Cluster Comput (2010) 13: 127–140
DOI 10.1007/s10586-009-0109-8

Service-oriented execution model supporting data sharing and adaptive query processing

Yongwei Wu · Jia Liu · Gang Chen · Qiming Fang · Guangwen Yang

Received: 15 May 2009 / Accepted: 15 October 2009 / Published online: 30 October 2009

Abstract To deal with the environment's heterogeneity, information providers usually offer access to their data by publishing Web services in the domain of pervasive computing. Therefore, to support applications that need to combine data from a diverse range of sources, pervasive computing requires a middleware to query multiple Web services. There exist works that have been investigating on generating optimal query plans. We however in this paper propose a query execution model, called PQModel, to optimize the process of query execution over Web Services. In other words, we attempt to improve query efficiency from the aspect of optimizing the execution processing of query plans.

PQModel is a data-flow execution model. Along with an adaptive query framework it used, PQModel aims to improve query efficiency and resource utilization by exploiting data and computation sharing opportunities across queries. A set of experiments, based on a prototype tool we developed, were carefully designed to evaluate PQModel by comparing it with a model whose query engine evaluates queries independently. Results show that our model can improve query efficiency in terms of both response time and network overhead.

Keywords Web service · Query processing · Data sharing · Data-flow execution model

Y. Wu (✉) · J. Liu · G. Chen · Q. Fang · G. Yang
Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing 100084, China
e-mail: wuyw@tsinghua.edu.cn

J. Liu
e-mail: liu-jia04@mails.tsinghua.edu.cn

G. Chen
e-mail: c-g05@mails.tsinghua.edu.cn

Q. Fang
e-mail: fangqiming@gmail.com

G. Yang
e-mail: ygw@tsinghua.edu.cn

1 Introduction

The promise of Web services (WSs) is to enable a distributed environment, in which any number of applications or application components can interoperate seamlessly among organizations in a platform-neutral, language-neutral fashion [10]. Due to the flexibility, extensibility and interoperability of Web services, Service-Oriented Architectures (SOA) are widely adopted for deploying pervasive computing environments [4, 33], which have the characteristics of high heterogeneity, high interoperability, and high-mobility, by modeling the available resources as services and providing mechanisms such as for service discovery, data management, security control etc.

Pervasive computing poses a number of challenges for data management [11]. One of the important challenges is the ability to combine data from a diverse range of data sources. In this paper, we address this challenge by querying over WSs for the pervasive computing environment adopting a Service-Oriented Architecture.

Query facilities [5, 9, 24, 25, 30, 31, 34] have been developed to support service-oriented queries, which enable users to access multiple WSs in a transparent and integrated fashion. Meanwhile, techniques on improving efficiency of service-oriented queries have been studied, which often include: (a) selecting the best WS from WSs with similar functionalities but from different service providers [13, 25]: Quality Of Web Service (QoWS), with parameters such as availability, latency, and fees, is usually considered as a key feature of distinguishing competing WSs, (b) ordering the selected WSs to form a best execution plan [9, 25, 30] determined according to the criteria of minimum query execution time, minimum monetary costs, or minimum monetary costs subjecting to a limit on the query execution time. These methods focus on generating optimal query plans. In this paper, we present a processing model PQModel, which focuses on the execution of query plans instead.

Our PQModel is an execution model that not only supports data/computation sharing but also facilitates adaptive query processing. Exploiting data/computation sharing could improve query efficiency and reduce resource consumption when overlapping WS requests occur in multiple concurrent queries. As opposed to existing query execution models (e.g., [30]), which typically evaluate queries independently by assigning a set of threads for each query, our PQModel adopts an operator-centric data-flow query execution model similar to most of data stream processing systems [6, 20]. Each query can be decomposed into a set of Web Service Processors (WSPs) and join operators, and the query is processed by routing each tuple through them. Concurrent queries are able to share WSPs and join operators during processing. Therefore, Adaptive data/computation sharing mechanisms can be developed, when such an operator-centric data flow execution model is used, to better utilize resources and improve query efficiency.

In terms of facilitating adaptive query processing, our PQModel utilizes an adaptive framework for Adaptive Query Processing (AQP) [16], which is an effective approach to correct bad initial decisions during query execution. WSPs of PQModel are able to capture running information (e.g., WSP rate, service cost and selectivity) of query execution. The component Event Handler of PQModel is used for assessing this information and identifying issues. The components Tuple Encapsulator, Thread Allocator and WSPs provide interfaces to respond to the related issues. Thus, different adaptive schemes, which are useful for handling various system changes, can be implemented in PQModel.

The rest of the paper is organized as follows. Section 2 describes preliminaries. Section 3 presents PQModel including its architecture and components. The design details of WSP are discussed in Sect. 4. The experimental results based on our prototype implementation are reported in Sect. 5. The related works are discussed in Sect. 6. Section 7 concludes the paper and discusses future work.

2 Preliminaries

In this section, we discuss the preliminaries of our work from three aspects: web service, service-oriented query, and query plan.

2.1 Web service

Web services are modeled as function calls in PQModel similar to the one described in [30].

Web services are modeled as function calls. Each Web service WS_i provides a function call like interface $X_i \rightarrow Y_i$: given values of attributes in X_i, WS_i returns values of attributes in Y_i. Applying the denotation of binding patterns [15], WS_i can be modeled as $WS_i(X_i^b, Y_i^f)$, where the values of the attributes in X_i, must be specified (or bound) while the values of the attributes in Y_i are retrieved (or free).

Moreover, we have to emphasize here, as a prerequisite, is that all WSs in our context are information providers, which implies that they operate on backend data sources in a read-only manner and therefore multiple concurrent WS requests with equivalent input values are then able to be merged into a single request.

2.2 Service-oriented query

The definition of query is given as follows:

$$\text{select } A_O \text{ from } I(A_I) \bowtie WS_1\big(X_i^b, Y_i^f\big) \bowtie \cdots$$
$$\bowtie WS_n\big(X_i^b, Y_i^f\big) \quad \text{where } P_1(A_1) \wedge \cdots \wedge P_m(A_m)$$

where A_O is the set of output attributes, $I(A_I)$ is the schema of the input table corresponding to the data input of a client, A_I is the set of input attributes, $WS_1, \ldots, WS_n$ are the queried WSs, and $P_1, \ldots, P_m$ are the predicates applied on the attributes $A_1, \ldots, A_m$ respectively.

The following is an example query:

Example 2.1 The following are three WSs:

(1) *getSalesPromotion*($city^b$, $market^f$): given the city name, it returns the markets in the given city holding promotional activities.
(2) *getAddress*($market^b$, $address^f$): given the market, it returns the address of the market.
(3) *getPrice*($market^b$, $product^b$, price^f): given the market and the product id, it returns the price of the product in the given market.

The following query Q attempts to find the addresses of markets, which are holding promotional activities in the given city and have the given product on sale with a price lower than 100.

Q: select address, price from $I(city, product)$

$\bowtie$ *getSalesPromotion*(*city*^{b}, *market*^{f})
$\bowtie$ *getAddress*(*market*^{b}, *address*^{f})
$\bowtie$ *getPrice* (*market*^{b}, *product*^{b}, *price*^{f})

where price $<$ 100.

2.3 Query plan

A query plan specifies the processing order of WSs in a query. Figure 1 shows a query plan of Example 2.1. We represent a query plan as a directed acyclic graph (DAG) that accepts an input table (e.g., I(*city, product*)) and produces answers.

(1) Each edge in the plan implies a producer/consumer relationship between nodes.
(2) Each node in the plan refers to either a WSP or a Join operator. A Join operator performs join on inputs from multiple precedent nodes. WSP is a newly defined operator. A WSP processes WS requests to a specified WS. Two implicit operators (selection and projection) are implemented in the WSP to filter out unnecessary data.

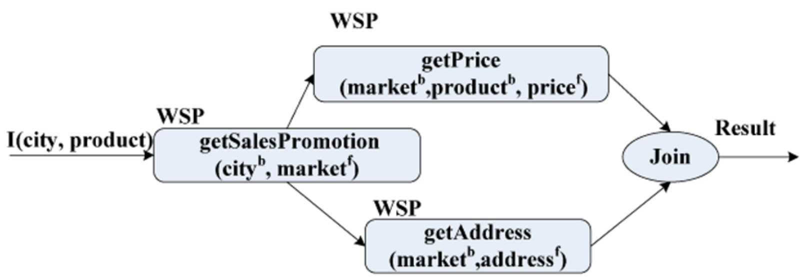
3 PQModel

We introduce our PQModel in this section by describing its architecture (Sect. 3.1), explaining query process (Sect. 3.2), and analyzing why PQModel is suitable to process service-oriented queries (Sect. 3.3).

3.1 Architecture

As shown in Fig. 2, the architecture of PQModel is essentially a dataflow style execution model, which is composed of a set of *WSP* instances (e.g., WSP_1 and WSP_2) and *Join* instances. PQModel maintains a thread pool, and the threads in the pool are continuously assigned to work for the *WSP* instances or *Join* instances. Each instance has an input buffer

Fig. 1 An example query plan



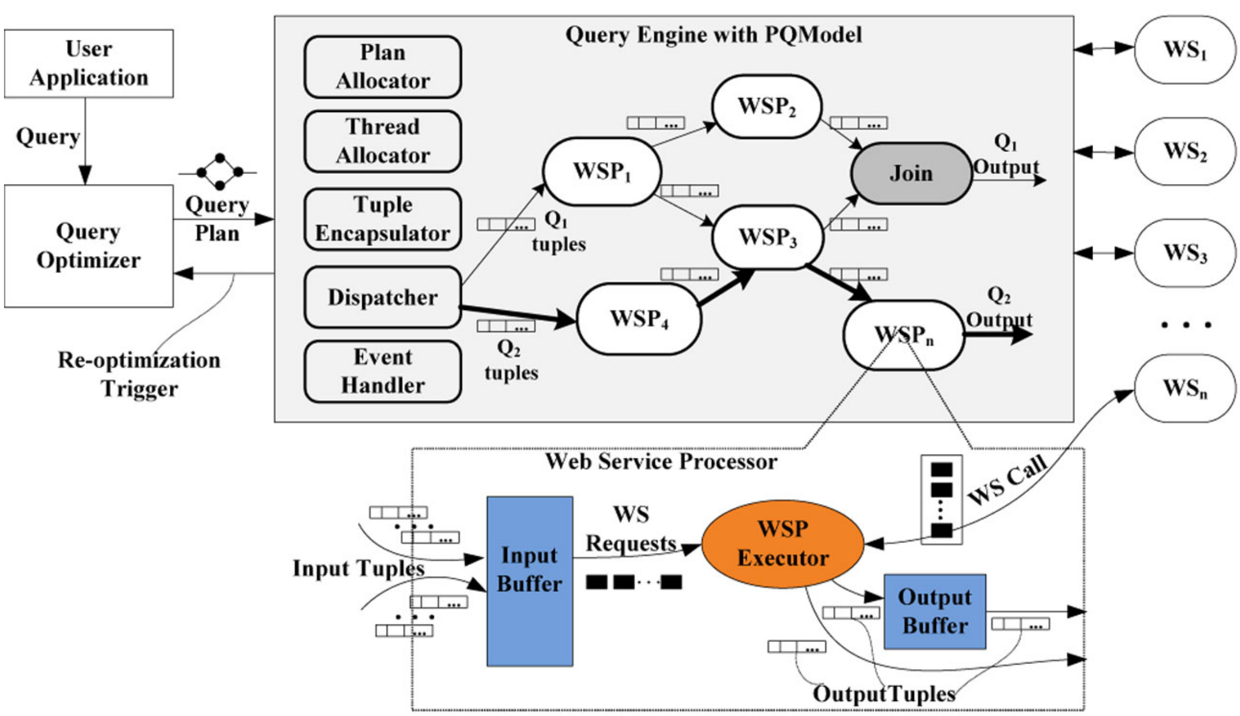


Fig. 2 Architecture of PQModel

for arriving requests (in the form of tuples) and one or more worker thread for processing requests. The input to PQ-Model is query plans (generated by optimizing algorithms as in [30]). Query plans can be broken into a set of tuples with route information after passing through the *Tuple Encapsulator*. The way a query is processed is by routing the tuples through a pipeline of the instances. For example, in Fig. 2, Q_1 can be processed by routing its tuples through WSP_1, WSP_2, WSP_3 and *Join* instance. Q_2 can be processed by routing its tuples through WSP_4, WSP_3 and WSP_n.

Each component of the architecture is described as follows:

(1) *WSP*: performs two functions concurrently. (I) Each WSP instance is in charge of service invocation for each WS. (II) WSP also collects running information during execution. Each incoming tuple of WSP instance contains a WS request. For instance, in Example 2.1, when a tuple containing "Beijing" arrives at *WSP*(*getSalesPromotion*), it should get the markets in the city of Beijing that holding promotional activities by invoking the WS *getSalesPromotion*. WS requests are generally can be processed in two modes: single mode and chunk mode. For single mode, the WS requests are processed one by one. In other word, each WS call contains only one WS request. For chunk mode, the WS requests are processed in chunks, that is, each WS call gets answers for a chunk of WS requests. Chunk mode is always adopted to make WS calls. As described in [30], each WS call usually has some fixed overhead, e.g., parsing SOAP/XML headers. Hence it can be very expensive to invoke a WS separately for each request. Sending requests to WSs in chunks (as shown in Fig. 2) can significantly reduce network overhead. PQModel uses chunk mode in default. As shown in Fig. 2, a WSP instance performs the following actions to serve an arriving tuple: WSP gets the WS request contained in the tuple by parsing its data, retrieves output values by making WS call, checks the relevant predicates, writes output data into the tuple's data, and routes the tuple to its next destinations. Each WSP can provide services for one or more queries. As shown in Fig. 2, if two concurrent queries (Q_1 and Q_2) contain the same WS (WS_3 in Fig. 2), then they can share the same WSP instance (WSP_3). The detailed design of WSP is discussed in Sect. 4.

(2) *Join*: is in charge of performing join on its input tuples and routing its output tuples for further processing.

(3) *Plan allocator*: prepares all required WSP instances and Join instances for every arriving query. In different situations, Plan Allocator performs the following operations: (I) creating a new WSP instance for a given WS, (II) destroying an existing WSP instance, (III) adding a query to a WSP instance, (IV) removing a query from a WSP instance, (V) creating a new Join instance, and (VI) destroying an existing Join instance.

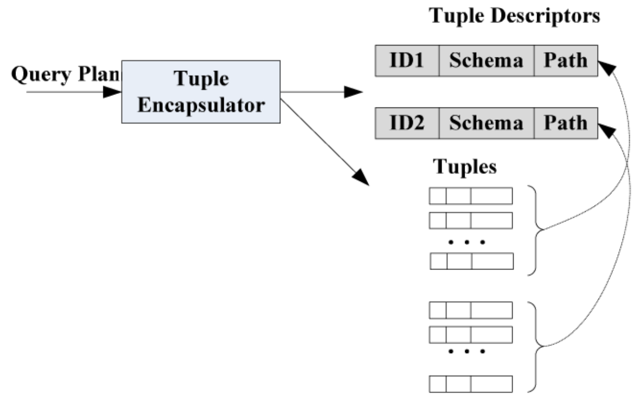


Fig. 3 Tuples with route information

(4) *Thread allocator*: decides how to allocate the threads in the thread pool to the WSP instances and join instances.

(5) *Tuple encapsulator*: is in charge of encapsulating data items in queries' input tables into tuples containing route information. Tuple Encapsulator produces one or more tuple descriptors and a set of tuples for each arriving query. More than one tuple descriptor, as the case shown in Fig. 3, can be generated if *Query Optimizer* allows different data items to follow different plans (e.g., [8]). The tuple descriptor is composed of three regions: descriptor ID, tuple schema and path. The tuple schema describes the list of attributes in each tuple; while the path describes the order in which the query's tuples are processed by WSP instances and Join instances. Each tuple also contains three regions: descriptor ID, data and route indicator. The descriptor ID points to the tuple descriptor containing the path that the tuple should follow. The data in a tuple is the values of the attributes, and its route indicator indicates the progress of the query processing. In our model, the component Dispatcher and all its operators are able to route tuples. According to the path specified in the tuple descriptor and the query processing progress given in the route indicator of a tuple, Dispatcher can decide the next destination(s) of the tuple. Thus, every tuple can be routed individually through WSP instances and Join instances for processing.

(6) *Dispatcher*: sends tuples to their first destination for queries.

(7) *Event handler*: is responsible of (I) receiving running information from WSPs, (II) assessing the information to identify whether there exist opportunities for improvement of plan performance, (III) making adaption decisions, and (IV) notifying other components to respond to the decisions.

3.2 Query process

PQModel takes two steps to process each query:

Step 1: Prepare query execution. Given an arriving query, this step prepares all the WSP instances and Join instances required by the query plan and encapsulates each data item into a tuple.

Step 2: Perform query execution. In this step, the Dispatcher dispatches the tuples prepared in Step 1 to WSPs for processing. The tuples are then routed through WSP instances and Join instances until the tuples are discarded or their answers are produced.

3.3 Analysis

In this section, we analyze why our PQModel is suitable to process service-oriented queries in terms of two goals: (1) reducing average response time of processing WS requests, and (2) adaptively allocating resources and changing query plans and therefore further improving query efficiency.

PQModel achieves the first goal by:

(a) Improve query efficiency by exploiting and reusing sharable WS requests, as generally the cost of WS request is expensive.
(b) Improve query efficiency by sharing WS calls. Recall that overall network overhead of WS requests can be reduced while processing WS requests in chunk mode, where a number of WS requests composing the same WS call can share the fixed part of overhead on making WS call. However, in the case of processing small queries (which means the number of WS requests needed to be processed is very small), the existing models (i.e., the models processing queries independently) could not fully take advantage of data/requests chunking. As opposed to these models, our PQModel utilizes operator sharing to combine multiple small chunks from different queries into a big one and therefore improves the efficiency of WS processing.
(c) Reduce average response time of WS requests by reducing tuples' waiting time while processing WS requests in chunks. This is possible because WSP allows us to combine WS requests from different queries into a chunk. Hence, the waiting time required to compose a chunk can be reduced.

PQModel achieves the second goal by using the generic adaptive framework proposed in [16] (the detailed discussion on adaptive strategies is omitted from this paper due to space limitation), which has the advantages of component reuse, more systematic AQP (adaptive query processing) development, and easy AQP deployment. The framework in [16] decomposes the feedback loop of adaptive query processing into three distinct phases namely monitoring, assessment and response, and uses three associated

```
AdaptivityComponent{
public:
  Queue inputQueue;
private:
  AdaptivityComponent[] subscribers;
  analyseNotification(Notification){}
  sendNotification(Notification,
       subscribers){}
    subscribe(){}
}
```

Fig. 4 The interface of adaptivity components

components: (1) monitoring component: acts as a source of notifications on the dynamic behaviour of the ongoing query execution, (2) assessment component: is to identify whether there exist opportunities for improvement of plan performance, and (3) response component: is responsible for making response decisions. Each component supports a publish/subscribe interface (as shown in Fig. 4) to provide and ask for services to and from other components, respectively. As shown in Fig. 4. Public attribute `inputQueue` is used for storing notifications from other components. Private function `analyseNotification(Notification)` is used to analyze input notifications. Function `sendNotification (Notification)` is used to publish events. Function `subscribe()` is used to register with other adaptivity components. To adopt the framework, in PQModel, WSPs is implemented as monitoring component, Event Handler is implemented as assessment component and response component, and the components of Tuple Encapsulator, Thread Allocator and WSPs are able to actuate response decision made by Event Handler. By conforming to the framework, and assembling different existing AQP techniques (e.g., [8, 19]), PQModel has opportunity to (1) implement adaptive resource allocation for resource utilization or workload balancing, and (2) implement adaptive plan modification for higher query efficiency.

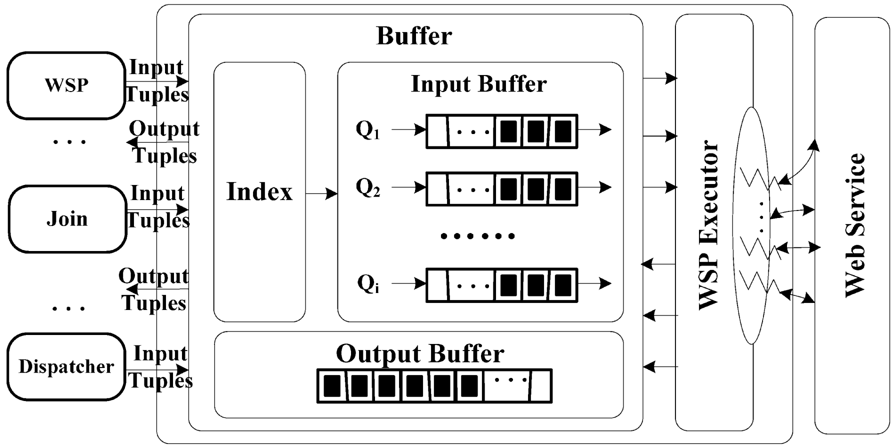
4 Web service processor

In this section, we present the design details of WSP by discussing its architecture and justifying how it facilitates exploiting data/computation sharing during WS processing (Sect. 4.1), and also its self-monitoring mechanism (Sect. 4.2).

4.1 WPS architecture

Recall that a WSP instance is used to process WS requests to a specified WS. As shown in Fig. 5, a WSP consists of two main components: a Buffer (Sect. 4.1.1) and a WSP Executor (Sect. 4.1.2). The buffer is used to store input and output tuples and the WSP Executor processes input tuples by invoking a WS.

Fig. 5 WSP in detail



To clarify how a WSP works, we first define several notations. $WSP(WS_i(X_i^b, Y_i^f))$ represents the WSP processing requests to WS_i. Let t be an arbitrary tuple arriving at $WSP(WS_i(X_i^b, Y_i^f))$. $V_i(t)$ denotes the data contained in t and $X_i(t)$ denotes the schema of $V_i(t)$. Before invoking WS_i, the values in X_i^b must be specified in tuple t, denoting as $V_i^b(t)$. That is, $X_i^b \subseteq X_i(t)$ and $V_i^b(t) \subseteq V_i(t)$ must be satisfied. After tuple t is processed by WS_i, the data in Y_i^f must be obtained. For any two tuples t_1 and t_2, if $V_i^b(t_1) = V_i^b(t_2)$, then both t_1 and t_2 can get values for Y_i^f by a sharing WS request though t_1 and t_2 come from different queries.

4.1.1 Buffer

As shown in Fig. 5, the buffer is composed of an input buffer, an index and an output buffer, which are described as follows respectively.

- Input buffer
  The input buffer deposits every input tuple waiting for service processing. As shown in Fig. 6a, the input buffer of WSP is implemented by an *ArrayList*. Every query is associated with a node of the ArrayList, which points to an input queue. The queue is implemented as a one-dimensional array with two pointers. One pointer is called *Head*, which points to the first element that can be taken away from the queue. The other is called *Tail*, which points to the position that can be used to place the next incoming element in the queue.
  Every arrival input tuple is stored in an input queue. Each element of the queue is specified as an object called *QueueElement*, which represents a WS request shared by a group of input tuples. As shown in Fig. 6b, a *QueueElement* contains three regions: *Input*, *Output*, and *Subscribers*. The input is an instance of X_i^b. The output is an instance of Y_i^f. The subscribers refer to the tuples containing the same values in the attributes X_i^b.

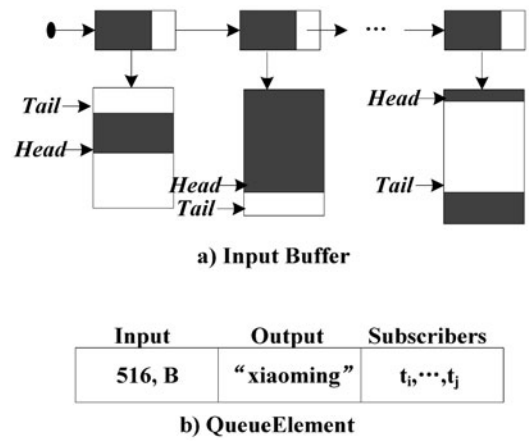


Fig. 6 Input buffer

- Index
  The index finds the input tuples stored in the input buffer. It optimizes the speed of identifying the tuples sharing the same WS request by the means of recording the input of each *QueueElement* and its locations in the input buffer.
  When an input tuple t arrives, WSP performs a lookup to the index for the value of $V_i^b(t)$ (the WS related input value in t) to see whether there is a matching WS request (represented by a *QueueElement*) that can be reused. If yes, there must be a request, say *QueueElement* $e1$, can be identified by the value returned by the index and then tuple t should be added into the $e1$'s subscribers. Otherwise, the following action should be sequentially taken: (1) a new *QueueElement*, say $e2$, with $V_i^b(t)$ in its input should be created, (2) tuple t should be added into the $e2$'s subscribers, (3) $e2$ should be inserted into the input queue, which is allocated to the query that contains tuple t, and (4) the value of $V_i^b(t)$ and the location of the *QueueElement* $e2$ are recorded into the index.
- Output buffer
  The output buffer is used to temporarily store the processed

tuples waiting to be dispatched. It is implemented as an array list. Its every node contains an output tuple and a pointer to the next node. Every output tuple is dispatched immediately if its next destination has sufficient space. Only those waiting for spare space are stored in the output buffer.

To dispatch the tuples in the output buffer, WSP scans the output buffer periodically. For each tuple t_i, if there is spare space in its next destination, t_i is removed from the output buffer and routed away. Otherwise, tuple t_i is kept in the output buffer to wait for spare space.

4.1.2 WSP executor

The WSP executor is a multithreaded executor. Each thread executes the following four tasks sequentially: (1) processing WS requests in the input buffer, (2) dispatching output tuples in the output buffer, (3) creating notifications for the Event Handler, and (4) analyzing the notifications from the Event Handler.

To process WS requests, each thread in WSP executor can apply either a single model or a chunk model. For single mode, the thread processes requests one by one, while for chunk mode the thread processes requests in chunks (assuming that the chunk size is $|S_c|$).

Let T_i be a thread working in chunk mode. Let q_i be an input queue. T_i gets its next WS request from q_i. T_i sequentially follows the following steps to perform its tasks: (1) Thread T_i takes requests from the input buffer. If the number of the requests in q_i, say $|q_i|$, is larger than or equal to the chunk size $|S_c|$, T_i takes a chunk of requests from q_i. Otherwise ($|q_i| < |S_c|$), thread T_i first takes all the available requests from q_i, and then takes a number ($\leq |S_c| - |q_i|$) of available requests from other input queues. (2) T_i extracts the input of each request and constructs a WS call. (3) After the WS call is formed, T_i invokes a remote WS for processing and retrieves answers arrived back. (4) Since each request is shared by one or more tuples in its corresponding subscribers, T_i writes output data back to each tuple in its subscribers. (5) T_i applies the relevant predicates specified in the query to each tuple. If any of the predicates is unmatched, the tuple is discarded. (6) T_i forwards the processed tuples. For each tuple t, (a) if its next destination is null (i.e., t is arriving at output point), T_i computes answers for the corresponding data items; (b) if the tuple's next destination is a WSP, then T_i validates that there is spare space or similar tuples in the tuple's next destination and then T_i dispatches tuple t; otherwise (i.e., no spare space or similar tuple in its next destination), T_i puts t into the output buffer; (c) If the tuple's next destination is a Join instance, T_i dispatches t to the Join instance. (7) T_i scans the output buffer to dispatch the tuples in it. (8) T_i creates notifications for the Event Handler by checking the variation of the running information. (9) T_i analyzes the notifications from the Event Handler and responds to it.

Thread T_i repeats the nine steps (1)–(9) until it is released from the WSP executor.

4.1.3 Discussion

As discussed in Sect. 3, one of the methods for PQModel to improve query efficiency is through operator sharing. This is ensured by the WSP architecture from the following points:

(1) WSP is able to exploit sharable WS requests. Concurrent tuples with equivalent values in WS's input parameters are grouped together to share the same QueueElement.
(2) WSP is able to share WS calls. This conclusion is very direct by the reason that threads in WSP executor are allowed to get WS requests from multiple input queues to make WS calls.
(3) WSP is able to reduce tuples' waiting time for WS processing while the WSP executor works in chunk mode. This is also resulted from that WSP executors are able to group WS requests from different queries to the same WS calls.

4.2 Self-monitoring mechanism

The other important functionality of WSP is to monitor cost and selectivity of each WS and the workload of each WSP instance. Monitoring and collecting this information is crucial for the Event Handler to validate if the query execution is still efficient.

Many metrics are available to measure the status of a WSP. We use the following three metrics as examples to illustrate the self-monitoring mechanism:

(1) Service selectivity. It is measured for each query because each query may contain distinct service-related predicates. For a service WS_i, service selectivity for query Q is measured as the average number of output tuples that WS_i produces for each input tuple after applying all its predicates related to WS_i in Q. The service selectivity of query Q_i is computed as $n^i_{\text{out}}/n^i_{\text{in}}$, where n^i_{out} denotes the number of tuples produced for Q_i and n^i_{in} is the number of tuples processed for Q_i.
(2) Service cost. For a service WS_i, its cost is measured as the average response time of each WS_i request. Service cost may depend on many dynamic factors such as the network conditions and the WS workload, thus service cost may change at all times. To estimate the recent cost, we adopt a window of a certain length and compute the average response time of WS requests in the window. The service cost for service WS_i in a window w_i is computed as $t_{\text{win}}/n_{\text{win}}$, where t_{win} is the total cost of the recent w_i WS calls and n_{win} is the number of WS requests processed by the recent w_i WS calls.

(3) WSP rate. It is a metric that reflects the processing power of a WSP instance. For a service WS_i, the rate of $WSP(WS_i)$ is measured as the number of WS_i requests that can be consumed in a time unit. It is computed as $n_{\text{con}}/t_{\text{span}}$, where t_{span} is the time elapsed since the number of threads in the WSP executor changed and n_{con} is the number of tuples processed by the WSP instance since the number of threads in the WSP executor changed.

After these metrics are obtained, WSPs can use them to drive adaption. Due to space limitation, we do not discuss the detailed adaptive techniques in this paper.

5 Evaluation

In this section, we propose our evaluation method and evaluation results. The experiments we performed for the evaluation are based on a prototype system, called SenGine, which realizes our PQModel and is implemented using Java. The overall experimental setup is discussed in Sect. 5.1, followed by the detailed discussion of each experiment and its results in Sect. 5.2.

5.1 Experimental setup

The experimental setup consists of two parts: the server side to deploy WSs and the client side to run SenGine. On the server side, we used Tomcat [2] as the application server and Axis tools [1] for WS deployment. In our experiments, each WS ran on an individual machine and provided data by issuing a SQL query to a Mysql (Version: 4.0.23) database deployed on a different machine. Several tables were created in the database, with different data characteristics. We will detail each table along with each experiment. On the client side, SenGine ran on a machine with 3 GHz Intel Pentium 4 CPU and 2 GB RAM. To demonstrate the effectiveness of our model, we compared our model with the one applied in [30], whose query engine evaluates queries independently by invoking a set of threads for each query. This model is denoted as the *independent* model in this section. Both the independent model and our PQModel are multithreaded. They both communicate with WSs using SOAP.

We compare average response time of queries and network overhead between PQModel and independent model. Network overhead in this section is measured as the total number of WS calls rather than communication traffic.

For the same query, the number of tuples generated by our PQModel and independent model are the same. But the total number of WS requests generated by our PQModel will be less than or equal to that of independent model because of WS requests sharing mechanism. So the number of WS calls generated by PQModel is also less than that of the independent model. Though the total number of WS requests is not linear with the network overhead or communication traffic. But the less number of WS requests will lead to less communication traffic definitely. So we use the number of WS calls replace Network overhead in our evaluation.

5.2 Experimental results

Table 1 shows all query templates used in following experiments.

5.2.1 Effect of sharing WS calls among queries

Initial experiments have been conducted to evaluate the effectiveness of sharing WS calls among queries, which is enabled by operator sharing.

We tested the average query response time and the total number of WS calls of running a collection of small queries. In this setting, independent model is hard to take advantage of chunk mode since the number of generated WS requests for each query is small, and our model can still take advantage of chunk mode since WS requests from different queries can share the same WS call. In this experiment, the queries are submitted to our prototype one by one every 0.5 second. Each query complies with the query template T_1 (see Table 1), but uses a different input table with only one data item in it. This design decision is made based on the fact that a query set containing queries that follow the same query template but use different parameters is frequently encountered in web applications, where every query is submitted through a web form corresponding to the same query template.

In T_1, selectivities of $WS_1, \ldots, WS_4$ are set to 1, which means they all provide data by accessing a table that returns exactly one tuple for each input value of the attribute $\{a\}$ (Sect. 3.2). Their service costs are 0.12, 0.16, 0.16, and 0.2 respectively. WS calls are processed in chunk mode with chunk size $|S_c| = 20$. For testing, we varied the number of queries from 100 to 1000. And in each run, the data items contained in queries' input tables are different and therefore we eliminate the impact of sharing WS requests, which is another advantage of our PQModel and the experiments on it will be discussed separately in Sect. 5.2.3.

Figure 7a reports on the average response time of the queries using two different models. Figure 7b reports on the

Table 1 Query templates used in experiments

T_1 : select a, b, c, d, e from $I(a) \bowtie WS_1(a^b, b^f) \bowtie WS_2(a^b, c^f)$ $\bowtie WS_3(a^b, d^f) \bowtie WS_4(a^b, e^f)$
T_2 : select a, b, c from $I(a) \bowtie WS_1(a^b, b^f) \bowtie WS_2(b^b, c^f)$
T_3 : select a, c, d from $I(a) \bowtie WS_1(a^b, c^f) \bowtie WS_2(c^b, d^f)$
T_4 : select b, c, d from $I(b) \bowtie WS_3(b^b, c^f) \bowtie WS_2(c^b, d^f)$

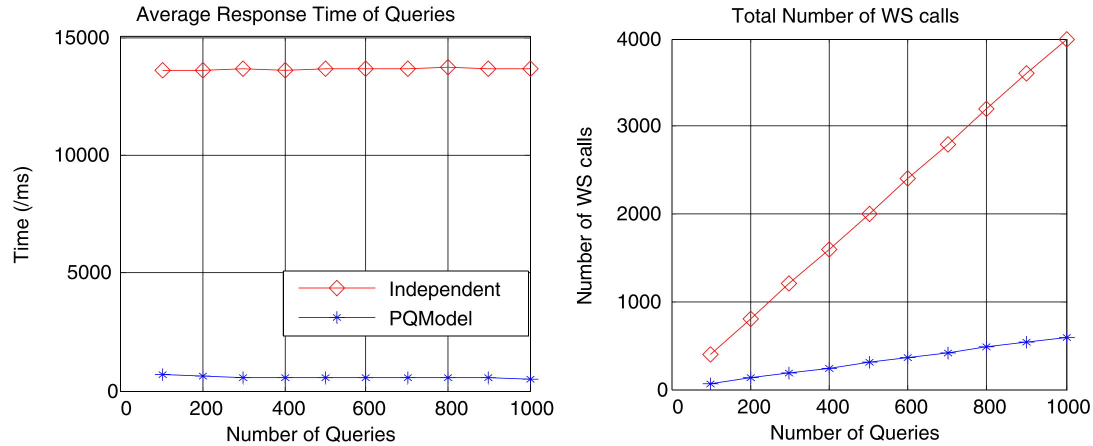


Fig. 7 Processing a collection of small queries complying with the same template

total number of the WS calls incurred by all queries. We can observe that both the required query response time and the incurred number of WS calls of PQModel are consistently and significantly smaller than that of the independent model. This is because the inter-arrival time (0.5 s) between two successive queries is not sufficient to fulfill a query for the independent model, which simply performs four WS calls ($WS_1, \ldots, WS_4$) for each query. However, our PQModel can group concurrent requests from different queries to the same WS together and process them in chunks, which can significantly reduce query response time and the number of incurred WS calls, as proved by the experiment.

5.2.2 Average response time in WS processing

In this experiment, we tested the average response time of queries with large input tables (1000 data items). For testing, we ran two queries Q_1 and Q_2 complying with the same query template T_1. Selectivities of $WS_1, \ldots, WS_4$ are set to 0.8, 0.6, 0.4, and 0.2 respectively. Service costs of $WS_1, \ldots, WS_4$ are 0.12, 0.16, 0.16, and 0.2 respectively. WS calls are processed in chunk mode with chunk size $|S_c| = 20$.

We submitted Q_2 immediately after Q_1. Both Q_1 and Q_2 have an input table with 1000 data items, and the data items contained in queries' input tables are different. We ran Q_1 and Q_2 for ten times. Figure 8 reports the average response time of Q_1 and Q_2 using two different models for each run. We can see that the average response time of PQModel is smaller than that of the independent model.

5.2.3 Effect of sharing WS requests

In this section, a set of experiments conducted to investigate the effectiveness of sharing WS requests are described. We ran a query Q_3 complies with the query template T_2 in

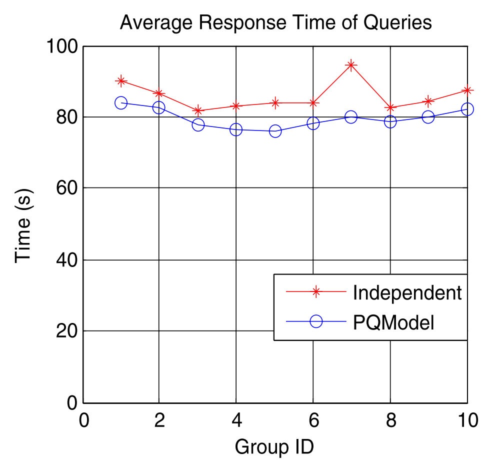


Fig. 8 Processing two large queries complying with the same template

Table 1. Since the output attribute $\{b\}$ of WS_1 is an input attribute of WS_2, thus WS_1 and WS_2 must be sequentially invoked in Q_3.

We setup two WSs for T_2: WS_1 and WS_2, which access the tables $Table_1(a, b)$ and $Table_2(b, c)$ respectively. We set 2000 tuples in both of the $Table_1(a, b)$ and $Table_2(b, c)$. For each tuple t_i $(1 \leq i \leq 2000)$ in $Table_1(a, b)$, the value in the attribute $\{a\}$ is i, the value in the attribute $\{b\}$ is randomly generated from the range 1 to r. For each tuple t_i $(1 \leq i \leq 2000)$ in $Table_2(b, c)$, the values in the attributes $\{b, c\}$ are both i. We set different values 2000, 40 and 80 to r to adjust the opportunities for sharing WS_2 requests. The input tables of Q_3 is $I_3(a)$. We varied $|I_3(a)|$ from 50 to 1000 for testing.

Figure 9a reports the average response time of the items in $I_3(a)$. We can observe that: (1) PQModel performs much better than the independent model when $r = 40$ and $r = 80$. This is because multiple WS_1 requests may generate equivalent inputs for WS_2, in which case multiple WS_2 requests may share the same WS_2 request; (2) in PQModel, gener-

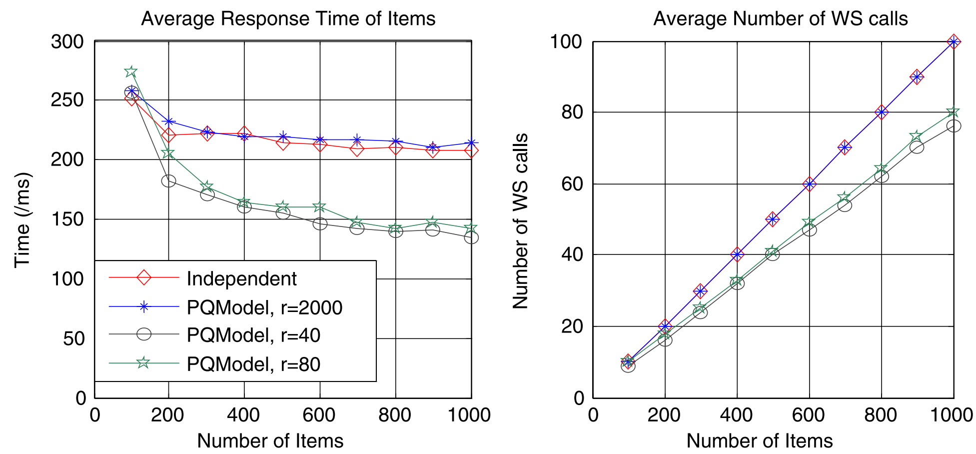


Fig. 9 Effect of sharing (intra-query) WS requests

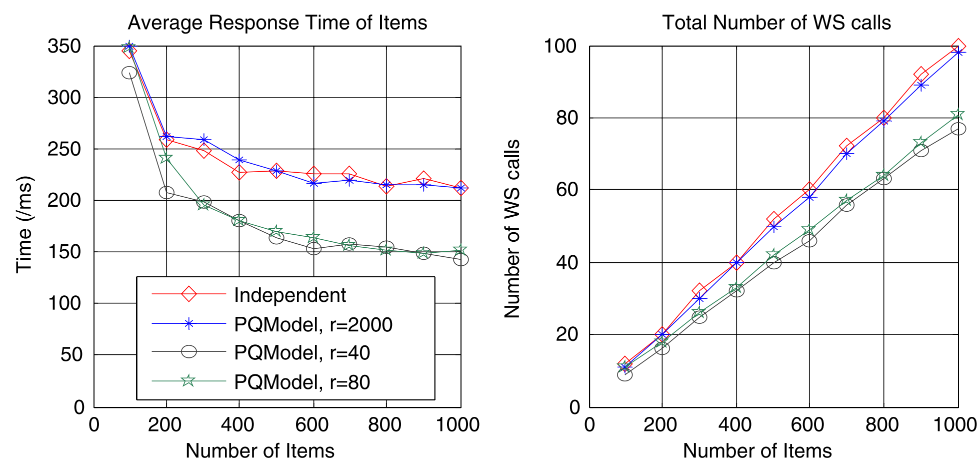


Fig. 10 Effect of sharing (inter-query) WS requests

ally, when r is smaller, the average response time of items is smaller. This is because the number of the opportunities to share WS_2 requests is bigger, when r is smaller.

Figure 9b shows the total number of the WS calls used to process Q_3. We can observe that: (1) the number of the WS calls caused by PQModel is smaller than that of the independent model when $r = 40$ and $r = 80$; (2) in PQModel, the number of the WS calls is smaller when r is smaller.

This experiment investigated the case of sharing WS requests within a single query. Next, we present the experiment conducted to investigate the case of sharing WS requests across multiple queries. We ran two queries Q_4 and Q_5 that follow the query template T_3 and T_4 in Table 1 respectively.

We setup three WSs for T_3 and T_4: WS_1, WS_2 and WS_3. Sharing WS_2 requests may happen when WS_1 and WS_3 generate equivalent inputs for WS_2. Here, we set WS_1 and WS_3 to access the same table $Table_1(a, b, c)$ in the database. We set WS_2 to access a table $Table_2(c, d)$ in the database. We set 2000 tuples in both $Table_1(a, b, c)$ and $Table_2(c, d)$. For each tuple t_i $(1 \le i \le 2000)$ in $Table_1(a, b, c)$, we set its value to be (i, i, j), where j is randomly generated from the range 1 to r. For each tuple t_i $(1 \le i \le 2000)$ in $Table_2(c, d)$, we set its value to be (i, i). Similarly, we adjusted the potential sharing opportunities between Q_4 and Q_5 by setting different values 2000, 40 and 80 to r. The input tables of Q_4 and Q_5 are $|I_4(a)|$ and $|I_5(b)|$. For each value of r, we varied both of the $|I_4(a)|$ and $|I_5(b)|$ from 25 to 500. Q_5 is submitted to the system immediately after Q_4.

Figure 10a reports the average processing time of the items in $|I_4(a)|$ and $|I_5(b)|$. Figure 10b reports the total number of the WS calls used to process Q_4 and Q_5. The number on the x-axis of these figures denotes the value of $|I_4(a)| + |I_5(b)|$. We can see that Figs. 10a and 10b have similar characteristics as Figs. 9a and 9b. Both response time and communication cost can benefit from sharing WS_2 calls. The difference is, in this experiment, multiple requests sharing the same WS_2 call may contain tuples from both Q_4 and Q_5.

5.2.4 *Effect of adaptive query plan modification*

In this section, we present an experiment to investigate the effectiveness of AQP techniques. For testing, we developed an AQP technique for correcting sub-optimal query plans in SenGine. The basic idea is as follows:

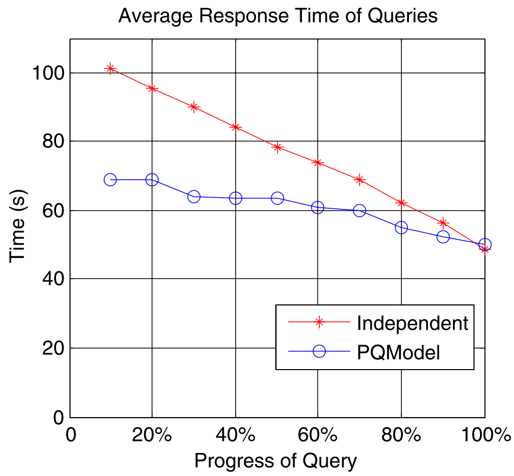


Fig. 11 Effect of adaptive query plan

Every time the Event Handler detects a deviation (larger than a specified threshold) of WS cost, it begins to analyze if the relevant query plans turn to sub-optimal. For each detected suboptimal plan, Event Handler notifies the Tuple Encapsulator the decision of query re-optimization. After receiving the notification, the Tuple Encapsulator triggers the query optimizer to perform query optimization (the optimization algorithm proposed in [30] is used), generates a new tuple descriptor containing new route information for the query, and sets tuples that have not been dispatched or encapsulated to point to the new tuple descriptor.

In this experiment, we ran a query Q_6 complies with the query template T_1 in Table 1. Selectivities of $WS_1, \ldots, WS_4$ are all set to 0.5. The input table of Q_6 contains 1000 data items without duplicate values. The initial service costs of $WS_1, \ldots, WS_4$ are 0.8, 0.12, 0.16, and 0.2 respectively. WS calls are processed in chunk mode with chunk size $|S_c| = 20$. After a certain time (here, we use the progress of query, denoted by p, representing the percentage of Q_6's items that have been dispatched), we changed the cost of WS_1 to 0.2 by adding a delay to each WS_1 call. For testing, we varied p from 10% to 100%.

Figure 11 reports the query processing time of Q_6 of two models. We can observe that: (1) for both models, the required query processing time is more while the cost change of WS_1 happens earlier; (2) PQModel performs much better than the independent model while there is a cost change during query processing (when $p < 100\%$), which means, adaptive query plans of PQModel exhibit improved efficiency compared to static query plans (the independent model).

6 Related work

Several streams of research related to our work are discussed in this section.

6.1 Answering queries over Web services

Query over WSs is first defined as a SQL-like query in WSMS [30], in which the authors focus on query optimization: arranging a query's WS calls into a pipelined execution plan to optimally exploit parallelism. PQModel differs from WSMS in two ways: (1) WSMS executes queries independently without sharing WSPs; however PQModel allows multiple queries to share WSPs. (2) WSMS uses static query plans for query processing; but PQModel uses adaptive query plans. Therefore, our PQModel performs better in terms of query efficiency and resources usage. In [31], an approach for integrating information from multiple bioinformatics data sources and services is proposed, where a data-flow execution model is applied. As opposed to our PQModel, the approach does not exploit data/computation sharing and AQP techniques. Instead, it investigates on constraints to reduce the access to the WSs. Multi-domain queries considered in [9] also need to query over two kinds of WSs: exact services and search services. The work presented in [9] focuses on query optimization rather than execution, which is our research point. Besides, we plan to distinguish between exact services and search services during query evaluation in the future work.

6.2 Data/computation sharing techniques

Data/computation sharing techniques have been widely studied in the context of traditional DBMS [18, 28], data integration systems [12, 26], and data stream systems [20]. Our technique is most closely relevant to QPipe [18]: a simultaneously pipelined query evaluation paradigm of RDBMS. QPipe changes the query engine philosophy from query-centric (one-query, many-operators) to operator-centric (one-operator, many-queries); thereby it can proactively detect common data and computation at execution time so that sharing could be possible. This is also what our PQModel wants to take advantage of; therefore higher query efficiency and better resource usage can be facilitated. QPipe considers RDBMS queries but PQModel considers service-oriented queries. QPipe exploits common data in relational operators while PQModel exploits sharable WS requests and calls in WSPs. Other data sharing techniques in RDBMS include: buffer pool management [27], result caching [12, 29] and multiple-query optimization (MQO) [26, 28]. Buffer pool management is not suitable for our context since PQModel uses network-based processing rather than disk-based processing. Result caching, which actually can be used in our context to cache results of WSs or queries with high reference frequencies and low maintenance costs, and MQO techniques, which can also be used in our context to identify reusable WS requests by generating a global query plan for a batch of queries in the phase of query optimization, will be considered in the future.

6.3 Adaptive query processing (AQP)

There is a large body of work on AQP techniques [6, 7, 14, 17, 19, 21–23, 32]; [7, 14] are two comprehensive surveys of the classical AQP techniques. Many proposed AQP techniques can be used in PQModel. First, the join can be realized as MJoin [32], a multi-way stream join algorithm that can adaptively spill overflowing inputs to disk and later join them to produce the final output. Second, the approach proposed in [8] can be used in PQModel to generate multiple query plans for each query, thus different tuples with different data properties in the query can be evaluated by different query plans. Third, the approach of interleaving planning and execution (e.g., Tukwila [19]) can be used in PQModel, thus PQModel can trigger re-optimization while the current query plan turns to suboptimal. Fourth, operator reconfiguration (e.g., [22, 23]) can be used in PQModel to adapt to workload imbalance and changes in resource availability.

6.4 Data-flow execution models

A number of data stream systems (e.g., CACQ [20] and Aurora [3]) have been developed along with data-flow processing models. CACQ [20] was implemented based on the eddy query processing framework [6], which enables very fine-grained adaptivity by routing each tuple adaptively across operators to process it. Besides, CACQ also provide sharing mechanisms. First, in CACQ, the path that each tuple takes through the operators is explicitly encoded in the tuple as tuple lineage, which enables sharing of operators between queries. Second, a predicate indexing operator called grouped filter is designed in CACQ to share selections. Third, unary operators called SteMs (State Modules) are adopted in CACQ to share Joins.

7 Conclusion and future work

Pervasive computing environments adopting a Service-Oriented Architectures need to query over Web services to combine data from multiple data sources. Query over Web services could be expensive. Some works have been done to improve query efficiency and better utilize resources by generating optimal query plans. Our PQModel however takes another way to achieve same objective: optimizing the process of query execution over Web services (i.e., optimizing the execution of query plans).

PQModel has two features. First, it is data-flow execution model. Concurrent queries in PQModel are able to share WSPs. Data/computation sharing detecting mechanism is designed in WSP for improving query efficiency and resource utilization. Second, PQModel adopts an adaptive framework. WSPs of PQModel are able to monitor running information during query plan. Event Handler is able to assess running information and identify events. The components of Tuple Encapsulator, Thread Allocator and WSPs are able to respond to events. Therefore, various AQP strategies can be developed in PQModel to lead to higher query efficiency. A set of experiments were conducted to evaluate the effectiveness of sharing and adaptivity. The experiment results clearly demonstrate that our PQModel can achieve performance improvement in terms of response time and network overhead.

In the near future, we plan to introduce and develop various AQP techniques based on the adaptive framework of our model.

Acknowledgements This work is supported by ChinaGrid project of Ministry of Education of China, Natural Science Foundation of China (60573110, 90612016, 60673152), National Key Basic Research Project of China (2004CB318000, 2003CB317007), National High Technology Development Program of China (2006AA01A108, 2006AA01A111, 2006AA01A101, 2004CB318000,), EU IST programme and Asia Link programme.

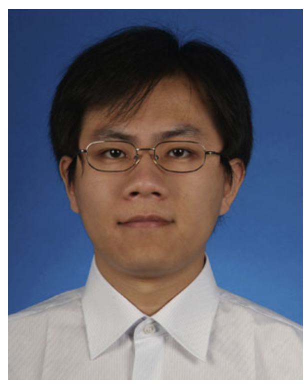
References

1. Apache axis. http://ws.apache.org/axis/
2. Apache Tomcat. http://tomcat.apache.org/
3. Abadi, D., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora: a new model and architecture for data stream management. Int. J. VLDB **12**(2), 120–139 (2003)
4. Acharya, S.: Application and infrastructure challenges in pervasive computing. In: NSF Workshop on Context-Aware Mobile Database Management (CAMM), January, 2002
5. Altinel, M., Brown, P., Cline, S., Kartha, R., Louie, E., Markl, V., Mau, L., Ng, Y.H., Simmen, D., Singh, A.: Damia—A data mashup fabric for intranet applications. In: Proc. of the 33rd Int. Conf. on Very Large Data Bases (VLDB) (2007)
6. Avnur, R., Hellerstein, J.: Eddies: Continuously adaptive query processing. In: Proc. of the 19th ACM SIGMOD Int. Conf. Management of Data (2000)
7. Babu, S., Bizarro, P.: Adaptive query processing in the looking glass. In: Conf. on Innovative Data Systems Research (CIDR) (2005)
8. Bizarro, P., Babu, S., DeWitt, D., Widom, J.: Content-based routing: Different plans for different data. In: Proc. of the 31st Int. Conf. on Very Large Data Bases (VLDB) (2005)
9. Braga, D., Ceri, S., Daniel, F., Martinenghi, D.: Optimization of multi-domain queries on the Web. In: Proc. of the 34th Int. Conf. on Very Large Data Bases (VLDB) (2008)
10. Chappell, D.A., Jewell, T.: Java Web Services. O'Reilly, 2002
11. Cherniack, M., Franklin, M.J., Zdonik, S.B.: Data management for pervasive computing. In: Proc. of the 27th Int. Conf. on Very Large Data Bases (VLDB) (2001)
12. Chidlovskii, B., Borghoff, U.M.: Semantic caching of Web queries. Int. J. VLDB **9**(1), 2–17 (2000)
13. Conti, M., Kumar, M., Das, S.K., Shirazi, B.A.: Quality of service issues in Internet Web services. IEEE Trans. Comput. **51**(6), 593–594 (2002)
14. Deshpande, A., Ives, Z.G., Raman, V.: Adaptive query processing. Found. Trends Databases **1**(1), 1–140 (2007)
15. Florescu, D., Levy, A., Manolescu, I., Suciu, D.: Query optimization in the presence of limited access patterns. In: Proc. of the 18th ACM SIGMOD Int. Conf. Management of Data (1999)

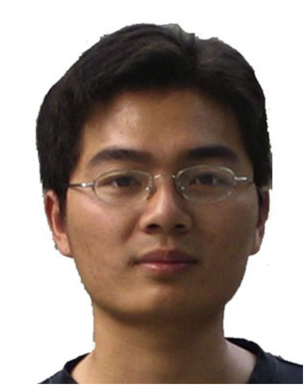
16. Gounaris, A.: Resource aware query processing on the grid. PhD thesis, School of Computer Science of the University of Manchester (2005)
17. Haas, P.J., Hellerstein, J.M.: Ripple joins for online aggregation. In: Proc. of the 18th ACM SIGMOD Int. Conf. Management of Data (1999)
18. Harizopoulos, S., Shkapenyuk, V., Ailamaki, A.: : Qpipe: a simultaneously pipelined relational query engine. In: Proc. of the 24th ACM SIGMOD Int. Conf. Management of Data (2005)
19. Ives, Z.: Efficient query processing for data integration. PhD thesis, University of Washington (2002)
20. Madden, S.R., Shah, M.A., Hellerstein, J.M., Raman, V.: Continuously adaptive continuous queries over streams. In: Proc. of the 21st ACM SIGMOD Int. Conf. Management of Data (2002)
21. Markl, V., Raman, V., Simmen, D., Lohman, G., Pirahesh, H.: Robust query processing through progressive optimization. In: Proc. of the 23rd ACM SIGMOD Int. Conf. Management of Data (2004)
22. Pang, H., Carey, M.J., Livny, M.: Partially preemptive hash joins. In: Proc. of the 12th ACM SIGMOD Int. Conf. Management of Data (1993)
23. Pang, H., Carey, M.J., Livny, M.: Memory-adaptive external sorting. In: Proc. of the 19th Int. Conf. on Very Large Data Bases (VLDB) (1993)
24. Petropoulos, M., Deutsch, A., Papakonstantinou, Y.: Interactive query formulation over Web service-accessed sources. In: Proc. of the 25th ACM SIGMOD Int. Conf. Management of Data (2006)
25. Quzzani, M.: Efficient delivery of Web services. PhD thesis, Virginia Polytechnic (2004)
26. Rubao, L., Minghong, Z., Huaming, L.: Request Window: An approach to improve throughput of RDBMS-based data integration system by utilizing data sharing across concurrent distributed queries. In: Proc. of the 33rd Int. Conf. on Very Large Data Bases (VLDB) (2007)
27. Sacco, G.M., Schkolnick, M.: Buffer management in relational database systems. ACM TODS **11**(4), 473–498 (1986)
28. Sellis, T.K.: Multiple query optimization. ACM Trans. Database Syst. **13**(1), 23–52 (1988)
29. Sivasubramanian, S., Pierre, G., Steen, M.V., Alonso, G.: Analysis of caching and replication strategies for Web applications. IEEE Internet Comput. **11**(1), 60–66 (2007)
30. Srivastava, U., Munagala, K., Widom, J., Motwani, R.: Query optimization over Web services. In: Proc. of the 32nd Int. Conf. on Very Large Data Bases (VLDB) (2006)
31. Thakkar, S., Ambite, J.L., Knoblock, C.A.: Composing, optimizing, and executing plans for bioinformatics web services. Int. J. VLDB **14**(3), 330–353 (2005)
32. Viglas, S., Naughton, J.F., Burger, J.: Maximizing the output rate of multi-join queries over streaming information sources. In: Proc. of the 29th Int. Conf. on Very Large Data Bases (VLDB) (2003)
33. Weiser, M.: The computer for the twenty-first century. Sci. Am. **265**(3), 94–104 (1991)
34. Yu, Q., Bouguettaya, A.: Framework for Web service query algebra and optimization. ACM Trans. Web (TWEB) **2**(1), 1–35 (2008)

Yongwei Wu received his Ph.D. degree in the Applied Mathematics from the Chinese Academy of Sciences in 2002. Since then, he has worked at the Department of Computer Science and Technology, Tsinghua University, as research Assistant Professor from 2002 to 2005, and Associate Professor since 2005. His research interests include grid and cloud computing, distributed processing, and parallel computing.

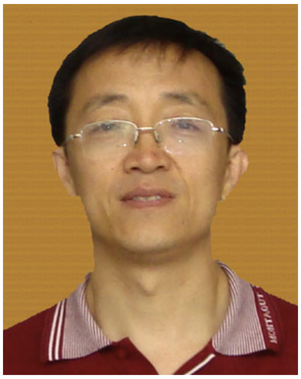
Jia Liu is a Ph.D. student in Computer Science and Technology Department, Tsinghua University. Her research interests include data grid, distributed data management, heterogeneous data integration and data mining.



Gang Chen is a Ph.D. student of Department of Computer Science and Technology, Tsinghua University. His research interest mainly focus on distributed and parallel computing system.



Qiming Fang is a Ph.D. student at Department of Computer Science and Technology, Tsinghua University, Beijing, China. His research interests include semantic web and ontology engineering, distributed computing and information retrieval.



Guangwen Yang is Professor of Computer Science and Technology, Tsinghua University, China. He is also an expert on "high-performance computer and its core software", the National "863" Program of China. He is mainly engaged in the research of grid computing, parallel and distributed processing and algorithm design and analysis.