Elastic Pipelining in an In-Memory Database Cluster

Li Wang[§‡], Minqi Zhou[§*], Zhenjie Zhang[‡], Yin Yang[⊥], Aoying Zhou[§], Dina Bitton[†]
[§]Institute for Data Science and Engineering, East China Normal University, China
[‡]Advanced Digital Sciences Center, Illinois at Singapore Pte. Ltd., Singapore
[⊥]College of Science and Engineering, Hamad Bin Khalifa University, Qatar
[†]Bitton Consulting, USA
{wang.li, zhenjie}@adsc.com.sg, {mqzhou,ayzhou}@sei.ecnu.edu.cn, yyang@qf.org.qa,
dinabitton@cs.wisc.edu

ABSTRACT

An in-memory database cluster consists of multiple interconnected nodes with a large capacity of RAM and modern multi-core CPUs. As a conventional query processing strategy, pipelining remains a promising solution for in-memory parallel database systems, as it avoids expensive intermediate result materialization and parallelizes the data processing among nodes. However, to fully unleash the power of pipelining in a cluster with multi-core nodes, it is crucial for the query optimizer to generate good query plans with appropriate intra-node parallelism, in order to maximize CPU and network bandwidth utilization. A suboptimal plan, on the contrary, causes load imbalance in the pipelines and consequently degrades the query performance. Parallelism assignment optimization at compile time is nearly impossible, as the workload in each node is affected by numerous factors and is highly dynamic during query evaluation. To tackle this problem, we propose *elastic pipelining*, which makes it possible to optimize intra-node parallelism assignments in the pipelines based on the actual workload at runtime. It is achieved with the adoption of new *elastic iterator model* and a fully optimized *dynamic scheduler*. The elastic iterator model generally upgrades traditional iterator model with new dynamic multi-core execution adjustment capability. And the dynamic scheduler efficiently provisions CPU cores to query execution segments in the pipelines based on the light-weight measurements on the operators. Extensive experiments on real and synthetic (TPC-H) data show that our proposal achieves almost full CPU utilization on typical decision-making analytical queries, outperforming state-of-the-art open-source systems by a huge margin.

1. INTRODUCTION

In-memory database achieves promising performance by keeping all tables and intermediate results entirely inside RAM, eliminating costly I/Os in traditional DBMSs. Query processing engines in the in-memory databases are designed to minimize the CPU cycles and memory access overhead in the query evalutaions. Such database systems were known to run well on high-end servers equipped with large RAM as well as newer-generation CPUs with many cores, large processor caches and super-scalar / SIMD capabilities. Although centralized in-memory databases are fast, the scalability is inherently limited by the amount of RAM and the number of cores within a single server, while in-memory parallel databases on a cluster of servers promise to overcome this limitation by distributed data storage and parallel data processing with multiple nodes [26]. Designing new query processing methods that fully utilize the power of a high-end server cluster, however, is highly challenging. Existing systems have focused on optimizing specific aspects of an in-memory cluster such as in-memory query optimizations [4, 8, 20, 23, 27, 30] and inter-node communication minimization [26]. The basic query processing framework still largely relies on existing solutions for traditional parallel databases.

We observe that traditional distributed query processing methods are designed based on very different assumptions, leading to potential waste on computation resource when used in an in-memory database cluster. Specifically, in these methods, a node usually materializes intermediate results before they are sent to other nodes over the network. However, in an in-memory database cluster, materialization is usually not considered as a good option due to its high memory consumption. Moreover, materialization increases query response time and incurs high memory access overhead by polluting the CPU cache [23].

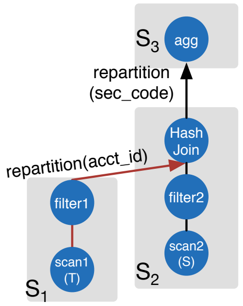
Pipelined execution [7] enabled by the iterator model [13], in which a node immediately sends its result tuples through network to its consumer node(s) for execution without concrete materialization, saves the overhead and memory usage in materialization. Pipelined execution also effectively reduces the query response time by parallelizing the data processing on the producer and consumer nodes and the network data transmission between them. However, pipelined execution is challenging, since it results in tight data synchronization between the query nodes in a pipeline. To fully unleash the power of pipelined execution on modern multi-core hardware, the system is expected to exploit the hardware resources, e.g., CPU cores and network bandwidth, and balance all the producer-consumer speeds in the pipelines. This is unfortunately nearly impossible under existing iterator model, due to the unpredictable and fluctuating workload on each node and the nature of static parallelism in the traditional iterator model. In the following, we illustrate the problems with a motivating example.

Figure 1 shows an example SQL query in a financial data warehouse system and its corresponding query plan. Assume that both tables are distributed across multiple nodes, and only S is partitioned based on the join attribute $acct\_id$. A popular join algorithm used in today's in-memory clusters is repartition join [6], with two phases: shuffle and local-join. Clearly, if materialization is adopted in query processing, huge amounts of intermediate results,

[*] Minqi Zhou is the corresponding author.

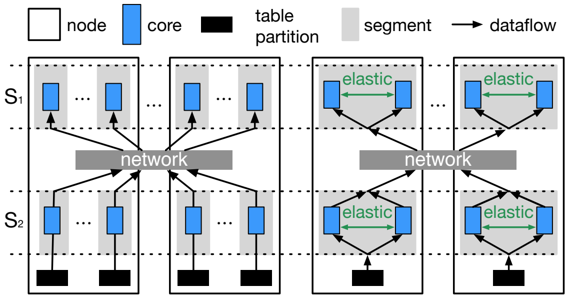
SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-3531-7/16/06…$15.00
DOI: http://dx.doi.org/10.1145/2882903.2882904

```
SELECT   sec_code, acct_id,
         sum(trade_volume),
         sum(entry_volume)
FROM     Trades T, Securities S
WHERE
T.trade_date = "2010-10-30" AND
S.entry_date = "2010-10-30" AND
T.acct_id = S.acct_id
GROUP BY T. sec_code, S.acct_id;
```



(a) Sql query (b) Query plan

Figure 1: An example SQL query in a financial warehouse system.



(a) Static pipeline (b) Elastic pipeline

Figure 2: Different parallelism mechanisms in traditional iterator model and elastic iterator model

e.g. shuffled partitions, are generated and kept in main memory. By employing pipelined execution, as shown in Figure 1(b), the SQL query is evaluated by an execution plan with two pipelines. The red arrow and the black arrow represent the two pipelines $P1$ and $P2$ respectively, which can be further decomposed into three segments, S_1, S_2 and S_3.

Let us zoom in on the first pipeline as shown in Figure 2(a). During its execution, both segment S_1 and segment S_2 are running simultaneously. In S_1, tuples from T satisfying $filter1$ are hash-partitioned on the join attribute and are sent to S_2. At the same time, S_2 receives the tuples from S_1 and constructs the hash table on-the-fly. One node in the in-memory database cluster could run multiple instances of S_1 and S_2 to achieve intra-node parallelism. As each segment instance works exclusively on a partitioned dataflow, this scheme results in a static pipeline, in which the intra-node parallelism must be determined at the compile time and hence cannot be modified on-the-fly. The main problem arises here: before query processing, the system must accurately estimate the throughput of the segments and generate query plan with proper intra-node parallelism in order to balance the workload in the pipeline and fully utilize the CPUs and network bandwidth. For example, too low parallelism of S_2 may lead to slow hash table construction, consequently blocking S_1 and under-utilizing CPU cycles and network bandwidth; conversely, too many cores on the instances of S_2 may exhaust S_1 or the network, leading to idle cores. Since the segment throughputs are affected by many factors such as operator selectivity, data distribution, implementation algorithms, and hardware features, it is very challenging for the query optimizer to foresee all those factors and generate query plans with optimal parallelism. Even worse, for a large dataset, the workload of each operator may fluctuate during runtime. In such case, the query processing with static parallelism can never be optimal. For instance, consider that tuples in T are roughly ordered by their insertion time, which correlates with attribute *entry_time*. The filter operator initially discards all tuples in T whose dates are before the selection criterion “2010-10-30", meaning no inputs for S_2. Accordingly, most cores should be given to S_1. Later on, the filter reaches a hotspot where almost all tuples match the selection criterion, causing a sudden burst of inputs for S_2. Thus, the core allocation should be dynamically adjusted to increase the speed of the join, and decrease that of the filter.

In this paper, we propose elastic pipelining (EP) to address the above challenges. The elastic pipelining consists of a novel elastic iterator model and a dynamic scheduler. The elastic iterator model enables query processing with dynamic intra-node parallelism as shown in Figure 2(b) and hence exempts the query optimizer from deciding the error-prone intra-node parallelism at compile time. The dynamic scheduler keeps tracks of the workload for each running segment and makes dynamic parallelism assignment to the segments in order to balance the workload in pipelines and fully leverage the available CPU cores and network bandwidth. Extensive experiments, using both a real financial dataset and synthetic TPC-H data, demonstrate that EP is highly effective: 5X faster than static pipelining. Meanwhile, our in-memory database cluster system implementing EP also outperforms Impala and Shark, two popular open-source distributed SQL query engines, by a huge margin. The main contributions of the paper are listed below:

1. We propose elastic pipelining to support online parallelism adjustment for query processing in the in-memory database cluster.
2. We introduce an elastic iterator model to add new elasticity functionalities to existing iterator model.
3. We design online scheduler algorithms to run lightweight resource scheduling on segments in the pipelines. uu
4. We evaluate the advantages of our proposal on both synthetic and real database workloads.

The rest of this paper is organized as following. Section 2 introduces the scenario we focus, shows the limitations of static pipelining and existing adaptive parallelism methods, and highlights the design philosophy of our elastic pipelining. Section 3 presents the elastic iterator model behind our elastic pipelining system. Section 4 discusses the dynamic scheduling strategy used in the system. Second 5 evaluates our proposal with experimental studies. Section 6 reviews the existing and related works in the literature, and Section 7 finally concludes the paper and highlight future works.

2. PRELIMINARIES

In this paper, we focus on query processing for in-memory parallel database systems running on clusters of servers with shared-nothing architecture, each of which is equipped with multicore processor(s) under CMP, SMP or NUMA architectures. The servers in the cluster, called *nodes* in the rest of the paper, are connected by local network. One node acts as the *master* to manage other *salve* nodes in the cluster. The master node is also responsible to query optimization and physical execution plan generation. The input tables are partitioned and distributed across the slave nodes, and entirely reside in main memory during query processing.

2.1 Iterator Model and Its Parallelism

In this part of the section, we introduce the core concepts in the traditional iterator model. We use the query plan in Figure 3 as the running example to explain these concepts.

Iterator: The iterator model [13] has been widely used in state-of-the-art database systems, because of its excellent generality and

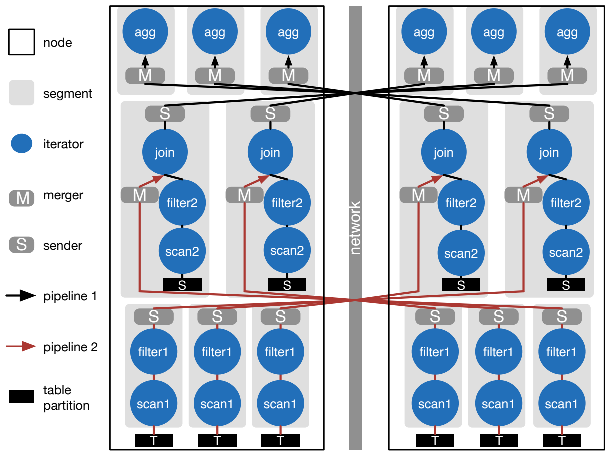


Figure 3: An example of iterator model and parallel execution

extensibility. All operators under iterator model follow a unified data-oriented interface, with basic functionalities of `open`, `next`, and `close` for input data manipulation. Every iterator is associated with a *state*, which maintains the runtime status and the important structure of the iterator. For hash join operator, for example, the state of the iterator maintains a hash table for the tuples from one of its input dataflow. For scan operator, for another example, the state of the iterator maintains a reference to the input file and the current reading cursor. The `open` function is responsible for state construction, such that the iterator is ready to emit output tuples when its `next` function is called. Some iterators, including hash join, aggregate and sort, are called *blocking iterators* or *pipeline breakers*, as they need to consume the entire dataflow on at least one input side to construct their state. Other iterators are called *non-blocking iterators*. The dataflow among iterators running on different nodes are connected by data exchange operator [13], which consists of *senders* and *mergers*. Senders and mergers are both non-blocking iterators.

Pipeline: Based on the concept of blocking iterators, an algebraic query plan is decomposed into pipelines, such that each pipeline forms a chain in the query plan tree and ends at certain pipeline breaker. In Figure 3, for example, there are two pipelines for the original query plan, including workflows as ($scan1$, $filer1$, $join$) and ($scan2$, $filter2$, $join$, agg) respectively.

Segment: To achieve *horizontal parallelism*, a fragment of a query plan can be duplicated into several identical *segment instances* or *segments* for short, each of which is responsible for a partition of the input dataflow for the pipeline and thus independently evaluated. The set of identical segments is called *segment group*. In Figure 3, for example, ($scan1$, $filter1$) has three identical segments on each node to achieve intra-node parallelism. Those segments receiving data from network are called *producer segments*, while those segments sending result tuples via network are called *consumer segments*. *Pipelined parallelism* is achieved by running the producer and the consumer segments simultaneously.

Stage: A pipeline may cover more than one segment. Within a pipeline, the set of iterators in the same segment is called a *local stage*, or *stage* for short. For instance, the iterators within the first pipeline in Figure 3 consists of two types of stages: $\{scan1$, $filter1$, $sender\}$ and $\{merger, join\}$. Similarity, a segment may belong to multiples pipeline and thus consists of several stages. When a segment is running, only one stage is active at a given time. In the active stage, the dataflow is release from the *stage beginner* and ends at the *stage ender*.

After the decomposition of the query execution plan on the segment level, each result segment is assigned to a node in the distributed system. As reported in [33], grouping tuples into blocks and block-at-a-time processing turns more efficient than tuple-at-a-time processing. In the rest of the paper, we assume the data blocks as the basic processing units in the system, although all of our proposals work well with tuple-at-a-time processing.

2.2 System Design Principles

In this part of section, we discuss the design principles for efficient in-memory query processing systems, especially from the perspective of desirable system features. In particular, we believe the following system features are important to maximize the computation utilization of the cluster and minimize the query response time.

DESIDERATA 1. ***Materialization-Free*** *property avoids any materialization of intermediate results during query processing.*

Materialization-free property allows the in-memory cluster to save unnecessary memory consumption and materialization overhead. This property is essentially important when a large amount of intermediate results are generated and transmitted across different nodes in the cluster.

DESIDERATA 2. ***CPU Efficiency*** *property minimizes CPU idle rate and maximizes query throughput per CPU cycle.*

In the in-memory database cluster, the response time for a given query largely depends on the CPU utilization and efficiency. When the CPUs are idle-free and processing the query in an effective manner, the query response time is minimized, especially when the queries are computation-intensive. This requirement is generally difficult to meet in distributed system, since it usually involves heavy synchronization among computation nodes in the cluster. As motivated in the introduction section, the workloads of querying processing logic is nearly unpredictable and varies at different phases, which leads to the third requirement below.

DESIDERATA 3. ***Self-Tuning*** *property allows the in-memory cluster to dynamically reallocate computation tasks based on the instantaneous workload of the system.*

Table 1: Comparisons on distributed processing techniques

Technique	Material.-Free	CPU Efficiency	Self-Tuning
[7, 19, 24]	✓	×	✓
[22]	✓	✓	×
Shark [31]	×	✓	✓
EP	✓	✓	✓

Table 1 classifies known methods according to the above desirable properties. [24] creates more query threads than the number of hardware cores and relies on the operating system to schedule them. However, this method keeps high CPU utilization at the expense of low CPU-efficiency, as it inevitably introduces expensive context switches and cache thrashing. [7, 19] decompose the queries into fine-grained, self-maintained executable units, each of which can be executed by a thread independently. Once a thread has processed one unit, it picks up another one based on a specific pickup algorithm to achieve desirable parallelism assignment. However, when using the methods to balance the workload in pipelines in our settings, the pickup algorithm will be much more complicated and time-consuming than its original design, leading to expensive scheduling overhead. [22] estimates the producing rate of operators, based on which the system allocates the optimal parallelism to achieve load balance. However, this method is not self-tuning. It leads to low CPU utilization when the workload within the pipelines is imbalanced due to the estimation errors or workload changes. In

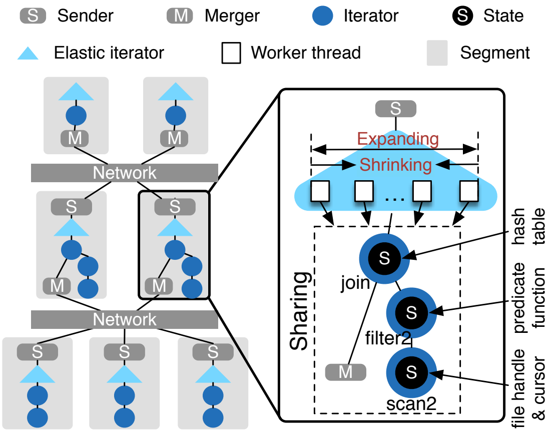


Figure 4: Overview of the elastic iterator model

Shark [31], materialization is run on the producer side to eliminate the necessity of synchronization among computation nodes, which simplifies the scheduling algorithm and improves the CPU utilization. However, materialization incurs high RAM consumption and inhibits pipelined parallelism.

As a conclusion of this section, to meet all these three requirements for distributed in-memory SQL query engine, it is essential to support *pipelining* with *dynamic parallelism* scheduling on the operators in the pipeline. In this paper, we will introduce *elastic pipelining* as a holistic solution.

2.3 Framework Overview of Elastic Pipelining

The system design principles in previous subsection motivate us to propose the framework of elastic pipelining. It consists of *elastic iterator model* and *dynamic scheduler*.

(1) The elastic iterator model is a set of implementation techniques to enable query processing with dynamic parallelism on the segment level. It generally updates the traditional iterator model with two functionalities *expand* and *shrink*, allowing the system to freely update the parallelism on each query segments on-the-fly. The elastic iterator model is introduced in Section 3.

(2) The runtime dynamic scheduler monitors the online status of all the running segments and makes decisions on parallelism update, in order to minimize the overall query response time and avoid computation resource waste. If one of the segments overproduces, i.e. generating tuples too fast for the network bandwidth or throughput of its producer/consumer segments, the scheduler calls the segment to shrink its parallelism. On the other hand, if a slow segment is identified as the performance bottleneck of the whole query, the scheduler calls the segment to expand parallelism, for the purpose of overall performance optimization. The dynamic scheduler is discussed in Section 4.

3. ELASTIC ITERATOR MODEL

In traditional iterator model, the input data flow is partitioned based on segments, such that each segment independently processes one of the partitions. Parallelism update on segment level is extremely difficult, since any insertion or removal of the segment involves state construction, destruction and migration across segments. In Figure 4, we present the internal structure of a segment from the query evaluation plan in Figure 3. Each segment is bound to an instance of iterator, within which the iterator state may contain a huge hash table for join operator, the predicate function for filter operator and file cursor for scan operator. If a new segment is created to increase the parallelism, the hash table must be redistributed among the segments accordingly. Such iterator state migration can be very time-consuming. It is thus more reasonable to support parallelism update within segment, by dynamically adjusting the number of worker threads within the segment.

An important feature of elastic iterator model is the state sharing mechanism among the worker threads. During state initialization, all worker threads collaboratively build the states, e.g. the hash table in Figure 4, in a unified memory space. Once the state construction completes, all worker threads can read the state when processing its own input data. Therefore, if a new worker thread is added into the elastic iterator, the initialization cost of the new thread is minimized. Similarly, when the system decides to decrease the parallelism by terminating a worker thread, the target thread can terminate quickly without iterator state migration. This feature enables the database query processing engine to revise the parallelisms of any running segment in milliseconds at runtime.

In the remainder of this section, we introduce the semantics of elastic iterator (Section 3.1) and optimizations used in the implementation of elastic iterator model on modern multi-core hardware (Section 3.2).

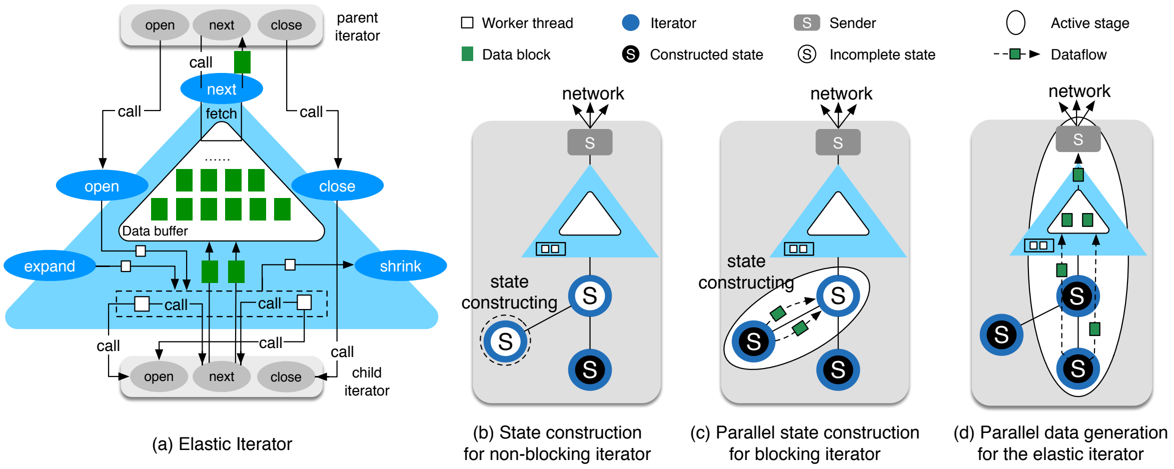
3.1 Semantics of Elastic Iterator

The elastic iterator is an extension to the conventional open-next-close protocol. In open-next-close protocol, all instances of iterators have the same data processing interface, by calling the `open`, `next` and `close` functions. There are two new functions in elastic iterator model, i.e. *expand* and *shrink*, to support runtime parallelism adjustment. This part of the section introduces the new semantics of the generic functions in the elastic iterator, to reflect the new support to elasticity.

Figure 5(a) illustrates the overall working mechanism of the elastic iterators. The execution of each segment is run by the corresponding elastic iterator, represented by a triangle in the segment. Generally speaking, the `open` function of the elastic iterator creates a number of worker threads for the segment and initializes a joint data buffer for all worker threads. The threads collaboratively process in parallel and insert the output blocks into the data buffer. The `next` function of the elastic iterator returns one data block from data buffer at a time, if there is any data block available; otherwise, the `next` function is blocked, until new data blocks arrive or *end-of-file* is reached. Data buffer connects the dataflow between the worker threads and the thread calling `next`, in an asynchronous way. Similar to the traditional iterator model, `close` function frees all the resource within the iterator and recursively calls `close` of its child iterators to terminate the entire iterator subtree.

The worker threads first call the `open` function of the child iterator to recursively construct the state of each iterator in parallel. Figure 5(b) and (c) demonstrate the process of state construction for a non-blocking iterator and a blocking iterator respectively. After the completion of state construction for all the iterators, the worker threads call the `next` function of the child iterator to retrieve an output data block at a time, and insert result data blocks into the data buffer, as demonstrated in Figure 5(d). To guarantee the completeness and accuracy of the query results, all the iterators support the thread-safe version of `open` and `next` functions. The pseudocode and revisions over traditional iterators for these functions are available in the appendix of the paper.

Before delving into the details of expansion and shrinkage operations, we first introduce the concepts regarding the status of the worker threads. Specifically, the worker threads in an active segment are in one of the following three statuses at any time. (S1)



(a) Elastic Iterator (b) State construction for non-blocking iterator (c) Parallel state construction for blocking iterator (d) Parallel data generation for the elastic iterator

Figure 5: Structure and work mechanism of the elastic iterator model

The worker thread is constructing the state of a non-blocking iterator's state as demonstrated in Figure 5(b). (S2) The worker thread is constructing the state of a blocking iterator's state as shown in Figure 5(c). (S3) The worker thread is processing the data for the elastic iterator's buffer as shown in Figure 5(d). The logics of the expansion and shrinkage operations rely on the status of the involved worker threads, as described below.

Expand: When expansion is called, the elastic iterator creates a new worker thread. As the new worker thread shares all the iterators' states with existing threads, it can immediately participate in the processing, either parallel state construction for the segment or parallel output data generation. To be specific, when the existing worker threads are in S1 or S2, the newly added worker thread joins the state construction work. For the hash table for join operator, for example, the new worker thread immediately reads the data and insert the tuples into the hash table based on the hash value of the keys. When existing worker threads are in S3, all worker threads are processing input data to generate output data blocks. In such case, the new worker thread simply calls the `next` function of its child iterator and inserts result data blocks to elastic iterator's buffer.

Shrink: When shrinkage is called, the elastic iterator chooses one of its worker threads to terminate. As the states of the iterators are shared by the worker threads, the terminating thread does not need to transfer the states to any other threads; instead, it only needs to guarantee that its termination does not introduce inconsistency to the states of the iterator. In particular, when the target worker thread is in S2 or S3, the data flow is fetched from the stage beginner (by calling the stage beginner's `next` function) and is fed into the state of the stage ender (or the buffer of the elastic iterator resp.). Therefore, once the thread has finished his current data block from the stage beginner, it can be terminated safely when the current data block is completely processed and the results are written to the state of the stage ender (or the buffer of the elastic iterator resp.). When the thread is in S1, it can terminate once the current state construction is completed. In our model, this termination strategy is achieved by adding checking operations at the beginning of each iterator's `open` function.

3.2 Implementation Optimizations

The implementation of the elastic iterators involves a variety of optimizations. Some of the techniques are borrowed from existing in-memory algorithms designed for multi-core processors, e.g. in-memory radix join [20], sort-merge join [3], NUMA-aware aggregation [30]. In the following, we list three novel methods, which are crucial for the performance of the parallel in-memory database.

(1) *Context Reuse*. Some iterators require each worker thread to maintain a private structure for auxiliary information, called *context*. In private/hybrid hash aggregation, for example, each worker thread contains a private hash table for partial aggregation results. When expansions and shrinkages are frequently called on these operators, the expensive reconstruction of these contexts is the nightmare for the parallel database system. To alleviate the potential problem, we do not destroy the context of the worker thread when it terminates. Instead, the context is kept, for the purpose of *context reuse*. When a new worker thread starts its operation, it first tries to find and reuse one of the existing contexts to skip the tedious context initialization. Each iterator is allowed to choose one from three different strategies of context maintenance, i.e. *void* mode, *processor* mode and *core* mode. In void mode, the locality is not considered and the available context can be reused by any worker thread. In processor mode, the worker thread can only reuse the contexts created by the cores on the same processor. This mode could potentially avoid remote memory access when reading the in-memory context in a NUMA architecture, if the context is still resident in the last level cache. In core mode, the worker thread can only reuse the context allocated by the same core, which provides opportunities of reusing the context resident in private CPU cache. For each iterator, the context maintenance strategy is selected according to the storage size of its context.

(2) *Order Preservation:* It is desirable for the elastic iterator model to have order preserving properties such that the tuple order in the dataflow produced by multiple threads is the same as that produced by a single thread. Without such property, elastic iterators cannot be used before any iterator that requires sorted data, limiting the usage of elastic iterator model. To do this, each block released from the stage beginner is given a unique sequence number according to its original order. After the data blocks are inserted into the elastic iterator's buffer, elastic iterator reorders them based on their sequence number, before they are later fetched by the parent iterator (typically the sender). The reordering is extremely fast, as it operates on the data blocks instead of tuples. However, it forces the iterator to be a pipeline breaker. To solve this problem, we implement stage beginner in such a way that data blocks taken by the same worker thread are in their original order. By doing this, the sequence numbers of data blocks inserted into the data buffer by the same worker threads are in increasing order. As a result, the elastic iterator only needs to merge the sorted data blocks generated

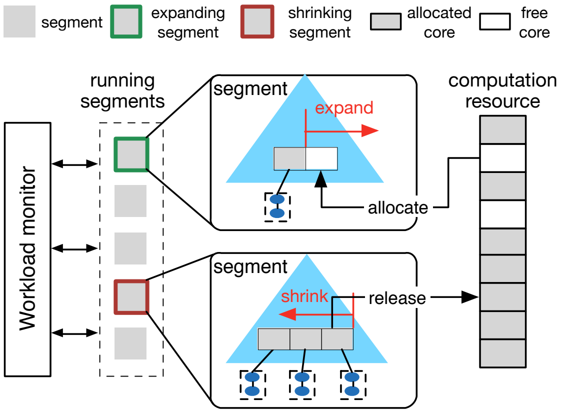


Figure 6: The interaction between segments and scheduler on a slave node.

Table 2: Table of Notations

notation	description
k	the number of nodes in the system
m	the number of CPU cores on a node
n	the number of segments
S_i	a segment
λ	the throughput of the pipeline
D_i	the service demand of S_i
V_i	the average visit rate of S_i
T_i	the processing rate of S_i
R_i	the normalized processing rate of S_i
δ_i	the selectivity of S_i
t_{ij}	the estimated processing rate of S_i with j worker threads

by different worker threads, which resolves the risk of becoming pipeline breaker.

(3) *NUMA Awareness:* In a NUMA architecture, it is more efficient to read data in local memory. Our elastic iterator model can easily achieve NUMA-awareness. When a worker thread asks for a data block from stage beginner for processing, the thread prefers to get a data block on the local memory, such that the subsequent data processing on that data block does not go through expensive remote memory access.

4. DYNAMIC SCHEDULER

In this section, we focus on the design and implementation of the dynamic scheduler, which is run on each slave node in the cluster to manage the runtime resource allocation. Figure 6 presents a concrete example of the scheduler on a slave node. The scheduler is run as an independent process on the slave node, on which it maintains a list structure for all active segments on the node. When a segment turns active, it is added to the end of the list, waiting for core assignment by the scheduler. The scheduler collects metrics from the threads and the corresponding segments, and keeps these statistical numbers in its own memory space. The scheduler evaluates possible performance improvement by core reassignment periodically. If the performance model identifies a better core assignment, the scheduler calls `shrink` on a segment to remove a core and calls `expand` on another segment to add the core. In the rest of the section, we will present the mathematical problem formulation of the scheduler (Section 4.1), introduce the basic optimization approach (Section 4.2), and discuss the metric collection procedure in the system (Section 4.3 and 4.4).

4.1 Problem Formulation

Assume that the parallel in-memory database system consists of k nodes, i.e. $\{N_1, N_2, \ldots, N_k\}$, each of which is equipped with m CPU cores. Given an executing plan for a query, there are a number of segment groups. Each segment group contains segments with identical processing logic, which are bound to different nodes for execution. We assume there are n concrete segments, denoted by $\mathbb{S} = \{S_1 \ldots, S_n\}$, running in all segment groups. The segment group without producer segments is called *input group*, while the segment group without consumer segments is called *output group*. The dataflow starts with the input group, flows into to its consumer segments based on certain partitioning scheme for execution. This process continues with the following segments, until the data flow reaches the output group. A segment is exclusively bound to a node for execution, based on execution plan for the query. Each node may host multiple segments. We use G_i to denote the segments bound to i-th node in the parallel system, such that $G_i \cap G_j = \emptyset$ for any $i \neq j$, and $\cup_i G_i = \mathbb{S}$.

The exact number of cores assigned to a particular segment S_i is called its *parallelism*, i.e. p_i for segment S_i. The throughput of the pipeline, denoted by $\lambda(p_1, \ldots, p_n)$ is measured as the aggregated processing rate of the segments in the input group, given the parallelism assignment $\{p_1, \ldots, p_n\}$. The goal of dynamic scheduling is to find an optimal core-to-segment assignment such that λ is maximized, under the constraint of valid core assignment to the segments, i.e.

$$\begin{aligned} \text{Maximize: } & \lambda(p_1, p_2, \ldots, p_n) \\ \text{s.t } & p_i \geq 0 \\ \forall G_j \ , & \sum_{S_i \in G_j} p_i \leq m \end{aligned} \tag{1}$$

Basically, the total number of cores assigned to local segments with each node is no larger than m. In the rest of the section, we will study the connection between λ and $\{p_1, \ldots, p_n\}$, as well as the solution to the optimization program above.

4.2 Scheduling Algorithm

According to [18] and [12], the active pipeline forms an open queuing system, such that the generation and processing of intermediate results in pipelines are modeled as queues for service. When there are n basic processing units running (segments in our setting) at the same time, regardless of the overall topology, the overall throughput of the system λ is determined by

$$\lambda = \frac{1}{\max_{1 \leq i \leq n} D_i} \tag{2}$$

In Equation (2), D_i is called *service demand* and can be calculated as $D_i = V_i / T_i$, where V_i and T_i are the *average visit rate* and the *processing rate* of segment S_i respectively. Specifically, the average visit rate is the average number of tuples received by segment S_i for each original input tuple at the input segment. The processing rate is the number of tuples the segment can process within a unit time. The details on the collections of these two types of metrics are available in Section 4.3 and Section 4.4.

Given the metrics $\{(S_i, V_i, T_i)\}$ available to the scheduler, by simple manipulation over Equation (2), we have:

$$\lambda = \min_{1 \leq i \leq n} \frac{T_i}{V_i} \tag{3}$$

In Equation 3, T_i/V_i is called the *normalized processing rate*, denoted by R_i, of the segment S_i, in which $1/V_i$ is also known as the normalization factor. Equation (3) shows that λ is purely decided by the bottleneck segment, i.e., the segment with the smallest

Algorithm 1: Scheduling Algorithm

```
Input: the overall throughput $\lambda$, the segment set $G$ on the
       node, penalty factor $\Delta$
Construct a segment set $U \subseteq G$ with normalized processing
  rate close to $\lambda$;
Construct a segment set $O = G - U$ with normalized
  processing rate significantly higher than $\lambda$;
Retrieve the parallelism of each segment $S_i \in G$;
Initialize an empty array $A$;
for $(S_i, S_j) \in U \times O$ do
    Evaluate the new normalized processing rates on $T'_i$ and
      $T'_j$ by Equation 4;
    if $T'_i \geq \lambda + \Delta$ and $T'_j \geq \lambda + \Delta$ then
        Insert $(S_i, S_j, T'_i, T'_j)$ in $A$;
if $A$ is not empty then
    Select a pair of $(S_i, S_j)$ with maximal throughput
      improvement;
    Adjust the parallelism for $S_i$ and $S_j$ by calling the
      functions in iterators;
```

normalized processing rate. This observation also simplifies the scheduling algorithm. It is unnecessary to conduct global parallelism assignment across nodes in the system. Instead, if all nodes are aware of the overall throughput λ of the whole system, they only need to run local optimization independently, simply to ensure the normalized processing rates of all the local segments are better than λ.

On the other hand, it is equally important to estimate the normalized processing rate under specified parallelism for each segment S_i. Fortunately, the average visit rate V_i also depends on the processing logic of precedent segments. It is thus sufficient to collect the statistics from the segments to generate accurate estimation on V_i. The processing rate T_i is obviously dependent of the parallelism p_i for segment S_i. The scheduler on a node maintains a scalability vector $(t_{i1}, t_{i2}, \ldots, t_{im})$, each entry t_{ij} in which is an estimation on the processing rate T_i on segment S_i when there are j cores assigned to S_i. We leave the discussions on the generation of these statistics to the following subsections.

Given all the statistics available to the scheduler, the pseudo codes of the scheduling algorithm are listed in Algorithm 1. Given the global throughput λ, the scheduler firstly partitions all its segments into two parts. The first segment set U contains underperforming segments with normalized processing rate close to λ, while the second segment set O contains over-performing segments with normalized processing rate significantly larger than λ. The algorithm then tests for all paris of $(S_i, S_j) \in U \times O$. For each (S_i, S_j), it evaluates the potential improvement to the throughput by moving one worker thread from S_j to S_i. The evaluation is based on the estimation of new normalized processing throughputs on S_i and S_j that

$$
\begin{aligned}
T'_i &= \frac{t_{i(p_i+1)}}{V_i} \\
T'_j &= \frac{t_{j(p_j-1)}}{V_j}
\end{aligned}
\tag{4}
$$

After enumerating all possible paris, the algorithm finally selects the best pair with the largest potential improvement.

4.3 Measurements on Average Visit Rate

In general, average visit rate is propagated through the pipeline along with the dataflow, from one segment to another. Apparently, the average visit rate for any segment in the input group is 1. Consequently, if we are aware of how the values are propagated among the segments, then we are able to obtain the average visit rate for each segment in the pipeline. In the following, we first discuss the method to calculate the average visit rate in the simplified cases where each segment group has only one segment, and then extend the method to the general ones.

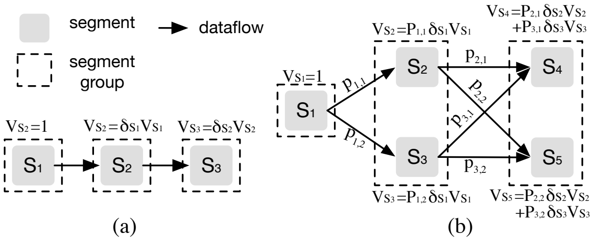


Figure 7: Propagation of average visit rates in the dataflow

Consider a pipeline as demonstrated in Figure 7(a), in which each segment group has only one segment. There is no partitioning on the dataflow; the output dataflow generated by a segment is entirely sent to one consumer segment for execution. Consequently, for any segment S_i, the average visit rate of its consumer segment is $\delta_i \cdot V_i$, in which δ_i is the selectivity of S_i, i.e., the ratio of the number of output tuples to the number of input tuples consumed.

Now we extend our method to the general cases, in which a segment group might contain several segments as shown in Figure 7(b). For a segment S_i with multiple consumer segments, the output dataflow is partitioned based on a certain partitioning scheme, e.g., hash partition. The contribution of average visit rate that S_i makes to its j-th consumer segment is $p_j \cdot \delta_i V_i$, where p_j is the percentage of output dataflow S_i sends to its j^{th} consumer segment. Similarity, for a segment S_j with multiple producer segments, its average visit is the sum of the average visit rates that all its producer segments contribute to it.

In our implementation, in order to obtain the average visit rates without introducing explicit communication among segments and nodes, we integrate instantaneous values of average visit rates in the dataflow. In particular, in the tail of each data block, we maintain the value of average visit rate for the tuples within the data block. The value is updated when the data block is processed by each segment or partitioned across the network. For a segment, it is kept updated to its instantaneous value of the average visit rate by reading tail information of its input data blocks.

4.4 Measurements on Processing Rate

The prediction of throughput plays an important role in the optimization procedure called by the scheduler. However, accurate prediction is very challenging, as the throughput of a segment is affected by numerous factors and is dynamic during the query execution. In this subsection, we present our method to achieve accurate, scalable estimation on the throughput with minimal efforts.

Our system maintains a data structure, called *scalability vector*, in the memory of the nodes. For each segment S_i, there is a vector, with the j-th entry in the vector as a value pair (t_{ij}, l_{ij}), in which t_{ij} is the measured instantaneous processing rate of S_i with j parallel worker threads and l_{ij} is the timestamp when the latest metric is recorded. During the execution of S_i, we periodically detect the instantaneous data processing rate and update the corresponding entry in the scalability vector, if the measured data processing rate is not under-estimated, i.e., the processing rate of S_i is not limited by that of other segments or the network bandwidth. As the scalability for a segment varies when running different stages, before the start of the execution of a stage, each entry in the scalability vector is initialized to be invalid, indicating that the actual value is not known.

At system time l, when estimating the processing rate of a segment S_i with a target parallelism p_i, the system checks the p_i-the entry in the scalability vector. If t_{ip_i} is valid and l_{ip_i} is a fresh record, i.e. $\|l_{ip_i} - l\| \leq \theta$ under the specified threshold θ, the system directly uses t_{ip_i} as the estimation of the processing rate. Otherwise, there is no existing record on the processing rate for direct estimation use. The system then uses the neighbor record in the scalability vector, as the current parallelism for S_i is either $p_i - 1$ or $p_i + 1$ before the scheduler tries a new round of optimization. The estimation on the processing rate is simply proportional to the number of cores, based on either $(p_i + 1)$-th record or $(p_i - 1)$-th record in the scalability vector. Note that such estimation may not be accurate but does not incur much assignment problem, as the scheduler only reassigns one core each time and inaccurate estimation could be quickly identified and corrected when next round of optimization comes into operation.

5. EXPERIMENT

5.1 Environment & Dataset

We implement the framework of elastic pipelining in CLAIMS [1], our in-memory parallel database prototype, tailored for efficient data analysis on clusters of shared-nothing multi-core servers. The system accepts SQL queries and employs an internal query optimizer to convert queries into physical execution plans with minimized cost. We employ block-at-a-time processing strategy in our elastic iterator model, with a block size of 64 KB to fit the L2 cache size on CPUs.

All the experiments run on a cluster with 10 identical nodes. The detailed hardware specifications of the nodes are listed in Table 3. The servers are inter-connected by a Gigabyte switch. The operating system is Redhat Enterprise 6.3 with kernel version 2.6.32.

Table 3: Hardware specifications of the server

Number of sockets	2
Number of physical (logical) Cores	12 (24)
Size of DRAM per NUMA socket	32 GB
L1 / L2 cache size (private)	32 / 256 KB
L3 cache size (shared within a NUMA socket)	15 MB

The evaluations are done over a synthetic dataset and a real dataset. The synthetic dataset is generated with TPC-H benchmark with scale factor at 100. The real-life dataset contains transactional records in Stock Exchange (SSE) in three months of year 2010. These transactional records are stored in two tables under schemas: Securities(*order_no*, *acct_id*, *sec_code*, *entry_date*, *entry_volume*) and Trades(*acct_id*, *sec_code*, *trade_date*, *trade_time*, *order_price*, *trade_volume*). Each table contains more than 840 million records. Without otherwise specification, all the tables in both datasets are hash-partitioned and kept on 10 nodes based on their primary key(s). Besides the standard TPC-H queries, we also test with another 5 synthetic queries, i.e. (S-Q1 - S-Q5), for TPC-H dataset, as well as 4 real queries (i.e., SSE-Q6 - SSE-Q9) commonly used on Stock Exchange dataset, to better demonstrate the properties of our proposals:

```
S-Q1: SELECT * FROM orders
      WHERE o_comment not like "%[word1]%[word2]";
S-Q2: SELECT * FROM orders
      WHERE o_orderdate < "[date]";
S-Q3: SELECT l_returnflag, l_linestatus,
             sum(l_quantity), avg(l_discount)
      FROM lineitem
      GROUP BY l_returnflag, l_linestatus;
S-Q4: SELECT l_commitdate, sum(l_quantity),
             avg(l_discount)
      FROM lineitem
      GROUP BY l_commitdate;
S-Q5: SELECT * FROM orders, lineitem
      WHERE l_orderkey=o_orderkey;
SSE-Q6: SELECT count(*)
        FROM  Trades T, Securities S
        WHERE S.sec_code = 600036  AND
        T.trade_date = "2010-10-30" AND
        S.acct_id = T.acct_id;
SSE-Q7: SELECT acct_id, sum(trade_volume)
        FROM Trades
        GROUP BY acct_id;
SSE-Q8: SELECT acct_id, sec_code sum(trade_volume)
        FROM Trades
        WHERE trade_date = "2010-10-10"
        GROUP BY acct_id, sec_code;
SSE-Q9: SELECT sec_code, acct_id,sum(trade_volume)
               sum(entry_volume)
        FROM Trades T, Securities S
        WHERE T.trade_date = "2010-10-30" AND
        S.entry_date= "2010-10-30" AND
        T.acct_id = S.acct_id
        GROUP BY T.sec_code, S.acct_id;
```

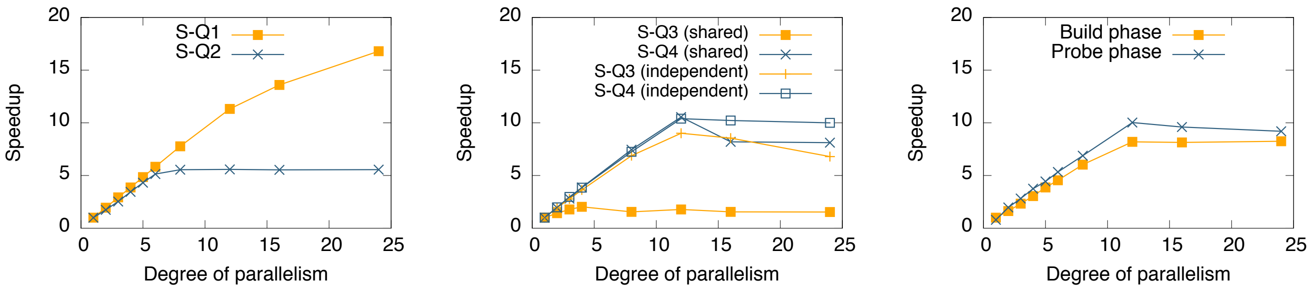
5.2 Elastic iterator model

Scalability: To evaluate the scalability of intra-segment parallelism enabled by our elastic iterator model, we run the synthetic queries (i.e., S-Q1 - S-Q5) on the TPC-H dataset and present the results in Figure 8. S-Q1 and S-Q2 filter tuples over the *orders* table. S-Q1 is computation-intensive because of the LIKE operator, while S-Q2 is data-intensive. S-Q3 and S-Q4 contain aggregations with group-by cardinalities of 4 and 250 million respectively in terms of the generated TPC-H dataset. S-Q5 evaluates equal join between two large tables. To evaluate the real scalability of a segment regardless of the network communication or workloads of other segments, we run this micro-benchmark over a non-partitioned TPC-H dataset on a single node, such that the query plan for each of the 5 queries contains only one segment .

Figure 8(a) illustrates the scalability of the filter operator (S-Q1 and S-Q2). For computation-intensive workload, e.g. S-Q1, the throughput of the filter operator increases almost linearly. In contrast, for data-intensive queries, e.g. S-Q2, the performance is no longer improved when the parallelism exceeds 8, due to the memory bandwidth limitation. Figure 8(b) presents the scalability of the hash aggregation operator (S-Q3 and S-Q4). S-Q3 and S-Q4 are evaluated under both shared-aggregation and independent aggregation algorithms, respectively. Due to the hash table contention caused by small group-by cardinality, S-Q3 scales poorly in shared aggregation algorithm. In contrast, as hash table contention rarely happens, S-Q3 with independent aggregation and S-Q4 with both algorithms scale well when hyper-threading is not involved. Figure 8(c) shows the scalability of join operator (S-Q5). As hash join is mainly data-intensive and lock-free structures are employed in hash table to avoid the latching cost, the query scales well both in building phase and probing phase without hyper-threading.

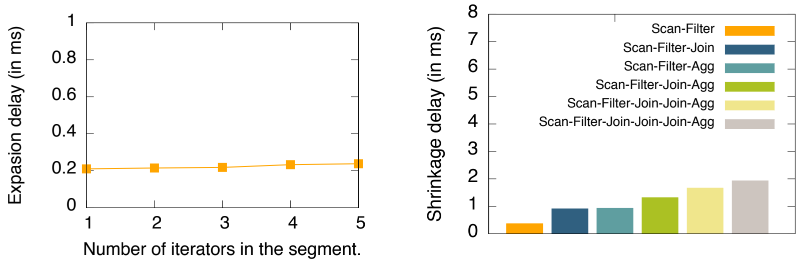
The above results confirm the good scalability of our elastic iterator model when the algorithms are properly configured. The results also show that the optimal parallelism for a given query relies on the query workload, data distribution and underlying algorithm implementation. They further imply the high risk of suboptimal query plan with poor thread parallelism chosen at compile time.

Overhead of expansion and shrinkage: We then evaluate the delay caused by expansion and shrinkage functions. The overhead determines the efficiency of parallelism adjustment and response to workload variance. The overhead of expansion is the time duration between worker thread creation and its beginning of data processing. The overhead of shrinkage is the time duration between the termination request on a worker thread and the complete termination of the thread. Figure 9(a) shows the average overhead with vary-



(a) *filter* operator (b) *hash aggregation* operator (c) *hash join* operator

Figure 8: Scalability of intra-segment parallelism of the elastic iterator model



(a) overhead of expansion (b) overhead of shrinkage

Figure 9: The efficiency of expansion and shrinkage operations

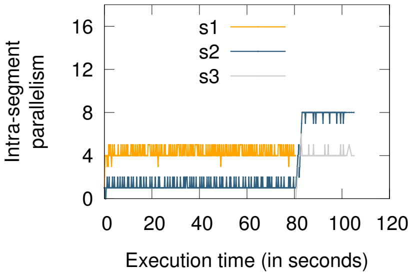


Figure 10: Parallelism dynamics of elastic pipelining on SSE-Q9

ing numbers of running iterators in the segments. As expected, the overhead of expansion is negligible and only slightly affected by the number of iterators in the segment. That is because the newly created thread does not need to process any data before catching up with other existing ones. In contrast, the overhead of shrinkage shown in Figure 9(b) depends on the number of iterators in the segment and their types. Before termination, the terminating thread must ensure its corresponding data block are finished by all the iterators in the active stage. Consequently, the overhead of shrinkage grows when the termination target is more time-consuming operators (e.g. join and aggregation) or a larger number of iterators are active in the segment. Nevertheless, the overhead of shrinkage is sufficiently small to support high flexibility of online parallelism adjustment.

5.3 Dynamic Scheduler

In this part of experiments, we show how our dynamic scheduler adaptively updates the parallelism for segments by an in-depth case study on SSE-Q9, which is daily issued to generate reports for Stock Exchange database. Table T is partitioned across 10 nodes on attribute “sec_code", while table S is partitioned on attribute “acct_id". As the equal hash join is run on attribute “acct_id", the dataflow of T has to be repartitioned on “acct_id" before joining with S. The query plan is generated after query optimization, as shown in Figure 1(b). In this query plan, there are two pipelines $P1$ and $P2$, and three segments S_1, S_2 and S_3, each of which runs on all of the 10 nodes. Without specific clarification, the initial intra-segment parallelism is set to 1.

Dynamic intra-segment parallelism: We keep track of the intra-segment parallelism for each stage on a randomly chosen node and show the results in Figure 10. When pipeline $P1$ begins to execute, S_1 is the performance bottleneck of this query due to the low selectivity of $filter1$. The scheduler detects such workload on-the-fly and lets S_1 expand to accelerate the data processing. As a consequence, the throughput of S_1 is improved and the new performance bottleneck moves to S_2, which will be expanded by the scheduler sequentially. As the throughput of data processing in S_1 and S_2 increases, the network communication between S_1 and S_2 finally becomes the performance bottleneck which prevents S_1 and S_2 from benefiting from further parallelism growth. The scheduler detects the network bottleneck and keeps S_1 and S_2 around the appropriate parallelism, such that the data processing rate of the two segments matches the network bandwidth. After the hash table construction, the query steps into pipeline $P2$. The scheduler quickly detects the workload on S_2 and S_3 and adjusts their parallelism for high CPU-efficiency and excellent load balancing.

Resilient to workload variance: To evaluate the adaptivity of our dynamic scheduler, we reorder the tuples in each partition of $Trade$ on attribute “trade_date" in ascending order, to simulate fluctuating selectivity on $filter1$ during query processing. The results for SSE-Q9 are shown in Figure 11. At the beginning of the query evaluation, the selectivity of $filter1$ is 0, as the dates of the data are all way before the date “2010-10-30" in filter condition. The scheduler detects that S_1 is under-producing and expands S_1. It also shrinks segment S_2 as there is no input data available for processing. As the query evaluation goes on, the selectivity of $filter1$ jumps to 1 when the tuples on “2010-10-30" are coming into the system. At this moment, S_1 turns over-producing due to the large selectivity of $filter1$, and is requested to shrink by the scheduler to ensure a proper producing rate matching the network bandwidth. Moreover, the scheduler starts to expand the “hibernating” segment S_2, as huge amount of tuples are flooding into S_2.

During the query evaluation, we run a CPU-intensive program periodically on each of the nodes to slow down the overall data processing throughput. We run SSE-Q9 again and monitor how our dynamic scheduler acts in such dynamic processing throughput. To better understand the behavior of the scheduler in response to workload variance, we let the program sleep 20 seconds for every 40 seconds. The parallelism of each segment during the query on a randomly chosen node is shown in Figure 12. When the inference program turns active, the scheduler immediately detects the low throughput of the segments and ask them to shrink for better CPU

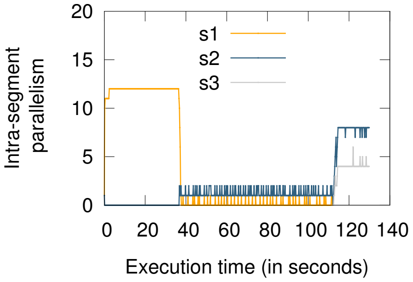


Figure 11: The adaptivity of dynamic scheduler to the selectivity fluctuation

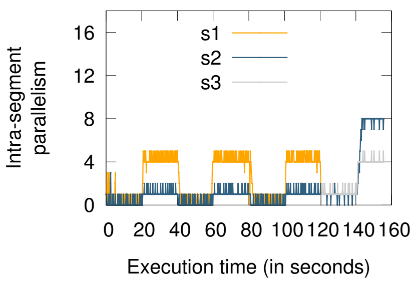


Figure 12: The adaptivity of dynamic scheduler to the interfering program

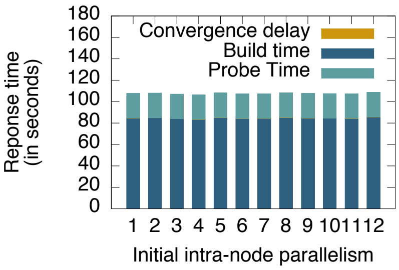


Figure 13: The robustness to the initial parallelism assignment

Table 4: Comparison on memory consumption (in GB)

	SSE-Q6	SSE-Q7	SSE-Q8	SSE-Q9
EP	3.56	1.41	2.0	2.3
SP	3.7	1.4	2.2	2.4
ME	23.4	4.2	11.3	16.5

utilization. In contrast, whenever the inference program pauses, the dynamic scheduler immediately identifies the potentials on the scalability of the segments, and expands them accordingly to leverage computation resource of CPUs.

Fast convergence and robustness to initial parallelism: To confirm the fast convergence of our dynamic scheduler in refining the parallelism and the robustness to the initial parallelism assignment specified at the beginning of query processing, we run SSE-Q9 with different initial intra-segment parallelism and report the complete query response time and convergence delays in Figure 13. The results show that as the query begins to evaluate or steps into a new stage, our scheduler can respond to the varying workload and readjusts to the optimal parallelism with a very short convergence delay. The results also demonstrate that the initial intra-segment parallelism does not obviously affect the query response time, which confirms the self-tunning feature of our elastic pipelining.

5.4 End-to-End evaluation

Comparison against static pipelining and materialized execution: The superiority of our elastic pipelining (EP) is verified by comparing with both static pipelining (SP) and materialized execution (ME). In ME, all the tuples generated by a segment are fully materialized before they are sent to the consumer segment(s) via network. Table 4 presents the memory consumption during query processing under the three processing frameworks. Due to space limitations, we only give the results on SSE dataset. As expected, ME consumes much more RAM compared with pipelined execution due to the materialization.

Table 7 shows the query response time of various queries under these three settings, respectively. As the query response time in SP and ME is significantly affected by the parallelism assignment, its efficiency is underestimated if the query optimizer fails to generate a query plan with the optimal parallelism. To avoid such underestimation, we manually register 10 different constant parallelism for each query and report only the best result as the strawman to our EP approach. In the parallelism assignment, we try to assign each segment with the optimal parallelism such that the segment is allocated with proper cores to accelerate data processing and to match the data processing rate of its downside/upside segments. The results show that SP outperforms ME, with benefits from pipelined parallelism among segments and the skip of materialization. However, the performance improvement by pipelined parallelism is no larger than 20% in most queries. That is because the imbalanced generating rates among consumer segments and producer segments in static pipelining severely discount the benefits of pipelining. In contrast, parallelism of each segment in elastic pipelining is dynamically adapted to the runtime workload for load balance within pipelines, which enhances the utilization of CPU cores and the network bandwidth. Consequently, elastic pipelining outperforms static pipelining by a huge margin.

Comparison to adaptive parallelism scheduling methods: We compare our elastic pipelining (EP) with two existing parallelism scheduling methods, namely implicit scheduling (IS) [24] and morsel-driven-parallelism (MDP) [19]. As the codes of their systems used in their experiments are not publicly available, we implement IS and MDP in our own system. In our IS implementation, each query plan consists of multiple segments on each node. One segment is bound to one thread for execution and the operating system is responsible for scheduling on the worker threads. In our MDP implementation, query segments are decomposed into fine-grained executable units, which are inserted to a waiting list and can be executed concurrently. A set of worker threads are created to process the executable units. Once a worker thread has processed a unit, it randomly picks up the next available one. We also implemented MDP$^+$, an enhanced version over MDP, in which each worker thread picks up the next unit based on our dynamic scheduling strategy that considers the scalability of the segments and the load balance among nodes. The aim of MDP$^+$ is to show how we can improve the performance of MDP by using a better scheduling strategy tailored to the in-memory database cluster. For MDP, we test a variety of executable unit sizes to find the optimal unit size at 64KB. We thus use 64KB as the executable unit size in our comparisons. To avoid the CPU under-utilization and reduce context switch cost, we test various concurrency levels for IS, MDP and MDP$^+$ respectively, as suggested in [24]. Concurrency level, denoted as c, is the ratio of worker threads to the number of hardware cores. We run all the SSE and TPC-H queries with these four methods respectively, and show the average response times under each method in Table 5 of all queries. To get a better perspective of the performance variances, we also report the average CPU utilization rate, cache miss ratio, context switches and scheduling overhead. CPU utilization rate is measured on the cores allocated to the query threads. Cache miss ratio is the average of L1, L2 and L3 data cache miss ratio, measured by Intel PCM. Context switch counts are obtained from the linux process profile. For MDP, MDP$^+$ and EP, the scheduling overhead is measured as the ratio of the aggregate scheduling CPU time to the query response time. As IS is scheduled by the OS, the scheduling overhead is not available.

The results show that IS and MDP perform poorly when c=1 (one thread per core), due to the low CPU utilization. Using more worker threads, e.g., c=5, helps to keep CPU busy, because if one thread is blocked due to imbalanced workload or inefficient network bandwidth, the thread can yield its core to another waiting

Table 5: Comparison with three baseline scheduling methods

	IS			MDP			MDP$^+$(unit size=64K)			MDP$^+$(unit size=8K)			EP
	c=1	c=2	c=5	c=1	c=2	c=5	c=1	c=2	c=5	c=1	c=2	c=5	c=1
CPU utilization rate(%)	27.0	53.5	88.2	36.2	66.9	93.5	59.6	76.4	97.3	87.2	99.5	99.6	99.6
context switches / second (×1000)	0.2	8.3	18.0	0.2	3.0	10.0	0.1	2.9	6.9	0.1	2.6	6.2	0.2
scheduling overhead (%)	n/a	n/a	n/a	0.21	0.92	1.54	0.49	1.27	2.82	3.36	6.22	17.41	0.20
cache miss ratio	0.41	0.52	0.75	0.42	0.54	0.76	0.46	0.52	0.75	0.21	0.56	0.73	0.41
response time (s)	202.3	144.7	123.4	172.2	131.4	120.7	138.1	121.3	111.9	145.4	131.1	122.7	54.3

thread. Unfortunately, using more worker threads may bring more context switches and hence ruins cache locality. Consequently, the benefits of higher CPU utilization brought by using more worker threads are discounted severely. Compared with IS and MDP, MDP$^+$ has better CPU utilization and better performance. That is because MDP$^+$ is equipped with our scheduling algorithm, which is capable of reassigning the cores from blocked segments to under-producing segments. It is interesting to find that MDP$^+$ always outperforms the other two methods, when IS, MDP and MDP$^+$ have the similar CPU utilization. Such phenomenon is due to our scheduling policy on MDP$^+$ with consideration on the scalability of segments. When a free core is available for multiple segments, the scheduling algorithm prefers to assign the core to the most scalable segment. Note that in MDP$^+$, if a thread is blocked by the network I/O, it cannot switch to another unit until the current unit is completely processed. As a result, a large unit size will cause delays in parallelism adjustment, which explains the relatively low CPU utilization for a large unit size. Using a smaller unit size could effectively improve the CPU utilization. However, as shown in Table 5, smaller unit size introduces expensive scheduling overhead, limiting the query performance. Although MDP$^+$ and EP use our dynamic scheduling strategy, EP is superior on cache locality and scheduling overhead, and thus significantly cuts the query response time. In our EP approach, CPU cores are always focusing on the data processing in their assigned segments until they are notified by the scheduler to migrate, which helps to retain good cache locality. Instead of running the time-consuming scheduling algorithm for each individual worker thread to make decision on a subsequent processing unit, our scheduling strategy in EP is called at much lower frequency while remains sensitive to workload change. This guarantees instantaneous response to changes on query processing with minimal scheduling overhead.

From another perspective, a key advantage of EP over baseline methods is that it always tries to improve the hardware utilization by revising the parallelism assignment according to the runtime workload, until the executing query reaches the limitation of the available hardware. To verify this claims, we divide the query execution time into slices and monitor the hardware utilization for each time slice. A time slices is called *high-utilized* if its average utilization of either CPU or network reaches a threshold, e.g., θ_u=0.95. The high utilization rate of a query is the percentage of high-utilized time slices. Table 6 presents the high utilization rates and the response times for the queries. Due to space limitation, we only show the results for TPC-H-Q1, TPC-H-Q9 and TPC-H-Q14, which respectively represent computing-intensive queries, network-intensive queries and mixed-workload queries. The results show that EP can always fully leverage the hardware resource, which partially explains the superior performance of EP over other methods.

Comparison to state-of-the-art systems: Finally, we compare the performance of elastic query processing engine with Impala and Shark, two state-of-the-art distributed SQL query engines in the open source community. For Impala and Shark, Parquet is used as the table format, which is reported to be the best option [11]. For sake of fair comparison on these systems, we carefully configure the setup in Impala and Shark to avoid disk I/O during the query evaluations. In particular, HDFS cache is enabled and all the tables are read before conducting comparison so that Impala and Shark can query over the cached data. Note that although Impala writes query results onto disk, this does not noticeably increase the query response time due to the very small output results size (smaller than 100KB) for the queries evaluated in our experiments. Also, we ensure that the intermediate results materialization in Shark is buffered by operating system cache with minimal disk I/Os, which is verified by monitoring Linux Process profile. Table 7 shows the query response times for a variety of queries on these three systems. Due to the limitation of our query engine, some TPC-H queries are not supported in CLAIMS and hence the results for those queries are not shown in the table. Largely due to the efficient data processing features in Impala, such as code generation, SIMD instructions etc., it outperforms Shark by a huge margin. However, with the huge benefits from scalable intra-segment parallelism as well as the sophisticated parallelism assignment enabled by elastic pipelining, CLAIMS is consistently 5 times faster than Impala in most cases.

Table 6: Comparison with baselines on hardware utilization

	high utilization rate (%)			Query response time (s)		
Queries	IS	MDP	EP	IS	MDP	EP
TPC-H-Q1	28.9	21.3	96.5	19.5	9.6	4.8
TPC-H-Q9	38.6	33.2	95.4	256.4	203.7	105.1
TPC-H-Q14	19.3	18.4	89.2	112.3	87.3	20.6

We hereby emphasize that, although the framework of elastic pipelining is proven to be efficient, it is possible to improve the performance by including other orthogonal optimization techniques. Firstly, since network data transmission is the major bottleneck in our settings, significant performance enhancement is expected by employing high-speed interconnect devices, such as InfiniBand. Second, we can further enhance the query performance of CLAIMS by integrating code generation features into the query engine. For instance, our recent investigations on our system show that filter operators gain a speedup by up to two orders of magnitude by simply employing LLVM [17] to generate query-specific code.

6. RELATED WORK

Centralized In-Memory Database. By maintaining data and intermediate results completely in main memory, in-memory databases employ new techniques to resolve new bottlenecks in query processing and exploit multi-core architecture and memory hierarchy for performance improvement. The existing studies can be generally classified into three categories, based on their targets and assumptions on the systems.

The first category covers research attempts with better data placement and organization in main memory. [10] shows column store is helpful to alleviate the memory access bottleneck. Light-weight data compression is introduced in [25] to reduce the data access overhead. The second category includes research works trying to improve cache behavior of database operators. Manegold et al. [21] propose radix join algorithm to accelerate in-memory equal join. Wang et al. [30] validates the possibility of improvement for

Table 7: Response time (in seconds) of various queries in CLAIMS, Shark and Impala

	CLAIMS				
Queries	ME	SP	EP	Shark	Impala
TPC-H-Q1	38.3	35.0	4.8	102.7	35.1
TPC-H-Q2	25.4	28.4	5.4	76.5	28.4
TPC-H-Q3	172.3	76.6	28.0	231.0	150.9
TPC-H-Q5	323.2	200.9	68.8	412.9	321.0
TPC-H-Q6	62.5	58.7	15.1	120.2	46.0
TPC-H-Q7	223.1	203.3	62.3	325.3	243.3
TPC-H-Q8	312.7	284.1	58.5	421.2	256.2
TPC-H-Q9	398.8	363.2	105.1	512.6	410.4
TPC-H-Q10	188.0	175.0	65.0	342.4	183.0
TPC-H-Q12	91.2	85.7	16.7	78.5	43.2
TPC-H-Q14	94.5	61.1	20.6	130.3	78.3
SSE-Q6	38.0	33.9	10.3	56.6	35.2
SSE-Q7	25.2	21.1	3.5	48.2	27.9
SSE-Q8	26.3	20.34	2.4	35.3	20.5
SSE-Q9	58.3	53.2	28.4	203.7	147.4

in-memory database by running aggregation queries under NUMA architecture. Albutiu et al. [3] propose extensive optimizations to achieve scalable sort-merge join in NUMA architectures. [15] revisits the parallel sort-merge join and hash join run over modem hardware. [5] further exploits the parallel join algorithms and shows the competitiveness of non-partitioning join. The third category contains research studies on data processing model and sophisticated compilation techniques. MonetDB and Vectorwise prove the power of column-at-a-time and block-at-at-a-time data processing model respectively to improve the code efficiency. Krikellas et al. show that query performance can be significantly improved by generating query-specific C code [16]. HyPer [23] and Impala [2] leverage LLVM [17] to generate specific, optimized code for a given query.

All these studies above are designed for centralized in-memory database running on one single machine. We hereby emphasize that, all those techniques are generally orthogonal to the proposal in this paper, and applicable in our prototype system for further performance improvement.

In-Memory Parallel Database. Recently, the research efforts on parallel relational query processing is shifting from disk-based architecture to the new in-memory processing paradigm, to keep the pace with the quick advances of cloud computing, abundance of RAMs in modern servers and the interactive querying requirements from real applications. A number of new systems are emerging in recent years, with new technologies to better address such demands. Shark [31], for example, processes queries on top of Apache Spark, with a data-centric computation resource allocation mechanism. It materializes huge amount of data for fault tolerance purpose and thus potentially wastes CPU cycles and memory space, to gain the benefits on efficient failure recovery. Imapla [2], known as the top performer in a wide group of tests [9] against Hive [28] and Presto [29], supports query processing under the massive parallel processing (MPP) paradigm. However, Impala does not fully exploit the computation power in modern multi-core hardware, as it uses single-threaded algorithms to process joins and aggregations [11]. In the following, we discuss two specific and general challenges for in-memory parallel database systems.

Firstly, inter-node data and intermediate result transmission becomes one of the major bottlenecks in multi-node in-memory parallel databases, due to the relative slow network bandwidth. There is extensive work on reducing the network communication. A general coloring theory is introduced in [14] to dispatch operators over the distributed nodes in order to minimize network communication. [32] minimizes the data repartitioning cost by reducing data repartitioning in the query plans. [26] proposes optimal data partition assignment strategy, to leverage data co-location for smaller network transmission cost. Our elastic pipelining technique assumes that the query execution plan on the distributed nodes is already optimized. The elasticity feature on the pipelines provides additional performance enhancement by fully utilizing the CPU cycles and network bandwidth.

Secondly, the workload imbalance among the pipelines significantly hinders the query processing efficiency. Manish et al. [22] propose a model on data producing rate of operators, together with intra-operator parallelism assignment strategy for load balancing. However, their method may not be applicable to the in-memory database cluster, as the data producing rates are more sensitive than traditional disk-based databases. In this paper, we simplify the load balance problem by adjusting the intra-segment parallelism at runtime rather than fixing the parallelism at compile time. There are also several adaptive parallelism methods proposed for online load balance in pipelined execution. [24] relies on the operating system scheduler to decide the runtime parallelism of the queries. It may result in expensive context switch cost and cache miss penalty. Our dynamic scheduler avoids the context switch cost by allocating one query thread per core and dynamically provisioning cores to segments based on the runtime workload. [19] achieves adaptive parallelism by decomposing query into self-contained, small-sized units. However, their method does not fully address the potential problem of parallel in-memory database. Our proposal provides a complete solution even when the operators interact via network connection.

7. CONCLUSION AND FUTURE WORK

In this paper, we propose elastic pipelining as a new approach to unleash the huge performance potential of pipelining framework for in-memory database clusters. We present elastic iterator model, as an extended version of traditional iterator model, to support runtime intra-segment parallelism update in a generic way. A dynamic scheduler is designed to monitor the workload of query execution, wisely choose the appropriate parallelism for each segment, which dramatically improves the overall throughput on data processing and minimizes the query response time. Extensive experiments on the real dataset and TPC-H dataset validate the great flexibility of our iterator model and effectiveness of our scheduling strategy.

In the future, we plan to explore the following directions for more performance improvement over the current system. Firstly, it is promising to allow runtime segment movement across nodes in the database cluster. Currently, the segment-to-node assignment is pre-defined and fixed during query execution. In extreme cases, the workloads on the nodes can be highly skewed. More flexibility on segment movement will enable the system to achieve better workload balancing for higher processing throughput. Secondly, the scheduling method can be further extended to handle multiple queries running at the same time. When there are a number of query executions running on the nodes, the scheduling strategy needs to be updated, in order to address the demands on the overall response time minimization on all queries.

8. ACKNOWLEDGMENT

This study is partially supported by National Science Foundation of China under grant No.61332006 and 863 Key project under grant No. 2015AA015303, the research grant for the Human-Centered Cyber-physical Systems Programme at the Advanced Digital Sciences Center from Singapore's A*STAR, and Infosys.

9. REFERENCES

[1] https://github.com/dase/claims.git.

[2] https://www.cloudera.com/products/apache-hadoop/impala.html.

[3] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *PVLDB*, 5(10):1064–1075, 2012.

[4] C. Balkesen, G. Alonso, J. Teubner, and M. T. Ozsu. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*, 7(1), 2013.

[5] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *SIGMOD*, pages 37–48, 2011.

[6] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in mapreduce. In *SIGMOD*, pages 975–986, 2010.

[7] L. Bouganim, D. Florescu, P. Valduriez, et al. Dynamic load balancing in hierarchical parallel database systems. In *VLDB*, pages 436–447, 1996.

[8] J. Cieslewicz and K. Ross. Adaptive aggregation on chip multiprocessors. In *VLDB*, pages 339–350, 2007.

[9] J. Erickson, M. Kornacker, and D. Kumar. New sql choices in the apache hadoop ecosystem: Why impala continues to lead. http://blog.cloudera.com/blog/2014/05/new-sql-choices-in-the-apache-hadoop-ecosystem-why-impala-continues-to-lead.

[10] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. Sap hana database: data management for modern business applications. *ACM Sigmod Record*, 40(4):45–51, 2012.

[11] A. Floratou, U. F. Minhas, and F. Ozcan. Sql-on-hadoop: Full circle back to shared-nothing database architectures. *PVLDB*, 7(12):1295–1306, 2014.

[12] T. Z. J. Fu, J. Ding, R. T. B. Ma, M. Winslett, Y. Yang, and Z. Zhang. Drs: Dynamic resource scheduling for real-time analytics over fast streams. In *ICDCS*, pages 411–420, 2015.

[13] G. Graefe. Volcano-an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[14] W. Hasan. *Optimization of SQL queries for parallel machines*. PhD thesis, Stanford University, 1995.

[15] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *PVLDB*, 2(2):1378–1389, 2009.

[16] K. Krikellas, S. D. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE*, pages 613–624, 2010.

[17] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–86, 2004.

[18] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., 1984.

[19] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age. In *SIGMOD*, pages 743–754, 2014.

[20] S. Manegold, P. Boncz, and M. Kersten. Optimizing main-memory join on modern hardware. *IEEE Trans. Knowl. Data Eng.*, 14(4):709–730, 2002.

[21] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing database architecture for the new bottleneck: memory access. *VLDB Journal*, 9(3):231–246, 2000.

[22] M. Mehta and D. J. DeWitt. Managing intra-operator parallelism in parallel database systems. In *VLDB*, pages 382–394, 1995.

[23] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[24] I. Psaroudakis, T. Scheuer, N. May, and A. Ailamaki. Task scheduling for highly concurrent analytical and transactional main-memory workloads. In *ADMS@ VLDB*, pages 36–45, 2013.

[25] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, et al. Db2 with blu acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, 2013.

[26] W. Rodiger, T. Muhlbauer, P. Unterbrunner, A. Reiser, A. Kemper, and T. Neumann. Locality-sensitive operators for parallel main-memory database clusters. In *ICDE*, pages 592–603, 2014.

[27] J. Teubner and R. Mueller. How soccer players would do stream joins. In *SIGMOD*, pages 625–636. ACM, 2011.

[28] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[29] M. Traverso. Presto: Interacting with Petabytes of Data at Facebook. https://www.facebook.com/notes/facebook-engineering/presto-interacting-with-petabytes-of-data-at-facebook/10151786197628920.pdf, 2014.

[30] L. Wang, M. Zhou, Z. Zhang, M. S. Shan, and A. Zhou. Numa-aware scalable and efficient in-memory aggregation on large domains. *IEEE Trans. Knowl. Data Eng.*, 27(4):1071–1084, 2015.

[31] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: Sql and rich analytics at scale. In *SIGMOD*, pages 13–24, 2013.

[32] J. Zhou, P.-A. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the scope optimizer. In *ICDE*, pages 1060–1071, 2010.

[33] M. Zukowski, M. van de Wiel, and P. Boncz. Vectorwise: A vectorized analytical dbms. In *ICDE*, pages 1349–1350, 2012.

APPENDIX

A. IMPLEMENTATION DETAILS

In this section, we introduce the implementation details of iterators in our elastic iterator model.

A.1 Elastic Iterator

Algorithm 2 presents the concrete workflow of the worker thread. When a new worker thread is started, it immediately calls child iterator's `open` function to initialize the entire sub-iterator tree. The new worker thread checks the returned status of `open` function. If the status is TERMINATED, it means that the expanded thread has received a terminate request during calling its child iterator's open. Then the thread exits to complete the shrinkage. Otherwise, the thread will repeatedly get data blocks from child iterator by calling `next` of child iterator and inserts the newly obtained data blocks into the data buffer until the calling of `next` returns status other than SUCCESS, which means either the input dataflow has been entirely consumed or the thread receives a terminate request.

Algorithm 2: Pseudocodes of the worker thread.

```
open_status=child_iterator.open()
if open_status=TERMINATED then
    /*open terminated due to terminal requrest*/
    return SHRINK_SUCCESS
/*open is successful, and call next*/
while true do
    next_ret=child_iterator.next()
    if next_ret=block then
        /*successfully get a block*/
        insert block to the buffer.
    else if next_ret=TERMINATED then
        /*received terminate request*/
        return SHRINK_SUCCESS
    else if next_ret=FINISH then
        /*child iterator is exhausted*/
        return FINISH
```

A.2 Modification to Existing Iterators

Before delving into the details, we first give the basic rules of revision over traditional iterator model to support the multi-threaded calling of `open` and `next` functions. Then we introduce our implementation of synchronization barriers, which is a fundamental function widely used in the implementation of elastic iterator model, to guarantee the correctness of multi-threaded query processing. Finally, we present the details of modification over existing iterators.

A.2.1 Modification Sketch

Implementation of `open`: The `open` function has two major tasks, including (a) calling the child iterator's `open` if necessary, and (b) initializing the state. For the first task, the corresponding code of traditional iterators does not need to be modified, as it is the responsibility of the child iterator to handle the multi-threaded calling of `open`. For the second task, the code should be carefully designed such that the state initialization is correct in case of multi-threaded calling of `open`. The way of handling multi-threaded calling of `open` depends on whether the workload of constructing its state is heavy enough such that multi-threaded processing is deserved.

For the non-blocking iterators such as filter and scan, they do not need to consume any data from the child iterator to construct their states, and hence the state construction is light-weighed and can be finished instantly. For instance, to construct the state of a filter iterator, the thread only needs to initialize the references to the predicate functions. For such operators, we only need to slightly modify the codes, such that only the worker thread first arriving at the initialization code is responsible for the state construction while other worker threads arriving later all wait at the synchronization barrier until the completion of state construction.

Blocking iterators, i.e., pipeline breakers, build their states with the data blocks obtained by calling child iterator's `next`. Compared with non-blocking iterator, the state construction in the blocking iterator is very heavy and deserves multi-threaded processing. Consequently, we let all the worker threads construct the state in parallel. As the child iterator is responsible for handling the multi-threaded calling of `next`, the blocking iterator only needs to achieve thread safety on `open` function, when the obtained data blocks are fed into the state by multiple threads concurrently. For instance, in the hash join iterator's `open`, it is necessary to protect the hash table only, by employing locks or atomic instructions.

Implementation of `next`: Recall that `next` function consumes data either from the iterator's state or from the child iterator, processes the data, and returns the processed data block to its caller each time. The way of supporting multi-threading in the `next` depends on whether the calling of `next` updates the state of the iterator.

For some iterators, such as filter and hash join, the codes in `next` only get access to iterator states in a read-only mode, and hence worker threads work independently of each other. Consequently, the `next` function under this category does not require any modification to support multi-threaded calling.

For the other iterators, such as scan, aggregation, and sort, the calling of `next` updates the states. `next` should be implemented in such a way that the updates to the state are thread-safe. Take hash aggregation iterator as an example. After state construction in `open`, the aggregated results are stored in hash table. `next` will return a block of tuples in hash table at a time. The update to the cursor that records the current read position in the hash table should be thread-safe by using locking or atomic instructions.

A.2.2 Synchronization Barriers

Synchronization barriers are widely used in our iterator implementation. A synchronization barrier forces multiple threads (the caller to the barrier) to wait until a specified number of threads have reached the particular point of execution where the barrier is injected. As the number of worker threads might change during query execution due to shrinkage and expansion operations, the barriers should be updated appropriately when a worker thread is terminated or created. To achieve this in our implementation, each barrier maintains a value, called $thread\_count$. The threads waiting at the barrier are blocked until the number of waiting threads reaches $thread\_count$. The initial value of $thread\_count$ is set to 0. When a newly created thread calls `open` of an iterator, it will call *registerToAllBarriers()* to increase the $thread\_count$ of all the barriers in this iterator by 1 such that the other worker threads have to wait this new thread if they arrived at any of the barriers. When a thread terminates, it will call *boradcastExitToAllBarriers()* to decrease the $thread\_count$ of all the barriers by 1 such that other threads waiting at the barriers will not wait this thread.

A.2.3 Modification Details

In the rest of this subsection, we will introduce the implementation details for the iterators one by one.

Scan: Algorithm 3 presents the mechanism of $scan$ iterator. $Scan$'s `open` initializes the read cursor for the target table. The workload in $scan$ is light-weight and can be finished by a single thread instantly. Consequently, only the worker thread who first calls $scan$'s `open` is responsible for initializing the read cursor. After `open` is finished, `next` will be called. Then the iterator gets one block from the read cursor and moves the cursor to the next reading position. To support multi-threaded calling of $scan$'s `next`, the movement of the read cursor should be atomic, either by locking or more efficient compare-and-swap (CAS) instruction. To achieve NUMA-awareness in a NUMA architecture, $scan$ maintains multiple read cursors, each pointing to a partition of the table on a NUMA socket. When $scan$'s `next` is called by a worker thread, $scan$ iterator prefers to return a block in local socket. As $scan$ is a pipeline beginner, if it detects a termination request when `next` is called, it will immediately return TERMINATE to its parent iterator such that the worker thread can terminate when all the tuples derived from the $scan$ iterator is processed properly.

Algorithm 3: Pseudocodes of the $scan$ iterator.

```
def open()
    registerToAllBarriers();
    if isFirstWorkerThread() then
        initialize the read cursor;
    Barrier->arrive();
def next()
    if detectedTerminateRequest() then
        return TERMINATE;
    /*get a data block, preferring blocks on local memory.*/
    if getBlockFromCursoe(block) =SUCCESS then
        return block;
    else
        return end-of-file;
```

Sender: Algorithm 4 presents the mechanism of $sender$ iterator. $Sender$ iterator is responsible for sending the data blocks from its segment to consumer (upside) segments. In our elastic iterator model, $sender$ is always the parent of expander iterator, thus it is never called by multiple threads. $Sender$'s `open` creates sending thread, which sends data block in the buffer to the $mergers$ of the upside segments according to a specified partitioning schema. After the sending thread is created, the $sender$ gets data blocks from elastic iterator and inserts the obtained data blocks to the sending buffer. $Sender$'s `next` returns end-of-file when all the blocks have been sent.

Algorithm 4: Pseudocodes of the $sender$ iterator.

```
def open()
    create sending thread;
    child->open();
    while child->next(block) =SUCCESS do
        insert block into sending buffer;
def next()
    blocked until all the blocks in sending buffer are sent;
    return end-of-file;
```

Merger: Algorithm 5 presents the mechanism of $merger$ iterator. A $merger$ is responsible for receiving data from the $senders$ of its downside segments. The first worker thread calling $merger$'s `open` is responsible for creating a merging thread. The merging thread receives data blocks from $senders$, and stores them in the $merger$'s buffer. In the NUMA architecture, $merger$'s buffer is split into several partitions, each of which is stored on a NUMA socket. And the data blocks received from $senders$ are inserted into each partition in a round-robin manner, for the purpose of balancing the workload among NUMA sockets. Note that creating a new merging thread is important in our elastic iterator model, because it guarantees that the $merger$ is able to receive data blocks when all worker threads have been terminated. When $merger$'s `next` is called, it gets a data block from its buffer, preferring to the data blocks on local socket. As $merger$ is the pipeline beginner in a segment, the termination code is inserted at the beginning of `next`.

Filter: The `open` of $filter$ iterator initializes the filter function. Similar to $scan$ iterator, the state can be constructed by a single work thread instantly. $Filter$ iterator's `next` does not modify its state when being called, thus $Filter$'s `next` is the same as traditional iterator.

Hash Join: Algorithm 6 presents the mechanism of $join$ iterator. $Join$ iterator has two input iterators, say L and R. In $join$ iterator's `open`, the tuples obtained by calling L's `next` are inserted into the hash table. To guarantee the correctness of building hash table by multiple threads, tuple insertion to the hash table should be atomic, by locking or CAS instruction. When L's `next` returns TERMINATE, it means that the worker thread received a terminate request. In such cases, the thread stop building the hash table and is terminated in Line 9.

Algorithm 5: Pseudocodes of the $merger$ iterator.

```
def open()
    registerToAllBarriers();
    if isFirstWorkerThread() then
        create merging thread;
    Barrier->arrive();
def next()
    if detectedTerminateRequest() then
        return TERMINATE;
    /*get a data block, preferring blocks on local memory.*/
    if getBlockFromBuffer(block) =SUCCESS then
        return block;
    else
        return end-of-file;
```

Algorithm 6: Pseudocodes of the $join$ iterator.

```
1  def open()
2      registerToAllBarriers();
3      left_child->open();
4      while left_child->next(block) =SUCCESS do
5          foreach tuple ∈ block do
6              atomic insert tuple into its corresponding hash bucket;
7      if detectedTerminateRequest() then
8          broadcastExitToAllBarriers();
9          return TERMINATE;
10     right_child->open();
11     Barrier->arrive();
12 def next()
13     while right_child->next(block) =SUCCESS do
14         foreach tuple ∈ block do
15             insert all the matching tuples in hash table to
               result_block;
16         return result_block;
17     return end-of-file;
```

After the hash table is built, $join$'s `next` will be called by multiple worker threads to get the join results. Once $join$ iterator's `next` is called by a worker thread, the thread asks for data blocks by calling R's `next`. By probing the hash table with the tuples from R, $join$ iterator returns join results in form of data blocks to its parent iterator. Once the tuples in R are exhausted, the iterator returns end-of-file. As the probe in equal join never modifies the hash table, it does not need any synchronization under multiple thread workers.

Hash Aggregation: Algorithm 7 presents the mechanism of $aggregation$ iterator. The `open` of $aggregation$ iterator reads data blocks from child iterator and updates the results of the corresponding entries in the hash table. As the same as $join$ iterator, aggregation is expensive and deserves multi-threaded processing. For multi-threaded aggregation, the updates to hash table should be protected by an atomic instruction or a lock. We provide two aggregation implementations, denoted as shared aggregation and hybrid aggregation. In shared aggregation, all the worker threads directly update the hash table, which is efficient when the group-by cardinality is large. In hybrid aggregation, a small private hash table is allocated to each worker thread to buffer the partial aggregation results, from which the overflowing entries are flushed into the global hash table. In hybrid aggregation, before a worker thread terminates, it stores its private hash table in the iterator's context for reuse. When a new worker arrives at the aggregation's `open`, it tries to get a unused private hash table allocated by the same core. By dong so, the hash table initialization cost can be avoided by reusing the context if the private hash table is still in cache. This feature is important to keep a short shrinkage delay. To improve the performance under NUMA architectures, the private hash table for each thread is allocated on the local NUMA socket, while the global hash table is partitioned across all the NUMA sockets for better load balance.

Algorithm 7: Pseudocodes of the *aggregation* iterator.

```
def open()
    registerToAllBarriers();
    child->open();
    if isHybridAggregation() then
        /*try to reuse private hash allocated by
        the same core*/
        private_hashtable=getOrCreateContext()(core
        mode);
    while child->next(block)=SUCCESS do
        foreach $tuple \in block$ do
            atomically aggregate $tuple$ into the global/private hash
            table;
    if detectedTerminateRequest() then
        broadcastExitToAllBarriers();
        if isHybridAggregation() then
            storeContext(private_hashtable);
        return TERMINATE;
    foreach private hash table pht in the context do
        atomically remove pht from context and flush the result into
        hash table.;
    Barrier->arrive();
def next()
    if detectedTerminateRequest() then
        broadcastExitToAllBarriers();
        return TERMINATE;
    if atomicGetBlockFromHashTable(block)=SUCCESS
    then
        return block;
    else
        return end-of-file;
```

After the `open` function of *aggregation* iterator finishes, the aggregation results are stored in the hash table. When *aggregation*'s `next` is called by a worker thread (from the parent iterator), the iterator organizes a data block from hash table and returns it to the parent iterator. The updates to the read cursor recording the current read position in the hash table should be thread-safe, again by employing locking or CAS instructions. As *aggregation* is a pipeline beginner, the worker thread return TERMINATION if it receives a terminate request, which is similar to *scan* iterator.

Sort: Algorithm 8 presents the mechanism of *sort* iterator. In *sort*'s `open`, the worker thread first calls the child iterator's `open`, and then repeatedly calls `next` function of the child iterator and inserts the obtained data blocks into its buffer. After all the data blocks from child iterator are inserted to the buffer, the worker thread gets one chunk from the buffer at a time, sorts the chunk, and inserts the sorted chunk into a list. Before the worker thread tries to obtain the next chunk to sort, it terminates if terminate request is detected. The chunk size is a trade-off between the shrinkage delay and the merging cost, which can be easily configured by empirical results in practice. After all the chunks are sorted, one worker thread is responsible for computing the global separators and assigns them to each worker thread such that each work can merge the data without synchronization. When *sort*'s `next` is called by a worker thread, the iterator returns a data block from the sorted buffer and updates the read cursor with locking or CAS instruction employed.

Algorithm 8: Pseudocodes of the *sort* iterator.

```
def open()
    registerToAllBarriers();
    child->open();
    while child->next(block)=SUCCESS do
        insert block into $buffer$;
    if detectedTerminateRequest() then
        broadcastExitToAllBarriers();
        return TERMINATE;
    while getChunkFromBuffer(&chunk)=SUCCESS do
        local sort $chunk$ and add $chunk$ into the
        $sorted\_chunk\_list$;
        if detectedTerminateRequest() then
            broadcastExitToAllBarriers();
            return TERMINATE;
    Barrier1->arrive();
    if isFirstWorkerThread() then
        compute the global separators keys and assign them to each
        worker thread;
    Barrier2->arrive();
    merge the data into $sorted\_buffer$ according to assigned
    separators keys;
    Barrier3->arrive();
def next()
    if detectedTerminateRequest() then
        broadcastExitToAllBarriers();
        return TERMINATE;
    if getBlockFromSortedBuffer(block)=SUCCESS then
        return block;
    else
        return end-of-file;
```