CarStream: An Industrial System of Big Data Processing for Internet-of-Vehicles

Mingming Zhang
SKLSDE, Beihang University
Beijing, China
zmm021@gmail.com

Tianyu Wo*
SKLSDE, Beihang University
Beijing, China
woty@act.buaa.edu.cn

Tao Xie
University of Illinois at Urbana-Champaign
Urbana, IL, USA
taoxie@illinois.edu

Xuelian Lin
SKLSDE, Beihang University
Beijing, China
linxl@act.buaa.edu.cn

Yaxiao Liu
Tsinghua University
Beijing, China
liuyaxiao12@mails.tsinghua.edu.cn

ABSTRACT

As the Internet-of-Vehicles (IoV) technology becomes an increasingly important trend for future transportation, designing large-scale IoV systems has become a critical task that aims to process big data uploaded by fleet vehicles and to provide data-driven services. The IoV data, especially high-frequency vehicle statuses (e.g., location, engine parameters), are characterized as large volume with a low density of value and low data quality. Such characteristics pose challenges for developing real-time applications based on such data. In this paper, we address the challenges in designing a scalable IoV system by describing CarStream, an industrial system of big data processing for chauffeured car services. Connected with over 30,000 vehicles, CarStream collects and processes multiple types of driving data including vehicle status, driver activity, and passenger-trip information. Multiple services are provided based on the collected data. CarStream has been deployed and maintained for three years in industrial usage, collecting over 40 terabytes of driving data. This paper shares our experiences on designing CarStream based on large-scale driving-data streams, and the lessons learned from the process of addressing the challenges in designing and maintaining CarStream.

1. INTRODUCTION

In recent years, the Internet-of-Things (IoT) technology has emerged as an important research and application area. As a major branch of IoT, Internet-of-Vehicles (IoV) has drawn great research and industry attention. Recently, the cloud-based IoV has benefited from the fast development of mobile networking and big data technologies. Different from traditional vehicle networking technologies, which focus on vehicle-to-vehicle (V2V) communication and vehicular networks [28, 36], in a typical cloud-based IoV scenario, the vehicles are connected to the cloud data center and upload vehicle statuses to the center through wireless communications. The cloud collects and analyzes the uploaded data and sends the value-added information back to the vehicles.

Similar to other IoT applications, the vehicle data are usually organized in a manner of streaming. Although each vehicle uploads a small stream of data, a big stream is merged in the cloud due to both the high frequency and the large fleet scale. Therefore, a core requirement in this cloud-based IoV scenario is to process the stream of big vehicle data in a timely fashion.

However, to satisfy such core requirement, there are three main challenges in designing a large-scale industrial IoV system to support various data-centric services. First, the system needs to be highly scalable to deal with the *big* scale of the data. Unlike other systems, the data in IoV are mostly generated by machines instead of humans, and this kind of data can be massive due to the high sampling frequency. Second, the system needs to make a desirable tradeoff between the real-time and accuracy requirements of data-centric services when processing large-scale data with low quality and low density of value. Remarkable redundancy exists in the data due to the high sampling frequency, which causes the low density of data value. However, such data should not be simply removed because it is challenging to identify which data will never be used by any supported service. Third, the system needs to be highly reliable to support safety-critical services such as emergency assistance. Once a service is deployed, it needs to run continuously and reliably along with the data keeping coming in. Unexpected service outage may cause severe damage of property or even life loss.

There can be multiple solutions for designing an IoV system. For example, the traditional OLTP (Online Transaction Processing) architecture uses a mature and stable database as a center to deploy services. In such architecture, the system collects data uploaded by vehicles and stores them into the database, and further provides services based on the analytical ability of the database subsystem. This solution is simple, mature, and can be highly reliable. How-

*Tianyu Wo is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 10, No. 12

ever, it cannot scale well with the fleet size growing because the database can easily become the bottleneck of the whole system. The OLAP (Online Analytical Processing) architecture is often used to deal with large-scale data, in which a data-processing subsystem, rather than a data-storage subsystem (such as a database subsystem), offers the core analytical ability. In such a system, the computing pressure is mainly on the data-processing subsystem, which often executes complex queries. Designing an OLAP-style processing subsystem for IoV scenarios often requires integrating multiple platforms such as Storm [5] and Spark Streaming [4,38].

In this paper, we address the challenges in designing a scalable, high-performance big data analytics system for IoV by describing CarStream, an industrial data-processing system for chauffeured car services. CarStream has connected over 30,000 vehicles in more than 60 cities, and it has multiple data sources including vehicle status data (such as speed, trajectories, engine Revolutions Per Minute (RPM), remaining gasoline), driver activities (such as starting a service, picking up a passenger), and passenger orders. CarStream provides multiple real-time services based on these data. In particular, we address the scalability issue by equipping CarStream with the ability of distributed processing and data storage. We then employ a stream-processing subsystem to preprocess the data on the fly, so that the problems of low data quality and low density of value can be addressed. Our solution achieves higher performance with the cost of more storage space. We also further accelerate the performance of CarStream by integrating an in-memory caching subsystem. Finally, we design a three-layered monitoring subsystem for CarStream to provide high reliability and manageability. The monitoring subsystem provides a comprehensive view of how the overall system runs, including the cluster layer, the computing-platform layer, and the application layer. This subsystem helps sense the health status of CarStream in real-time, and it also provides valuable information of the system to developers who are improving the reliability and performance of the system continuously.

CarStream has been deployed and maintained for three years in industrial usage, in which we keep improving and upgrading the system, and deploy new applications. In this process, we have learned various lessons, which are valuable for both the academic and industrial communities, especially those who are interested in designing a streaming system of big data processing. In this paper, we share our experiences including the system designs and the lessons learned in this process.

In summary, this paper makes the following main contributions:

- An industrial system of big data processing that integrates multiple technologies to serve high performance to IoV applications.
- A three-layered monitoring subsystem for reliability assurance of long/durable-running and safety-critical stream-processing applications.
- A set of lessons learned throughout the three-year process of designing, maintaining, and evolving such industrial system of big data processing to support real-world chauffeured car services.

The rest of this paper is organized as follows. Section 2 describes the background of CarStream, including the business background and the data. Section 3 describes the overall architecture of CarStream. Section 4 discusses the implementation detail of CarStream, including the stream-processing subsystem, the heterogeneous big-data management subsystem, and the three-layered monitoring subsystem. Section 5 presents the lessons learned. Section 6 summarizes the related work and Section 7 concludes the paper.

2. BACKGROUND OF CARSTREAM

The concept of IoV is originated from IoT. There can be multiple definitions of IoV. For example, traditional IoV defines a vehicular network connected by vehicles through the Radio Frequency Identification (RFID) technology. A technology known as Vehicle to Vehicle (V2V) can further share traffic and vehicle information through running vehicles, providing a local-area solution for transportation safety and efficiency. With the wide adoption of mobile Internet, traditional IoV has evolved to a wide-area network deployment that combines in-vehicle network and inter-vehicle network. In such IoV, the vehicle sensors are connected to the Electronic Control Unit (ECU) through CAN-BUS, and a cloud-centric vehicular network is constructed with vehicles connected to the backend servers through the mobile network. The backend servers act as a central intelligent unit that collects, analyzes, stores the data uploaded by vehicles, and further send the value-added information back to vehicles.

By successfully fulfilling many requirements in the transportation domain, IoV has been for years one of the most popular applications in the IoT paradigm. But few large IoV deployments exist, due to the high deployment and maintenance cost for individually owned cars. Private car services, such as Uber [9] and Lyft [8], are successful Internet-driven industrial examples. The underlying system supporting such services can be regarded as a simplified version of IoV where smartphones, instead of dedicated sensors, are used to connect passengers, drivers, and vehicles. This technology has given rise to a huge market targeted by many similar companies established in recent years. To implement an IoV platform, the basic functional requirements should at least include vehicle-data collection, storage, analysis, and services. Vehicles can upload driving data, including instant locations and vehicle-engine statuses, to the backend servers. The data can be used in both real-time and offline vehicle-management applications.

In this paper, we discuss the design and implementation issues of a cloud-centric IoV system named CarStream. CarStream is based on a real-world chauffeured car service company in China, named UCAR. Such service is similar to the Uber private-car service where the drivers take orders from passengers through mobile phones and drive them to their destinations. But the UCAR service also has several unique characteristics: the fleet belongs to the company, and drivers are employees of the company. This centralized business model provides a good opportunity for deploying an Onboard Diagnostic (OBD) connector, a data-sampling device for vehicles. The OBD is connected to the in-vehicle Controller Area Network bus (CAN-BUS). Under a defined interval, the OBD collects vehicle parameter values such as speed, engine RPM, and vehicle error code. The collected data are transmitted back to the backend server through an integrated wireless-communication module. With the help

Table 1: Functionalities and applications provided in CarStream.

Type	Name	Note
Data quality	Outlier Detection	Detect outlier data from vehicle-data streams.
	Data Cleaning	Filter outlier data or repeated data.
	Data Quality Inspection	Analyze data quality from multiple dimensions and generate reports periodically.
	Data Filling	Fill the missing data or replace the wrong data in the streams with various techniques.
Fleet Management	Electronic Fence	Based on user-defined virtual fence (on map) and vehicle location, detect the activities of fence entering and leaving.
	Vehicle Tracking	Track vehicle status including location, speed, engine parameter, error, and warning in real-time. Provide fast feedback to drivers and the control department.
	Fleet Distribution	Fast index and visualize the dynamic fleet distribution in the space dimension; support the vehicle scheduling.
	Gasoline Anomaly Detection	Detect abnormal gas consumption in case of stealing or leaking.
	Fleet Operation Report	Produce statistics of the driving data; offer a regular report on the running status of the fleet, such as daily mileage, gas consumption, valid mileage.
	Driving Behavior Analysis	Evaluate driving skills in terms of safe and skillful driving.
	Driving Event Detection	Detect driving events, such as sudden braking, from data stream.
Decision Making	Driver Profit Assessment	Calculate the profitable mileage and gasoline consumption of a certain trip by using historical data.
	Order Prediction	Based on historical order data, predict the order distribution in the future so that cars can be dispatched in advance.
	Dynamic Pricing	Design a flexible price model based on the real-time passenger demand to achieve maximum profit, e.g., increasing the price for peak hour.
Other	System Monitoring	Provide a comprehensive view for the system-health status.
	Multi Stream Fusion	Merge multiple streams such as the driving events and the GPS locations to extract correlated patterns.
	Trajectory Compression	Losslessly compress huge historical trajectory data.

of such near real-time vehicle data, we can monitor the driving behavior in a fine granularity and react in a timely manner to ensure the safety and quality of the trip services. The data also provide a direct view of vehicle status, being valuable in real-time fleet scheduling and long-time operational management.

Besides the OBD connector, each driver also uses a mobile phone to conduct business-related operations such as taking orders, navigation, changing service status. Therefore, the mobile phone generates a separate data stream, namely business data, and uploads the data to the server in a different channel.

CarStream is designed to process those vehicle data and business data as a back-end service. In general, CarStream collects, stores, and analyzes the uploaded data in terms of vehicle and driver management. So far, a major part of the fleet, over 30,000 vehicles, are connected to CarStream. Those vehicles are distributed among 60 different cities in China. Each vehicle uploads a data packet to the server every 1 to 3 seconds when the vehicle is running. Thus, the backend system needs to process nearly 1 billion data instances per day. The scale of the fleet had expanded from the initial 1500 to 30,000 within a few months; this business expansion poses a horizontal scalability request to the underlying platform.

Figure 1 illustrates the multiple data that are collected and processed in CarStream during a typical day. After running for 3 years, totally 40 TB vehicle data are collected by CarStream. The data can be roughly classified into four types: vehicle status, passenger order, driver activity, and trajectory data, as shown in Figure 1. The data in CarStream are not only large in volume but also comprehensive.

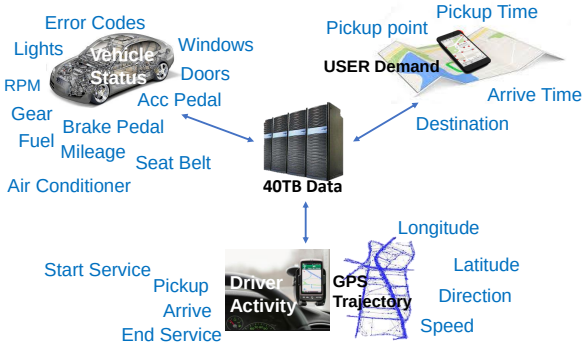

Figure 1: The data processed in CarStream.

However, underneath the abundant data are a series of data-quality problems that need to be addressed. For example, a full set of the vehicle-data parameters contain more than 70 items, but because the vehicles are designed by multiple manufacturers, different subsets of the whole parameter set may be collected from different vehicles. Some vehicles have gear parameter data while others do not. Meanwhile, the running of the vehicle always causes vibration, which in turn affects the correctness of sensor readings such as the remaining gasoline level. Other data-quality problems that CarStream faces include disorder and delay, which may be caused by multiple factors such as distributed processing

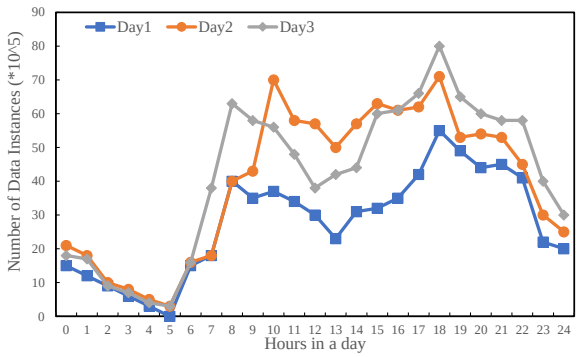

Figure 2: The data amount changes with time.

and network problems. Per the statistics of our dataset, nearly 10% data are disordered or delayed. Such problems may have a significant influence on different aspects of the system, especially on the processing accuracy.

A special characteristic of the data stream in CarStream is that the data stream is subjected to the traffic pattern: the data amount uploaded in peak hours can be as much as 80 times larger than that are in off-peak hours. For this reason, CarStream also needs to deal with burst streams. Figure 2 illustrates the data amount of 24 hours in three days, and clearly shows a traffic-like pattern with morning and evening peak hours.

Based on the data, CarStream provides multiple functionalities and applications to facilitate the fleet management for chauffeured car service. As shown in Table 1, the applications have a dependency on each other. *Data filling*, for example, relies on data cleaning, which in turn has a dependency on outlier detection. The application of fleet operation report has a strong dependency on the functionalities of data cleaning and outlier detection because most of the gasoline data are very inaccurate due to the influence of vehicle vibration. *Multi stream fusion* is another application that other applications depend on. Because the data are collected from multiple sources with different frequencies, the fusion (or matching) becomes a common requirement. For example, an interesting application in CarStream is to detect abnormal gasoline consumptions. In this application, we need to merge the data stream of driver activities with the data stream of vehicle statuses so that we can learn whether the gasoline consumption is reasonable. Sudden drops of the level of remaining gasoline are detected by the system, but a further analysis of the data suggests that most of these phenomena are caused by the influence of parking position. However, such phenomena can also be caused by gasoline stealing, which was not expected by fleet managers.

The applications in CarStream also differ on the performance requirements, as we can infer from Table 1. Some applications may have high-performance requirements as they are time-critical or safety-critical missions. The various application requirements increase the complexity in designing an integrated system that fits all of the listed applications.

3. OVERALL ARCHITECTURE

Based on the aforementioned challenges and the requirements, we design CarStream, a high-performance big data analytics system for IoV. In this section, we present the overview of CarStream and its comparison with other possible solutions. As illustrated in Figure 4, CarStream is an integrated system with multiple layers involved. The components that are marked by the dotted line are the core parts of CarStream and therefore are discussed in detail in this paper.

There can be multiple design choices for fulfilling IoV requirements. The traditional OLTP/OLAP solutions adopt a database-centric design. In such architecture, the uploaded driving data are normally stored in a relational database; the services are implemented based on the queries of the database. Such solution is simple, mature, and easy to deploy and maintain, qualifying such solution to be an option for IoV scenarios with a small-scale fleet. However, for large-scale scenarios, the database-centric architecture would suffer from the poor throughput of data reading and writing. Since the relational database is designed to fulfill the common requirements of transactional processing; it is hard to tailor the database platform in a fine-grained manner, dropping the unnecessary indexing and transactional protection, which consequently cause bad system performance.

Nowadays, as applications related to streaming data become popular, many stream-processing architectures have been proposed. The architecture comprising distributed buffering, stream processing, and big data storage is widely adopted. Such solution can satisfy many IoV scenarios. However, it is not sufficient to deploy the stream processing alone. The comprehensive and evolving application requirements, as well as the varying data characteristics, require both online and offline processing ability. A good design to well connect different loosely coupled components is a key to the success of the system.

3.1 Data Bus Layer

The data bus layer provides a data-exchange hub for system modules to connect different computing tasks. This layer is implemented with a distributed messaging platform with fault-tolerant, high performance, and horizontally scalable ability. This layer also aims at providing a buffer for adjusting the stream volume of downstream data-processing procedures. By eliminating the time-varying spike of data streams, the downstream processing subsystems can take a routine pace to deal with the streams, without worrying about the system capacity being exceeded.

3.2 Processing Layer

The processing layer includes two parts: the online stream-processing subsystem and the offline batch-processing subsystem. The stream processing subsystem is designed to provide a preprocessing functionality to the streams in a way that the query pressure of the database can be reduced. Without preprocessing, the applications based on the raw data would bring huge read pressure to the database. Since the large volume of raw data will be continuously stored, the writing burden cannot be reduced. The system would suffer from the read and write operations simultaneously if they both apply to a single target. An extracted subset of the raw data can be generated and maintained for further query by preprocessing. The result of preprocessing is incrementally updated in a data-driven manner. Such solution satisfies many IoV applications that depend more on the extracted data subset rather than the whole dataset. In this manner, the read and write pressure of the database can be

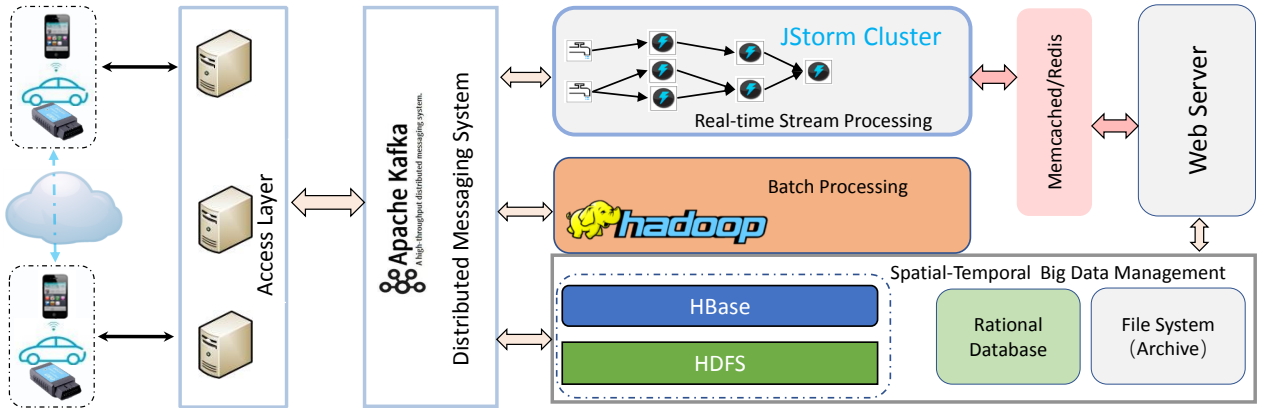

Figure 3: An overview of CarStream.

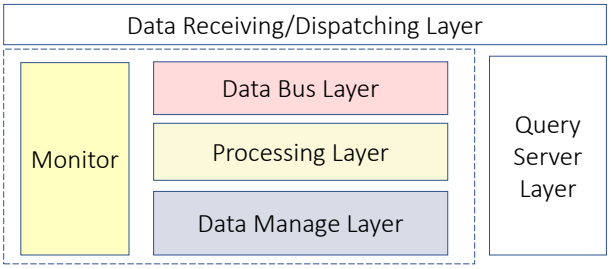

Figure 4: The system architecture of CarStream.

separated, leading to a more friendly architecture to scale to a larger data volume and more applications.

The batch-processing subsystem is employed to satisfy the complex analysis requirements that involve more historical data. Moreover, considering that the low quality of the raw data may affect the accuracy of stream processing, the result of batch processing can also play a role as an auxiliary measurement to evaluate the quality of the stream-processing result. Combining both the stream processing and batch processing, CarStream can satisfy the needs of a wide range of workloads and use cases, in which low-latency reads and frequent updates of data are required.

3.3 Data Management Layer

The data management layer adopts a heterogeneous structure constructed with multiple databases. In CarStream, both NOSQL and SQL databases are adopted to achieve high performance. The design decision is based on a comprehensive analysis of the IoV applications. We store the data in different storage systems per the access requirements of the applications.

More specifically, the most frequently accessed data are stored in the in-memory caching subsystem so that the data can be accessed and updated frequently. By using an in-memory key-value data structure, the in-memory caching subsystem can achieve a milliseconds latency and a high throughput in data updating and exchanging. Choices of in-memory caching can be multiple, such as Memcached [25] and Redis [19]. These platforms are originally designed for caching objects of a web server to reduce the pressure of the database. The in-memory caching employed here acts as the fast media of data update and exchange for applications. On one hand, the processing subsystem keeps updating the current result into the caching; on the other hand, the web servers keep fetching the processing results from the caching. An application example here is vehicle tracking, in which the stream processing subsystem updates the statuses of the vehicles, while the users check these statuses simultaneously through the web browser. Then, the structured and preprocessed data, which can be some smaller data streams than the volume of raw data, are stored in RDBMS and being accessed with SQL. By leveraging the mature object-relation mapping frameworks, it is much easier for RDBMS to be integrated to build web user interfaces. Finally, all the data generated in CarStream are archived in the distributed NOSQL big data storage subsystem, which supports only the non-realtime offline data processing. This architecture increases the system complexity, but it also accommodates the various requirements of different applications and can reduce the risk of data loss in a single-system failure.

3.4 System Monitoring

System monitoring is a unified name for a three-layered monitoring subsystem of CarStream. We monitor the system from three different layers: infrastructure, computing platform, and application. It is necessary to monitor each of the layers. The infrastructure monitoring is used to view either live or archived statistics covering metrics such as the average CPU load and the network utilization of the cluster. The monitoring infrastructure provides not only a real-time health view of the hardware but also a guidance for adjusting the system deployment in terms of load balancing and resource utilization. For the computing platform layer, we monitor the platforms such as the stream-processing subsystem, the batch-processing subsystem, and the databases to track the running status of the platform software. Specifically, the applications in CarStream are monitored because the monitoring information collected for the applications can directly reflect the health status of the provided services. Monitoring the application layer can provide useful information for service-failure prevention or recovery.

4. IMPLEMENTATION PARADIGM

The overall structure of CarStream and the connections between different modules are illustrated in Figure 3. In this section, we discuss the detailed implementation of CarStream from three main parts: the processing subsystem, streaming data management subsystem, and the three-layered monitoring subsystem. We show how to combine different tech-

Table 2: Application requirements of CarStream.

Requirement	Detail
Scalability	With the increase of the fleet scale, we need to be able to improve the ability of the processing platform. We also need to be able to add additional capacity to our data-storage subsystem.
Low Latency	After data are uploaded by a vehicle, they must be processed immediately. The processed result should be provided with low latency.
High Throughput	On one hand, the system needs to be able to receive and store all the uploaded data. On the other hand, the system needs to be able to provide a fast query for multiple users.
High Reliability	CarStream needs to provide a highly reliable data-analytics system to users, including drivers, fleet managers, and passengers, to ensure the dependability of the services.
High Availability	CarStream needs to provide sustainable services that are robust enough to tolerate various unstable environmental conditions.

nologies to accomplish a high-performance data-analytics system for IoV. For each part, we present the problems in the implementation along with the insights of solving these problems.

The requirements of applications in CarStream are summarized in Table 2.

4.1 Processing Subsystem

As introduced in the preceding sections, the data processed in CarStream are uploaded by vehicles. With the continuous vehicle running, the data stream formed by data packets comes continuously. Thousands of small streams are merged into a big stream in the cloud. Such data are naturally distributed; therefore, a natural solution to process the data is using a distributed platform for stream processing. There can be multiple technological solutions for data-stream processing, such as Storm [5], JStorm, Flink [1], Spark Streaming [4, 38], Flume [2], Samza [3], and S4 [32].

Apache Storm [5] is a distributed real-time computating platform for processing a large volume of high-velocity data. Based on the master-worker paradigm, Storm uses topology to construct its computing tasks. Storm is a fast, scalable, fault-tolerant, and reliable platform; it is easy to set up and operate. JStorm is a Java version of Storm while Storm is implemented with Clojure, making it inconvenient for big-data developers who are mostly familiar with Java or C++. Compared with Storm, JStorm is more stable and powerful. Apache Flink [1] is an open-source stream processing framework for distributed, high-performance, always-available, and accurate data streaming applications. Spark Streaming [4,38] is an extension of the core Spark API; such extension enables scalable, high-throughput, fault-tolerant stream processing of live data streams. In Spark Streaming, data can be processed using complex algorithms expressed with high-level functionalities. Apache Flume [2] is a distributed, dependable, and available service for efficiently collecting, aggregating, and moving a large volume of log data or streaming event data. Targeting at processing log-like data, Flume also provides a simple extensible data model for online analytics applications. Apache Samza [3] is a relatively new framework for distributed stream processing. It uses Apache Kafka for messaging, and Apache Hadoop YARN to provide fault tolerance, processor isolation, security, and resource management. S4 [32] is a general-purpose, distributed, scalable, fault-tolerant, pluggable platform that allows programmers to easily develop applications for processing continuous unbounded streams of data. There are also some other stream processing platforms such as Amazon Kinesis, which runs on Amazon Web Service (AWS), and IBM InfoSphere Streams.

The preceding platforms share some common characteristics in that they are all scalable and high performance, and can process millions of data instances per second. They all provide basic reliability guarantee and support multiple programming languages. It is not easy to generally assess the pros and cons of each system. However, for the given requirements, if the decision has been narrowed down to choosing between those platforms, the choice usually depends on the following considerations:

- **System Cost**. Cost is an important factor in selecting a technology since small enterprises or research groups might not be able to afford to pay for services that are not free.
- **Development Cost**. Development cost includes the human cost of deploying the infrastructure platforms and developing the applications. It would be easier to develop a comprehensive data-analytics system if the selected platforms have good support to be integrated together. Meanwhile, the deployment could also be part of the human cost as sometimes it is easy to sink days or weeks of the development time into making open-source platforms to be a scale-ready, production environment.
- **Upgrade Cost**. Business requirements change over time, and the corresponding applications also need to change accordingly. In an earlier stage of a production system, such evolution is especially frequent. Developers often suffer from such circumstance; however, the problem can hardly be attributed to the processing system if the platforms support an easy upgrade of the applications in a way that developers can easily reuse the existing work, so that the upgrade can be less painful.
- **Language Paradigm**. This consideration includes two aspects. The first is what language developers want to use to develop their applications. The second is what language the infrastructure is developed with. Developers prefer to choose the language that they are familiar with to develop their applications. Once they want to improve the infrastructure platforms, it would be more convenient if the platform is developed with the language that developers are familiar with.
- **Processing Model**. Different platforms support different processing models. For example, Storm uses topology to organize its steps of processing the data streams. The processing model decides which platform fits best in implementing the applications.

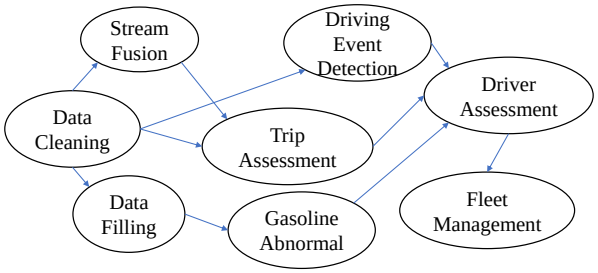

Figure 5: An example of dependency among applications.

After a considerable investigation and experimental exploration, we adopt JStorm as the basic stream-processing platform in CarStream. The main reason is that, besides the basic high-performance, distributed, fault-tolerant characteristics of stream processing, JStorm also uses topology as the computing task model; this characteristic is especially suitable for implementing the aforementioned IoV applications. As discussed earlier, most IoV applications either have a dependency on each other or share similar functionalities. An example of such dependency is shown in Figure 5. It is common to develop and deploy an independent computing task for each application. But sometimes we might need to merge different tasks to improve resource utilization and development efficiency. With topology as the programming model, developers can integrate multiple applications by merging similar processing tasks or split a huge task into smaller (and thus more reusable) function modules. Smaza provides another solution by using KAFKA, a distributed messaging platform, as a data hub. Each task outputs its processing results to KAFKA, and such results can be further used by other tasks.

Another reason of using JStorm is that both its implementation and supported programming language are Java, a language that most big data developers are familiar with. Therefore, when it is necessary to modify the platform during the development, it is easier to modify JStorm than other platforms such as Storm, which is implemented with Clojure. CarStream also employs Hadoop as the offline processing subsystem. Applications in IoV require not only real-time stream processing but also large-scale batch processing. However, such requirement is not the only reason why batch processing is used here. The data-quality issue is another reason. According to our empirical study of the data, nearly 10% data are disordered or delayed, and less than 0.1% data are substantially delayed, or even lost. Although such data occupy a minor part of the whole dataset, they still affect the stream processing significantly in terms of accuracy and system performance. Consider multi-stream matching as an example. The basic requirement is matching stream A with stream B. In this task, if some data of stream B are delayed or lost, then the data in stream A may not find their matches. As a result, the program has to wait or at least buffer more data instances for stream A in case the delayed data in stream B might come later. This solution is common, but sometimes it can be risky because increasing the buffer length may cause memory issues. In our implementation, we detect the data-quality issue and split the bad part of the data for offline processing. Online stream processing deals with only the good part of the data. Offline processing runs with a given interval when most data are ready. When an application launches a request, both online and offline processed results are queried. This combination of online with offline is widely used in CarStream to achieve real-time performance for most data, and CarStream tries its best to offer a better result than performing online processing only.

The data bus layer in CarStream is implemented with KAFKA, one of the most powerful distributed messaging platforms. KAFKA can be easily integrated with most stream processing platforms and hence is widely used as industrial solutions. In fact, there are also many other similar platforms that can be used as the data bus, such as MetaQ, RocketMQ, ActiveMQ, and RabbitMQ.

4.2 Streaming Data Management Subsystem

CarStream needs to manage terabytes of driving data and further to provide services based on the data analytics. The stream is formed by a great number of data instances, each of which takes a hundred bytes. CarStream collects nearly a billion of data instances every day. The value density of such data is very low, as few useful information can be extracted, due to the high sampling frequency. Moreover, the data value decreases fast with time because most of the applications, especially the time-critical applications, in IoV care more about the newly generated data. Though, the historical data cannot be simply compressed or discarded because the data might still be useful for certain applications.

We try to address the issue of data storage in IoV from two aspects. The first aspect is data separation. We identify those applications that have high real-time requirements so that we can further identify the corresponding hot data (such as the vehicle current status). We then separate the hot data with the cold data and try to put them into different storage platforms with different performance. The second aspect is preprocessing. Big dataset scan for the database is painful, but the data appending (write) can be fast. Therefore, we try to extract a small dataset with a higher density of value by using stream processing. We then separate this part of data with the raw data. In this manner, the storage subsystem can provide high throughput for data write, and high performance for the queries that are based on the small processed data. Furthermore, we put some hot data in in-memory cache to achieve a real-time data access.

To construct a data management subsystem, we adopted three different storage platforms in CarStream, including in-memory caching, relational database, and NOSQL big data storage. There are multiple selections for each kind of storage platform. Many NOSQL databases, such as Cassandra, HBase, and MongoDB, can be adopted. In practice, we use HBase as the archive storage platform for the raw data. HBase can easily integrate with Hadoop to achieve big data batch processing ability. HBase provides a high-performance key-value storage, which perfectly fits the requirements of storing the raw vehicle data. The raw data include vehicle status, vehicle trajectories, driver activities related data, and user-order related data. Those data use ID (vehicle ID, user ID, driver ID) and timestamp as the index key. In the technical decision making procedure, we conducted experiments on managing raw driving data with RDBMS such as Mysql and Postgresql. In the test, we use $< ID, timestamp >$ as the index. The result shows that the relational databases have to maintain huge index files to index the big dataset. In another word, the data access performance of RDBMS does not scale with data volume.

However, as a distributed key-value storage, HBase has the scalability with data volume.

Postgresql database is used to store relational data, and some of the processed data, to facilitate the queries from the web server. Compared with other relational databases such as MySQL, DB2, Sybase, the Postgresql has more comprehensive extended spatial functions. This feature is especially important because IoV applications have many location related processing requirements. Even though such requirements are mostly satisfied by the processing subsystem, the database still needs to provide several spatial related query abilities. Postgresql has a spatial database extension named PostGIS, which offers many spatial related functions that are rarely found in other competing spatial databases.

In-memory caching plays a key role in accelerating the real-time applications in CarStream. Memcached and Redis are popular in-memory caching platforms; they both provide high-performance key-value storage. Memcached provides extremely high performance on key-value based caching; therefore, the earlier version of CarStream adopts Memcached to be the data buffering and high-performance data storage media. Later we found that Memcached has limitation in its functionalities and supported data types. As a comparison, Redis supports more data types and data accessing functionalities; Redis also provides data persistence in case of memory data lost. We then switched to Redis to seek for more support on multiple data types. A typical application example of using in-memory caching to improve performance is real-time vehicle tracking. In such application, the vehicle uploads data to the server, and the data are further processed by the stream processing subsystem; the processed results are updated to Redis for further use. Both the data producer (processing subsystem) and consumer (web server) can achieve high performance with this solution.

4.3 Three-Layered Monitoring

Monitoring how the system runs is an essential method to provide highly dependable services [26]. By monitoring the system, maintainers can take corresponding actions timely when something goes wrong. CarStream can be roughly separated into three layers, the infrastructure layer, which includes the cluster and the operating system; the computing platform layer, which includes the processing platforms such as Hadoop, Storm; and the application layer that is formed by the applications. To assure high dependability, each of the three layers needs to be monitored.

Typically, for infrastructure layer, developers often use cluster monitoring tools such as Nagios [29], Ganglia [6], or Zabbix [10] to monitor the memory, network, I/O, and CPU load of the servers. Such monitoring tools also provide a useful dashboard to help to visualize the monitoring. The processing subsystem is formed by multiple processing platforms, which usually provide monitor tools to track how the platform works. It is also an option to use the third-party tools to monitor these platforms [23]. The monitoring of infrastructure layer and computing platform layer provide useful information for assuring service dependability. Infrastructure monitoring has become one of the basic system deployment and maintenance requirements. In CarStream, we monitor the infrastructure with Ganglia. Compared with other monitor tools, Ganglia provides powerful functions to monitor the cluster performance from a comprehensive view,

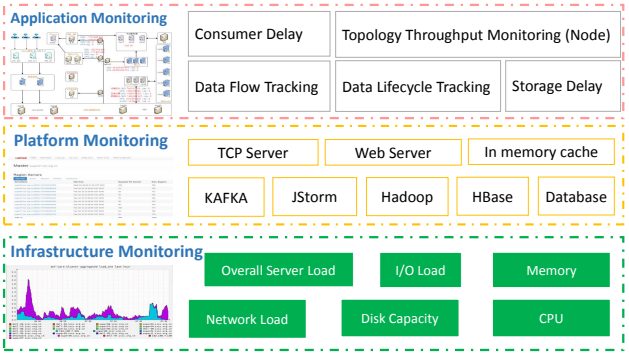

Figure 6: A user interface illustration of the three-layered monitoring subsystem.

it also enables users to define their own monitor items. Ganglia collects cluster information with very low cost. Through the dashboard provided by Ganglia, developers can obtain an overall view of the whole cluster. For computing platform monitoring, our monitoring subsystem simply integrates the monitoring APIs provided by each platform.

However, our maintenance experience on CarStream suggests that application monitoring is also essential. The applications in IoV run permanently with the streaming data continuously come. Meanwhile, the applications are always evolving with business changes. The evolving requires abundant information of the old versions of applications including how each application runs. For example, *Is the parallelism enough for a specific application? How the application works on handling the burst stream spikes? Which task is the bottleneck in the processing topology (which task caused the congestion)?* The answers to these questions are highly related to the monitoring of the application layer. From this perspective, the monitoring of application layer is as important as the monitoring of the infrastructure layer and the computing platform layer. Therefore, we empowered application layer monitoring into the monitoring subsystem of CarStream. In practice, we embedded into each application a module to monitor the parameters that can reflect the performance and the health status of the applications; such parameters include the buffer queue length, the streaming speed, heartbeat, etc. Those parameters are gathered into a shared in-memory database and further being analyzed in near real-time by a center monitor server. Moreover, we employ anomaly detection algorithms on these collected parameters to predict the potential system failures. The monitoring server sends an alert message to maintainers when an abnormal is detected.

With the help of this monitoring subsystem, the reaction time to system problems in CarStream can be reduced from hours to seconds. Figure 6 illustrates the three-layered monitoring subsystem from a user interface perspective. The abundant application-running information that have been collected are further used in improving the alert accuracy of the monitoring. For example, we update the alerting thresholds to more proper ones based on the analysis of historical monitoring information; therefore, the spam alert messages, which are the pain point of maintainers, can be significantly reduced. The daily number of alert messages of CarStream is reduced from tens to just a few routine notifications.

4.4 Performance Evaluation

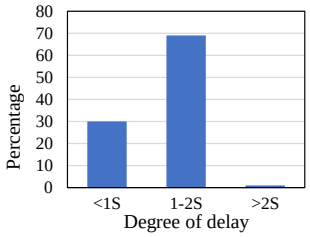

Figure 7: A statistic of the delay in data processing.

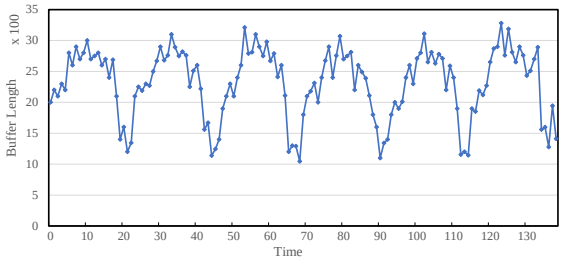

Figure 8: The fluctuation of buffered queue length of stream processing subsystem.

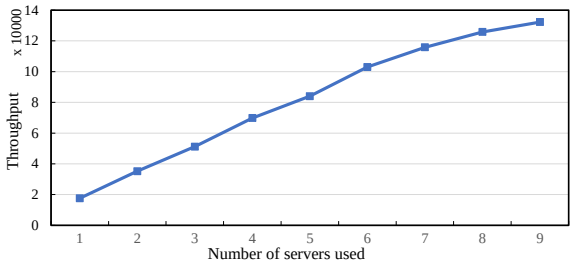

Figure 9: The scalability evaluation of CarStream.

The performance of each platform used in CarStream is evaluated by researchers and the platforms are proving to be high performance. Here, to answer the question that whether the integrated system still scales with data volume and have high performance, we conduct an end-to-end performance evaluation of the entire stream-processing procedure from the data receiving to the data query. The processing subsystem is deployed on a virtualized cluster with 25 servers, each server has 16GB memory, a 2-core 2.0 GHz CPU, 2*10 Gbps Emulex NIC. A dedicated server, which has 2*Intel XEON E5-2630v3, 64GB memory, 2*120G SSD and 102TB HDD, is allocated to deploy the database. The HBase and HDFS are deployed in a non-virtualized environment with three 12-node clusters. The total capacity of HDFS is above 600TB. We test the system from the following aspects.

Throughput. CarStream can easily handle the data stream uploaded by 30,000 vehicles. We further simulate a high data workload by injecting the historical dataset back into the system with high frequency. With the same deployment, the total processing throughput of CarStream has reached 240,000 data instances per second (each data instance is around 100KB).

End-to-end delay. This test shows whether the system can satisfy the low-delay requirement of time-critical applications. We use real-time vehicle tracking as the test application. In this application, the data are uploaded from the vehicles and go through a series of platforms including the buffering platform, stream processing platform, in-memory caching platform, and then finally being queried by the web server. For the end-to-end delay, we compare the timestamps between the data being generated and the data being processed. The result, as illustrated in Figure 7, shows that the average delay is less than 2 seconds; such delay is acceptable in this scenario. We also monitor the length of the queue inside of the computing tasks; such queue length may also reflect the processing delay in an indirect way. The result is shown in Figure 8. It can be inferred from the evaluation result that the buffering subsystem performs well on buffering the workload of peak hours; the buffering slightly increases the processing delay, but the processing subsystem can catch up on dealing with the workload quickly.

Processing scalability. Scalability is a critical issue for CarStream as the fleet scale increases over time. We test the processing scalability of CarStream by increasing the deployment scale of the buffering and processing subsystems. In this test, we use the electronic fence as the foundation application. We deploy the processing subsystem on one server and increase the number of servers gradually. By injecting a large volume of data into the system, we evaluate the relationship between the throughput and the system scale. The evaluation result, as illustrated in Figure 9, shows a near linear scalability of the processing subsystem. This performance can be attributed to two reasons, the distributed design of the processing platform, and the naturally distributed characteristic of the vehicle data. As the fleet scales up, we can always partition the data into small streams and deploy high-performance processing tasks accordingly.

5. LESSONS LEARNED

Lesson Learned 1: Application-level monitoring is necessary for achieving high reliability.

Infrastructure monitoring and computing-platform monitoring are the most commonly adopted techniques for assuring system reliability. Doing so can allow us to take timely reactions when a problem occurs. We can also use the monitoring information to help to utilize system resources by adjusting the deployment. However, we learn through the maintaining of CarStream that infrastructure-level and platform-level monitoring are insufficient to provide highly dependable services, especially in safety-critical scenarios. Our insight is that, it is necessary to monitor the application-level because application monitoring provides direct information to answer key questions such as *which processing step is the bottleneck?* or *what caused the cutoff of the data stream?* By monitoring the key parameters in an application, such as local buffer length, processing delay, and processing rate, we can have a deeper understanding of the application's runtime behavior. These parameters can be collected and analyzed on the fly, helping operators to take actions for service failure prevention or recovery.

Lesson Learned 2: Low data quality widely exists in IoV. A fixing model extracted from historical data patterns can be helpful.

According to our experience, IoV applications usually face a severe issue of low data quality, such as data loss, data disorder, data delay, insufficient data, and wrong data, etc. There are multiple causes of those problems. For example, the data disorder can be attributed to the distributed processing mode of the system. Because the data are processed concurrently so that later data are possible to be

Table 3: Common data-quality issues in IoV scenarios.

Category	Problem	Consequence	Cause	Solution
Lack of data	Data Loss	No result/ Inaccurate result	Network failure	Redundant deployment. Interpolation by data patterns.
			Software fault	
	Insufficient Data	Inaccurate result	Physical limitation (vehicle limitation)	Interpolation by data patterns.
Wrong data	Disorder	Wrong result/ Inaccurate result	Distributed nature of processing platform	Delay and wait. Dropping out-of-date data. Order guarantee design (in app layer). Fixing with prediction.
			Network transition	
			Store and forward design of data nodes	
	Wrong/Outlier Data	Wrong result	Hardware malfunction	Outlier detection. Data cleaning. Fixing with data patterns.
			Inaccurate sensors	
			Other physical reasons	

processed earlier than the earlier-arrived data, resulting in a disordered sequence. The disorder can also be caused by the parallel transmission of the network or the defect of the processing in the application layer. Similarly, the problem of wrong/outlier data can be attributed to multiple factors such as hardware malfunction, the sensor jitter, or other physical reasons (e.g., vehicle vibration). In IoV scenarios, these problems usually need to be addressed in real time. A general list of the problems and the corresponding causes are summarized in Table 3.

There can be multiple solutions to these problems. Let us take dealing with data disorder as an example. A basic solution is delay-and-wait: the processing is pended until all the data instances within the sliding window are ready. This solution compromises real-time performance to achieve the accuracy. Another solution is providing sequence guarantee in the application when designing the system. For example, the data of each vehicle are processed by the same processing node. Hence, the processing is paralleled for the fleet but is sequential for any specific vehicle. This solution also has its own limitation: it sacrifices the flexibility and scalability of the processing subsystem.

In CarStream, we employ a data-driven solution to address the data-quality problem. In particular, we make use of the patterns in the data and drive a fixing model to represent the data patterns, and then we use the fixing model online to improve the data quality. Our empirical study on the data reveals clear patterns from the data; such data nature is largely due to the regularity in driving. For example, the trajectory generated along a road follows the trend of that road, and drivers would maintain the vehicle in a stable status when driving on a highway. During the acceleration stage, most drivers would take a steady acceleration. The model extraction takes some time but the online running can be fast and efficiency. Note that such solution is only applicable to the problem of local data quality where a small part of the stream is missing or disordered.

Lesson Learned 3: In large-scale scenarios, the linked-queue-based sliding-window processing may cause a performance issue.

In stream-processing applications, the sliding window is one of the most commonly used functionalities. Typically, the sliding window is implemented with a linked queue. A new data is appended to the queue when the data comes. Meanwhile, when the queue has reached the maximum length, the oldest data will be removed. The enqueue and dequeue operations have high performance when the dataset is relatively small. However, when it comes to large-scale data streams, such linked-queue-based solution may suffer from poor performance due to the frequent memory-management (allocate and free) operations.

To improve the performance, we suggest using a Round-Robin Queue (RRQ) to implement the sliding window. The RRQ reuses memory to improve the performance of processing. Compared with the linked-queue-based solution, the runtime of processing the same quantity of data with RRQ can be reduced by 80%.

Lesson Learned 4: A single storage is usually insufficient; a heterogeneous storage architecture is necessary for managing large-scale vehicle data.

As a typical big data processing scenario, IoV needs to manage a huge quantity of vehicle data. In this scenario, data management faces severe challenges: the data volume is huge, and the applications are of variety such that different data-access requirements need to be satisfied. For example, decision-making-related applications require a large throughput of the data access, while time-critical applications require higher data-access performance. Based on our experience, there might not be a one-for-all storage platform that satisfies the requirements of all the applications, and a heterogeneous storage architecture is usually necessary for maximizing the system performance.

We summarize two tips on how to improve the database performance from an architecture perspective.

- Separate hot data with archived (cold) data; use in-memory caching as the exchange media for the hot data.
- Avoid scanning a big dataset by extracting a small dataset with preprocessing.

Lesson Learned 5: Single platform may not be sufficient for multiple requirements in IoV.

Simplicity is an important principle in system design: the benefit of a simple design includes ease of maintenance, upgrade, and use. For general computing system, high performance often comes with simplicity of design. However, for IoV system, which has multiple complex application requirements, simple design may not be sufficient to fulfill the requirements of each application. It's also hard for a monolithic system design to be flexible enough to keep up with the changing of service requirements. In CarStream, we adopt a

multi-platform design for the IoV applications. We carefully evaluate the requirement of each application and choose the best platform to deploy the application. Designing in such manner indeed increases the system complexity and redundancy, but by using a robust data bus, processing of multiply data flows can be loosely coupled and fulfill the requirements of IoV together.

6. RELATED WORK

IoV has become an increasingly important area in both industry and academia. IoV is, in fact, an integrated technology including big data processing, distributed systems, big data management, wireless communication, vehicle design technology, human behavior analysis, etc. For a cloud-based IoV, big data processing and management are especially critical.

Various high-performance platforms for big data processing have been proposed. Some examples are Storm [5], Spark [4], Samza [3], S4 [32], Flume [2], and Flink [1]. However, it is still difficult to smoothly integrate different technologies to develop a system for complex scenarios in IoV. Companies providing private car services, such as Uber [9] and Lyft [8], develop complex systems for data processing, by integrating online processing, offline processing, and big data management, to achieve an ecosystem for serving intelligent transportations.

As a key technology of IoV, system design for stream processing [31, 33, 34] is a challenging task. Cherniack et al. [21] discuss the architectural challenges in designing two large-scale distributed systems (Aurora and Medusa [11]) for stream processing; the paper explores complementary solutions to tackle the challenges. At Google, a dataflow model [13] is designed to process unbounded, out-of-ordered data streams. Google also provide CloudIoT [7], a data-processing solution that targeting at IoT scenario. CloudIoT aims at providing a unified programming model for both batch and streaming data sources from IoT. Twitter has been using Storm to process streaming data for years. Kulkarni et al. [30] discuss the limitations of Storm in processing the increasing volume of data at Twitter, and they design and implement Heron, a new streaming data processing engine to replace Storm. For online and offline integrated processing, Twitter also propose Summingbird [18], a framework that integrates batch processing and online stream processing. Kinesis [16] is an industrial solution provided by Amazon as a big data processing platform that runs on Amazon Web Service (AWS). Kinesis provides powerful functionalities for users to load and process streaming data of IoT or other scenarios.

Chen et al. [20] discuss multiple design decisions made in the real-time data processing system of Facebook and their effects on ease of use, performance, fault tolerance, scalability, and correctness. Borthakur et al. [17] propose Facebook Messages, a real-time user-facing application built on the Apache Hadoop platform. Arasu at al. [15] propose an online stream query system named *STREAM*. *STREAM* uses Continuous Query Language (CQL), a query language that is similar to SQL, to query the continuous streaming data. Multiple-stream fusion is an important problem in various stream-processing scenarios such as those in IoV [37]. Ananthanarayanan et al. [14] propose Photon, a fault-tolerant and scalable system for joining continuous data streams. Gedik et al. [27] investigate the problem of scalable execution of windowed stream join operators on multi-core processors, and specifically on the cell processor. The management and analysis of big spatial data are important foundational abilities for IoV. Fernandes et al. [24] recently present TrafficDB, a shared-memory data store, designed for traffic-aware services. Cudre-Mauroux et al. [22] propose TrajStore, a dynamic storage system optimized for efficiently retrieving all data in a spatio-temporal region. Hadoop-GIS [12] and LocationSpark [35] are two high-performance platforms, which are based on Hadoop and Spark, respectively, designed for large-scale spatial data processing.

7. CONCLUSION

In this paper, we have described our experience on addressing the challenges in designing CarStream, an industrial system of big data processing for IoV, and the experience of constructing multiple data-driven applications for chauffeured car services based on this system. CarStream provides high-dependability assurance for safety-critical services in IoV by including a three-layered monitoring subsystem. The monitoring subsystem covers from the application layer down to the infrastructure layer. CarStream further leverages in-memory caching and stream processing to address the issues of real-time processing, large-scale data, low data quality, and low density of value. CarStream manages the large volume of driving data with a heterogeneous data-storage subsystem. We have also shared our lessons learned in maintaining and evolving CarStream to satisfy the changing application requirements in IoV. So far, our system can handle tens of thousands of vehicles. In our future work, we plan to evolve CarStream into a system with micro-service architecture to better maintain the system and to develop new applications when the fleet scales up.

8. ACKNOWLEDGMENT

This work is supported by grants from China Key Research and Development Program (2016YFB0100902), the National Natural Science Foundation of China (91118008, 61421003). Tao Xie's work is supported in part by NSF under grants no. CCF-1409423, CNS-1434582, CNS-1513939, CNS-1564274. We would like to thank UCAR Inc for the collaboration and the data that are used in this paper.

9. REFERENCES

[1] Apache Flink project. `http://flink.apache.org/`. Accessed: 2017-02-28.

[2] Apache Flume project. `http://flume.apache.org/`. Accessed: 2017-02-28.

[3] Apache Samza project. `http://samza.apache.org/`. Accessed: 2017-02-28.

[4] Apache Spark project. `http://spark.apache.org/`. Accessed: 2017-02-28.

[5] Apache Storm project. `http://storm.apache.org/`. Accessed: 2017-02-28.

[6] Ganglia monitor system. `http://ganglia.info/`. Accessed: 2017-02-28.

[7] Google Internet of Things (IoT) solutions. `https://cloud.google.com/solutions/iot/`. Accessed: 2017-02-28.

[8] Lyft engineering blog. `https://eng.lyft.com/`. Accessed: 2017-02-28.

[9] Uber engineering blog. https://eng.uber.com/. Accessed: 2017-02-28.

[10] Zabbix monitoring solution. http://www.zabbix.com/. Accessed: 2017-02-28.

[11] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[12] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz. HadoopGIS: A high performance spatial data warehousing system over mapreduce. *In Proc. VLDB Endow.*, 6(11):1009–1020, 2013.

[13] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *In Proc. VLDB Endow.*, 8(12):1792–1803, 2015.

[14] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman. Photon: Fault-tolerant and scalable joining of continuous data streams. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 577–588, 2013.

[15] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. STREAM: The Stanford data stream management system. In *Data Stream Management*, pages 317–336. 2016.

[16] J. Barr. Amazon Kinesis Analytics: Process streaming data in real-time with SQL. https://aws.amazon.com/cn/blogs/aws/amazon-kinesis-analytics-process-streaming-data-in-real-time-with-sql/. Accessed: 2017-02-28.

[17] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, et al. Apache Hadoop goes realtime at Facebook. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 1071–1080, 2011.

[18] O. Boykin, S. Ritchie, I. O'Connell, and J. Lin. Summingbird: A framework for integrating batch and online mapreduce computations. *In Proc. VLDB Endow.*, 7(13):1441–1451, 2014.

[19] J. L. Carlson. *Redis in Action*. Manning Publications Co., 2013.

[20] G. J. Chen, J. L. Wiener, S. Iyer, A. Jaiswal, R. Lei, N. Simha, W. Wang, K. Wilfong, T. Williamson, and S. Yilmaz. Realtime data processing at Facebook. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 1087–1098, 2016.

[21] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. B. Zdonik. Scalable distributed stream processing. In *Proc. CIDR*, volume 3, pages 257–268, 2003.

[22] P. Cudre-Mauroux, E. Wu, and S. Madden. Trajstore: An adaptive storage system for very large trajectory data sets. In *Proc. 26th IEEE International Conference on Data Engineering (ICDE)*, pages 109–120, 2010.

[23] D. J. Dean, H. Nguyen, P. Wang, and X. Gu. Perfcompass: Toward runtime performance anomaly fault localization for infrastructure-as-a-service clouds. In *Proc. HotCloud*, 2014.

[24] R. Fernandes, P. Zaczkowski, B. Göttler, C. Ettinoffe, and A. Moussa. TrafficDB: HERE's high performance shared-memory data store. *In Proc. VLDB Endow.*, 9(13):1365–1376, 2016.

[25] B. Fitzpatrick. Distributed caching with Memcached. *Linux journal*, 2004(124):5, 2004.

[26] Q. Fu, J.-G. Lou, Q.-W. Lin, R. Ding, D. Zhang, Z. Ye, and T. Xie. Performance issue diagnosis for online service systems. In *Proc. 31st IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 273–278, 2012.

[27] B. Gedik, P. S. Yu, and R. R. Bordawekar. Executing stream joins on the cell processor. In *Proc. 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 363–374, 2007.

[28] M. Gerla. Vehicular cloud computing. In *Proc. 11th Annual Mediterranean Ad Hoc Networking Workshop (Med-Hoc-Net)*, pages 152–155. IEEE, 2012.

[29] D. Josephsen. *Building a monitoring infrastructure with Nagios*. Prentice Hall PTR, 2007.

[30] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter Heron: Stream processing at scale. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 239–250, 2015.

[31] D. Namiot. On big data stream processing. *International Journal of Open Information Technologies*, 3(8):48–51, 2015.

[32] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Proc. IEEE International Conference on Data Mining Workshops (ICDMW)*, pages 170–177, 2010.

[33] R. Ranjan. Streaming big data processing in datacenter clouds. *IEEE Cloud Computing*, 1(1):78–83, 2014.

[34] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34(4):42–47, 2005.

[35] M. Tang, Y. Yu, Q. M. Malluhi, M. Ouzzani, and W. G. Aref. LocationSpark: A distributed in-memory data management system for big spatial data. *In Proc. VLDB Endow.*, 9(13):1565–1568, 2016.

[36] M. Whaiduzzaman, M. Sookhak, A. Gani, and R. Buyya. A survey on vehicular cloud computing. *Journal of Network and Computer Applications*, 40:325–344, 2014.

[37] J. Xie and J. Yang. A survey of join processing in data streams. In *Data Streams*, pages 209–236. 2007.

[38] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proc. 24th ACM Symposium on Operating Systems Principles*, pages 423–438, 2013.