Multi Instance Task in Data Stream Management System

Mrs.Swapna Vanguru, R.Suhasini,
Assistant Professor, Assistant Professor,
Computer Science & Engineering,
CMR College of Engineering and Technology, Hyderabad, India

Abstract— Now a days “Big data” is becoming part of the day. It is a collection of data sets so large and complex that it becomes difficult to process, manage, and analyses data in a just-in-time manner. The rapid growth in the amount of data created in the world continues to accelerate and surprise us. Moreover, big data is more complicated to handle efficiently. In a data stream management system (DSMS), multi-task in a service and multi-instance in a task are commonly used. In other words, a service is composed of input, output, and tasks: The input constantly reads new messages and emits them to the stream. The output is also written streams in the specified format (e.g., a file). A task is executed often in response to newly arrived stream from the input or previous task emission. When a task would interfere with processing in other tasks, the task must be necessary to run multiple copies of the source (i.e., a single task). So, a single task splits multi-instance where resides in physical instances. Today, there are many big data processing frameworks to handle large volume of big data. In this paper, we will discuss in top level projects of the Apache Software Foundation (ASF), is a general-purpose data processing platform. They have a wide field of application and are usable for dozens of big data scenarios. Apache Spark is based on resilient distributed datasets (RDDs). This in-memory data structure gives the power to spark functional programming paradigm. It is capable of big batch calculations by pinning memory. Especially, Spark Streaming wraps data streams into mini-batches. It collects all data that arrives within a certain period of time and runs a regular batch program on the collected data. While the batch program is running, the data for the next mini-batch is collected.

Index Terms—bigdata, data streaming, multi-instance, task (key words)

INTRODUCTION:
Every second an incredible amount of data is being generated around the world. According to the world has created 90% of all its data in the last two years. Such staggering number indicates that the data nowadays is not only large in its size but also implies the fast speed in its generation and changes. The amount of data generated is so large that it is not only difficult to manage but also process using traditional data management tools. This data is referred as big data in today’s context. There are two fundamental aspects that have changed the way data needs to be processed and thus needs a complete paradigm shift: the size of data has evolved to the amount that it has become intractable by existing data management systems, and the rate of change in data is so rapid that processing also needs to be in real-time. To tackle the challenge of processing big data, MapReduce framework was developed by Google, which needs to process millions of webpages every second. Over the last five years with the availability of several open-source implementations of this framework such as Hadoop, it has become a dominant framework for solving big data processing problems. However, recently it was observed that “Volume” is not the only challenge of big data processing and that the speed of data generation is also an important challenge that needs to be tackled for processing sensing data which is continuously being generated. There are a number of applications where data is expected to be ingested not only in high volume but also with high speed, requiring real-time analytics or data processing. These applications include: stock trading, network monitoring, social media based business analytics, and so on. More specifically, there is a high demand of real-time data processing on the Internet or in sensors-related business models as the data generated need to be analyzed dynamically.

For example, Google needs to count the clicks of websites in real time to decide which webpages are popular, and then use this information to leverage the advertisement fees to earn benefits. Besides, there is also a value associated with each dataset that varies with time. For example, static pages may have validity/value for some months; blogging may have for days, and Twitter messages may be valuable for less than a day. To process and analyze a data set which is changing in its value/validity quite fast, it is not productive to apply traditional method of “store and then analyze later” approach. The reason for this is obvious: firstly, such a large amount of data itself is not easy to manage, and secondly, by the time one will start the analysis, data may lose its value. Since the data almost needs to be processed in a real-time manner, the latency in processing the data should be quite low when compared to batch data processing systems such as Hadoop. The limitation of the existing approach in managing and analyzing high volume and velocity data in real-time has led to development of sophisticated new distributed techniques and technologies. The processing of data with such features is defined as “data stream processing” or “stream processing” or sometimes called stream computing. The research in the area of stream processing can be divided into three areas:
• Data stream management systems where online query languages have been explored;
• Online data processing algorithms where the aim is to process the data in single pass; and finally
• The stream processing platforms/engines, which enable implementation and scaling of stream processing-based applications.
Given the high business demand of stream processing platforms, in this chapter, we are specifically focused on the analysis of different stream processing platforms/engines and developing taxonomy of their different features. Current stream processing platforms/systems borrow some features from dataflow systems developed in the 1960s to low level signaling networks developed during the 1980s, and then from data stream management systems developed during the 1990s. Stream processing platforms enable specifically the execution and deployment of real-time data processing applications in a scalable and fault-tolerant manner.

In recent years, due to different challenges and requirements posed by different application domains, several open source platforms have emerged for real-time data stream processing. Although different platforms share the concept of handling data as continuous unbounded streams, and process them immediately as the data is collected, they follow different architectural models and offer different capabilities and features. For both research and business communities who want to adopt the stream processing technology, it is important to understand what capabilities and features different stream platforms offer to its end users.

II. RELATED WORKS

1. Fault-tolerant Schemes for Stream Processing Applications:

The working principle of FT schemes for stream processing applications is to ensure the reliability (correctness of results in spite of failures (both hardware and software failures)). Two key metrics to evaluate an FT scheme are the cost and the recovery performance. Active replication and upstream backup are two widely applied FT mechanisms in distributed systems. Active replication ensures a minimum recovery delay but introduces constant costs, at least doubling resource consumption. The method is applied in earlier stream processing systems or in data engines, that are hosted by a small cluster of machines. It becomes inefficient or even impractical in Distributed Stream Processing Systems (DSPS) where the application scale increases rapidly. Upstream backup has been widely applied, in recent DSPS because it can achieve a resalable recovery delay and only introduces extra cost during failure recovery. Upstream tasks keep data in output buffers as backups for downstream tasks. Upon failure, the restarted task can obtain previous data from its upstream backup(s), and it reprocesses them to recover its status. Combined with checkpointing/recovery upstream backup can support applications that depend on the complete history of previous data. However, the upstream backup scheme introduces backup dependencies among tasks. The recovering task depends on its upstream backup task(s). When its upstream backup task(s) is (are) not adjacent to it, the task(s) in between the upstream backup(s) and the recovering task will also be involved in reprocessing data.

2. FAILURE EFFECT MODELING AND UPSTREAM BACKUP STRATEGIES:

There is a trade-off between backup overhead and recovery latency. Existing works model the backup overhead as a separate resource, i.e. storage resource. Setting all tasks as backup tasks provides that all required backup data are available right away. This introduces the minimum recovery latency; but maintaining backup data introduces runtime overheads. Decreasing the number of backup tasks reduces the overhead. The other extreme is to perform backup only on the input streams that minimize the backup overhead. Because all the upstream tasks are involved recursively in a task recovery, this strategy introduces maximum recovery latencies. Related works study backup strategies for different objectives. seeks materialization configurations that minimize the end-to-end query time. We introduce the Fault-Tolerant Configuration (FTC) problem that seeks the minimum number of backups with guaranteed recovery latency. We study the relationship between recovery latency and backup configuration for the task-level and processor-level failures in and, respectively. A stream topology is similar to the assembly line model in production systems. Existing works analyze the processing throughput considering the effects of unreliable machines based on the queuing theory. The issue of starvation and/or the blocking of failure-free processing caused by a failure is studied. But these works usually assume that extra repairmen are reserved and dedicated the failure recoveries. Previous works assume that there is always reserved resource for recoveries. The recovery cost and failure-free processing cost are considered separately. This is impractical when resource is always limited. This paper aims for a unified model that considers both the recovery cost and the failure free processing cost as a single resource requirement, and it study the corresponding task allocation problem.
C. TASK ALLOCATION STRATEGIES:

Task Allocation Strategies Earlier related works focus on the modelling of task resource requirements and the relationship between resource assignment and processing performance (both throughput and latency). The resource requirement of a task, hereafter referred to as the weight of a task, represents the share of resource (i.e., computational, memory, and bandwidth capacity) required to ensure the processing performance according to input rate. It is common to assume that task weights are given as an input.

Eidenbenz et al. give a theoretical analysis of this problem and prove its NP-hardness. They propose an approach to compute optimal resource assignments for each task in a given stream topology when the stream topology is a series-parallel decomposable graph. At present, a fast heuristic algorithm considering both computational and bandwidth resource requirements, and it uses throughput as the performance metric. Recent works also focus on enhancing the processing latency for both static and dynamic task weights.

The task allocation problem is usually modelled based on the Bin Packing Problem (BPP), which is a well-studied combinatorial optimization problem. For the general BPP, please refer to surveys. However, related works focus on the task allocation problem in a failure-free scenario that does not take the effects of failures into account. In our previous work [23], we study a task allocation problem that uses the recovery latency as a constraint. This work assumes reserved resource is available for recoveries and does not consider the failure effects on the failure-free processing performance. This paper re-visits the task allocation problem from a novel perspective that considers the recovery cost and the failure-free processing cost as a unified resource requirement.

II. MULTI INSTANCE TASK:

In a data stream management system (DSMS), multitask in a service and multi-instance in a task are commonly used. In other words, a service is composed of input, output, and tasks: The input constantly reads new messages and emits them to the stream. The output is also written streams in the specified format (e.g., a file). A task is executed often in response to newly arrived stream from the input or previous task emission. When a task would interfere with processing in other tasks, the task must be necessary to run multiple copies of the source (i.e., a single task). So, a single task splits multi-instance where resides in physical instances.

Figure 1 represents an example of multi-instance task. A single instance task b is connected to the Task a. If the Task b has not an appropriate ability to handle streams, the status of Task b changes a single instance to multi-instance. A multi-instance task gives transparent operations which are equivalent to a single instance task’s operation. Task a just calls a send() to communicate with multi-instance Task b. In fact, the send() method is provided by our own transport protocol.

I. Key Value Approach

A key value fashion with three keys

Messages in key-value fashion are distributed in a fan out fashion to all connected peers. Input stream (i.e., <1, & A, #>) publishes all Task 2’s instances. They receive data stream via a filtering operation. The string of filtering operation is used by keys. For example, the receiver side of Task 2, Instance 1 has a filtering operation such as is_special_chars() function. The sender side of each instance is similar to round-robin data stream model in Figure 2. The result data of each output port is forwarded to a central instance (i.e., Task 2, Instance 2) for aggregating data sequences.
3. Port Reusability

In general, a user in DSMS can submit many services, which conduct tasks associated with data streams. For example, a user develops a word count program that monitors the count of each word from an infinite stream of Twitter feeds. The program may consist of a stream’s input task from Twitter, one or more split sentence tasks from the input task, a word count task, and a result output task. If the user would like to collect data streams from other data sources in addition to Twitter, he/she can make use of other services’ tasks through port reusability. Figure 5 shows an example of port reusability between two services. User 1, Service B exports port 0 via a Uniform Resource Identifier (URI) format. The URI format is task://InputTask#2.0. This reusable design builds profit by eliminating the use of expensive data streams among tasks and reducing network costs.

III TEST RESULTS:

1. DAG Deployment

The Service Designer is a powerful Eclipse-based integrated development environment (IDE) which enables developers to design ease-of-use Directed Acyclic Graph (DAG) application. This includes capabilities for source code generation, graphical DAG development and integration, XML editing and input, output, and task settings. Stream processing is based upon a record at a time. InputTask receives Twitter feed streams from Twitter website. It can forward raw data to Multi-instanceSplitTaskRR, Single-instanceTask, and Multi-instanceSplitTaskKV. These three tasks conduct a split job with various ways. Split data is assigned to two kinds of merge task such as Multi-instanceMergeTask and Single-instanceMergeTask. The count of each word from the Twitter example is collected by OutputTask. The following figure represents an in-depth DAG graph. Multi-instance tasks (i.e., Multi-instanceSplitTaskRR and Multi-instanceSplitTaskKV) are generated by our system. If the system has a burden of processing, then its job is much heavier. This will lead to expand the scope of parallel processing associated with two Multi-instance tasks.

![Simple DAG Graph](image-url)

We implemented simple codes associated with channel communication. It includes multi-instance tasks and round-robin/key-value data stream model. Using these codes, we conducted the performance evaluation of sample DAG graph in Following.
IV CONCLUSION

In this paper, we have proposed the stream processing system associated with stream data processing in the big data platform. This is a prototype with basic communication module and task module. The proposed prototype provides multi-instance tasks, channel communication methods, and port reusability. We also explained a sample DAG graph and an Eclipse-based graphic designer for deployment and conducted performance evaluation. In experiments, the proposed data channel management provided parallel task execution from splitting jobs, and it is useful to process big data even when they are overloaded.

References