Adaptive Parallelization of Queries over Dependent Web Service Calls

Manivasakan Sabesan and Tore Risch

Department of Information Technology, Uppsala University
Sweden

msabesan@it.uu.se
Tore.Risch@it.uu.se

***Abstract*— We have developed a system to process database queries over composed data providing web services. The queries are transformed into execution plans containing an operator that invokes any web service for given arguments. A common pattern in these query execution plans is that the output of one web service call is the input for another, etc. The challenge addressed in this paper is to develop methods to speed up such *dependent* calls in queries by parallelization. Since web service calls incur high-latency and message set-up costs, a naïve approach making the calls sequentially is time consuming and parallel invocations of the web service calls should improve the speed. Our approach automatically parallelizes the web service calls by starting separate *query processes,* each managing a parameterized sub-query, a *plan function*, for different parameter tuples. For a given query, the query processes are automatically arranged in a multi-level process tree where plan functions are called in parallel. The parallel plan is defined in terms of an algebra operator, *FF_APPLYP,* to ship in parallel to other query processes the same plan function for different parameters. By using *FF_APPLYP* we first investigated ways to set up different process trees manually. We concluded from our experiments that the best performing query execution plan is an almost balanced bushy tree. To automatically achieve the optimal process tree we modified *FF_APPLYP* to an operator *AFF_APPLYP* that adapts a parallel plan locally in each query process until an optimized performance is achieved. *AFF_APPLYP* starts with a binary process tree. During execution each query process in the tree makes local decisions to expand or shrink its process sub-tree by comparing the average time to process each incoming tuple. The query execution time obtained with *AFF_APPLYP* is shown to be close to the best time achieved by manually built query process trees.**

I. Introduction

There is a common need to search information supplied by *data providing web services* that return a set of objects for a given set of parameters without any side effects. For example, consider a query to find *USAF Academy's Zip code* and the *State* where it is located. The three different data providing web service calls in this query are *GetAllStates* [3] to retrieve all the states, *GetInfoByState* [19] to get all the Zip codes within a given state, and *GetPlacesInside* [4] to provide all the places having a given Zip code. A naïve implementation of the example query makes 5000 calls sequentially and takes nearly 2400 seconds to execute. The reason is that each web service call incurs high latency and message set-up costs.

Queries calling data providing web services often have a similar pattern where the output (e.g. state) of one web service call is the input for another web service call (e.g. *GetInfoByState*), i.e. the second call is *dependent* on the first one, etc. A challenge here is to develop methods to speed up queries requiring such dependent web service calls.

In our approach a web service call is considered as an expensive function call where the result is a collection. It is likely that making parallel invocations of such calls will speed up the performance of queries with several dependent web service calls. To improve the response time, we present an approach to parallelize the web service calls while keeping the dependencies among them. With the approach separate *query processes* are started in parallel, each calling a parameterized sub query, called a *plan function,* for a stream of parameter tuples. Each plan function encapsulates a web service call.

The approach is implemented in the *Web Service MEDiator (WSMED)* system [15] that extends a main memory functional DBMS [14] with primitives to call web services. WSMED enables general query capabilities over data accessible through any data providing web service by reading the WSDL meta-data description. Queries are expressed in SQL. To enable simple queries to complex collections returned by web services, WSMED automatically generates flattened views of the result collections as tables.

For a given query the WSMED optimizer first produces a non-parallel plan where web service operations are called as functions. The query processor then automatically reformulates the non-parallel plan into a parallel one where web service operations are called in parallel while keeping the required dependency among the calls. The algebra operator, *FF_APPLYP* (First Finished Apply in Parallel), ships a plan function in parallel to other query processes and then calls the shipped plan function in parallel for a stream of parameter tuples.

Multi-level execution plans are generated with several layers of parallelism in different query processes. This forms the *process tree* for the query. Each child query process delivers back the result data from the shipped plan function to its parent process asynchronously. The number of children processes below a parent query process is called its *fanout.* During execution a coordinator query process first initiates the communication with its child query processes and then ships

1084-4627/09 $25.00 © 2009 IEEE
DOI 10.1109/ICDE.2009.148

IEEE computer society

in parallel to the children their plan functions. Then a stream of different parameter tuples for the plan functions is shipped in parallel to the children. At any point in time every process in the tree executes one plan function for a specific parameter tuple. The results from the children are delivered to the parent in parallel as streams.

The performance is often improved by setting up several web service calls to the same operation in parallel rather than to call the operation in sequence for different parameters. Normally there is an optimal number of parallel calls for a given web service operation. It is therefore important to figure out an optimized process tree for an execution plan by automatically arranging the available query processes for best performance. We first evaluated *FF_APPLYP* for different process trees by setting different fanouts manually. We tested flat and bushy process trees over existing real web services. Based on the experiments we concluded that a process tree rather close to a balanced tree performed best.

The exact properties of the composed web service operations and the computing environments involved in the calls are usually unknown. Therefore an optimal process tree is very difficult to produce using traditional query optimization assuming a cost-model describing these properties. WSMED therefore adaptively achieves an optimized process tree by run-time monitoring of the plan function calls. For the adaptation we modified *FF_APPLYP* to an operator *AFF_APPLYP* that dynamically modifies a parallel plan locally and greedily in each query process. We compared the operator *AFF_APPLYP* to the process tree with best effort manual process arrangement.

In summary the contributions of our work are:

- We define an algebra operator *FF_APPLYP* to distribute a plan function among child query processes for parallel calls with different parameter tuples.
- An algorithm is implemented to transform a central plan into a parallel plan by introducing *FF_APPLYP* operators calling plan functions encapsulating each web service call.
- Experiments with using *FF_APPLYP* showed that the best execution time for queries with dependent joins is achieved with a bushy tree rather close to a balanced one.
- To automatically optimize the parallel plan, we developed another algebra operator *AFF_APPLYP* that locally adjusts an initial balanced binary process tree adaptively until best performance is obtained.

The rest of this paper is organized as follows. In Section 2, we provide a motivating scenario used in experiments in terms of existing web services. Query process arrangements using *FF_APPLYP* are presented in Section 3. The query processing details are explained in Section 4. Experimental results and the *AFF_APPLYP* operator are presented in Section 5. Related work is analyzed in Section 6, and Section 7 summarizes and indicates future directions.

II. Motivating Scenario

The class of queries we consider here is dependent-join [7] queries, which in their simplest form can be expressed as:

$$f(x-, y+) \wedge g(y-, z+)$$

The predicate *f* binds *y* for some input value *x* and passes each *y* to the predicate *g* that returns the bindings of *z* as result. Thus, *g* depends on the output of *f*. The predicates *f* and *g* represent calls to parameterized sub queries, which in our case are execution plans encapsulating data providing web service operations. Inputs parameters are annotated with '-' and outputs with '+'.

We made experiments with two different queries calling different web service operations provided by different publicly available service providers.

A. Query1

In the first test case we used the SQL *Query1* in Fig. 1 that finds information about places located within 15 km from each city whose name starts with 'Atlanta' in all US states. In the query we utilize the web service operations *GetAllStates* [3], *GetPlacesWithin* [3], and *GetPlaceList* [17]. For a given web service WSMED automatically generates *operation wrapper functions* (OWFs) based on the WSDL definitions of the web service operations. Each OWFs encapsulates a data providing web service operation for given parameters and emits the result as a flattened stream of tuples. Each OWF defines an SQL view of a web service operation. SQL queries can be made over these views with the restriction that the input values of the OWFs must be known in the query. In Fig. 1 the three OWFs *GetAllStates, GetPlacesWithin,* and *GetPlaceList* define views encapsulating web service operations with the same names. The query returns a stream of 360 result tuples. A naïve central sequential execution plan invokes more than 300 web service calls.

```
Select   gl.placename,gl.state
From     GetAllStates gs, GetPlacesWithin gp,
         GetPlaceList gl
Where    gs.State=gp.state and gp.distance=15.0
         and gp.placeTypeToFind='City' and
         gp.place='Atlanta' and
         gl.placeName=gp.ToPlace+' ,'+gp.ToState
         and gl.MaxItems=100 and
         gl.imagePresence='true'
```

Fig. 1 *Query 1* defined in SQL

The OWF *GetAllStates* presents information of US states as a set of tuples <*name*, *type*, *state*, *latDegrees*, *lonDegrees*, *latRadians*, *lonRadians*>. However, we are only interested in the values of the attribute *State*. The OWF *GetPlacesWithin* returns a set of tuples <To*City, ToState, GeoPlaceDistance_Distance*> for given place (*'Atlanta'*), state (*gs.State*), distance (*15.0*), and kind of place type to find (*'City'*). The OWF *GetPlaceList* retrieves a set of places <*placename, state, country, placeLon, placeLat, availableThemeMask, placeTypeId*, *population*> given a specification of a place (concatenate To*City+', '+ToState*), the maximum number result tuples (*100*), and a flag indicating whether places having an associated map are returned.

Fig. 2 shows the automatically generated OWF *GetAllStates,* which flattens the result from the web service operation named *GetAllStates*. An OWF is generated based on the WSDL definition of a web service operation. Any web service operation can be invoked by the built-in function *cwo* (line 14). Its parameters are the URI of the WSDL document that describes the service, the name of the service, the operation name, and the input parameter list for the operation. The web service operation *GetAllStates* has no input parameters ({}).

```
1.   create function GetAllStates()-> Bag of
            <Charstring name, Charstring type,
             Charstring state, Real latDegrees,
             Real lonDegrees, Real latRadians,
             Real lonRadians> as
2.   select  GeoPlaceDetails['Name'],
3.           GeoPlaceDetails['Type'],
4.           GeoPlaceDetails['State'],
5.           GeoPlaceDetails['LatDegrees'],
6.           GeoPlaceDetails['LonDegrees'],
7.           GeoPlaceDetails['LatRadians'],
8.           GeoPlaceDetails['LonRadians']
9.   from    Sequence out,
10.          Record GetAllStatesResult ,
11.          Record GetAllStatesResult1,
12.          Sequence GetAllStateResult2,
13.          Record GeoPlaceDetails
14.  where   out=cwo('http://codebump.com/services
             /PlaceLookup.wsdl', 'GeoPlaces',
             'GetAllStates', {})and
15.          GetAllStatesResult1 in out and
16.          GetAllStatesResult2 =
             GetAllStatesResult1
             ['GetAllStatesResult']and
17.          GetAllStateResult in
             GetAllStatesResult2 and
18.          GeoPlaceDetails=GetAllStatesResult['G
             eoPlaceDetails'];
```

Fig. 2 Automatically generated OWF *GetAllStates*

The result from *cwo* is bound to the query variable *out* (line 14). It holds an object representing the output from the web service operation temporarily materialized in WSMED's local store. The OWF converts the output XML structure from the web service operation call into records and sequences. The result *out* is here a sequence from which elements are extracted (line 15) into the *GetAllStatesResult1* record structure using the *in* operator. The records have only one attribute named *GetAllStatesResult* whose values are assigned to another sequence structure *GetAllStatesResult2* (line 16). An attribute *a* of a record *r* is accessed using the notation *r[a]*. Each element record from the sequence *GetAllStatesResult2* is bound to the variable *GetAllStateResult* (line 17). The values of the attribute *GeoPlaceDetails* are assigned to the *GeoPlaceDetails* record with attributes *Name, Type, State, LatDegrees, LonDegrees, LatRadians,* and *LonRadians* (line 18). The OWFs *GetPlacesWithin* and *GetPlaceList* are automatically generated analogously.

B. *Query2*

The second case, *Query2* in Fig. 3, finds the zip code and state of the place 'USAF Academy'. A naïve sequential plan invokes more than 5000 web service calls. Here also three different dependent web services are involved. *GetAllStates* is the same as in *Query1*. *GetInfoByState* is provided by the USZip [19] web service to retrieve all zip codes for a given state as a single comma separated string (*gi.GetInfoByStateResult*). *getzipcode* is an helping function defined in WSMED that extracts the set of zip codes (*gc.zipcode*) given a string of zip codes (*gc.zipstr*). The OWF *GetPlacesInside* is supported by the Zipcodes [4] web service provider and returns for a given zip code a set of tuples <*ToPlace, ToState, Distance*> where *ToPlace* is a place located within the zip code area, *ToState* is the state of the place, and *Distance* is the distance from the place to the origin of the given zip code area.

```
select   gp.ToState, gp.zip
From     GetAllStates gs, GetInfoByState gi,
         getzipcode gc, GetPlacesInside gp
Where    gs.State=gi.USState and
         gi.GetInfoByStateResult=gc.zipstr and
         gc.zipcode=gp.zip and
         gp.ToPlace='USAFAcademy'
```

Fig. 3 *Query2* defined in SQL

III. WSMED Process Arrangement

The web service metadata in a WSDL document is first imported and stored in the WSMED local database [15]. A query is processed by a coordinator process *q0.* Fig. 4 gives an example of a process tree generated by the WSMED query optimizer. Every query process on each level can be connected with a number of child processes and all the processes on the same level execute the same plan function but with different parameters.

In Fig. 4, *q1* is connected with *q3, q4,* and *q5.* The plan function in the coordinator *q0* encapsulates the OWF *GetAllStates*, while the plan functions of the processes in level one encapsulate the OWF *GetPlacesWithin* for different states. On level two the plan function calls the OWF *GetPlaceList* for different place specifications.

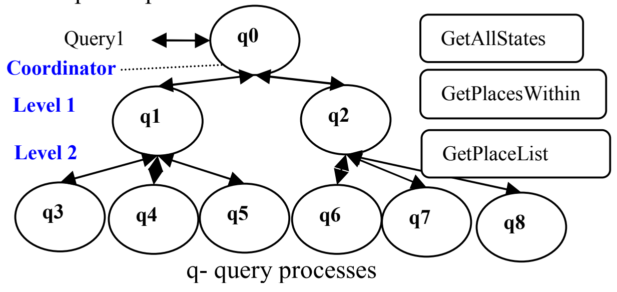

Fig. 4 Process tree

The coordinator *q0* first generates a central plan containing calls to the OWFs. It then automatically reformulates the central plan to incorporate parallel web service calls by inserting algebra operators *FF_APPLYP* in the execution plan whenever an OWF is encountered. For each OWF a plan function is generated that encapsulates a fragment of the central execution plan embodying the OWF call. When the algebra operator *FF_APPLYP* is executed in process *q0,* it first ships in parallel to its children in level one (*q1,q2*) the same plan function definition that encapsulates *GetPlacesWithin*. Then it ships in parallel different parameter tuples to the shipped plan function installed in the children processes ready for execution. Analogously, the *FF_APPLYP* operators executing in the level one processes send another plan function definition to the level two processes (*q3,q4,q5,q6,q7,q8*). Each query process initially receives its own plan function definition once before execution. When the level two processes receive data from the wrapped web service operation *GetPlaceList*, the results will be returned asynchronously as streams to the processes in level one, and finally the results are streamed to the coordinator process.

A. FF_APPLYP

The operator *FF_APPLYP* enables parallel invocation of a plan function for different parameter tuples delivered as an input stream to *FF_APPLYP*. *FF_APPLYP* has the signature: *FF_APPLYP(Function pf, Integer fo, Stream pstream) → Stream result*

It ships in parallel to *fo* number of child query processes the definition of the same plan function *pf.* Then it ships one by one parameter tuples from *pstream* to each of the children. The result stream from a call to *pf* for a given parameter tuple is sent back to *FF_APPLYP* asynchronously as a stream of tuples, *result*.

In our first experiments the fanout *fo* is set manually for each level. This allows us to analyze different process trees. In Fig. 4 the fanout on level one is $fo_1=2$ and on level two $fo_2=3$. The coordinator *q0* at level zero first initializes the two child processes *q1* and *q2*. Then *q0* ships the plan function encapsulating the web service operation *GetPlacesWithin* to the children (*q1, q2*). When all plan functions are shipped it starts picking parameter tuples one by one from *pstream*, to send down to the plan function started in the children. In *q0* the stream *pstream* is a stream of state names produced as the result of the plan function that encapsulates the web service operation *GetAllStates*. When the first round of parameter tuples are shipped to all children, *FF_APPLYP* will broadcast that it is ready to receive results. Whenever a result tuple is received from some child it is directly emitted as a result of *FF_APPLYP*. When a child completed the processing of a plan function for a given parameter tuple it sends an *end-of-call* message to *FF_APPLYP*. When the parent receives an end-of-call message from a child it will ship the next pending parameter tuple from *pstream* to the idle child process. When there are no pending parameter tuples in *pstream* and no pending results from the child processes, *FF_APPLYP* is finished.

IV. QUERY PARALLELIZATION IN WSMED

Fig. 5 illustrates the query processor in WSMED [15]. The *calculus generator* produces from a given user query defined in SQL an internal calculus expression in a Datalog dialect [13]. The symbol '_' represents an anonymous result variable.

Query1 is transformed into the following calculus expression:

```
 Query1(pl,st) :-
        GetAllStates()                     AND
        GetPlacesWithin('Atlanta',_,
                       15.0,'City')        AND
        GetPlaceList(_, 100,'true')
```

With naïve query optimization the calculus expression is translated by the *central plan creator* into the algebra expression in Fig. 6. The central plan creator uses a simple heuristic web service cost model based on the signatures of web service operations assuming that web service operations are expensive. The algebra expressions contains calls to the *apply* operator γ [6], which applies a plan function for a given parameter tuple. The naïve central query execution plan with γ can be directly interpreted but with very bad performance since many web service operations are applied in sequence.

The plan first executes the OWF *GetAllStates* returning a stream of tuples <*st1*>. Each of these tuples are fed to the next OWF *GetPlacesWithin* called by the apply operator with the given argument tuple *('Atlanta', st1, 15.0, 'City')* returning a stream of tuples <*city, st2*>. The built in function *concat* is then applied on each argument tuple *(city,',',st2)* producing a stream of strings *str*. Finally the OWF *GetPlaceList* is applied on each argument tuple *(str,100,'true')* returning a stream of tuples <*pl,st*>.

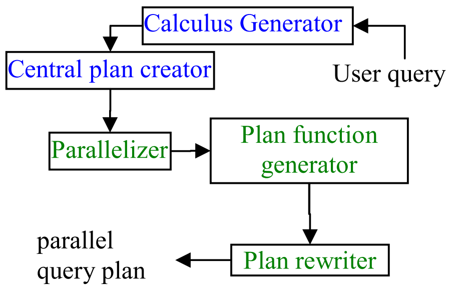

Fig. 5 Query Processor

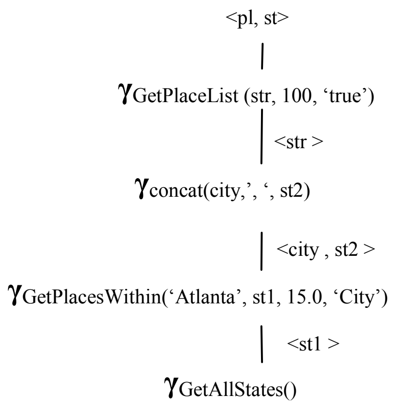
<pl, st>
γGetPlaceList (str, 100, 'true')
<str >
γconcat(city,', ', st2)
<city , st2 >
γGetPlacesWithin('Atlanta', st1, 15.0, 'City')
<st1 >
γGetAllStates()

Fig. 6 Central query plan - *Query1*

The *parallelizer* in Fig. 5 takes as input a central plan (e.g. the one in Fig. 6) and identifies there the parallelizable OWFs. Since the parallelization is based on parameter streams, OWFs not having input parameters are not considered. For example, the plan in Fig. 6 can be parallelized for the OWFs *GetPlacesWithin* and *GetPlaceList*, but not for *GetAllStates.* The parallelizer splits the plan into one section for each parallelizable OWF starting from the bottom. The first section, flattening the result from the call to the web service operation *GetAllStates*, is executed in the coordinator. The next section contains the calls to *GetPlacesWithin* and *concat*. The final section contains only the call to *GetPlaceList*.

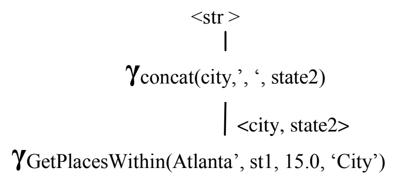
<str >
γconcat(city,', ', state2)
<city, state2>
γGetPlacesWithin(Atlanta', st1, 15.0, 'City')

Fig. 7.Plan function *PF1* wrapping *GetPlacesWithin*

For each parallelizable section the *plan function generator* creates a plan function that encapsulates a parallelizable call to an OWF. For example, the plan function *PF1* in Fig. 7 encapsulates the OWF *GetPlacesWithin*. It has the signature *PF1(Charstring st1) → Stream of Charstring str*. Analogously *PF2* in Fig. 8 flattens the web service operation *GetPlaceList* to return a stream of tuples <*pl, st*> and has the signature *PF2(Charstring str) → Stream of <Charstring pl, Charstring st>*.

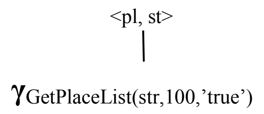

Fig. 8 Plan function *PF2* wrapping *GetPlaceList*

Finally, the *plan rewriter* transforms the central query by inserting the algebra operator *FF_APPLYP* for each generated plan function. Fig. 9 shows the final parallelized execution plan with two calls to *FF_APPLYP* (*FF_γ*).

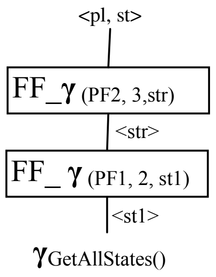

Fig. 9.Parallel execution plan-*Query1*

Analogously *Query2* is initially compiled into the central plan in Fig. 10. The central plan first executes the OWF *GetAllStates* to return a stream of tuples <*st1*>. These outputs are fed to the next OWF *GetInfoByState* returning a stream of single comma separated strings *zstr*. For each *zstr* the γ operator applies the user defined helping function *getzipcode* to produce a stream of extracted zip codes *zc*. Then the OWF *GetPlacesInside* is applied for each *zc* returning a stream of tuples <*st, pl, zc*>. Finally the *equal* function is applied to check if *pl* is equal to '*USAF Academy*' and returns stream of valid tuples <*st ,zc*>.

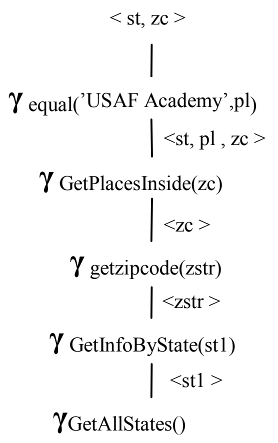

Fig. 10 Central query plan- *Query2*

The parallelizer splits the first parallelizable section (call to OWF *GetAllStates*) to execute in the coordinator. The next parallelizable section contains the calls to *GetInfoByState* and *getzipcode*. The final section contains only the call to *GetPlacesInside* and *equal*. Then the plan function generator creates plan functions to encapsulate the parallelizable OWFs. The plan function *PF3* in Fig. 11 encapsulates *GetInfoByState*. It has the signature:
PF3(Charstring st1) → Stream of Charstring zc.

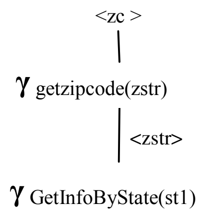

Fig. 11 Plan function *PF3* wrapping *GetInfoByState*

PF4 in Fig. 12 wraps the OWF *GetPlacesInside* and returns <*st,zc*>. It has the signature:
PF4(Charstring zc) → <Charstring st, Charstring zc>.

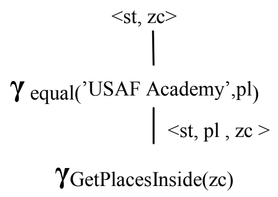

Fig. 12 Plan function *PF4* wrapping *GetPlacesInside*

Finally, the plan rewriter transforms the central query by inserting *FF_γ* for each generated plan function as illustrated in Fig. 13.

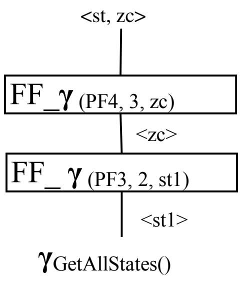

Fig. 13 Parallel execution plan-*Query2*

V. Experiments

We compared the query execution times for *Query1* using the central execution plan in Fig. 6 with the parallel plan in Fig. 9 (for *Query2* we compare the plans in Fig. 10 and Fig. 13). To analyze different process trees, we set manually a *fanout vector* with fanouts for the different process tree levels to evaluate the query execution times. The tests were run on a computer with a 3 GHz single processor Intel Pentium 4 with 2.5GB RAM. We evaluated the following process trees:

- *Flat tree* (Fig. 14): The fanout vector has fo_2=0 ($\{fo_1,0\}$) in which case both OWFs are combined into the same plan function executed at the same level.
- *Unbalanced tree* (Fig. 15): Fanout vector $\{fo_1,fo_2\}$, $fo_1 \neq fo_2$
- *Balanced tree*: the fanouts are equal, i.e. $fo_1 = fo_2$

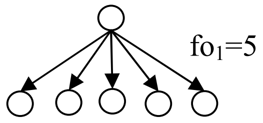

Fig. 14 Flat tree

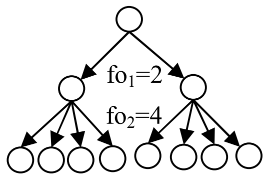

Fig. 15 Unbalanced tree

The total number of query processes N needed to execute the parallel queries is $N = fo_1 + fo_1 * fo_2$.

In general, there should be an optimum shape of the process tree based on properties of the web service calls, which are not known. The experiments investigate the optimum tree topology for up to 60 query processes.

Fig. 16 illustrates the execution times in seconds for *Query1* by varying the values of fo_1 and fo_2. It shows the lowest execution time region is achieved within the range 50 - 60 sec. The fastest execution time 56.4 sec for fanout vector *{5,4}* outperformed with speedup 4.3 the central plan (244.8 sec). Fig. 17 shows that the best execution time for *Query2* is achieved within the range of 1200-1400 sec. The best execution time 1243.89 sec for fanout vector *{4,3}* outperformed with speed up of nearly 2 the central plan (2412.95 sec).

We notice from the experiments that the best execution time for both queries is achieved close to, but not exactly for, balanced trees, (*Query1*: fo_1=5, fo_2=4 , *Query2*: fo_1=4, fo_2=3).

A. Adaptive apply, AFF_APPLYP

To automatically achieve an optimized process tree, we developed another algebra operator *AFF_APPLYP* (Adaptive First Finished Apply in Parallel) to replace *FF_APPLYP,* but requires no explicit fanout argument.

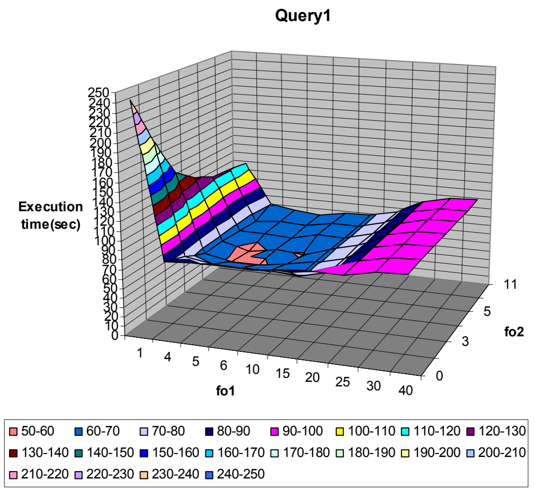

Fig. 16 Execution time for *Query1*

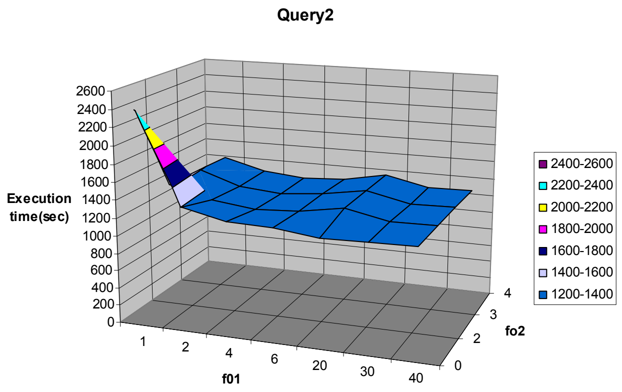

Fig. 17 Execution time for *Query2*

Based on the observation that the best parallelization is close to a balanced tree, *AFF_APPLYP* adapts the process plan at run time starting with a binary tree. Each node locally monitors the execution times of its children to dynamically modify its subtrees *AFF_APPLYP* does the following:

1. *AFF_APPLYP* initially forms a binary process tree (Fig. 18) by always setting fanout to 2, the *init stage.*
2. A *monitoring cycle* for a non-leaf query process is defined as when it has received the same number of end-of-call messages as its number of children. After the first monitoring cycle *AFF_APPLYP* adds *p* new child processes. Adding new processes is called an *add stage*. In Fig. 19, *p=1* and therefore query process *q0* adds one new process *q7* at level 1, while *q1* and *q2* add *q10* and *q11* at level 2, respectively.

3. When an added node has several levels of children the init stages of the children's *AFF_APPLYs* will produce balanced binary sub–trees. That is, *q7* adds *q8* and *q9*.
4. *AFF_APPLYP* records per monitoring cycle *i* the average time t_i to produce an incoming tuple from the children. If t_i decreases more than a threshold (set to 25%) the add stage is rerun. If t_i increases we either stop or run a *drop stage* that drops one child and its children. In Fig. 20, *q2* adds *q12,* while *q0* drops *q7,* and *q7* drops *q8* and *q9*.

We experimented with different values of *p* and different change thresholds, with and without the drop stage. The results for 25% change are shown in Fig. 21. The fanout values are exact for *FF_APPLYP* while fo_1 and fo_2 for *AFF_APPLYP* are average fanouts. The measurements include the adaptation times.

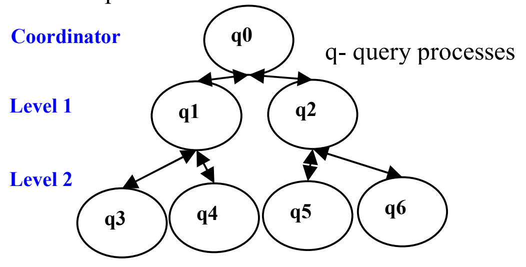

Fig. 18 Binary process tree

We notice that for *Query1* the execution time with *p*=4 and no drop stage comes close to the execution time of the best manually specified process tree, while for *Query2* the execution with *p*=2 and no drop stage is the closest one.

We concluded in both cases that execution time with *p=2* and no drop stage is close to the execution time of the best manually specified process tree (*Query1* 80%, *Query2* 96 %) and further dropping processes make insignificant changes in the execution time.

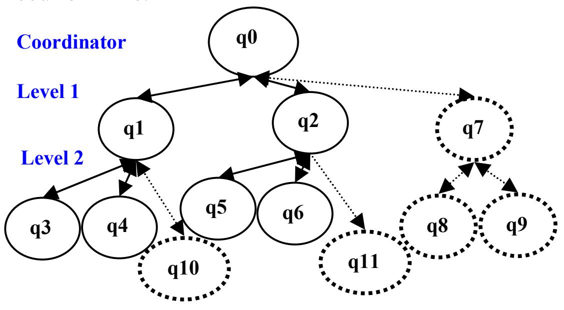

Fig. 19 Adding processes

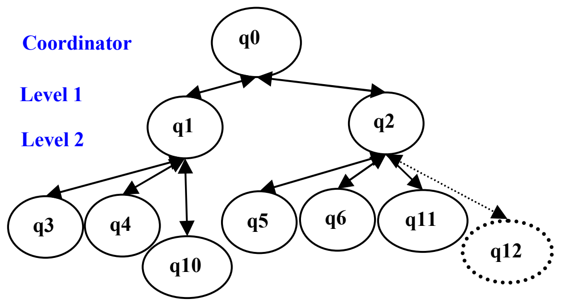

Fig. 20 Adding and removing processes

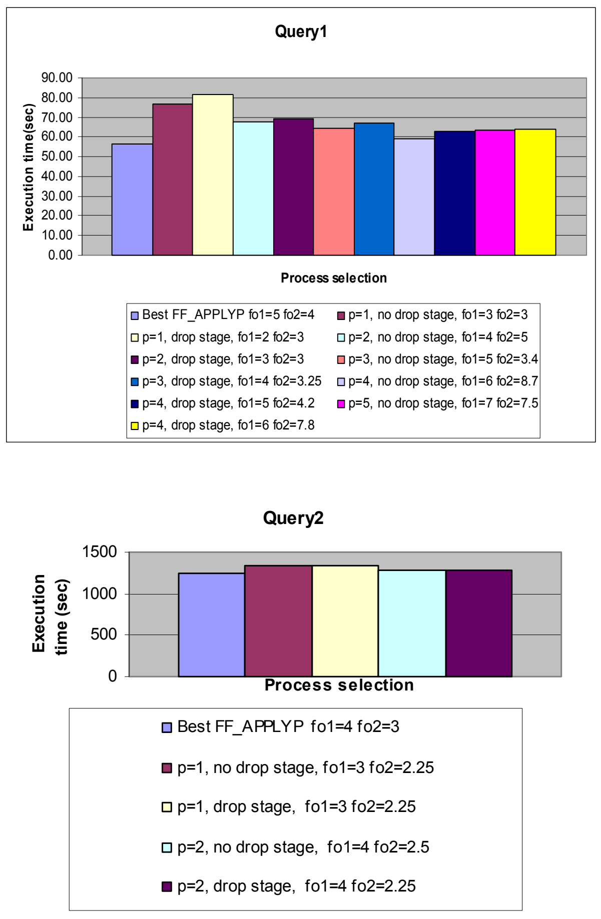

Fig. 21 Execution time with AFF_APPLYP

VI. Related Work

BPEL [2] proposes workflow primitives to manually invoke parallel web service calls. It requires a lot of effort on the part of the programmer to manually identify sections of the code to run in parallel, and to specify dependencies among the calls. In contrast, WSMED automatically compiles a given query over composed data providing web services by generating an adaptive, parallel, and optimized workflow.

In [1] an approach is described for optimizing web service compositions by procedurally traversing ActiveXML documents to select embedded web service calls. It demonstrates the gain obtained by maximizing parallelism achieved by invoking calls to *independent* web services in a query. Conversely, WSMED adaptively parallelizes *dependent* web service calls.

WSQ/DSQ [9] handles high-latency calls to web search engines by launching asynchronous materialized dependent joins later joined in the execution plan using a special operator. In contrast, WSMED produces non-blocking multi-level parallel plans based on streams of parameter tuples passed to parallel sub plans without any materialization.

WSMS [16] proposed an approach for pipelined parallelism among dependent web services to minimize the query execution time. By contrast, we parallelize by partitioning parameter tuple streams. Furthermore, WSMS didn't propose any adaptive parallelization, lacked support for code shipping, and couldn't make parallel calls to the same web service. In contrast we propose a strategy to adaptively produce a parallelized plan where *AFF_APPLYP* invokes parameterized plans calling web services in parallel.

Like two-phase parallel query optimization [11] WSMED also generates a parallelized query execution plan from an initial central query plan. However, WSMED adaptively parallelizes dependent joins by generating plan functions that are called in parallel using the adaptive operator *AFF_APPLYP*, while [11] focused on static inter-operator parallelism in distributed databases based on a static cost model.

The plan function and parameter tuple shipping phase of *FF_APPLYP* is similar to the map phase of *MAPREDUCE* [5]. However, *MAPREDUCE* is more of a programming model than a query operator and is not dynamically rearranging query execution plans as *AFF_APPLYP*.

In [10] run time adaptation of buffer sizes in web service calls is investigated, not dealing with adaptive parallelism on web service calls at the client side.

The formal basis for using views to query heterogeneous data sources is reviewed in [8][18]. *Chocolate* [12] extends the federated database capabilities of *DB2/UDB* by automatically creating views of web services from WSDL descriptions, similar to the OWF generation in WSMED. However, Chocolate does not deal with adaptive parallelization of the web service calls in a query as WSMED.

VII. CONCLUSIONS AND FUTURE WORK

We presented an approach to automatically parallelize queries with dependent web service calls. The algebra operator *FF_APPLYP* was first defined in order to parallelize calls to parameterized sub plans partitioned for different parameter tuples. We did experiments by manually arranging different process trees with different fanouts. From the experiments we concluded that the optimum process fanout is close to, but not exactly, a balanced tree. To adaptively find the best process tree we devised an algebra operator *AFF_APPLYP* that starts with a balanced binary process tree and then each non-leaf process locally adapts the process sub-trees by adding and removing children until an optimum is reached, based on monitoring the flow of result tuples from the children. The adaptive method obtained performance close to the best manually specified process tree.

Our algebra operators *FF_APPLYP* and *AFF_APPLYP* can handle parallel query plans for a query with any number of dependent joins. We would like to generalize the strategy for queries mixing both dependent and independent web service calls, as well bushy trees. Further we need to investigate different process arrangement strategies with the algebra operators.

ACKNOWLEDGMENTS

This work is supported by Sida and the Swedish Foundation for Strategic Research under contract RIT08-0041.

REFERENCES

[1] S. Abiteboul et al., Lazy query evaluation for active XML, *Proc. of the 2004 ACM SIGMOD*, 227–238, 2004.

[2] T. Andrews et al., Business Process Execution Language for Web Services version 1.1., *http://ifr.sap.com/bpel4ws/*, 2003

[3] codeBump, GeoPlaces web service, http://codebump. com /services /PlaceLookup.asmx

[4] codeBump, Zipcodes web service, http://codebump.com/services /ZipCodeLookup.asmx

[5] J.Dean, and S.Ghemawat, MAPREDUCE: Simplified Data Processing on Large Clusters*, Communications of the ACM*, 51(1), 107-113, 2008

[6] G. Fahl, and T. Risch, Query Processing over Object Views of Relational Data, *The VLDB Journal* , 6(4), 261-281, 1997

[7] D.Florescu, A.Levy, I.Manolescu and D.Suciu, Query Optimization in the Presence of Limited Access Patterns, *Proc. of ACM SIGMOD '99,* 311-322, 1999

[8] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A.Rajaraman, Y. Sagiv, J.D. Ullman, V. Vassalos, and J.Widom, The TSIMMIS Approach to Mediation: Data Models and Languages, *Journal of Intelligent Information Systems*, 8(2): 117-132, 1997

[9] R.Goldman, and J.Widom, WSQ/DSQ: a practical approach for combined querying of databases and the Web, *Proc. of 2000 ACM SIGMOD Intl. Conf. on Management of Data,* 285-296*,* 2000.

[10] A. Gounaris, et al., Robust runtime optimization of data transfer in queries over Web Services, *Proc. of ICDE 2008,* 2008

[11] W.Hasan, D.Florescu, and P.Valduriez, Open Issues in Parallel Query Optimization, *SIGMOD Record*, 25(3), 1996

[12] V.Josifovski, S.Massmann, and F.Naumann, Super-Fast XML Wrapper Generation in DB2: A Demonstration, *Proc. International Conference of Data Engineering, (ICDE'03)*, 756-758, 2003

[13] W. Litwin, and T. Risch, Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates, *Proc. IEEE Trans. on Knowledge and Data Engineering,* 4(6), 517-528, 1992

[14] T.Risch, V.Josifovski, and T.Katchaounov, Functional Data Integration in a Distributed Mediator System*, Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data*, Springer, 211-238, 2003

[15] M.Sabesan and T.Risch, Web Service Mediation Through Multi-level Views, Proc. International Workshop on Web Information Systems Modeling (WISM 2007), 755-766, 2007

[16] U.Srivastava, J.Widom, K.Munagala, and R.Motwani, Query Optimization over Web Services, *Proc. Very Large Database Conference (VLDB 2006),* 2006

[17] TerraServer, TerraService, http://terraservice.net/webservices.aspx

[18] J.D.Ullman, Information Integration Using Logical Views, *Proc. 6th International Conference on Database Theory (ICDT '97)*, 19-40, 1997

[19] USZip, http://www.webservicex.net/uszip.asmx