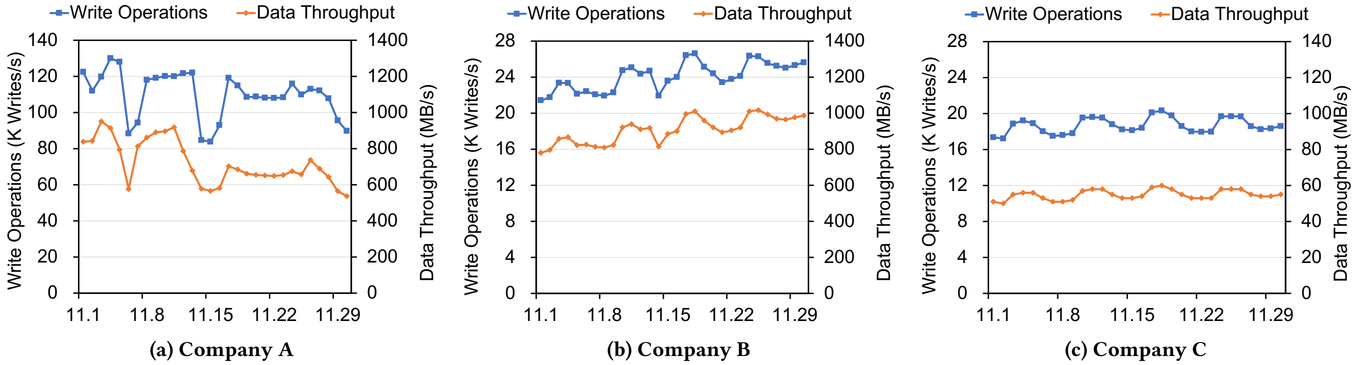


Figure 1: The average write throughput for each day of a month. The blue line indicates the number of write operations per second (K/s). The orange line indicates the data throughput (MB/s).

compared to A and B, around 55MB/s (Figure 1c). It should be noted that at certain peak times, such as in the morning, a large number of vehicles are in intensive use. Consequently, the data throughput at the peak periods is even higher than the values shown in Figure 1. For example, the data throughput of B reaches 1.68 GB/s at certain times on Nov. 18.

Based on the analysis above, we can summarize the characteristics of IoV write workload as follows: 1) a large number of metrics; 2) extremely high write throughput; 3) update-style writing.

Query workload. To understand how IoV data is queried, we use query logs from the three companies over one week and conduct a statistical analysis of the filter conditions used in these queries. The general query pattern can be summarized as follows: given several filter conditions for vehicles, a time range, and a set of metrics, the database returns all data of the specified metrics from the qualified vehicles for the given period. We observe that the majority of queries are single-vehicle queries, where a specific vehicle ID is given. Only a small proportion of queries use vehicle attributes as filters to retrieve data from multiple vehicles. An example of the single-vehicle query using data in Table 1 is as follows:

```
Select BMS_volt, DCU_temp, VehicleID, timestamp
from IOV_Table where VehicleID = v001
and timestamp between 1699254000 and 1699257600
```

`BMS_volt` and `DCU_temp` are the two metrics being queried.

Table 3: Overview of queries from different companies.

Company	#Query	Ratio of single-vehicle queries
A	1062	100%
B	496377	100%
C	470888	99.79%

First, Table 3 shows the overview of queries from the three companies. During one week, B and C both have over 470,000 queries, while A only performs about 1,000 queries. This is because the queries in A are from the handling of user inquiry tickets. Only when a vehicle experiences a fault, the vehicle owner submits a ticket. After receiving the ticket, experts and engineers in A examine the data from the vehicle during the fault period to perform a diagnosis. In the case of B and C, a large number of queries come from scheduled tasks that export IoV data to various downstream data-driven applications, such as driving behavior analysis. As shown in the third column in Table 3, almost all queries are single-vehicle queries, with only about 0.2% of queries from C involving multiple-vehicle data. This is because applications that provide online services typically collect data on a per-vehicle basis to render customized, precise information [23].

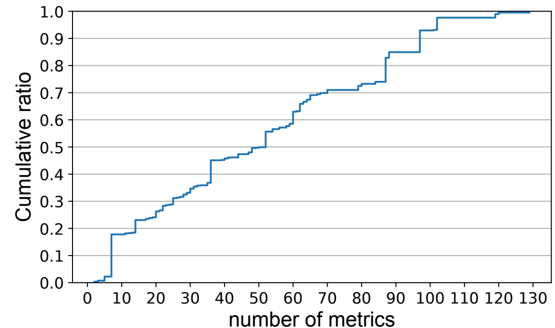
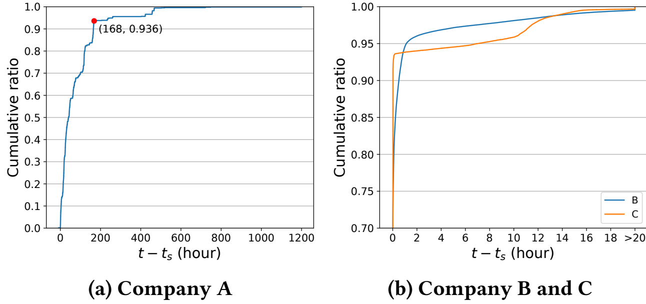


Figure 2: Cumulative distribution of the number of target metrics in A's queries.

Second, we investigate the number of metrics involved in these queries. Queries from B and C involve all metrics, *i.e.*, they query all metrics from a vehicle over a period. This type of query allows downstream applications to perform a variety of flexible and complex computations on the queried data. Different from B and C, each query in A is only interested in a few metrics, no more than 130, as shown in Figure 2. Compared to the total number of metrics in A, namely 2999, the percentage of metrics required per query is less than 4.4%. However, the diversity of components and applications in IoV systems can generate a large number of query metric combinations, even if the number of metrics per query is relatively small. Recall that the queries in A are used to process user tickets. For different types of tickets, the engineers determine which metrics are needed based on their experience. Therefore, instead of retrieving the entire row of data, they query the values of certain metrics. For example, when an electric vehicle (EV) has experienced a power

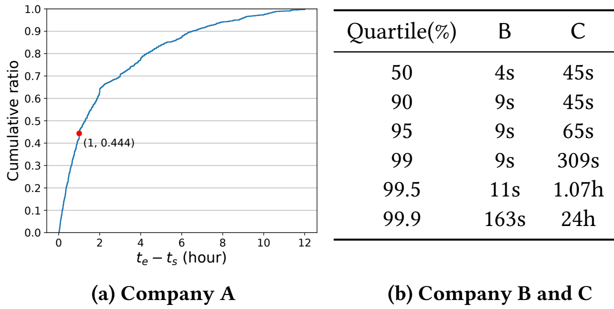
supply problem, the on-call engineers usually look into some of the metrics in the Drive System, Battery Management System (BMS), and Thermal Management System (TMS).



(a) Company A (b) Company B and C

Figure 3: Cumulative distribution of $t - t_s$ in query from different companies

Third, for a query at time t, which retrieves data within the time range $[t_s, t_e]$, we investigate its temporal characteristics in terms of the following two measurements: 1) the interval between the data start time and the query time, $t - t_s$; and 2) the length of the interested time range, $t_e - t_s$. As shown in Figure 3, the majority of queries focused on recently uploaded data. Specifically, 93.6% of queries in A (Figure 3a) concern data uploaded within 1 week, *i.e.*, 168 hours. It is more evident in B and C, where more than 94% of queries (Figure 3b) retrieve data within 1 hour. It indicates that newly generated data is of much higher value in IoV system and obsolete data is less frequently queried. For the length of time range in a query, Figure 4a shows that in A, 44.4% of queries retrieve data for no more than 1 hour and that all queries demand time ranges of less than 12 hours. Table 4b presents the quantiles of $t_e - t_s$ of queries for B and C. Almost all (over 99.5%) of the queries in B have a time range of a few seconds, and the rest retrieve data for minutes. Queries in C require no more than 24 hours, 95% of which query one-minute-long data.



(a) Company A

Quartile(%)	B	C
50	4s	45s
90	9s	45s
95	9s	65s
99	9s	309s
99.5	11s	1.07h
99.9	163s	24h

(b) Company B and C

Figure 4: Cumulative distribution of $t_e - t_s$ in query from different companies.

In conclusion, query workloads in the IoV system are dominated by queries for short time ranges of newly generated data. Depending on various purposes, some queries demand entire data rows, while others touch only a small number of metrics. Therefore, the underlying database needs to be able to: 1) retrieve recent data quickly; and 2) handle queries requiring either a small or a large number of metrics efficiently.

Table 4: Write throughput (the number of write operations per second) of different databases.

#Metrics	MongoDB	HBase	InfluxDB
100	85400	199468	156500
200	75700	111102	69100
500	62550	73205	31500
1000	21200	20295	8740
2000 (similar to B)	7752	9651	4230
3000 (similar to A)	4595	6416	3150

2.3 Motivation for IoV Database Designs

In summary, data ingestion and query in the IoV system mainly bring three challenges to the database as follows: 1) extremely high data ingestion rate; 2) a huge number of metrics per vehicle with update-style writing; 3) diverse patterns for querying a small or large number of metrics.

To motivate the design of a tailored database for IoV system, we conduct two experiments on three widely used databases to evaluate their performance on IoV write and query workloads. As IoV data contains massive metrics that are unknown to the database before they are written, we choose MongoDB [14] and HBase [8] as baselines, which can handle schema-free multi-metric data and are widely used for IoV data management in practice. Besides, InfluxDB [13] is also covered as a popular time-series database [12], which is good at serving queries with a few metrics. To have a fair comparison, we deploy the three databases with the same configuration of 3 nodes, each of which has 8 cores and 64GB RAM.

IoV write workload. Table 4 shows the write throughput of these databases when writing data with different scales. We generate different numbers of metrics to simulate the metric scale from hundreds to thousands in the IoV system. Besides, we load the data using update-style writing: splitting metrics values of the same vehicle with the same timestamp into ten write operations.

As shown in the table, when the number of metrics reaches the scale of B and A (*i.e.*, 2000 and 3000, respectively), these databases struggle to achieve satisfactory performance. The best-performing HBase achieves a throughput of 9.6K when writing data with 2000 metrics, which is 960 rows per second. This is significantly lower than the throughput of B, which is 24K on average (see in Figure 1b). Due to update-style writing, when MongoDB writes data, it must first retrieve the target document and then update with new metrics, leading to poor performance. InfluxDB organizes compressed data chunks in memory for each metric, while HBase treats each metric in every row as an individual entry. These impose prohibitive overheads for massive metrics. As a result, their performance degrades significantly as the number of metrics increases.

IoV query workload. Here we investigate the query performance. We first load the data of 2000 metrics into these databases, with each vehicle containing 3 hours of data. During the testing phase, we evaluate the query throughput (*i.e.*, executed queries per second) of these databases under single-vehicle query conditions.

Table 5: Query throughput (QPS) of different databases.

#Metrics selected	MongoDB	HBase	InfluxDB
2000	104	9.8	10.3
100	232	21.5	109
20	249	25.7	553
1	315	39.6	7030

We run queries that retrieve different numbers of metrics, namely 1, 20, 200, and 2000. Each query retrieves 10 minutes of data for a single vehicle, where the metrics are randomly selected. Table 5 shows the results. As a row-oriented database, MongoDB reads an entire row each time. This allows MongoDB to efficiently handle queries that involve a large number of metrics, but incurs significant read amplification when querying only a few metrics. As can be seen, MongoDB performs well when querying 2000 metrics, but even when the number of metrics is reduced to 1, the throughput gain is still marginal. In contrast, InfluxDB is a column-oriented database that can read contiguous data from a given metric with few I/Os. However, as the number of required metrics increases, the overhead will multiply. As a result, InfluxDB has higher query throughput when querying 1 or 20 metrics, but performance drops dramatically as the number of metrics increases. For HBase, each metric value is treated as a key-value item, where both metric name and timestamp are part of the key. This means that querying with specific metrics and time range requires a large number of key comparisons, leading to unacceptable performance.

Summary. The following findings can be drawn from the above experiments and challenges. First, existing typical database systems are unable to meet the high data ingestion throughput for multi-metric data. Second, queries retrieving either a small number or a large a large number of metrics cannot be both efficiently handled by a single system. To fill the gap, we design Lindorm-UWC (§3), an ultra-wide-column distributed database tailored for the IoV system. It provides high-throughput writing and querying for massive multi-metric IoV data. To be specific, we propose an ultra-wide-column storage engine that combines the advantages of both row-oriented and column-oriented storage to handle different requests (§3.3). We also employ a cold-hot separation mechanism to reduce the cost of managing massive amounts of data (§3.4).

3 SYSTEM DESIGN

In this section, we first give an overview of Lindorm-UWC. After that, we discuss its major components in detail.

3.1 System Overview

Lindorm-UWC is a distributed database serving the IoV system. Similar to our previous work Lindorm-TSDB [42], Lindorm-UWC is also an engine for Lindorm [6]. It shares some common components, *e.g.*, the underlying file system, with other engines. As illustrated in Figure 5, Lindorm-UWC consists of three major components: LDServer, LDMaster, and a shared distributed file system. The LDServer is the main computation node that manages data and processes read and write requests. As a distributed database, Lindorm-UWC can handle data of different scales by horizontally scaling out LDServers. The LDMaster is the management node that

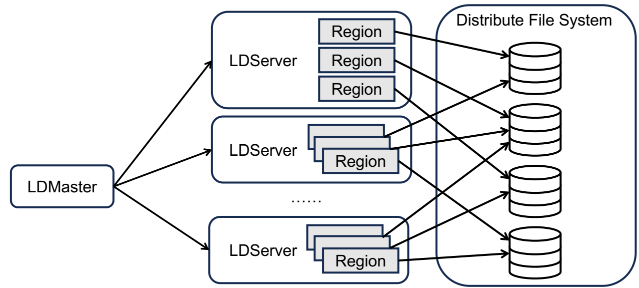


Figure 5: Lindorm-UWC system overview. Each LDServer manages data from multiple regions. LDMaster is responsible for load balancing and failover recovery of LDServers.

is responsible for load balancing and failover recovery in the cluster. The LDMaster itself is deployed in an active-standby configuration, relying on Zookeeper [11] to achieve high availability. The data in Lindorm-UWC is physically stored in the shared file system, which can be accessed by all LDServers. This architecture avoids the migration of large volumes of data between LDServers, while also ensuring system resilience and high availability.

Table 6: Data model of Lindorm-UWC.

Primary keys		Non-primary keys				
VID	time	metric1	metric2	metric3	···	metricN
v1	t1	60	11.5		···	1
v2	t2	70		60		0
v3	t3	45		55		2

Table 6 describes the data model of Lindorm-UWC. Multi-metric data is organized by rows, each of which contains primary keys serving as an identifier for that row and other non-primary key attributes. Lindorm-UWC supports dynamic columns. While inserting data, users only need to specify the primary keys without having to explicitly pre-define non-primary key schema. Clients can write any non-primary key columns, or metrics, to Lindorm-UWC. The blank cells in Table 6 show the case that some metrics are not contained in a write. This is important in IoV systems to support update-style writing and non-fixed metrics. As discussed in §2.2, the majority of queries in IoV systems retrieve data from a single vehicle over a certain period. Therefore, the **vehicle ID** and the data **timestamp** are usually set as the primary keys.

Lindorm-UWC distributes data across regions based on the concatenation of the primary keys (use "primary key" later in the paper for simplicity). Each LDServer manages a disjoint subset of regions and handles read and write requests accordingly. The LDMasters balance loads across the LDServers by scheduling the distribution of regions among nodes and automatically split the region with extremely high loads into smaller regions. Thanks to the partitioned data management and load balancing, Lindorm-UWC can handle massive and rapidly growing data seen in IoV systems.

To handle the writing and querying of multi-metric IoV data, we propose an ultra-wide-column storage engine based on the LSM (Log-Structured Merge-tree) model. Each region deploys an independent storage engine. In this storage engine, data is first written

to the Memtables residing in memory. During this process, values of all non-primary keys in one write operation are merged into a flattened JSON data structure, thereby avoiding the need to insert into each column individually. For update-style writing, We extend the update semantics of LSM-tree for data from different metrics but with the same primary key. With these optimizations, the write throughput of multi-metric data can be significantly improved. When flushing a Memtable to disk as an SSTable where data is organized in rows by the primary key, we maintain an additional data copy of that SSTable in a columnar storage format. Similarly, the compaction process also exports a columnar-storage-format file for the newly compacted SSTable. The coexistence of row-based and column-based storage files facilitates efficient querying of either a small or a large number of metrics.

Due to the massive amounts of data in IoV systems, it is critical to reduce storage costs. We design a mechanism for separating cold and hot data based on our findings that most queries in IoV systems prioritize more recent data. Specifically, newly written data is considered hot and is stored on high-performance storage media. Data that has been in the system for a certain time is considered cold and is moved to less expensive media. To retain the processing efficiency of cold data, we design a *time-window compaction* strategy to reduce the number of compaction operations involving cold files. Besides, to accelerate queries, we maintain additional metadata and index data in hot storage for cold data, which are always accessed first to determine the necessity to read cold data.

3.2 Distributed Architecture

To handle the vast amounts of data generated in IoV systems, Lindorm-UWC leverages a distributed architecture, where data are partitioned into multiple regions for parallel data ingestion and query. However, unlike DevOps systems that run consistently and steadily, different vehicles produce non-uniform data ingestion rates due to differences in their models, running time, *etc.*, and are prone to creating traffic bursts due to unstable network connectivity. This diversity poses a challenge to achieve good load balancing between regions and to handle the emergence of unexpected hotspots.

Data partition. Lindorm-UWC partitions data based on the primary key, which in IoV systems is concatenated from the vehicle ID and timestamp. Each region is responsible for a continuous range of primary keys. When a write request or a query request reaches an LDServer, it will be forwarded to the corresponding regions whose ranges intersect with that of the request. In Lindorm-UWC, compute and storage are disaggregated. To be specific, A region is logically managed by a single LDServer, and physically, its data is stored in a shared distributed file system. When balancing the system load, it becomes convenient to redistribute regions across LDServers without having to migrate massive data [25, 42].

Region swap. Some regions may turn into hotspots when they experience a high volume of write and query requests. These hotspots can cause a backlog of requests on some LDServers, while other LDServers' resources remain underutilized. In such cases, the LDMaster rebalances the load across LDServers by reallocating the ownership of regions.

Firstly, the LDMaster assesses the workload of all regions to identify the hotspots. In particular, the LDMaster keeps real-time statistics of the consumed CPU time of each region as a metric, which reflects the overall load of writing data, querying data, and data persistence in a region. At regular intervals, the LDMaster performs *region swap*. First, the LDMaster calculates the total CPU consumption of all regions on each LDServer as a load indicator for that LDServer. It then sorts the LDServers in descending order based on this value. According to the sorted result, the LDMaster swaps the regions of the first-ranked (*i.e.*, the highest loaded) LDServer and the last-ranked LDServer. The regions for exchange are selected by the following rules, as shown in Figure 6:

1. Calculate the difference in loads, δ, between the two LDServers.
2. Find the region with the smallest CPU use, R_s, from the last-ranked LDServer.
3. Find the maximum CPU usage region, R_b, whose CPU use is no more than $\frac{\delta}{2}$ higher than that of R_s from the first-ranked LDServer.
4. If the CPU use of R_b is higher than that of R_s, then swap the ownerships of R_b and R_s.
5. Repeat step 1~4 until δ is higher than a pre-defined threshold.

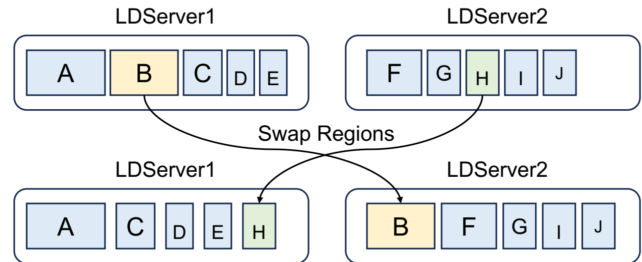


Figure 6: LDServer1 and LDServer2 swap their regions. The length of a region represents its CPU consumption.

Region split. Typically, region swaps based on CPU consumption can accomplish load balancing in most cases. However, if there are very few regions (*e.g.*, inappropriate initial configuration) or if some particular regions are extremely hot, the load balancing cannot be achieved by simply swapping regions. To deal with such situations, the LDMaster splits regions with very high CPU usage. For a region to be split, the LDMaster uses request sampling to find a split point within its primary key range. In detail, the LDMaster tracks a small portion of the recent request traffic for the region and then selects a split point to ensure that the traffic in the two sub-ranges is as equal as possible. After the region is split into two subregions from the split point, the LDMaster transfers these subregions to LDServers with lower loads.

During this process, to quickly complete the region split and minimize interruption of normal data read and write operations, we propose the *cascaded split* strategy based on the *reference split* strategy [8]. The key idea of the reference split strategy is to create a reference to ensure the usage of the original data during data file migration. When a parent region is split into two subregions, each file is divided into a top-half part and a bottom-half part. These parts are then moved to the corresponding subregions. Physically moving the files costs a significant amount of time, during which the subregions cannot process query requests. To solve this issue,

the reference split strategy creates references for the new files as long as the split process begins. The reference points to the file being split and records the positional information (whether it is the top or bottom part). In this way, subregions can locate files through references to handle read requests. Writing new data in subregions remains unaffected due to the LSM's append-write mode. Once the data migration is complete, these references are deleted.

We extend the *reference split* strategy to allow subregions to create references to existing references in the parent region, named *cascaded split* strategy. This is used to handle the case where the hotspot occurs in a subregion before the split process is complete. Figure 7 describes the process of undergoing cascaded splits involving 5 regions, O and A to D. O is initially split into two regions A and B. Before the data migration is complete, A contains references to the files in O and is split into regions C and D soon. At this point, C and D both have references to the references in A. When a query arrives in C or D, it can use the positional information on the multi-level references to quickly locate a precise starting scan point in each file to be retrieved in O. As a result, regions B, C, and D are all able to execute queries. Moreover, the cascaded split strategy can reduce the cost of data migration. Since A is split again, data transfer from O to A can be immediately halted, and data in O can be moved directly to C and D instead.

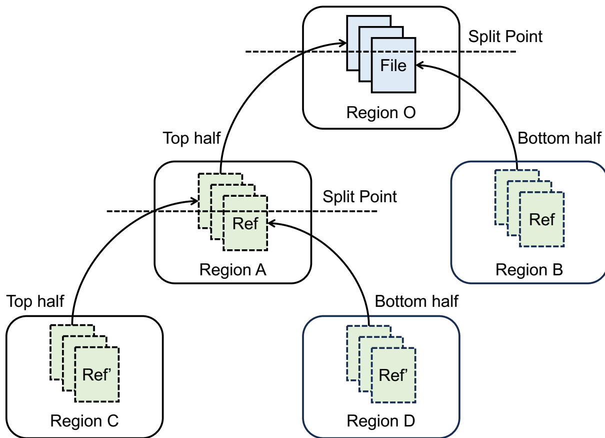


Figure 7: Cascaded split on a hot region O. O splits into A and B, soon A begins to split into C and D.

3.3 Ultra-wide-column Storage Engine

To address the challenges of rapid ingestion of multi-metric data, as well as queries accessing varying numbers of metrics, we propose an *ultra-wide-column storage engine* (UWCSE) based on LSM-tree. UWCSE integrates the advantages of both row-oriented and column-oriented storage systems with low additional overhead. In each region, there is an independent UWCSE responsible for efficiently executing read and write operations routed to that region.

Merged key-value item. As demonstrated in Figure 8, data is first written to an append-only Write-Ahead Log (WAL) in the shared storage to ensure data durability. Subsequently, the data is written into a Memtable. Recall that IoV data comprises a vast array of metrics. Separating data by metrics can introduce an excessive

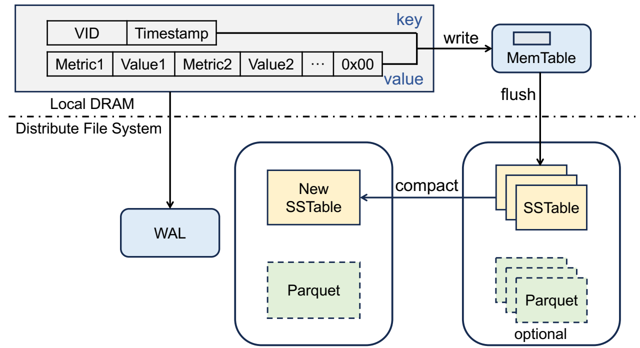


Figure 8: The overview of UWCSE. Data is organized in row-based format in memory and SSTable files. An additional columnar storage file is generated for each SSTable.

amount of key-value (KV) items and associated overhead, such as organizing data chunks for each metric in time-series databases. To address this issue, we design a *merged key-value item* format. Specifically, UWCSE merges data of multiple metrics in a single write operation into one KV item, as shown at the top of Figure 8. Specifically, we use the primary keys of data, *i.e.*, the vehicle ID and timestamp, as the key. This sorted order of keys aids in quickly filtering the time-series data for a specific vehicle (single-vehicle query in §2.2). For the value part, we encode the data of metrics into a serialized byte sequence similar to a flattened JSON string without nested structures. The part of each metric contains its type, name, and value. This encoding allows multiple metrics generated by a vehicle at the same time to be stored in a contiguous space. Therefore, UWCSE can quickly read the required data when faced with queries that retrieve entire rows or a large number of metrics.

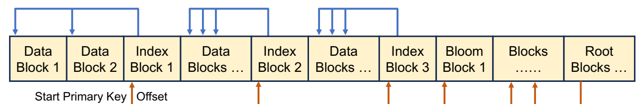


Figure 9: The Structure of SSTable. Each entry in an index block (root block) records the start primary key and the offset of a data block (index block or bloom block).

SSTable. When a Memtable's size exceeds a certain threshold, a flush operation is triggered to persist the Memtable to disk storage, forming an SSTable. As shown in Figure 9, an SSTable consists of many fixed-size (*e.g.*, 32KB) blocks, which are divided into four types: Data Block, Index Block, Bloom Block, and Root Block. A Data Block stores sorted key-value items that are compressed (*e.g.*, with ZSTD, LZ4, etc.) to save space. In an Index Block, each entry records the starting primary key and offset of the corresponding Data Block. It indexes a sequence of Data Blocks, helping to filter out those Data Blocks that do not contain the queried key. The Bloom Blocks store a partitioned Bloom Filter for the SSTable, each of which is a Bloom Filter for a range of primary keys. The final part of the SSTable are Root Blocks, which store the starting primary key and offset for each Index Block and Bloom Block, forming a multi-level index structure. When retrieving a certain key within

an SSTable, UWCSE can locate the Bloom Block and Index Block that may contain the key, by first looking into Root Blocks. In addition, Root Blocks contain the SSTable's metadata, including file information, time range, and column (metric) set. The column set will be used when generating the columnar storage files.

SSTables are regularly merged to avoid a very large number of small files, known as the compaction process. As mentioned in §2.2, update-style writing is common in IoV systems, where data that share the same primary keys but differ in non-primary key attributes are written to the database and stored in different files. Faced with update-style writing, UWCSE extends the update semantics of LSM-tree with the following mechanisms: during compaction and read requests, two KV items with the same key will be merged into a new item by concatenating the two values. This is referred to as the *merge-on-read* strategy. It is effective in IoV systems characterized by high-rate writes, infrequent reads, and the absence of deletion or modification. It ensures that query results are correct, without significantly impacting write throughput.

Columnar storage file. During the flush and compaction processes, in addition to forming a new SSTable, a corresponding columnar storage file containing the same data is also generated. The columnar storage file stores data in an open data format, Apache Parquet [10], and is used to reduce the read amplification for data extraction from a small subset of columns, *i.e.*, metrics. In detail, users can specify the access to the columnar storage files for certain queries that target a small number of metrics. When such a query arrives, the target SSTables are first filtered based on the vehicle ID and time range predicates of that query. Then, the associated columnar storage files are used to retrieve the target columns with fewer I/O operations compared to accessing the SSTable.

Generating a columnar storage file for a new SSTable has two steps: 1) Obtaining the columnar schema; 2) Parsing merged key-value items. First, the metadata in SSTable's Root Blocks records the set of metrics, *i.e.*, columns, of the data in the SSTable. For a compaction operation, the union of the column sets of the involved SSTables can be obtained, serving as the metadata for the new SSTable as well as the schema for the corresponding Parquet file. Second, key-value items from the SSTables are traversed in primary key order through a merge sort. For an item, we de-serialize the value part into structured multi-metric data as a row. It is then written into the columnar storage file using Parquet's interface. In the flush process, the columnar storage file is generated from the Memtable. Since data is originally written in the multi-metrics form, UWCSE directly maintains the column set and raw data for each Memtable to quickly generate the SSTable and Parquet file.

When generating new SSTables and columnar storage files, LDServer failure may interrupt the process. To ensure that an SSTable and its corresponding columnar storage file are always consistent on the disk. We perform consistency checks during crash recovery. Specifically, a new SStable and its columnar storage file are first generated as temporary files. Then, these two files are renamed atomically to final files in turn. When the LDServer breaks down, there will be two cases. The first is that no final files are generated, which indicates that data in the temporary files may be incomplete. Therefore, UWCSE will use the WAL or the original SSTables involved in the compaction to restart the generation process. The second is that the final SSTable file has been generated but the columnar storage file does not exist. This means that the temporary files are successfully generated, thus UWCSE can try to rename the columnar storage file again.

3.4 Cold-Hot Data Separation

We implement a cold-hot data separation mechanism on Lindorm-UWC to reduce the cost of storing massive volumes of IoV data. Similar to TSDBs [46], Data is classified as hot or cold based on the time when the data is written. Newly written data is considered hot and is stored on high-efficiency, high-cost storage mediums such as SSD. When the time since the data has been written exceeds a pre-defined threshold, such as 7 days, the hot data becomes cold data. Cold data is then stored on more cost-effective storage mediums like HDD and OSS [2]. This is because, in IoV systems, queries are more focused on the most recent data. Additionally, UWCSE, based on LSM, creates new files for newly written data, which inherently orders files by time and makes it straightforward and convenient to divide them into hot and cold tiers.

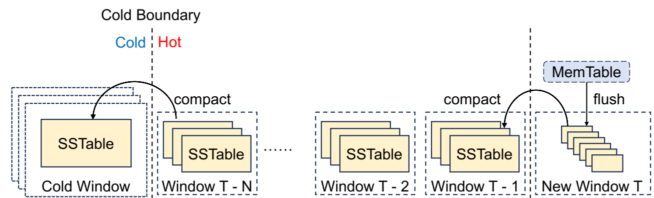
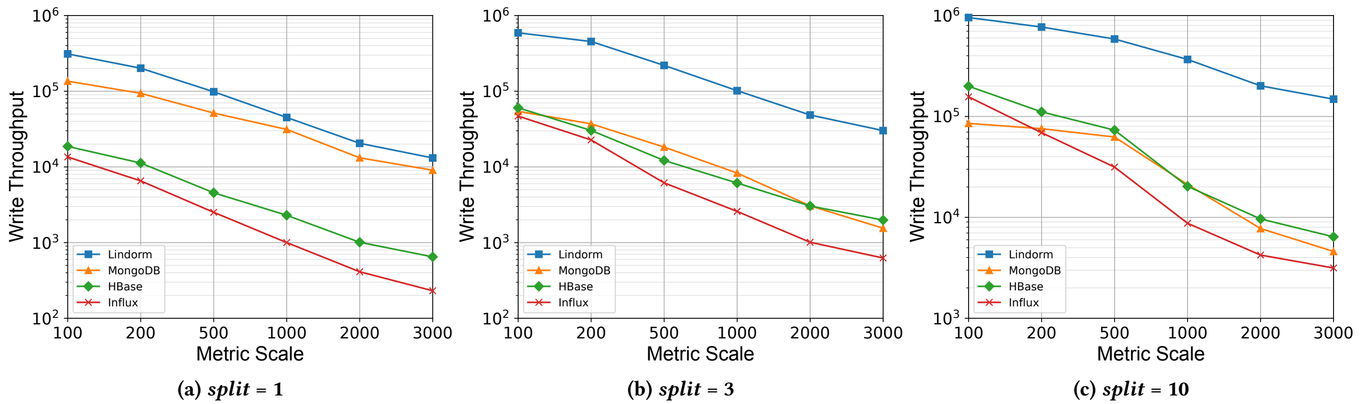


Figure 10: UWCSE periodically generates a new hot time window. After a pre-defined period, the hot window turns to cold, and all its SStables are merged into one.

Time-window compaction. Compaction is used to control the number of SSTable files in UWCSE. A compaction operation involves reading several small SSTables and merge-sorting them to create a new SSTable, which is then written to the storage medium. During this process, if there is data read from or written to the cold storage medium, the cost of compaction can be very high. To address this issue, we employ a *time-window compaction* strategy to avoid accessing the cold storage medium too frequently. As shown in Figure 10, we divide SSTables into non-overlapping time windows, typically one day per window. Windows are classified into cold and hot windows based on chronological order. In the time-window compaction strategy, compaction only merges SSTables in the same window. In a time window, a *universal compaction* strategy [36] is used to determine when to perform compaction and which SSTables to involve. When a hot window turns to cold, all of its SSTables are merged into a single SSTable and then transferred to the cold storage. Cold SSTables compact at a very low frequency, simply to keep the number of files under a reasonable level. In IoV systems, most queries do not require time ranges that span multiple cold windows. Therefore, Cold files that are not merged do not significantly affect query performance.

Cold data filtering acceleration. When dealing with a query, Lindorm-UWC first identifies all SSTables that may contain the query results using filters (*e.g.*, bloom filters and time range filters), and then it scans each file. To facilitate the filter process, Lindorm-UWC redundantly stores the Bloom Blocks and metadata



(a) ***split*** **= 1** (b) ***split*** **= 3** (c) ***split*** **= 10**

Figure 11: Write throughput at different *split* and *metric_scale*.

for filtration in Root Blocks of cold SSTables in the hot storage. This mechanism is referred to as *cold meta cache*. As a result, the target SSTables can be filtered by only accessing the hot storage. In case the query results are indeed in cold files, most Data Blocks in these files will not contain the required data. Therefore, UWCSE also caches Index Blocks of cold files. These indexes in hot storage are searched first to locate data blocks that satisfy the primary key requirements in the query. The cold storage is the last to be accessed to get to the concrete data.

4 EVALUATION

We evaluate Lindorm-UWC in three aspects. First, we comprehensively evaluate Lindorm-UWC's performance on data ingestion (§4.2) and query (§4.3), by comparing it to baseline systems in §2.3 (*i.e.*, MongoDB, InfluxDB, and HBase). We then validate that though UWCSE deploys the columnar storage replica for query acceleration, the additional overhead introduced during data ingestion is acceptable (§4.4).

4.1 Experiment Setup

We deploy the databases and clients required for the experiments on Alibaba Cloud. For each database system, we directly purchase corresponding Alibaba Cloud products [4–7] with consistent hardware configurations (3 servers, each with 8 cores and 64GB of RAM). We deploy an Alibaba Cloud Elastic Compute Service (ECS) [3] server as a client to generate write and query requests, which has 128 cores and 512 GB of RAM.

Data generation. We develop a benchmark suite in Java to generate data of different scales as well as single-vehicle query requests, based on the workload characteristics summarized in §2.2. Specifically, the benchmark generates multi-metric time-series data for each vehicle, where the number of metrics, *metric_scale*, can be adjusted as needed. The amount of data generated for each vehicle is consistent and is determined by pre-set parameters: the number of days, the time span per day that each vehicle generates data. To simulate update-style writing, the benchmark divides a row containing all metric values of a single timestamp into *split* rows with the same timestamp, where *split* is also an adjustable parameter as *metric_scale*. We generate datasets of 10,000 vehicles with different *metric_scale* and *split* for the subsequent experiments.

4.2 Write Performance

We first evaluate the write throughput of each database, *i.e.*, the number of write operations processed per second. We use Java API to write data for all systems, through their respective batch-write interfaces. We fix the batch size, so that each batch contains forty write operations. Then, we tune the number of client concurrency to achieve the best performance for each database. Figure 11 shows the results of the writing performance.

In Figure 11a, we write data to databases without update-style writing. The write throughput of Lindorm-UWC is 79% higher than MongoDB on average. It is more than an order of magnitude higher than HBase and InfluxDB. This is because Lindorm-UWC and MongoDB treat multiple metric values in a single write operation as one row, without processing each metric value individually as InfluxDB and HBase do. Compared to MongoDB's B+tree storage, our ultra-wide-column storage as a LSM-tree is more conducive to writing massive data, resulting in better performance in Lindorm-UWC. InfluxDB has the worst performance, as it organizes multi-metric data into time series, introducing extra overhead of maintaining indexes and data chunk structures for each metric. When following update-style writing in data ingestion (*e.g.*, Figure 11b and Figure 11c, *split* > 1), the write throughput of MongoDB drops severely, leading to an order-of-magnitude gap with Lindorm-UWC. This is because when dealing with multiple writes on the same primary key (*i.e.*, vehicle and timestamp), MongoDB needs to fetch the existing document for that key and then apply updates.

4.3 Query Performance

In query evaluation, we first write data to each database with 2000 metrics and a total period of 24 hours, where each vehicle contains 3 hours of data. Then, we send single-vehicle query requests with different filter conditions to the databases. To be specific, for a query, it has 2 values for the length of its time range filter (*i.e.*, 10 and 60 minutes), and 4 values for the number of metrics required (*i.e.*, 1, 20, 100, and 2000). Combining these two filters can produce 8 types of query. We measure the query throughput of each query type, which is the number of queries completed per second. During the test, we minimize the data overlapping among different queries' results as well as randomize the sequence of query requests to reduce the impact of caches in these databases.

Table 7: Query throughput of different filter conditions.

System	10-minute query				60-minute query			
	2000	100	20	1	2000	100	20	1
Lindorm	**173**	217 (74)*	(421)	**(7524)**	**27.0**	31.3 (**51.2**)	(219)	(1474)
MongoDB	108	**222**	243	244	16.9	33.3	35.9	36.3
HBase	9.84	21.5	25.7	39.6	1.73	3.76	4.39	7.00
InfluxDB	11.5	109	**544**	7093	2.57	50.5	**240**	**1716**

∗ The values in brackets are results obtained by searching columnar storage files in the UWCSE.

Table 7 presents the results for the 10-minute query (the left side) and the 60-minute query (the right side). For the 10-minute query, the results for databases other than Lindorm-UWC have been discussed in §2.3. In short, row-oriented storage, *e.g.*, MongoDB, is more advantageous for querying massive metrics (*e.g.*, 2000); while column-oriented storage time-series database, *e.g.*, InfluxDB, is better suited for querying a few metrics (*e.g.*, 20 and 1). Lindorm-UWC stores two SSTables whose data are organized as both row and column oriented files. As a result, the throughput of 2000-metric queries in Lindorm-UWC is 1.6× as high as that of MongoDB. The document structure in MongoDB (BSON) is more complex and leads to high network traffic when returning query results. When the number of queried metrics becomes fewer, Lindorm-UWC can improve performance as linearly as InfluxDB. Therefore, for queries retrieving no more than 20 metrics, Lindorm-UWC and InfluxDB show similar performance. Note that Lindorm-UWC can achieve 30.8 times the throughput of MongoDB for 1-metric queries. When processing 100-metric queries, Lindorm-UWC can complete 217 queries per second by searching SSTables, greater than the throughput (*i.e.*, 74, the value in the bracket) of reading columnar storage files. MongoDB has the best performance in this case (*i.e.*, 222), indicating that row-based storage is more advantageous here.

In the case of the 60-minute query, the results for querying 1, 20, and 2000 metrics show a similar trend as the 10-minute query. It is worth noting that InfluxDB outperforms MongoDB for 100-metric queries. The query throughput obtained by reading columnar storage files is also higher in Lindorm-UWC (51.2 > 31.3). This is because long time-range queries will hit more rows, incurring more I/Os in row-oriented databases. In contrast, in column-oriented databases, query I/Os mainly depends on the number of queried metrics, and is therefore less affected by the number of hit rows. However, quantifying the overhead of these two storage types under a certain query is difficult. Currently, Lindorm-UWC requires the user to manually specify the storage type for a query.

4.4 Columnar Storage Overhead

Now we evaluate the system load of the UWCSE during data ingestion. With this engine, generating corresponding columnar storage file for an SSTable file incurs additional computation overhead and disk I/O overhead. We measure these overheads by monitoring the system load metrics of Lindorm-UWC with a fixed write throughput under the following two configurations: 1) generating **no** columnar storage file; 2) generating columnar storage files. In this evaluation, we write data to databases with *metric_scale* of 2000 and *split* of 1. During the process, we fix the write throughput to 15,000 (slightly higher than the MongoDB's performance, see Figure 11a). We record the values of CPU usage, disk write throughput (MB/s), and the number of disk I/Os per second (IOPS) for both systems, as shown in Table 8.

Table 8: System load when writing data with 2000 metrics.

System	CPU usage	Disk throughput	Disk IOPS
w/o columnar store	67%	322 MB/s	3066
with columnar store	88%	373 MB/s	3365

As can be seen, generating columnar storage files increases the CPU usage by 21%/67% = 31.3%. The CPU resources are mainly utilized to reorganize the multi-metric data stored in rows into the columnar format. Disk resource consumption increases by 15.8% and 9.8% in throughput and IOPS, respectively, which is mainly from writing columnar storage files. Compared to WALs and SSTables that are composed of rows of various metrics, the columnar storage files store values from the same metric in a contiguous space and apply effective compressions. This reduces the disk overhead to some extent. As shown in §4.3, accessing files in suitable storage formats for different queries can greatly improve the query performance. It introduces acceptable overhead when maintaining columnar storage files and still sustains high write performance.

5 LESSONS LEARNED

The discussed features of Lindorm-UWC that are optimized for IoV scenarios were not conceived all at once, but emerged gradually through an iterative development during the process of business support. This section provides lessons we have learned and insights we have gained during this journey.

Where to solve update-style writing. As we mentioned in §2.2, different components on a vehicle may generate time-unaligned data. Their timestamps will be aligned to a certain precision only when written to the database. Hence, the metrics cannot be compacted on vehicle side. Lindorm-UWC adopts a JSON-like encoding way to cope with a large number of metrics in a write operation, and uses a merge-on-read strategy to deal with update-style writing. Some users, *e.g.*, B,merge multiple data from the same vehicle and the same timestamp before they are written to the database. However, this puts certain barriers on the technical capabilities of the users. It often requires the maintenance of additional components (such as Flink) to align the data in terms of the time window dimension, which also delays the data persistence. Therefore, we choose to optimize the database to simplify the users' workflows.

Time-window compaction. Before adopting the time-window compaction, each region's files had to wait for the major compaction cycle (usually 14∼20 days) to be moved to cold storage. In IoV scenarios where massive new data is generated daily, this cannot meet users' expectation for total storage cost reduction. Moreover, the major compaction strategy involves merging all files. This means that files already in cold storage have to be repeatedly read and written, which is time-consuming. To address this issue, we have proposed a strategy that only compacts hot files and we shortened the execution interval. alleviating the costs of compaction. Finally, the time-window compaction is developed to further reduce the number of compaction operations among the hot files.

Out-of-order data processing. Another common issue in IoV scenarios is the existence of out-of-order data. Vehicles in those areas that lack network access may buffer data locally. Once connectivity is restored, the buffered data becomes out-of-order data insertion. Beside, other out-of-order cases also occur commonly, such as historical data importation. Conventional time-series databases usually require special handling of these data (*e.g.*, using separated storage areas or time windows) to resolve their negative impacts on write and query performance [34, 44]. Fortunately, in Lindorm-UWC, out-of-order data is not an issue. The primary reason is that Lindorm-UWC's regions are usually small (not exceeding 8GB), allowing the total number of files within a region to be maintained at a low level. Even if queries for out-of-order data require merging or sorting multiple files, the impact on query performance is marginal. Besides, out-of-order data in Lindorm-UWC does not immediately enter cold storage and impact write throughput, because the division of windows is based solely on the time data is written.

Pre-splitting vs. Cascaded split strategy. Hotspot issues are common in the early stages of a business, where there are only a few available regions. This can be mitigated by having users implement a pre-splitting strategy in advance to create more regions. However, this approach is not convenient for users, as it is difficult to determine a reasonable pre-splitting value that suits different business scales. The feature of cascaded split strategy enables Lindorm-UWC to automatically and quickly disperse hotspots when they arise in the business, reducing the need for user intervention.

6 RELATED WORK

Time-series databases. Apart from IoV applications, there exist many other time-series databases (TSDBs) oriented toward IoT and monitoring scenarios. TDEngine [18] and IoTDB [44] are for IoT systems. They organize metrics in a column-oriented storage. TDEngine requires users to manually define a schema for each table, which makes it more difficult to optimize when the set of metrics from write operations is uncertain. IoTDB supports deployment in standalone, edge computing, and cloud environments, which improves computation and data storage efficiency at all levels of the IoT system and simplifies data transfer. For DevOps monitoring scenarios, Gorilla [39] and ByteSeries [43] compress time-series data structures to reduce space consumption. InfluxDB [13, 29, 30] proposes TSM based on LSM according to the characteristics of time-series data, including the optimizations of WAL writing, file format, and compaction policy. Lindorm-TSDB [42] designs an advanced execution engine to support low-latency responses to those queries that hit multiple timeseries in monitoring systems.

Overall, an important feature of TSDBs is that their data organization is on a per timeseries basis. However, this feature introduces non-negligible overhead and decreases write throughput in IoV systems that always contain a large number of metrics and vehicles. In addition, the column-oriented storage in TSDBs may consume a high amount of I/Os when a query retrieves many metrics. For other systems like MySQL and KV-store, it also requires heavy system modifications to support time-series data management.

Hybrid systems. Some hybrid systems combine the advantages of row-oriented and column-oriented storage to handle the diverse needs of data analysis and writing. The combination of HBase [8] and Hive [9] is a traditional hybrid system solution. The data is first written to a Hbase system and then exported to a Hive system for more complicated analysis. Besides, HTAP (Hybrid Transactional/Analytical Processing) system is also a widely-used approach, *e.g.*, Hologres [33], TiDB [27], PolarDB [45]. Hologres writes data to both row-oriented and column-oriented storage files. TiDB copies data to columnar storage by adding an observer to the Raft group. PolarDB implements an IMCI (In-Memory Column Index) feature to reorganize an in-memory data copy in a column-based format to accelerate analytical requests. These hybrid systems are designed for those queries with more complex filter conditions than the online queries in IoV workload (*i.e.*, only targeting different numbers of metrics from a certain time range and a single vehicle). Hence, higher overheads can be expected in existing hybrid systems. For instance in HTAP systems, AP engines usually have different partition strategies from TPs engines, introducing data transfer overheads. In comparison, Lindorm-UWC takes the same partition strategy for row-based and columnar data by maintaining the one-to-one relationship between an SSTable and a Parquet file.

Hot/cold data separation. When a huge amount of data is managed, how to reduce the storage cost must be seriously considered. In conventional KV database systems, frequently accessed items are regarded as hot data and the others are cold. In order to store both hot and cold data on appropriate storage media, many works focus on hot data identification (*e.g.*, SA-LSM [50], Mutant [49], Leaper [48], PrismDB [41]), hot/cold data migration (*e.g.*, RocksDB [17], Cassandra [35]), hot storage data structure design (*e.g.*, SpanDB [22], PrismDB [41], X-Engine [28]), and cold data query optimization (*e.g.*, Monkey [24]). In the cases that these systems work for, data with different keys exhibit different levels of hotness. However, in IoV systems (or time-series data management systems), the hotness is usually time-dependent. Therefore, time-series databases separate hot and cold data by time. TimeUnion [46] groups data into different types of time windows (hot, cold, and out-of-order) and merges multiple windows to optimize the query for historical data. Lindorm-UWC tries to avoid compacting files from different windows for better hot data processing performance, since queries for old data are rare in IoV systems.

7 CONCLUSION

In this paper, we conduct the first empirical study on IoV workloads of data read and write using real-world IoV records and queries, and spot three key challenges to underlying database systems. Based on the valuable findings obtained from this study, we propose Lindorm-UWC for managing multi-metric IoV data, which is a distributed database with a cold-hot data separation mechanism. Lindorm-UWC partitions data into different regions and supports automated load balancing. In each region, Lindorm-UWC adopts an ultra-wide-column storage engine that stores data in both row-oriented and column-oriented storage formats to efficiently deal with massive arrives and diverse queries. We evaluate the performance of Lindorm-UWC, and the results show that Lindorm-UWC is effective in high-throughput multi-metric data ingestion and diverse multi-metric queries.

REFERENCES

[1] 2023. 2023 Global Automotive Connectivity Executive Survey. https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/corporate-business-building-to-unlock-value-in-automotive-connectivity. Last accessed: 2024-07-14.

[2] 2024. Alibaba Cloud OSS. https://www.alibabacloud.com/product/object-storage-service. Last accessed: 2024-07-14.

[3] 2024. Alibaba ECS. https://www.alibabacloud.com/product/ecs. Last accessed: 2024-07-14.

[4] 2024. AlibabaCloud HBase. https://www.alibabacloud.com/product/hbase. Last accessed: 2024-07-14.

[5] 2024. AlibabaCloud InfluxDB. https://www.alibabacloud.com/product/hitsdb_influxdb. Last accessed: 2024-07-14.

[6] 2024. AlibabaCloud Lindorm. https://www.alibabacloud.com/product/lindorm. Last accessed: 2024-07-14.

[7] 2024. AlibabaCloud MongoDB. https://www.alibabacloud.com/product/apsaradb-for-mongodb. Last accessed: 2024-07-14.

[8] 2024. Apache HBase. https://hbase.apache.org/. Last accessed: 2024-07-14.

[9] 2024. Apache Hive. https://hive.apache.org/. Last accessed: 2024-07-14.

[10] 2024. Apache Parquet. https://parquet.apache.org/. Last accessed: 2024-07-14.

[11] 2024. Apache ZooKeeper. https://zookeeper.apache.org/. Last accessed: 2024-07-14.

[12] 2024. DB-Engines Ranking of Time Series DBMS. https://db-engines.com/en/ranking/time+series+dbms. Last accessed: 2024-07-14.

[13] 2024. InfluxDB. https://docs.influxdata.com/influxdb/v2.6/. Last accessed: 2024-07-14.

[14] 2024. MongoDB. https://www.mongodb.com/. Last accessed: 2024-07-14.

[15] 2024. Prometheus. https://prometheus.io/. Last accessed: 2024-07-14.

[16] 2024. Prometheus Node exporter. https://github.com/prometheus/node_exporter. Last accessed: 2024-07-14.

[17] 2024. RocksDB. https://rocksdb.org. Last accessed: 2024-07-14.

[18] 2024. TDengine. https://tdengine.com/. Last accessed: 2024-07-14.

[19] 2024. Time Series Benchmark Suite. https://github.com/timescale/tsbs. Last accessed: 2024-07-14.

[20] Saif Al-Sultan, Moath M Al-Doori, Ali H Al-Bayatti, and Hussien Zedan. 2014. A comprehensive survey on vehicular ad hoc network. *Journal of network and computer applications* 37 (2014), 380–392.

[21] Ansif Arooj, Muhammad Shoaib Farooq, Aftab Akram, Razi Iqbal, Ashutosh Sharma, and Gaurav Dhiman. 2022. Big data processing and analysis in internet of vehicles: architecture, taxonomy, and open research challenges. *Archives of Computational Methods in Engineering* 29, 2 (2022), 793–829.

[22] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. 2021. {SpanDB}: A fast,{Cost-Effective}{LSM-tree} based {KV} store on hybrid storage. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 17–32.

[23] JiuJun Cheng, JunLu Cheng, MengChu Zhou, FuQiang Liu, ShangCe Gao, and Cong Liu. 2015. Routing in internet of vehicles: A review. *IEEE Transactions on Intelligent Transportation Systems* 16, 5 (2015), 2339–2352.

[24] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2018. Optimal bloom filters and adaptive merging for LSM-trees. *ACM Transactions on Database Systems (TODS)* 43, 4 (2018), 1–48.

[25] Christian Garcia-Arellano, Hamdi Roumani, Richard Sidle, Josh Tiefenbach, Kostas Rakopoulos, Imran Sayyid, Adam Storm, Ronald Barber, Fatma Ozcan, Daniel Zilio, et al. 2020. Db2 event store: a purpose-built IoT database engine. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3299–3312.

[26] Chaochen Hu, Zihan Sun, Chao Li, Yong Zhang, and Chunxiao Xing. 2023. Survey of Time Series Data Generation in IoT. *Sensors* 23, 15 (2023), 6976.

[27] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[28] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An optimized storage engine for large-scale E-commerce transaction processing. In *Proceedings of the 2019 International Conference on Management of Data*. 651–665.

[29] InfluxData Inc. 2024. InfluxDB TSM. https://docs.influxdata.com/influxdb/v1/concepts/storage_engine/. Last accessed: 2024-07-14.

[30] InfluxData Inc. 2024. InfluxQL. https://docs.influxdata.com/influxdb/v1/query_language/. Last accessed: 2024-07-14.

[31] IoTDB. 2024. Application of Apache IoTDB in the Construction of Chang'an Intelligent Automobile Data Platform. https://www.timecho.com/archives/2022-iotdb-summit-chang-an-qi-che-huang-li. Last accessed: 2024-03-13.

[32] Baofeng Ji, Xueru Zhang, Shahid Mumtaz, Congzheng Han, Chunguo Li, Hong Wen, and Dan Wang. 2020. Survey on the internet of vehicles: Network architectures and applications. *IEEE Communications Standards Magazine* 4, 1 (2020), 34–41.

[33] Xiaowei Jiang, Yuejun Hu, Yu Xiang, Guangran Jiang, Xiaojun Jin, Chen Xia, Weihua Jiang, Jun Yu, Haitao Wang, Yuan Jiang, et al. 2020. Alibaba hologres: A cloud-native service for hybrid serving/analytical processing. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3272–3284.

[34] Yuyuan Kang, Xiangdong Huang, Shaoxu Song, Lingzhe Zhang, Jialin Qiao, Chen Wang, Jianmin Wang, and Julian Feinauer. 2022. Separation or not: On handing out-of-order time-series data in leveled lsm-tree. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 3340–3352.

[35] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS operating systems review* 44, 2 (2010), 35–40.

[36] Qizhong Mao, Steven Jacobs, Waleed Amjad, Vagelis Hristidis, Vassilis J Tsotras, and Neal E Young. 2021. Comparison and evaluation of state-of-the-art LSM merge policies. *The VLDB Journal* 30 (2021), 361–378.

[37] Zhisheng Niu, S Shen, QY Zhang, et al. 2017. Space-air-ground integrated vehicular network for immersive driving experience. *Chinese J. Internet of Things* 1, 2 (2017), 17–27.

[38] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33 (1996), 351–385.

[39] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. 2015. Gorilla: A fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1816–1827.

[40] Meikel Poess, Raghunath Nambiar, Karthik Kulkarni, Chinmayi Narasimhadevara, Tilmann Rabl, and Hans-Arno Jacobsen. 2018. Analysis of tpcx-iot: The first industry standard benchmark for iot gateway systems. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 1519–1530.

[41] Ashwini Raina, Jianan Lu, Asaf Cidon, and Michael J Freedman. 2023. Efficient Compactions between Storage Tiers with PrismDB. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 179–193.

[42] Chunhui Shen, Qianyu Ouyang, Feibo Li, Zhipeng Liu, Longcheng Zhu, Yujie Zou, Qing Su, Tianhuan Yu, Yi Yi, Jianhong Hu, et al. 2023. Lindorm TSDB: A Cloud-Native Time-Series Database for Large-Scale Monitoring Systems. *Proceedings of the VLDB Endowment* 16, 12 (2023), 3715–3727.

[43] Xuanhua Shi, Zezhao Feng, Kaixi Li, Yongluan Zhou, Hai Jin, Yan Jiang, Bingsheng He, Zhijun Ling, and Xin Li. 2020. ByteSeries: an in-memory time series database for large-scale monitoring systems. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 60–73.

[44] Chen Wang, Jialin Qiao, Xiangdong Huang, Shaoxu Song, Haonan Hou, Tian Jiang, Lei Rui, Jianmin Wang, and Jiaguang Sun. 2023. Apache IoTDB: A time series database for IoT applications. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.

[45] Jianying Wang, Tongliang Li, Haoze Song, Xinjun Yang, Wenchao Zhou, Feifei Li, Baoyue Yan, Qianqian Wu, Yukun Liang, ChengJun Ying, et al. 2023. PolarDB-IMCI: A cloud-native HTAP database system at alibaba. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–25.

[46] Zhiqi Wang and Zili Shao. 2022. TimeUnion: An Efficient Architecture with Unified Data Model for Timeseries Management Systems on Hybrid Cloud Storage. In *Proceedings of the 2022 International Conference on Management of Data*. 1418–1432.

[47] Fangchun Yang, Shangguang Wang, Jinglin Li, Zhihan Liu, and Qibo Sun. 2014. An overview of Internet of Vehicles. *China Communications* 11, 10 (2014), 1–15. https://doi.org/10.1109/CC.2014.6969789

[48] Lei Yang, Hong Wu, Tieying Zhang, Xuntao Cheng, Feifei Li, Lei Zou, Yujie Wang, Rongyao Chen, Jianying Wang, and Gui Huang. 2020. Leaper: A learned prefetcher for cache invalidation in LSM-tree based storage engines. *Proceedings of the VLDB Endowment* 13, 12 (2020), 1976–1989.

[49] Hobin Yoon, Juncheng Yang, Sveinn Fannar Kristjansson, Steinn E Sigurdarson, Ymir Vigfusson, and Ada Gavrilovska. 2018. Mutant: Balancing storage cost and latency in lsm-tree data stores. In *Proceedings of the ACM Symposium on Cloud Computing*. 162–173.

[50] Teng Zhang, Jian Tan, Xin Cai, Jianying Wang, Feifei Li, and Jianling Sun. 2022. SA-LSM: optimize data layout for LSM-tree based storage using survival analysis. *Proceedings of the VLDB Endowment* 15, 10 (2022), 2161–2174.