# MapReduce-Based Data Stream Processing over Large History Data

Kaiyuan Qi[1,2], Zhuofeng Zhao[1], Jun Fang[1], and Yanbo Han[1]

[1] Cloud Computing Research Center,North China University of Technology, No.5 Jinuanzhuang Road,100144 Beijing, China
[2] Institute of Computing Technology, Chinese Academy of Sciences, No.6 Academy South Road,100144 Beijing, China
{qikaiyuan,zhaozf}@software.ict.ac.cn, yhan@ict.ac.cn

**Abstract.** With the development of Internet of Things applications based on sensor data, how to process high speed data stream over large scale history data brings a new challenge. This paper proposes a new programming model RTMR, which improves the real-time capability of traditional batch processing based MapReduce by preprocessing and caching, along with pipelining and localizing. Furthermore, to adapt the topologies to application characteristics and cluster environments, a model analysis based RTMR cluster constructing method is proposed. The benchmark built on the urban vehicle monitoring system shows RTMR can provide the real-time capability and scalability for data stream processing over large scale data.

**Keywords:** data stream processing, large scale data processing, MapReduce.

## 1 Introduction

With the development of IoT (Internet of Things), real-time sensor data based data stream processing has become the key to IoT applications. When dealing with continuous data stream, processing systems must immediately react and response. Because the finite systems cannot handle the full information of infinite stream, the window mechanism is usually adopted to designate the boundary, within which the accumulated data is called history data. With the improvement of data acquisition and transmission technologies, high data stream speed makes accumulating large scale historical data in a short period possible. Meanwhile, the long-term, comprehensive and accurate requirements of current data stream processing applications also entail the enlargement of history data scale. Take the urban vehicle monitoring system as an example, which collects running vehicle information by sensor devices, and based on the data automatically identifies fake license cars and other illegal cars. These applications, in front of the data stream and historical data, should complete the computations between the both inputs in real time. And the expansion of the window, the increment of data objects (such as vehicles) and the increase of each object data (such as vehicle information), result in the large scale of historical data. With the trend, how to guarantee real-time data stream processing over large-scale historical data, i.e. to

C. Liu et al. (Eds.): ICSOC 2012, LNCS 7636, pp. 718–732, 2012.

provide scalable data stream processing for history data, became a new challenge to IoT and cloud computing.

Traditional study on scalability of data stream processing can be divided into two categories. In the centralized environments, subject to limited memory, scalability is guaranteed by sacrificing the quality of service, such as synopsis data [3] and admission control [1]. In the distributed environments, where the data stream processing network is consisting of multiple operators, scalability is supported by balancing the distribution of operators across multiple nodes [2]. However, the processing capacity is still limited by the window size a single node can handle and scalability is insufficient in the case of large scale historical data.

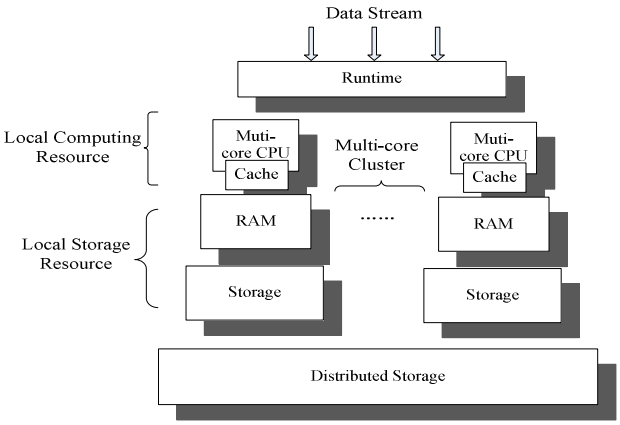



**Fig. 1.** Multi-core cluster architecture

Data stream processing over large scale history data needs breaking limitation of a single node. Today, in order to support large scale data processing, shared-nothing architecture is universal used, as well as 4-tier storage structure of cache, memory, storage and distributed storage. In this architecture, as shown in Fig.1, multi-core CPU forms the local computing resource, and memory and external storage forms the local storage. Under shared-nothing architecture, MapReduce [4] programming model is a core technology to solve large-scale data processing and has been widely adopted. However, the existing MapReduce methods, such as Hadoop[1] and Phoenix [5], are designed for batch processing for static persistent data. Provided continuous data is treated in this way, if the batches processed each time are small, then the system overhead is too large to fulfill real-time requirement, mainly in: 1) the runtime need to be initialized from the scratch, and history data need to be loaded and processed repeatedly, 2) there exists much synchronization and transmission overhead between Map and Reduce phases. If the batches are large, then the processing latency is added.

To support data stream processing, MapReduce should be extended by preprocessing and caching to avoid the repeated overhead on each data stream arrival, and by

[1] Apache Hadoop, `http://hadoop.apache.org/`

pipelining and localizing to reduce the synchronization overhead between stages and the data transmission cost between nodes. This paper proposes a real-time MapReduce model (RTMR) to support such kind of data processing. Furthermore, from the view of constructing RTMR cluster, there exists many possible combinations of node configurations and topologies for different applications and networks, how to build optimal architecture is a problem. This paper also proposes an adaptive RTMR cluster constructing method by establishing and analyzing the RTMR performance model.

## 2 Real-Time MapReduce Model

### 2.1 Key/value Data

In the era of big data, key/value model gradually replace the relational model to become a mainstream data processing model.

**Definition 1.**Key/value data is a 2-tuple {*key*,*val*},in which *key* is key word and *val* is a set of 2-tuple { *attr*,*con*},in which *attr* is attribute and *cont* is its content.

Key/value data is only related to data with the same keys, and independent of processing environment. Because of environment independence, key/value data has the nature of parallel processing and intermediate results can be saved as current state without additional information. Furthermore, the details of parallel processing, load balance and fault tolerance can be hidden from the abstract programming model, programmer can only focus on operations on key/value data.

**Definition 2.** Key/value algebra is a kind of abstract language used for operations on key/value data.

Similar with relational algebra, key/value data algebra also includes set operations (union, intersection and difference), special operations (Cartesian product, selection, projection, concatenation and division), comparison operation (>, < and etc) and logic operations (not, and, or).

### 2.2 RTMR Theory

The definition of MapReduce model is [4]:

$$\text{Map: } k_1,v_1 \rightarrow \text{List}<k_2,v_2>$$
$$\text{Reduce: } k_2,\text{List}<v_2> \rightarrow list(v_2)$$

in which Map phase turns the key/value pairs $<k_1,v_1>$ into pairs $< k_2,v_2>$, and Reduce phase performs operation *list* on the structure $\text{List}<v_2>$ of each $k_2$. Supposing the pending data is *D*, Map intermediate results for *D* is *I*, *M* represents the Map method, *R* represents the Reduce method, and *list* represents Reduce operation, then the above process can be denoted as $MR(D)=R(M(D))=list(I)$.

MapReduce takes full advantage of key/value model: it provides sufficient semantics to parallelly process large-scale data through a simple programming interface, and shield task scheduling and data management from programmers. However, the existing batch processing based MapReduce cannot meet the real-time requirement of data stream processing. In order to extend the real-time capability of MapReduce, we prove MapReduce is no less expressive than ($\preceq$) key/value algebra at first.

**Theorem 1.** key/value ⪯ MapReduce

*Proof.* In MapReduce, selection and projection operators can be implemented in Map, and other key/value algebra operations can be implemented in Reduce. Therefore, MapReduce is more expressive than key/value algebra.

**Definition 3.** For the function $F:S\rightarrow O$, if there exists a function $P:O\times O\rightarrow O$, satisfying $F(D+\Delta)=P(F(D),F(\Delta))$, then $F$ is mergeable.

**Definition 4.** For the data set $D$ and its subsets $D_1,D_2,...,D_n$, if $D_1 \cap D_2 \cap ... \cap D_n = \phi$ and $D_1 \cup D_2 \cup ... \cup D_n = D$, then $D_1,D_2,...,D_n$ is called a partition on $D$.

**Definition 5.** For the key/value data set $D=\{<key,value>\}$ and the key set $K$, the collection $\{d \mid d.key \in K, d \in D\}$ is called a selection of $D$ on $K$, denoted by $\sigma_K(D)$.

By the above definitions, we can see MapReduce has the following properties:

1. Map is distributive, i.e., the Map of the union of two data sets is equal to the union of the Map of the two sets, $M(D+\Delta)=M(D)+M(\Delta)$
2. Reduce is distributive, i.e., if $K_1,K_2,..,K_n$ is a partition on the key set of intermediate results I, then $list(I)=list(\sigma_{K_1}(I))+list(\sigma_{K_2}(I))+...+list(\sigma_{K_n}(I))$

In the traditional batch processing based MapReduce, overhead for repeated processing history data is the key factor to restricting real-time capability, therefore, the large-scale historical data should be preprocessed and cached.

**Theorem 2.** *list* is mergeable ⇔ MapReduce is mergeable.

*Proof.* According to the properties of MapReduce, for data $D$ and increment $\Delta$,

$$\begin{aligned} &MR(D+\Delta)\\ &=R(M(D+\Delta))\\ &=R(M(D)+M(\Delta))\\ &=list(I_D+I_\Delta) \end{aligned}$$

If *list* is mergeable, then

$$\begin{aligned} &list(I_D+I_\Delta)\\ &=list(list(I_D),list(I_\Delta))\\ &=R(MR(D),MR(\Delta)) \end{aligned}$$

That is, if *list* is mergeable, then MapReduce is mergeable, and vice versa.

Theorem 2 shows that by caching MapReduce intermediate results of history data preprocessing, repeated processing overhead can be avoided every time data stream arrives. The above process can be denoted as $MR(D+\Delta)= \text{list}(I_D+I_\Delta)=MR(\Delta|I_D)$.

In the existing MapReduce, another major factor to constraining real-time processing capability is synchronization overhead between phases, which caused by Reduce phase waiting to sort all the Map results. In fact, theorem 2 also shows that there is no data dependency between Map and Reduce phases, so synchronization can be eliminated by the asynchronous pipeline. In pipeline, Map and Reduce phases use buffers to communicate. Each Map task puts the results into buffers immediately after processing, and Reduce task obtains data asynchronously from the buffer to process. MapReduce also includes the synchronized method like Partition, Combine and Sort.

In pipeline manner, Partition and Sort can be completed respectively in Map and Reduce phase. As for Combine method, whether use it or not can be decided by the data compression effect, the algorithm will be detailed in 4.2.

In addition, the data transmission between nodes constrains the processing capability of MapReduce as well. In order to save data transmission cost, local computing resources should be fully taken advantage of to complete MapReduce.

**Theorem 3.** If $K_1,K_2,...,K_n$ is a partition of key set of MapReduce intermediate results $I$, then the MapReduce of increment $\Delta$ over $I$ satisfies

$$MR(\Delta|I)=\mathrm{MR}(\Delta|\sigma_{K_1}(I))+MR(\Delta|\sigma_{K_2}(I))+...+MR(\Delta|\sigma_{K_n}(I))$$

*Proof.* According to the properties of MapReduce, for intermediate results $I$ and increment $\Delta$,

$$MR(\Delta|I)=\mathrm{list}(I+I_\Delta)$$
$$=\mathrm{list}(\sigma_{K_1}(I+I_\Delta))+\mathrm{list}(\sigma_{K_2}(I+I_\Delta))+...+\mathrm{list}(\sigma_{K_n}(I+I_\Delta))$$

For Reduce, the selection of intermediate results on $K_1$ is only relevant to $K_1$, i.e.

$$MR(\Delta|\sigma_{K_1}(I))=\mathrm{list}(I_\Delta+\sigma_{K_1}(I))$$
$$=\mathrm{list}(\sigma_{K_1}(I_\Delta+\sigma_{K_1}(I)))=\mathrm{list}(\sigma_{K_1}(I_\Delta+I))$$

Similarly, $MR(\Delta|\sigma_{K_2}(I))=\mathrm{list}(\sigma_{K_2}(I_\Delta+I))$

$$.....$$
$$MR(\Delta|\sigma_{K_n}(I))=\mathrm{list}(\sigma_{K_n}(I_\Delta+I)$$

Hence, $MR(\Delta|I)=MR(\Delta|\sigma_{K_1}(I))+MR(\Delta|\sigma_{K_2}(I))+...+MR(\Delta|\sigma_{K_n}(I))$

Theorem 3 shows that MapReduce can be localized by distributing intermediate results across the cluster. And because of avoiding data transmission, partitioning the intermediate results properly can guarantee the scalability of the cluster.

### 2.3 RTMR Model

Theorem 1 gives the necessary and sufficient condition that MapReduce is mergeable. However, this condition is only applies to some aggregate operations. For other operations not mergable, an intermediate results cache structure apt to randomly read and write can also be formed by grouping, sorting and indexing when preprocessing.

**Definition 6.** In RTMR, the [$k_2$,List<$v_2$>] and *list*($v_2$) of MapReduce model are called intermediate results.

Following the idea of Metis [6], intermediate results are stored in memory using Hash B+ tree, which is of high performance, as shown in Fig.2. In Hash B+ tree, keys $k_2$ with the same Hash value are grouped in the same Hash table entry as B+ tree, [$k_2$, *list*($v_2$)] are organized as a linked list in the B+ tree leaf node, and *list*($v_2$) is stored in the B+ tree leaf node. If $k_2$ has a unique Hash value, Hash table can be allocated enough entries to avoid Hash conflict and tree search, then the complexity of insertion and search operation is only $O(1)$. If the Hash value of $k_2$ is not unique, the complexity of insertion and search is just $O(1)+O(\log n)$. In order to enlarge the capacity, files

in the SSTable [7] structure are constructed at the external storage to store intermediate results. SSTable consists of an index block and several 64 KB data blocks, as shown in Fig.3, which allocates disk space for Hash table entries in blocks. In data stream processing, if desired intermediate result Hash entry is not in memory but in the external storage and the memory space isn't enough, memory replacement occurs.

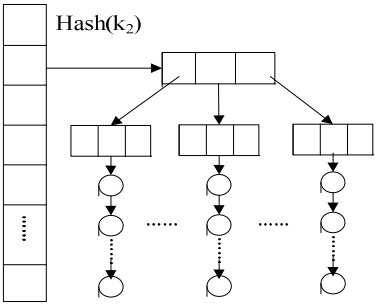



**Fig. 2.** Intermediate result structure

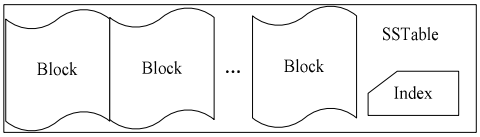



**Fig. 3.** SSTable structure

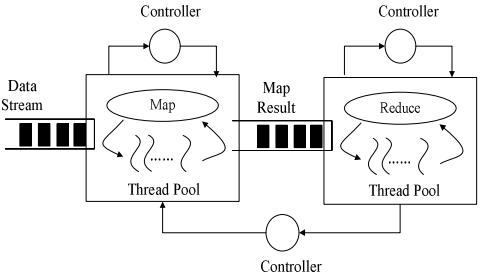



**Fig. 4.** RTMR architecture

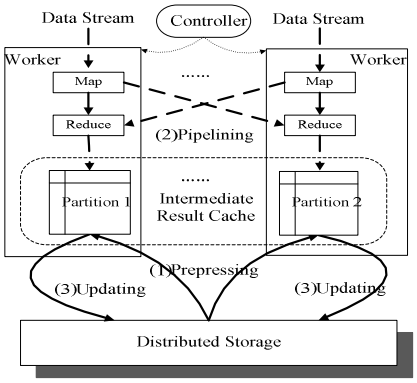



**Fig. 5.** Staged pipeline

In order to support data stream processing, RTMR constructs staged pipeline between Map and Reduce phases, as shown in Fig.4. In pipeline, each stage is comprised of the thread pool, input buffer and intra-stage controller, and shared resources such as threads are allocated by extra-stage controllers. Staged pipeline reduces the initialization overhead on each batch processing by the thread pools, and eliminates the synchronization between phases though event driven buffers. Furthermore, the real-time processing capability of staged pipeline can be improved by intra-stage batch adjustment and extra-stage thread pool control.

Based on the above designs, we propose a real-time MapReduce (RTMR) model for data stream processing over large scale data, which works as (Fig. 5):

1. Intermediate result caching. Preprocess history data resulting in intermediate results, and partition and distribute the results across the worker nodes according to the Hash value on $k_2$.

2. Pipelining. MapReduce proceed in asynchronous way that Map phase groups the data stream by the Hash function on $k_2$ and transmit the data to corresponding Reduce node to compute with intermediate results according to range partitions.
3. Data updating. Update the local results to the distributed storage.

In RTMR, worker is responsible for maintaining the local cache and staged pipeline, and controller is responsible for RTMR job scheduling, fault tolerant and scalability guarantee. This paper mainly focuses on the RTMR model and architecture.

## 3 Adaptive RTMR Cluster

RTMR cluster architecture is decided by the Map/Reduce node configuration and topology. In RTMR, to take full advantage of local computing resource, Map nodes also act as Reduce nodes, and the architecture in the configuration of *x* Map nodes is called RTMR(*x*). For an example of 4-node cluster, the architectures RTMR(1), RTMR(2) and RTMR(4) configured 1, 2 and 4 Map node are shown in Fig.6 (a) (b) (c), respectively. Under different application characteristics, node capacities and network environments, how to construct optimal architecture is a key issue. For the data stream processing system, the goal of adaptively constructing architecture is to minimize the average data processing delay.

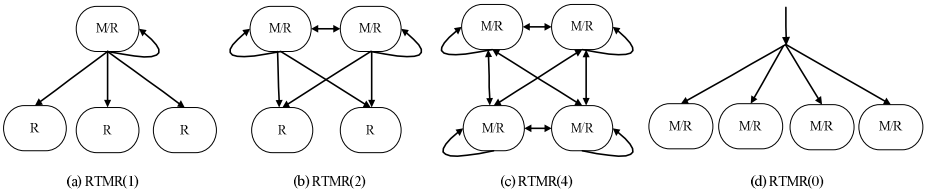



**Fig. 6.** RTMR architectures

### 3.1 RTMR Performance Model

Because the data stream arrival and processing is very similar with the queuing model, and thus queuing theory is a natural selection of the performance model of data stream system [8]. Previous work [3] and our statistics analysis on real scenarios show that data stream arrival process can be modeled as a Poisson process.

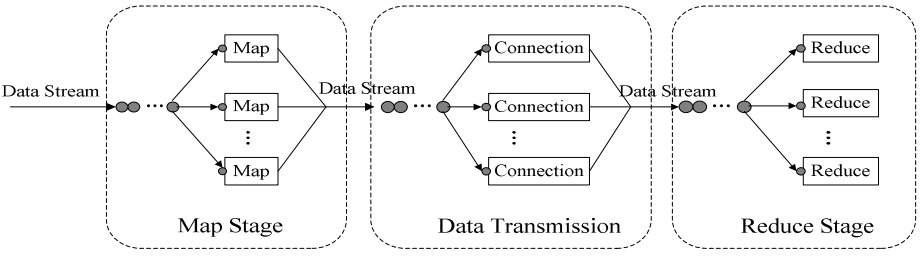



**Fig. 7.** RTMR performance model

Assuming that the arrival data stream is the most simple flow, the Map processing rate, network transmission speed and Reduce processing rate depends on negative exponential distribution, then RTMR cluster can be modeled as a cascade of three queuing system shown in Fig.7. Based on queuing theory [8], for the M/M/*c* queuing system which has *c* service units, when data stream speed is $\lambda$ and processing rate of each unit is $\mu$, the average processing delay is

$$L_q(c,\lambda,\mu) = \frac{(c\rho)^x \cdot \rho}{c! \cdot (1-\rho)^2} P + \frac{1}{\mu} \quad , \tag{1}$$

in which

$$P = \left[ \sum_{k=0}^{c-1} \frac{\rho^k}{k!} + \frac{c\rho^c}{c! \cdot (c-\rho)} \right]^{-1} .$$

$$\rho = \frac{\lambda}{\mu}$$

For a *n*-node RTMR cluster, supposing the number of Map nodes is *x*, Map processing rate of each node is $\mu_m$, then Map stage under the data stream speed $\lambda_m$ is equivalent to a M/M/*x* queuing system. According to equation (1), the Map stage average data processing delay is

$$L_m(x,\lambda_m,\mu_m) = L_q(x,\lambda_m,\mu_m) . \tag{2}$$

For a *n*-node RTMR cluster, *x* of *n* Reduce nodes shared with Map (processing rate $\mu_{r1}$) is equivalent to a M/M/*x* queuing system, the other *n-x* exclusive Reduce node (processing rate $\mu_{r2}$) is equivalent to an M/M/*n-x* queuing system. Therefore, the Reduce stage average data processing delay under data stream speed $\lambda_r$ is

$$L_r(x,\lambda_r,\mu_{r1},\mu_{r2}) = \frac{x}{n} L_q(x,\frac{x}{n}\lambda_r,\mu_{r1}) + \frac{n-x}{n} L_q(n-x,\frac{n-x}{n}\lambda_r,\mu_{r2}) \quad . \tag{3}$$

In RTMR(*x*), the output connections of each Map node are *n*-1, and the input connections of each Reduce node are *x*. If nodes is connected by the switch and the bandwidth between two nodes is $\mu_n$, then each connection bandwidth is inversely proportional to the total number of connections, that is

$$\mu'_n = \frac{\mu_n}{x+n-1} \quad .$$

On each connection, the data speed is $\lambda_r / xn$, according to the M/M/1 queuing model [10], the network delay of data stream through one Map node is

$$L_n'(x,\lambda_r,\mu_n) = \frac{1}{\frac{\mu_n}{x+n-1} - \frac{\lambda_r}{xn}} \quad .$$

And the average network delay of data stream parallelly through all Map nodes is

$$L_n(x,\lambda_r,\mu_n)=\frac{L_n'(x)}{x}=\frac{1}{\frac{x\mu_n}{x+n-1}-\frac{\lambda_r}{n}} \quad . \tag{4}$$

In RTMR, too many threads will cause additional overhead such as context switching and critical resources competition. Corresponding to the connections, each pair of Map and Reduce nodes exists $n$+$x$-1 threads receiving and sending data. If the delay factor is $\varepsilon$, then the extra delay of data stream passing through 1 Map node is

$$L_e'(x)=(n+x-1)\cdot\varepsilon \quad ,$$

And the average extra delay of data stream parallelly through all Map nodes is

$$L_e(x)=\frac{L_e'(x)}{x}=\frac{(n+x-1)\cdot\varepsilon}{x} \quad . \tag{5}$$

Above all, the data stream processing delay of RTMR($x$) is

$$L(x)=L_m(x,\lambda_m,\mu_m)+L_r(x,\lambda_r,\mu_{r1},\mu_{r2})+L_n'(x,\lambda_r,\mu_n)+L_e(x) \quad . \tag{6}$$

### 3.2 Model Analysis

In general, adaptive constructing RTMR is to analyze the extreme value of equation (6) to determine $x$. Given space limitation, instead of mentioning the solution of the extreme value of $L(x)$, two more practical architectures are discussed.

In RTMR, Combine method can be used to reduce the data transmission. If the data compression rate of Combine method is $\tau$, then the data speed of Reduce stage is $\lambda_r'=\tau\lambda_r$, and if the Map processing rate is down to $\mu_m'$, then the data processing delay of RTMR ($x$) is

$$L_c(x)=L_m(x,\lambda_m,\mu_m')+L_r(x,\tau\lambda_r,\mu_{r1},\mu_{r2})+L_n'(x,\tau\lambda_r,\mu_n)+L_e(x) \quad .$$

Then for applications that exists Combine, whether implement Combine or not is decided by comparing $L(x)$ with $L_c(x)$.

Under the current IoT environment, the data speed is limited by acquisition terminal bandwidth, and the preliminary processing such as filtering and encoding has been completed by the communication servers, so it only occupy a small part of CPU time to accomplish the data receiving, transformation, selection and projection as well as partitioning and combining. In this case, the impact of Map Processing on Reduce performance on the shared node can be ignored, i.e. $\mu_r=\mu_{r1}\approx\mu_{r2}$.Then the Reduce stage can be considered as M/M/$n$ system, the processing delay is

$$L_r(x,\lambda_r,\mu_{mr})=L_q(n,\lambda_r,\mu_r) \quad ,$$

which is independent of $x$. Due to equations (2) (4) (5) are monotonically decreasing functions of $x$. Thus, $L(x)$ is a monotone decreasing function, i.e., for a $n$-node cluster, RTMR($n$) contributes to the minimum delay by the most highly parallel processing.

Besides, by theorem 3 we know another architecture shown in Fig.6 (d): the intermediate results are cached across distributed nodes; each node redundantly receiving the data stream, in pipeline manner, filters the data in the charge of itself at Map phase and processing the data over the local cache at Reduce phase. This architecture is defined as RTMR(0). In RTMR(0), if the existing computing and storage resources cannot satisfy the real-time requirements, the cluster can be scaled up to more nodes by repartitioning and moving the cache data. Due to avoiding data transmission and extra latency, the data stream through each node is equivalent to passing through a M/M/1 Map stage plus a M/M/1 Reduce stage, the delay of RTMR(0) is

$$L'(0) = \frac{1}{\mu_m - \lambda_m} + \frac{1}{\mu_r - \frac{\lambda_r}{n}} \quad .$$

And the processing delay of data stream parallelly passing through $n$ nodes is

$$L(0) = \frac{L'(0)}{n} = \frac{1}{n\mu_m - n\lambda_m} + \frac{1}{n\mu_r - \lambda_r} \quad .$$

Apparently for these applications, adaptively constructing RTMR cluster is comparing $L(n)$ with $L(0)$.

## 4 Evaluation

In this section, we utilize the real-time urban traffic data processing applications as the benchmark to evaluate RTMR.

In a large city, where license plates reach $10^7$, the peak will reach 10 MB/s if comprehensively capturing running vehicle data (1 KB for each item, about 10 000 items/s). Meanwhile, if the data have been stored for 1 day, history data will reach 1 TB. In the benchmark, three typical applications are adopted, which are all from real scenario of urban traffic monitoring system and can be regarded as the representative use cases out of related references[9-14].

Fake-licensed car is determined by space-time contradiction. For each item of real-time vehicle data at certain points, retrieve all the historical items at other points within the maximum time threshold, and if the time difference between the two items is less than the time threshold for the two points, the vehicle is suspected to be fake-licensed. The RTMR algorithm is implemented as: for each license plate of item, Map indexes its entry in Hash table grouped by plates; Reduce locates its list in the B+ tree, checks time difference with each historical data, and updates the list.

Traffic statistics application reports the vehicle counts of all the monitoring points, the RTMR algorithm is: for each item of real-time data occurred at certain point, Map indexes its entry in Hash table grouped by monitoring points; Reduce finds the immediate result in the B+ tree to merge and update it.

Traffic flow analysis application calculates the average travel speeds between two points to provide traffic guidance, the RTMR algorithm is: for each item of data captured at certain point, Map transforms the data into GPS coordination data and indexes its entry in Hash table grouped by monitoring points; Reduce finds the list in the

B+ tree, inserts the real-time data, eliminates the overdue data, and periodically merge immediate results within the window to compute the average travel speed.

In the above 3 RTMR algorithms, Hash function hash($k$)=$k$ mod $2^{20}$ can be used to group data items, and the intermediate result Hash table has $2^{20}$ entries, each storing data of $10^7/2^{20}\approx10$ license plates.

RTMR cluster is set up on the 2×4 cores 2.0 GHz CPU, 32 GB RAM and 250 GB disk servers, using a 4×4 cores 2.4 GHz CPU, 64 GB RAM server as control node, and the cluster is connected by 1 Gbps Ethernet and switches. Additionally, Load Runner 9.0 is deployed in a dual-core 3.0 GHz CPU and 4 GB RAM server to simulate data stream. In order to evaluate the scalability, on the basis of random and local characteristics of vehicle data stream, we evenly partition immediate result ranges across the cluster and simulate the uniform distribution stream. The method is: First, use the decimal interval $(0,10^8]$ to simulate license plates. Second, if there exists n nodes, select n subsets on immediate result ranges of n nodes $P_1',P_2',...,P_n'$, satisfying $|P_1'|+|P_2'|+...+|P_n'|=10^5$, and then generate loads for each node cyclically. Third, for node $i$, select a random entry $t$ in $P_i'$, select a random number $x$ in the interval(0,10) and regard $2^{20}x+t$ as the license plate of the data item, at last, randomly set its point and add its timestamp.

Base on the benchmark, each experiment is conducted 10 tests, and at each test we sample results for 10 minutes at steady state of the stream processing system, taking the averages as the final results.

### 4.1 Adaptive Architecture Analysis

First, we analyze the adaptive architecture of the 3 applications shown in Table 1.

**Table 1.** Benchmark applications

| Application | Map | Reduce |
|---|---|---|
| Traffic count | The compression rate of Combine method is effective | Merging |
| Fake-licensed car | No combine | Comparing and updating |
| Traffic flow analysis | Data transform costs much overhead, and compression rate is 0 | Merging and updating |

For the traffic statistics application, compression ratio can be effective to reduce the cost of data transferring, so the Combine method should be adopted.

For the fake-licensed car monitoring, because of the absence of the Combine and other operations in Map phase, satisfying $\mu_r = \mu_{r1} \approx \mu_{r2}$, then the only thing for adaption is to compare $L(n)$ with $L(0)$. Experiment 1, under a 4-node cluster, compares the data processing performance of RTMR(0),RTMR(1), RTMR(2) and RTMR(4) over different history data scale. As Fig.8 shown, no matter what scale, the processing capabilities of RTMR(1), RTMR(2) and the RTMR(4) are promoting with the increase of Map nodes. In addition, when the data speed exceeds 15 MB/s, RTMR(0) is less powerful than the other 3 architectures, this is because in the broadcast mode,

when it come to high speed data stream, receiving data and processing Map stage on each node occupies too much CPU time, which reduces the CPU time for the Reduce and thereby constrains the overall performance. With the history data scaling up, performances of all the architectures decrease, when the speed drops to 15 MB/s, RTMR(0) starts to be the most powerful, this is because the receiving and Map processing overhead on each node no longer affect the Reduce stage, and meanwhile avoid the data transmission cost.

For traffic flow analysis, although there exists Combine method, it cannot reduce the data transmission significantly because its Reduce need to maintain all the data within the window. Furthermore, its Map method includes expensive GPS coordinate transformation operation, dissatisfying $\mu_r = \mu_{r1} \approx \mu_{r2}$ .The results of the extreme value analysis of $L(x)$ under the 4-node cluster are: from 0 to 200 GB data scale, $x = 2$; from 200 to 600 GB, $x = 3$; from 600 to 800 GB, $x = 1$. Experiment 2 compares the data stream processing performance of RTMR(1), RTMR(2), RTMR(3) and RTMR(4) over different historical data scale. Fig.8 shows the empirical results are consistent with the model analysis.

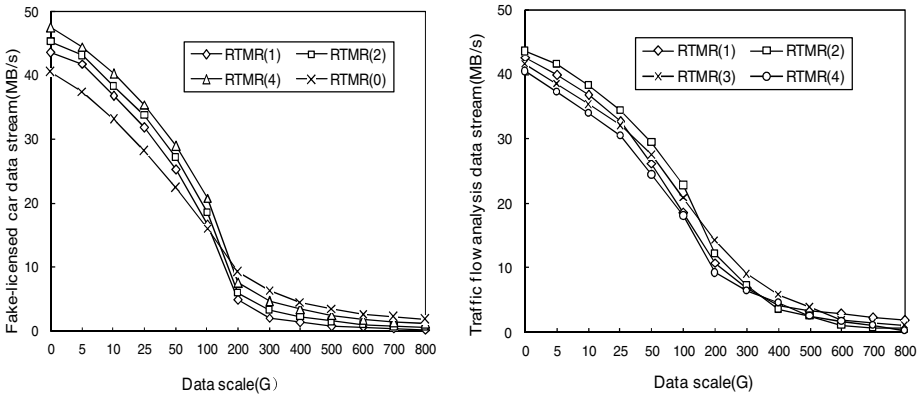



**Fig. 8.** Analysis of adaptive architecture

## 4.2 Scalability Analysis

Experiment 3 and 4 compare the scalability of RTMR(0) and RTMR($n$). Experiment 3, at the fixed data stream 2 MB/s, tests historical data scale the cluster can handle when adding nodes. Fig.9 shows, the promotion trend of RTMR(0) capacity is approximately linear, which is because RTMR(0) minimize the data transmission and synchronization between nodes which affects the enhancement of parallel throughput by distributing intermediate results and localizing. And the reason why RTMR(0) doesn't achieve linear scaling is that local file read and write overhead increases when intermediate results scaling up. As for RTMT($n$), as the nodes are added, data receiving and transferring cost between nodes increases significantly, thus limiting historical data scale that can be handled. Experiment 4, at the fixed intermediate results 50 GB for each node, tests the data stream the cluster can process when adding nodes.

Fig.10 shows that, as the node increases, although the data sending-receiving and transmission costs increase, but RTMR($n$) is more scalable than RTMR(0) in data stream speed, this is because RTMR($n$) distributes data stream to be processed parallelly across nodes, while RTMR(0) is restricted by the increasing CPU overhead of receiving data and processing Map. Specifically, when the data stream speed is less than 15 MB/s, the growth of RTMR(0) processing capability is approximately linear, and when the speed is more than 15 MB/s, the growth slows down.

From the experiences of using RTMR to solve vehicle monitoring, the data stream speed in current IoT environment, constrained by the bandwidth, is far less than 15 MB/s. In the situation of large scale history data, RTMR(0) is more adaptive.

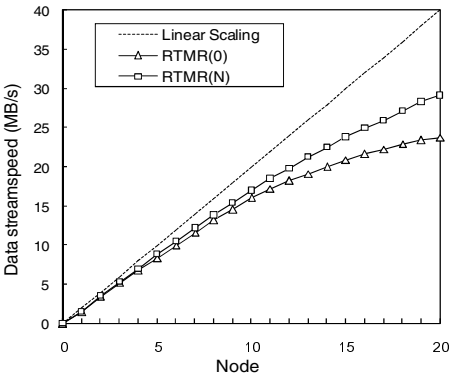



**Fig. 9.** Scalability analysis for data stream

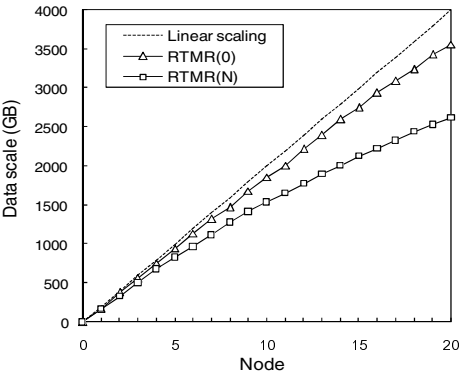
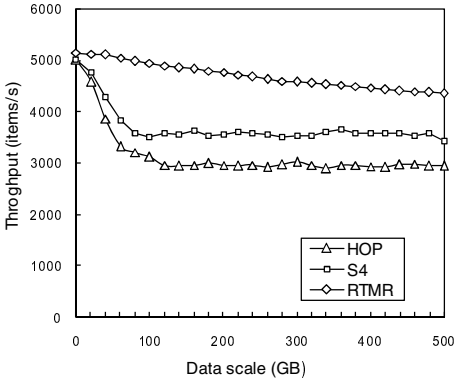



**Fig. 10.** Scalability analysis for history data



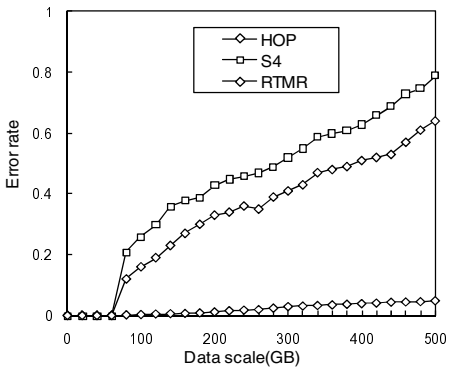




**Fig. 11.** Real-time capacity and Error rate

### 4.3 Real-Time Performance Analysis

Experiment 5 compares the real-time capacity of the architecture S4, HOP and RTMR(0). Due to the lack of preprocessing, S4 and HOP process history data repeatedly on each stream arrival. Hence, in order to compare data stream processing

capacity over large-scale data, preprocessing logic is inserted into the benchmark implementation of S4 and HOP. All architectures are set up on 2 nodes, the data stream is fixed at 5 MB/s. Fig.11 show that when the scale of intermediate results is less than 32 GB, the throughputs of HOP and S4 are fairly high because a single node can accommodate all the intermediate results, while the RTMR(0) is higher due to the utilization of staged pipeline. When the intermediate results are more than 32 GB and distributed to two node memory, the throughputs of HOP and S4 decrease rapidly because of increasing data transmission and synchronization overhead between nodes, while RTMR(0) is still very high owing to localization. When the intermediate results reach 64 GB, since the scale is beyond the cache capacity, throughputs of S4 and HOP are steady and the error rates increase with the data scaling up, whereas RTMR(0) can reduce the error rate and maintain a relatively high throughput due to the expansion of local intermediate result storage.

### 4.4 Related Work

Real-time improvement for MapReduce has become a research hotspot. Increment processing Percolator [10] and iteration processing Twister [11] and Spark [12] promote performance of large scale data processing in the way of random storage access and intermediate result cache. However, these methods are still batch based processing for static data increment. HOP [13] and S4 [14] extends the real-time processing ability of MapReduce by pipelining and distributed processing elements respectively, but they still do not focus on large scale history data, due to the lack of the support to preprocess history data and cache immediate results, and the mechanism to adaptively configure the cluster instead of relying on experience or experiment way.

## 5 Conclusions

The difficult of data stream processing over large scale historical data is guaranteeing both real-time capacity and scalability. And the contributions of this paper are:

- Improving the real-time data stream processing performance of MapReduce by caching, pipelining and localizing.
- Proposing a model analysis based RTMR cluster constructing method which can configure the Map/Reduce nodes and topologies adaptively according to application characteristics and network environments.
- Showing that RTMR(0) is practically effective to support data stream processing over large scale data in current IoT environment.

Programming model and cluster architecture is the basis of RTMR, and in addition, load skew is another key factor to restrict the scalability of the RTMR cluster. So the next work is to guarantee load balance of RTMR by static history data distribution and dynamic date stream load adaption.

**Acknowledgments.** This research has been funded by the National Natural Science Foundation of China under Grant No. 60903137, No. 61033006.

## References

1. Motwani, R., Widom, J., Arasu, A., et al.: Query processing, resource management, and approximation in a data stream management system. In: 1st Biennial Conference on Innovative Data Systems Research, pp. 176–187. ACM Press, New York (2003)
2. Abadi, D.J., Ahmad, Y., Balazinska, M., et al.: The design of the Borealis stream processing engine. In: 2nd Biennial Conference on Innovative Data Systems Research, pp. 277–289. ACM Press, New York (2005)
3. Jin, C.Q., Qian, W.N., Zhou, A.Y.: Analysis and management of streaming data: A survey. Journal of Software 15(8), 1172–1181 (2004)
4. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. ACM Communication 51(1), 107–113 (2008)
5. Ranger, C., Raghuraman, R., Penmetsa, A., et al.: Evaluating MapReduce for multi-core and multiprocessor systems. In: 13th International Conference on High Performance Computer Architecture, pp. 13–24. IEEE Computer Society, Washington (2007)
6. Kaashoek, F., Morris, R., Mao, Y.: Optimizing MapReduce for multicore architectures. Technical Report, MIT Computer Science and Artificial Intelligence Laboratory (2010)
7. Chang, F., Dean, J., Ghemawat, S., et al.: Bigtable: A distributed storage system for structured data. In: 7th Symposium on Operating Systems Design and Implementation, pp. 205–218. USENIX Association, Berkeley (2006)
8. Diao, Z.J., Zheng, H.D., Liu, J.Z., et al.: Operational Research. Higher Education Press, Beijing (2010)
9. Shah, M.A., Hellerstein, J.M., Chandrasekaran, S., et al.: Flux: An adaptive partitioning operator for continuous query systems. In: 19th International Conference on Data Engineering, pp. 25–36. IEEE Computer Society, Washington (2003)
10. Peng, D., Dabek, F.: Large-scale incremental processing using distributed transactions and notifications. In: 9th USENIX Symposium on Operating Systems Design and Implementation, pp. 251–264. USENIX Association, Berkeley (2010)
11. Ekanayake, J., Li, H., Zhang, B., et al.: Twister: A runtime for iterative MapReduce. In: 19th ACM International Symposium on High Performance Distributed Computing, pp. 810–818. ACM Press, New York (2010)
12. Zaharia, M., Chowdhury, N.M., Franklin, M., et al.: Spark: Cluster competing with working sets. In: 2nd USENIX Conference on Hot Topics in Cloud Computing, pp. 1–10. USENIX Association, Berkeley (2010)
13. Condie, T., Conway, N., Alvaro, P., et al.: MapReduce online. In: 7th USENIX Symposium on Networked Systems Design and Implementation, pp. 313–328. USENIX Association, Berkeley (2010)
14. Neumeyer, L., Robbins, L., Nair, A., et al.: S4: Distributed stream computing platform. In: 10th IEEE International Conference on Data Mining Workshops, pp. 170–177. IEEE Computer Society, Washington (2010)