Fault Tolerant Resource Allocation for Query Processing in Grid Environments

Deniz Cokuslu, Abdelkader Hameurlain, Kayhan Erciyes

Fault-tolerant resource allocation for query processing in grid environments

Deniz Çokuslu*

Software Engineering Department,
Izmir University,
Gursel Aksel Bulvari, Uckuyular, 35350 Izmir, Turkey
Email: deniz.cokuslu@izmir.edu.tr
*Corresponding author

Abdelkader Hameurlain

Institut de Recherche en Informatique de Toulouse (IRIT),
Paul Sabatier University,
118 Route de Narbonne, 31062 Toulouse, France
Email: hameurlain@irit.fr

Kayhan Erciyes

Computer Engineering Department,
Izmir University,
Gursel Aksel Bulvari, Uckuyular, 35350 Izmir, Turkey
Email: kayhan.erciyes@izmir.edu.tr

Abstract: In this paper, we propose a new algorithm for fault-tolerant resource allocation for query processing in grid environments. For this, we propose an initial resource allocation algorithm followed by a fault-tolerance protocol. The proposed fault-tolerance protocol is based on the passive replication of stateful operators in queries. We provide theoretical analyses of the proposed algorithms and consolidate our analyses with the simulations.

Keywords: resource allocation; fault-tolerance; distributed query processing; grid systems; distributed databases.

Reference to this paper should be made as follows: Çokuslu, D., Hameurlain, A. and Erciyes, K. (2015) 'Fault-tolerant resource allocation for query processing in grid environments', *Int. J. Web and Grid Services*, Vol. 11, No. 2, pp.143–159.

Biographical notes: Deniz Çokuslu received his BSc (2004) and MSc (2007) degrees from Department of Computer Engineering, Izmir Institute of Technology. He received his PhD (2012) degree in Information Sciences from Ege University, İzmir, Turkey and Paul Sabatier University, Toulouse, France. He worked for Yaşar Holding, Astron Project Development Office between 2003 and 2004 as an SAP Software Developer. He also worked for Izmir Institute of Technology as a research assistant between 2004 and 2012 and as an instructor until 2013. Since then, he is working as an Assistant Professor in Izmir University, Software Engineering Department, İzmir, Turkey.

Abdelkader Hameurlain is full Professor in Computer Science at Paul Sabatier University (IRIT Laboratory) Toulouse, France. His current research interests are in query optimisation in parallel and large-scale distributed environments, and mobile databases. He has been the general chair of the International Conference on Database and Expert Systems Applications (DEXA'02). He is Co-editor-in-Chief of *Transactions on Large-Scale Data- and Knowledge-Centered Systems* (LNCS, Springer). He was guest editor of two special issues of *International Journal of Computer Systems Science and Engineering* on 'Mobile Databases' and 'Data Management in Grid and P2P Systems'.

Kayhan Erciyes received a BSc degree in Electrical Engineering and Electronics from the University of Manchester, MSc degree in Electronic Control Engineering from the University of Salford and a PhD degree in Computer Engineering from Ege University. He was a visiting scholar at Edinburgh University Computer Science Dept. during his PhD studies. He worked as visiting and tenure track faculty at Oregon State University, University of California Davis and California State University San Marcos, all in the USA. He is a faculty member of Computer Engineering Department and Rector of Izmir University, Izmir, Turkey.

1 Introduction

Grid systems are differentiated from distributed and parallel systems by their large-scale, dynamic and heterogeneous characteristics (Foster and Kesselman, 2004). These characteristics raise additional challenges to the distributed query processing domain such as resource discovery, resource selection, resource allocation, autonomous computing, monitoring, replication, caching, security issues and many others (Gounaris et al., 2005). From those, resource allocation is one of the most important steps that directly affect the performance of query execution. However finding near optimal resource allocation alone may not be sufficient for efficiently processing queries in grid environments. Defining the policies in case of node failures during query execution is also very important and should be included in the resource allocation method. Since grid environments are dynamic, eventual node failures are likely during the execution of queries. These failures may be very costly if the queries are long running and if the system is not designed fault-tolerant. Therefore fault-tolerance must be considered in processing queries in grid environments.

In the current literature, there can be found many resource allocation algorithms with dynamicity support (Gounaris et al., 2005; Silva et al., 2006; Venugopal et al., 2006; Kotowski et al., 2008; Gounaris et al., 2009; Paton et al., 2009; Quiane-Ruiz et al., 2009). Although these studies provide resource allocation methods by considering dynamicity of resource properties, none of them consider failure of nodes during execution of stateful operators in queries. A query operator is considered to be stateful if its execution requires storage of any kind of state such as a hash table (Bestehorn et al., 2010). Since stateful operators like hash join require recovery of states of the nodes in case of node failures, dynamic resource allocation methods are not sufficient in such cases. We have found few studies for fault-tolerant query processing in the current literature (Smith and Watson, 2005; Smith and Watson, 2007; Taylor, 2008; Bestehorn et al., 2010). Although these studies provide fruitful algorithms, none of them is specialised on processing stateful query operators in grid environments.

In this paper, we propose a resource allocation algorithm for fault-tolerant query processing in grid environments based on passive replication of stateful operations in queries. For finding the Initial Resource Allocation (IRA) scheme, we first propose the IRA algorithm. After completion of the IRA we propose the Fault-Tolerant Resource Allocation (FTRA) algorithm in which each allocated node applies a replication policy by itself according to the type of the task that it executes.

The contribution of this section is therefore twofold. First, we propose the IRA algorithm by taking advantages of Single Join Operator Resource Allocation (SJORA) algorithm which is presented by Cokuslu et al. (2012b). Second, we propose the FTRA algorithm that presents fault-tolerance for stateful query operators in grid environments. With IRA, we struggle large-scale and heterogeneity characteristics, with FTRA we deal with the dynamicity characteristic of the grid environments taking hash join query requirements into consideration.

In this paper, we assume that the node that posts the query and the nodes in which base relations reside are fault-tolerant or stable by default during the execution of the query. We consider a query as consisting of hash join operators which are composed of atomic tasks, namely scan, build and probe. Scan tasks act as providers to both build and probe tasks by reading tuples from their storage units and sending them to their corresponding tasks. Build tasks receive tuples from their scan tasks and create hash tables for that operator. Probe tasks are blocked during the execution of their corresponding build tasks. After build tasks complete, probe tasks start receiving tuples from their corresponding scan tasks and check for matching tuples in the hash table. Probe tasks pipeline matched tuples through their successor tasks in the query tree. More detailed description of distributed hash join operator can be found in the work of Özsu and Valduriez (2011).

The structure of this paper is as follows; in Section 2 we present a brief literature survey about the current resource allocation studies. In Section 3, we propose the IRA algorithm followed by the FTRA algorithm for query processing in grid environments. We present detailed design, analyses and simulations for each algorithm. Finally in Section 4, we present our conclusions.

2 Related work

There can be found many studies in the literature, which examine resource allocation for query processing in grid environments (such as Gounaris et al., 2004; Gounaris et al., 2005; Kant and Grosu, 2005; Mandal et al., 2005; Soe et al., 2005; Gounaris et al., 2006; Silva et al., 2006; Venugopal et al., 2006; Bose et al., 2007; Kotowski et al., 2008; Liu and Karimi, 2008; Gounaris et al., 2009; Paton et al., 2009; Quiane-Ruiz et al., 2009). A very detailed literature survey related to these studies is presented by Cokuslu et al. (2012a). These studies either present static resource allocation algorithms in which the resource allocation is performed once, and allocated nodes sustain execution until the tasks are completed, or they present dynamic resource allocation algorithms in which allocation is dynamically modified during the execution of the tasks according to the monitored status of the resources. Although dynamic resource allocation algorithms take dynamicity of the characteristics of the environment into account, none of these algorithms consider the case in which node failures occur. Still, there can be found many other studies in the current literature, which examine fault-tolerant distributed query processing in different environments (Abadi et al., 2005; Balazinska et al., 2008;

Bestehorn et al., 2010; Chandrasekaran et al., 2003; Hwang et al., 2005; Hwang et al., 2007; Kwon et al., 2008; Smith and Watson, 2005; Smith and Watson, 2007). However, few of them are applicable in grid environments especially for processing stateful query operators such as hash joins (Bestehorn et al., 2010).

More precisely, Smith and Watson (2005) presented a fault-tolerant query processing system for distributed query processing. Their study is an extension to their previous work *OGSA-DQP* (Alpdemir et al., 2004) with the addition of *fault detector* and *fault handler* modules. In their study, the queried node generates a query plan, performs an IRA and initiates the execution of the query. For the fault-tolerance, the algorithm performs a rollback recovery protocol. Smith and Watson (2007) discuss failure recovery alternatives in query processing in grid environments. In their study, they examined three failure recovery alternatives, namely *restart*, *reduce* and *replace*. Bestehorn et al. (2010) presented a fault-tolerant query processing algorithm in structured P2P systems. In their study, they examined the query operations in two classes: namely stateless and stateful operations. The proposed method examines fault-tolerance over these operations with two different perspectives: fault-tolerant routing and replication. The study exploits the functionalities proposed by the CAN peer-to-peer system for selecting the backup peers. A very detailed survey study related to fault-tolerant distributed query processing can be found in the work of Taylor (2008). In his study, Taylor examines different fault-tolerant distributed query processing algorithms in three classes, namely: (a) upstream backup (Smith and Watson, 2007; Smith and Watson, 2005), (b) active standby (Abadi et al., 2005; Balazinska et al., 2008; Chandrasekaran et al., 2003) and (c) passive standby (Hwang et al., 2005; Hwang et al., 2007; Kwon et al., 2008).

Considering our analyses related to fault-tolerance in query processing, we have found many studies examining this problem. Although these studies present valuable algorithms, to the best of our knowledge, we cannot find studies designed especially for grid environments, which examine fault-tolerance in stateful query operators. Most of the examined studies are focused on stream processing which does not include stateful operators. Moreover most of these studies are designed for P2P systems. Although there are many similarities between grid systems and P2P systems, in many aspects they have distinct differences. Therefore, we believe that the characteristics of grid environments and requirements of query processing tasks should be focused in order to find suitable RA algorithms for fault-tolerant query processing in grid environments.

3 Algorithms

In this section, we propose an algorithm for FTRA for query processing in grid environments. To realise this, we first propose an IRA algorithm by extending the SJORA algorithm which is presented by Cokuslu et al. (2012b). Then, after setting the IRA, we propose FTRA algorithm by introducing a fault-tolerance module that realises passive replication of stateful operators in the queries.

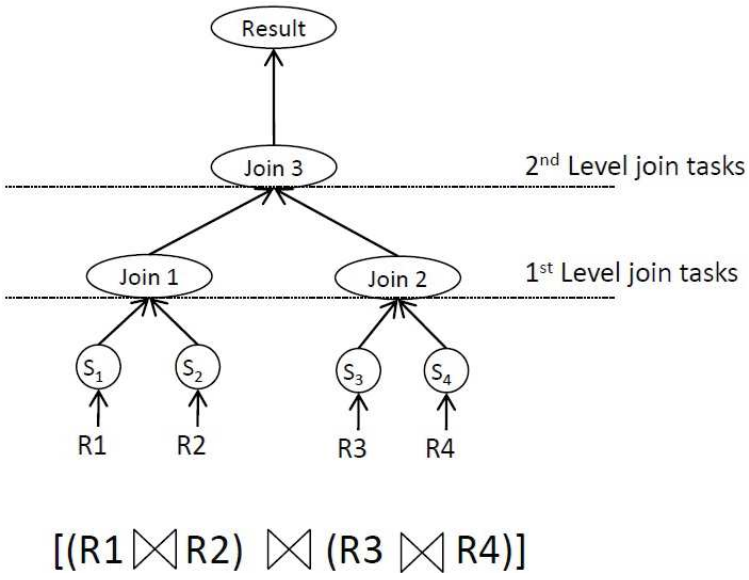
3.1 Initial Resource Allocation (IRA) algorithm

In the IRA algorithm, we aim at finding suitable nodes for all tasks that compose the join operators in the entire query. However, since the residing nodes of the temporary relations cannot be determined beforehand, we design the IRA algorithm in bottom-up fashion starting from allocation of resources to the operators that use base relations. In

this section, we examine hash join queries, which consist of one or more join operators, as a use case for query operators. We assume that the optimised query operator tree is provided explicitly. The relations that are involved in the query are assumed to be horizontally partitioned into the grid without replication. We consider a query as consisting of hash join operators which are composed of atomic tasks namely, scan, build and probe. A brief description of these tasks is given in the introduction section and detailed are presented in the work of Özsu and Valduriez (2011).

In SJORA algorithm, Cokuslu et al. (2012b) find and allocate suitable resources for queries consisting of a single join operator assuming scan tasks are already allocated in the nodes in which the base relations reside. Since queries may consist of more than one join operator, it is necessary to extend the SJORA algorithm by materialising the allocation of all tasks in the entire query. In this section, we propose the IRA algorithm, which allocates nodes for the queries that consist of multiple join operators. For the IRA algorithm, we use tree representation of queries. An example query with its tree representation is shown in Figure 1. Vertices in the query tree represent tasks and directed edges represent data flow between tasks. In IRA algorithm, we consider that queries consist of join operators, and join operators consist of scan, build and probe tasks. Each join operator in the query may be composed of more than one pair of build and probe tasks that execute concurrently over the different data partitions. The build and probe tasks in each pair are tightly coupled. Therefore, we decided to allocate each pair of build and probe tasks in the same resource. For simplicity, in the rest of this paper, these pairs of build and probe tasks are named as join tasks.

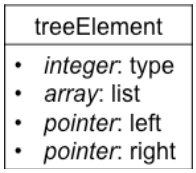
Figure 1 An example query and its query tree



In SJORA algorithm, Cokuslu et al. (2012b) assumed that the partitions of base relations are known at the end of the resource discovery stage. However, for the temporary relations, it is not possible to find out their physical locations before precedent join tasks are allocated. Therefore, in IRA algorithm, we execute multiple instances of SJORA algorithm starting from the join tasks that are located at the lowest level of the query tree.

These kinds of join tasks are marked as first-level join tasks as shown in Figure 1. Then, we allocate the join tasks that are located at the second level in the query tree. We repeat this process until all tasks are allocated in the query. To realise this, we apply post-order tree traversal on the query tree. The generation of the query tree involves query optimisation steps such as query reordering. Since this kind of optimisation is out of scope of this paper, we assume that the query tree is already provided exclusively. The IRA algorithm is executed by the queried node. A snapshot of the IRA is stored in the query tree that resides on the queried node. Each vertex in the query tree contains a data structure called *treeElement*, as shown in Figure 2. The *treeElement* contains four principal fields, namely *type*, *list*, *left* and *right*. *Type* field indicates the type of the task whether it is a scan or a join task. *List* field contains a list of allocated nodes for the task, which is initially empty. *Left* and *right* fields point to the left and right subtrees, respectively, in the query tree.

Figure 2 Data structure for the treeElement



The IRA algorithm is shown in Algorithm 1. The algorithm runs recursively a post-order tree traversal (lines 1–6). In line 7, the algorithm processes the visited vertex. If the visited vertex is a scan task, IRA algorithm fills list field of the *treeElement* with the list of nodes in which the partitions of the processed relation reside (line 8). Else, if the visited vertex is a join task, the list field is filled with the results of the SJORA algorithm. In this step, the list of partitions of base or temporary relations is provided by the child vertices (line 10). The estimated sizes of the temporary relations (*sizes*) are assumed to be known by external estimations. The ids of most distant nodes and the distance between them for the partitions of temporary relations (*nodeA*, *nodeB* and *dist*) are provided by the topology control module of the resource discovery algorithm, which is presented by Cokuslu et al. (2010). At the end of execution of the IRA algorithm, all the vertices in the query tree contain the list of selected nodes for the corresponding task. At this stage, the queried node sends the query tree to all selected nodes in order to complete allocation of nodes and to start execution of the query.

Algorithm 1 IRA algorithm

Input: *Root of the query tree*

Output: List of nodes to be allocated for each operator

1: *IRAAlgorithm(treeNode)*
2: **if** *treeNode is empty* **then**
3: *return*
4: **else**
5: *IRAAlgorithm(treeNode.left)*
6: *IRAAlgorithm(treeNode.right)*

```
7:    if treeNode.type = SCAN then
8:       treeNode.list = nodes in which partitions reside
9:    else if treeNode.type = JOIN then
10:      treeNode.list = SJORA(treeNode.left.list ∪ treeNode.right.list, nodeA,
nodeB, dist, sizes)
11:   end if
12: end if
13: return
14: end
```

3.1.1 Analysis

In this section, we provide time and message complexity analyses of the proposed algorithm, IRA.

Theorem 1: *The IRA algorithm has O(jnN) time complexity, where j is the number of join operators in the query, n is the number of scan nodes and N is the number of nodes in the grid system*.

Proof: The IRA algorithm uses the SJORA algorithm (Cokuslu et al., 2012b) for each join operator in the query tree. Since the time complexity of the SJORA algorithm is $O(nN)$, the time complexity of the IRA algorithm is $O(jnN)$ for a query that consists of j join operators.

Theorem 2: *The IRA algorithm has $O(jnN^2)$ message complexity, where j is the number of join operators in the query, n is the number of the scan nodes and N is the number of the nodes in the grid system.*

Proof: The IRA algorithm uses the SJORA algorithm (Cokuslu et al., 2012b) for each join operator in the query tree. Since the message complexity of the SJORA algorithm is $O(nN^2)$, the total message complexity of the IRA algorithm is $O(jnN^2)$ for a query that consists j join operators.

3.1.2 Simulations

In this section, we present evaluation of the IRA algorithm by simulation. We compare our algorithm with a comparative algorithm (CA) that reflects the common properties of the recent resource allocation algorithms (Gounaris et al., 2004; Gounaris et al., 2006). The main idea behind the comparative algorithm is similar to the algorithm proposed by Gounaris et al. (2004, 2006). The algorithm ranks the nodes in the grid according to their properties. In our case, one of the most significant properties that influence the execution of queries is the connection speed of the nodes. Therefore, the CA algorithm ranks the nodes according to their connection speeds. Then the ranked nodes are sorted and the algorithm starts to allocate nodes starting from the top of the list. When addition of a new node does not lead to a performance increase, the algorithm terminates. The CA algorithm traverses the query tree and allocates resources for each join operator in the

query. We have implemented IRA and CA algorithms in ns2 simulation environment and measured the cost of resource allocation process and duration of execution of a sample query.

We have generated grid simulation scenarios consisting of 100 through 800 nodes. Each node in the scenario represents a uniprocessor computer in the grid system that has arbitrary connections to other nodes in the environment. The bandwidths of duplex connections between nodes are randomly assigned between 1 and 10 Gbps. In our simulation scenarios, we have simulated resource allocation and execution of a sample query consisting of three join operators, which joins four relations in total. The formal representation of the sample query is shown in Figure 3 where R1, R2, R3 and R4 present relations and ⋈ presents the join operator.

Figure 3 Formal representation of the sample query

$$\{(R1 \bowtie R2) \bowtie (R3 \bowtie R4)\}$$

Each relation is horizontally partitioned into five arbitrary scan nodes in our scenarios. The distribution of the scan nodes is realised randomly over the simulated environment. Each scan node is assumed to store a partition of a base relation of size 50 GB. Each scan node stores only one partition.

We have collected test results for the cost of IRA and cost of query execution. Figure 4 shows the IRA process for the sample query. As it can be seen in Figure 4, the cost of the IRA algorithm is higher than the CA. This is because the IRA algorithm processes its entire candidate list to find the best possible resource allocation within its candidates and measures the communication speeds from each candidate node to all scan nodes while calculating the estimated query duration. This overhead results in a worse performance than the CA initially in return for a better selection of resources. However, it can be seen in Figure 4 that the IRA algorithm scales well with the number of nodes in the grid environment. The cost of resource allocation process remains nearly constant as the number of nodes increase. This is caused by the limitation of the candidate resource search space. In each scenario, the algorithm examines nearly the same number of candidate resource in the IRA. Like the IRA, the CA also remains nearly constant as the number of nodes increase. This is because in CA the algorithm stops adding new resources when the performance increase reaches to a threshold limit, instead of examining all resources in the candidate list. Otherwise, the CA would examine every resource in the grid environment. This approach may miss better resource allocation combinations with higher number of resources. However, it is conceptually impossible to evaluate all possible resource allocation combinations.

Figure 5 shows the execution cost of the sample query. It can be seen in Figure 5 that the resources allocated by the IRA algorithm execute the query faster than the resources that are allocated by the CA. This is because, although the resources that are allocated by the CA are the highest ranked nodes in the grid, they might be placed far from the scan nodes, which may result in slower data transfer rates. On the other hand, the resources that are allocated by the IRA algorithm are closer to the scan nodes in terms of connection speeds. For that reason, IRA algorithm ensures allocation of more effective resources in terms of the communication performances with the scan nodes.

Figure 4 Initial resource allocation costs

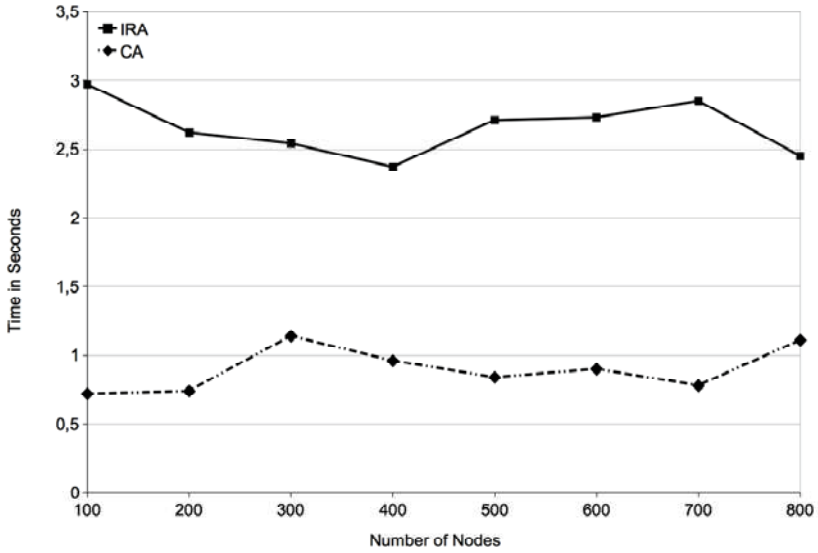
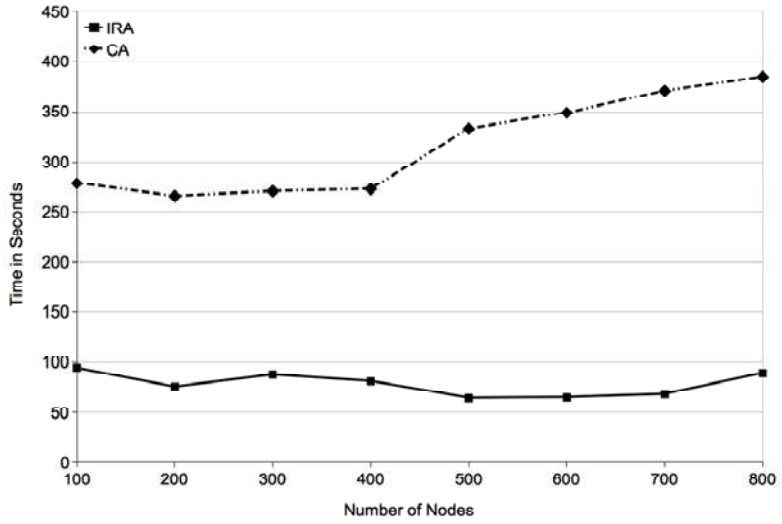


Figure 5 Sample query execution costs with resources allocated by IRA and CA



Regarding the simulation results, which are shown in Figures 4 and 5, it can be considered that the IRA algorithm is more preferable if the cost of IRA does not exceed the estimated query execution durations. In our simulation scenarios, the durations of query executions are much higher than the costs of the resource allocation processes. Therefore, in such cases, the IRA algorithm might be considered as a better alternative to the existing resource allocation algorithms that are based on ranking functions.

3.2 Fault-Tolerant Resource Allocation (FTRA) algorithm

In this section, we propose the FTRA algorithm for query processing in grid environments based on the passive replication of stateful join operators in the queries. The FTRA algorithm works in succession with the IRA algorithm and is responsible for the fault-tolerance of the nodes that are allocated for the join tasks. The nodes that execute tasks are named as master nodes; and the nodes that are allocated for the fault-tolerance purpose are named as backup nodes or replicas. It is assumed that a master node and its replica do not fail at the same time.

The FTRA algorithm is composed of four steps: (a) replica selection, (b) query execution & backing up, (c) failure detection and (d) failure recovery. These steps are defined as follow:

1 *Replica selection:* The replica selection step is realised before the master node starts its execution. When the IRA algorithm completes, each master node determines a backup node by choosing its closest available neighbour. The selection of the closest available neighbour as the replica aims at minimising the replication overhead that will be caused by the FTRA. In this step, we assume that a master node has always at least one available neighbour for replica selection.

2 *Query execution & backing up*: After the replica selection step, the master node starts executing the query while backing up its state for fault-tolerance. In order to avoid synchronisation issues between replicas and master nodes, the replication scheme in this step is chosen to be passive replication. In passive replication, the states of the master nodes are backed up to their replicas periodically. The states depend on the type of tasks that are being executed. More precisely, build tasks generate hash tables for the join operators. The failure of a node during the execution of build task results in the loss of the hash table that is generated so far. Therefore, during the execution of the build task the state is composed of the hash table and the sequence number of the last tuple that is received. On the other hand, probe tasks receive tuples from their predecessors, check the hash table for occurrences and send results to their successors in a pipelined fashion. Execution of a probe task means that the build task is already terminated and the hash table is already constructed. It also implies that the hash table is already backed up. Therefore, during the execution of the probe task the state is composed only of the sequence number of the last received tuple. The update interval of the replication period is determined heuristically. The states of master nodes are backed up incrementally. In other words, during the execution of build task, each time the state of the master node is backed up, only the additions to the hash table is transferred to the replica node since the last check point.

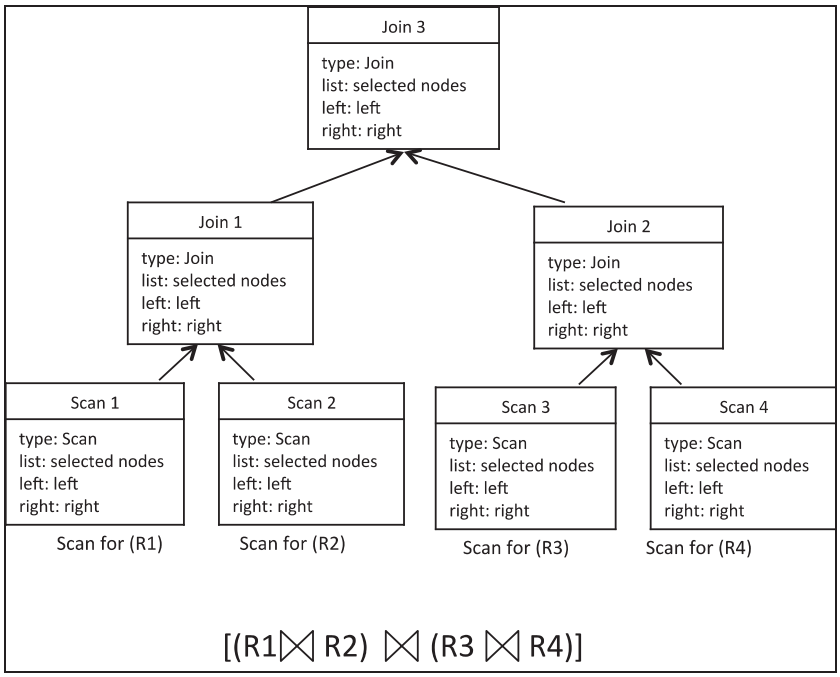
3 *Failure detection:* In FTRA, the failure detection is held by both master nodes and replicas. Failures of nodes are detected by exploiting the periodical backup messages. The replica nodes monitor failures of their master nodes and master nodes monitor failure of their replicas by examining delivery of backup and their acknowledgement messages.

4 *Failure recovery:* Whenever a replica node detects failure of its master node, it replaces itself with its master and notifies its predecessor and successor nodes for the change. Before it starts acting as a master node, it requests a replica node for itself

by choosing its closest available neighbour. On the other hand, if a master node detects failure of its replica, it requests a new replica and backs up its last state entirely once and continues incremental backing up later.

The algorithm is executed successively to the IRA algorithm that is proposed in the Section 3.1. When the IRA algorithm is executed beforehand, the algorithm outputs a tree structure, namely query tree, which contains the tree representation of the query with the selected nodes included for the IRA. A sample query tree is shown in Figure 6.

Figure 6 Example query tree that is used for input of the FTRA algorithm



Each vertex in the query tree is composed of the data structure *treeElement* that is shown in Figure 2. The FTRA algorithm inputs the query tree and starts processing it by using post-order tree traversal. Each time the algorithm visits a vertex in the query tree, it sends a request message to resources that are selected by the IRA algorithm in order to inform them about the initial allocation. This allows selected nodes to know their successor and predecessor nodes to communicate with. The formal presentation of the FTRA algorithm in the queried node is shown in Algorithm 2.

Algorithm 2 FTRA algorithm in the queried node

Input: *Root of the query tree*

Output: Query execution

1: *FTRAAlgorithm(treeNode)*

2: **if** *treeNode is empty* **then**

3: *return*

4: **else**

```
5:     FTRAAlgorithm(treeNode.left)
6:     FTRAAlgorithm(treeNode.right)
7:     if treeNode.type = SCAN then
8:       send AllocScanReq to the each node in treeNode.list
9:     else if treeNode.type = JOIN then
10:       send AllocJoinReq to the each node in treeNode.list
11:    end if
12:  end if
13:  return
14: end
```

After finishing the query tree traversal, the queried node waits for the resulting tuples of the query as a pipelined fashion. The other nodes in the grid environment, which receive request messages, involve in the query execution according to the algorithm that is suitable for their types. Basic algorithm steps in the nodes that are assigned for different types of tasks are shown in Algorithm 3. In Algorithm 3, the execution of FTRA algorithm in a node that executes a scan task is presented. A scan node will read tuples from its storage unit and sends them to its successor nodes. In order to start this operation, it first waits for all its successor nodes to become ready to receive tuples (line 2). When all its successor nodes become ready, it starts the scan operation (line 4). Upon finishing the data to be sent, the node indicates this by sending a specific message to its successor nodes (line 6). Notice that it is assumed that scan nodes are stable during the execution of the query therefore they do not require any fault-tolerance algorithm.

Algorithm 3 Basic steps of FTRA algorithm in a scan node

Input: *Received messages*

Output: Execution of the required steps

```
1: Upon reception of AllocScanReq message:
2:    Wait StartScan message from all successor nodes
3:    When all successor nodes send StartScan message
4:    Start sending tuples to the corresponding successor nodes
5:    When the data is completely consumed
6:     Send ScanEnd message to all successor nodes
7: end
```

The execution steps of the FTRA algorithm in a node that is assigned for a join task are shown in Algorithm 4. After being allocated for the task, a join node selects a replica for itself that is the nearest neighbour to it (line 2). Then it claims its readiness to its predecessor nodes (line 3). While receiving tuples from its scan nodes, the join node constructs the hash table (line 4). At the same time it backs up its state periodically in its replica node (line 5). If failure of the backup node is detected, the join node selects another backup node and backs up its entire state once (lines 6–8). When construction of the hash table is finished, the node wakes-up the scan nodes that hold relations for the probe phase and starts probe operation (lines 9 and 10). At this stage, the join node only backs up the sequence number of the scan messages (line 11). When the probe operation is finished, the node informs its successor by sending a specific message (line 12).

Algorithm 4 Basic steps of FTRA algorithm in a join node

Input: *Received messages*

Output: Execution of the required steps

1: *Upon reception of AllocJoinReq message:*

2: *Send ReplicaReq message to the nearest neighbour*

3: *Upon receiving ReplicaResp, send StartScan message to all predecessor nodes*

4: *Build the hash table using the received tuples from the scan nodes*

5: *Periodically send hash table updates to the backup node*

6: **if** *the backup node is failed* **then**

7: *Select another backup node*

8: **end if**

9: *When build task is finished, send StartScan message to the scan nodes that hold partitions of the relation that is used in the probe task*

10: *Execute probe task using the received tuples*

11: *Periodically send last received tuple information to the backup node*

12: *When probe task is finished, send ScanEnd message to the successor node*

13: **end**

Finally in Algorithm 5, the algorithm steps for a node that is allocated for replication is presented. After agreeing with the master node (lines 1 and 2) the backup node periodically accepts backup messages and stores the state information of its master (line 3). If failure of the master node is detected, the backup node informs the predecessor nodes of its master about the failure to suspend data flow (lines 4 and 5). Then it selects a backup node for itself (line 6) and itself starts acting as a master node. It then makes its predecessor nodes to resume the communication and continue the execution of the query (lines 7and 8).

Algorithm 5 Basic steps of FTRA algorithm in a backup node

Input: *Received messages*

Output: Execution of the required steps

1: *Upon reception of ReplicaReq message:*

2: *Send ReplicaResp message to the master node*

3: *Record periodical state updates*

4: **if** *master node is failed* **then**

5: *Send AllocUpdate message to all predecessors of the master node in order to suspend communication*

6: *Select a backup node by sending ReplicaReq message to the nearest neighbour*

7: *Change type of task to join*

8: *Send StartScan message to all predecessors to resume the query execution as a master node*

9: **end if**

10: **end**

3.2.1 Analysis

In this section, we present time and message complexity analyses of the FTRA algorithm.

Theorem 3: *The overhead of the FTRA algorithm to the standard query execution in terms of time complexity is O(j), where j is the number of join operators in the query*.

Proof: The standard execution of a query in a grid environment consists of execution of scan, build and probe tasks. In FTRA algorithm, additionally to the standard execution, replica selection is required for each join operator before starting the execution. For each join operator, selection of a replica node involves four steps. Since there are j join operators in the query, the total time overhead of the FTRA algorithm is (4j) which is defined as O(j).

Theorem 4: *The overhead of the FTRA algorithm to the standard query execution in terms of message complexity is O(jnm), where j is the number of join operators in the query, n is the number of scan nodes and m is the number of allocated nodes for the join tasks*.

Proof: In FTRA algorithm, since allocated nodes become ready to receive tuples from their precedent nodes only after they set their backup nodes, beginning of scan tasks are subject to synchronisation with their successors. This synchronisation requires each allocated node to send a message to its scan nodes. Therefore, for each join operator, the FTRA algorithm requires *nm* message exchanges. Therefore, the total message overhead of the FTRA algorithm is *O(jnm)*.

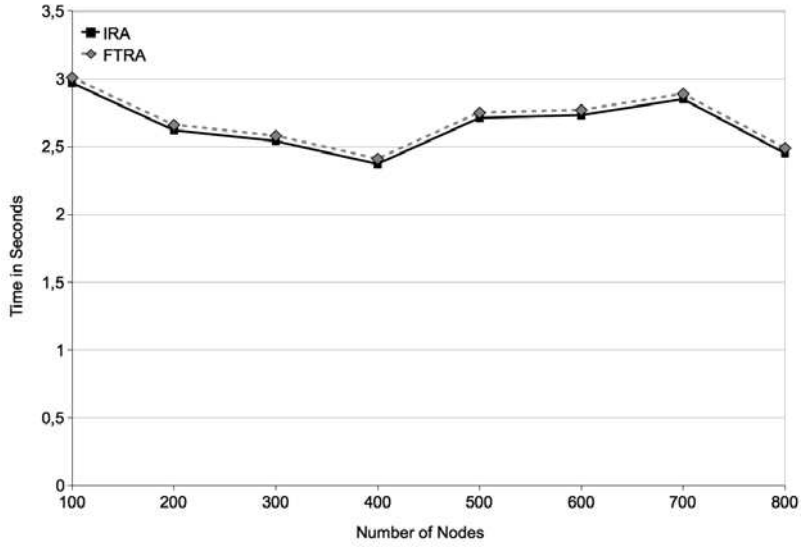
3.2.2 Simulations

In this section, we present quantitative evaluations and simulation results for the FTRA algorithm. We compare our algorithm to the IRA algorithm and examine the overhead that is caused by the replication.

For the simulation scenario, we used the same simulation setting that is used in the Section 3.1.2. We measured the cost of the resource allocation process.

Figure 7 shows the initial costs of IRA and FTRA algorithms. As it can be seen in Figure 7, there is a slight difference between the IRA and the FTRA. In all different simulation scenarios, the FTRA requires a constant amount of time in order to provide fault-tolerance. This overhead is caused by the selection of the replica nodes. In FTRA, after completion of the IRA, the algorithm executes the finite state machine steps which are shown in Figure 7. In our scenarios, since the number of join operators and number of scan nodes are constant, the overhead of the FTRA algorithm is bound by the number of allocated nodes in the IRA. During our simulations, we observed that the number of allocated nodes by the IRA algorithm does not alter significantly. Therefore, the overhead of the FTRA algorithm remains nearly constant as the number of nodes increase. Regarding the simulation results, it can be considered that the FTRA can be a strong alternative to the classical resource allocation algorithms in the case in which the queries are long running and fault-tolerance overhead is acceptable.

Figure 7 Cost of the FTRA algorithm



4 Conclusions

Resource allocation for query processing in grid systems is a very important step that directly affects the performance of query execution. Since grid environments are differentiated from other distributed environments by their large-scale, dynamic and heterogeneous nature, the resource allocation algorithm should address all problems that are caused by these characteristics. In this paper, we have first proposed an IRA algorithm, which allocates resources for a query consisting of multiple join operators by exploiting the SJORA algorithm which is presented by Cokuslu et al. (2012b), aiming at addressing scalability and heterogeneity problems. Then, aiming at addressing the dynamicity problems, we proposed an FTRA algorithm for query processing in grid environments that recovers from node failures during the execution of the query. Although there can be found many studies in the current literature, to the best of our knowledge, we cannot find any study which focuses on the dynamicity of nodes in the grid environment in terms of node failures. For that reason, we aimed at contributing the query-processing domain by proposing fault-tolerance in resource allocation.

We presented our algorithms in detail and proposed complex analyses. Then, we strengthened our perspectives by the use of quantitative analyses and simulations. We showed that the IRA algorithm outperforms the similar existing algorithms in the cases in which the query execution durations are much higher than the cost of the IRA. For the FTRA, the simulation results showed that the algorithm is a very favourable algorithm with slight overhead to the IRA algorithm. Since it provides fault-tolerance, it can be preferred in situations in which the queries are long running and in environments in which node failures are likely. It can be concluded that the proposed algorithms can be considered to be scalable and strong alternatives to the existing algorithms in the defined cases.

References

Abadi, D., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y. and Zdonik, S. (2005) 'The design of the borealis stream processing engine', *CIDR'05: 2nd Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, pp.277–289.

Alpdemir, M., Mukherjee, A., Gounaris, A., Paton, N., Watson, P., Fernandes, A. and Fitzgerald, D. (2004) 'OGSA-DQP: a service for distributed querying on the grid', in Bertino, E., Christodoulakis, S., Plexousakis, D., Christophides, V., Koubarakis, M., Böhm, K. and Ferrari, E. (Eds): *Advances in Database Technology – EDBT 2004*, Springer, Berlin, Heidelberg, pp.858–861.

Balazinska, M., Balakrishnan, H., Madden, S.R. and Stonebraker, M. (2008) 'Fault-tolerance in the borealis distributed stream processing system', *ACM Transaction on Database System*, Vol. 33, pp.1–44.

Bestehorn, M., Von der Weth, C., Buchmann, E. and Böhm, K. (2010) 'Fault-tolerant query processing in structured P2P-systems', *Distributed and Parallel Databases*, Vol. 28, pp.33–66.

Bose, S.K., Krishnamoorthy, S. and Ranade, N. (2007) 'Allocating resources to parallel query plans in data grids', *GCC'07: Proceedings of the Sixth International Conference on Grid and Cooperative Computing*, IEEE Computer Society, pp.210–220.

Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S.R., Reiss, F. and Shah, M.A. (2003) 'TelegraphCQ: continuous dataflow processing', *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, San Diego, California, ACM, pp.668–668.

Cokuslu, D., Erciyes, K. and Hameurlain, A. (2010) 'A maximum degree self-stabilizing spanning tree algorithm', *ISCIS'10: The 25th International Symposium on Computer and Information Sciences*, Springer Verlag, pp.393–396.

Cokuslu, D., Hamuerlain, A. and Erciyes, K. (2012a) 'Resource allocation for query processing in grid systems: a survey', *International Journal of Computer Systems Science and Engineering*, Vol. 27.

Cokuslu, D., Hamuerlain, A., Erciyes, K. and Morvan, F. (2012b) 'Resource allocation algorithm for a relational join operator in grid systems', *Proceedings of 16th International Database Engineering & Applications Symposium, IDEAS12*, Prague, Czech, pp.139–145.

Foster, I. and Kesselman, C. (2004) *The Grid: Blueprint for a New Computing Infrastructure*, Chicago, IL.

Gounaris, A., Sakellariou, R., Paton, N.W. and Fernandes, A.A.A. (2004) 'Resource scheduling for parallel query processing on computational grids', *5th IEEE/ACM International Workshop on Grid Computing*, pp.396–401.

Gounaris, A., Sakellariou, R., Paton, N.W. and Fernandes, A.A.A. (2006) 'A novel approach to resource scheduling for parallel query processing on computational grids', *Distributed and Parallel Databases*, Vol. 19, pp.87–106.

Gounaris, A., Smith, J., Paton, N.W., Sakellariou, R., Fernandez, A.A.A. and Watson, P. (2005) 'Adapting to changing resource performance in grid query processing', in Pierson, J-M. (Ed.): *Data Management in Grids*, Springer, Berlin. Heidelberg, pp.30–44.

Gounaris, A., Smith, J., Paton, N. W., Sakellariou, R., Fernandes, A.A. and Watson, P. (2009) 'Adaptive workload allocation in query processing in autonomous heterogeneous environments', *Distributed and Parallel Databases*, Vol. 25, pp.125–164.

Hwang, J-H., Balazinska, M., Rasin, A., Cetintemel, U., Stonebraker, M. and Zdonik, S. (2005) 'High-availability algorithms for distributed stream processing', *Proceedings of the 21st International Conference on Data Engineering*, IEEE Computer Society, pp.779–790.

Hwang, J-H., Xing, Y. and Zdonik, S. (2007) 'A cooperative, self-configuring high-availability solution for stream processing', *ICDE, IEEE 23rd International Conference on Data Engineering*, Istanbul, pp.176–185.

Kant, U. and Grosu, D. (2005) 'Double auction protocols for resource allocation in grids', *ITCC'05: Proceedings of the International Conference on Information Technology: Coding and Computing*, IEEE Computer Society, pp.366–371.

Kotowski, N., Lima, A.A.B., Pacitti, E., Valduriez, P. and Mattoso, M. (2008) 'Parallel query processing for OLAP in grids', *Concurrency and Computation: Practice and Experience*, Vol. 20, pp.2039–2048.

Kwon, Y., Balazinska, M. and Greenberg, A. (2008) 'Fault-tolerant stream processing using a distributed, replicated file system', *Proceedings of VLDB* Endowment, Vol. 1, pp.574–585.

Liu, S. and Karimi, H.A. (2008) 'Grid query optimizer to improve query processing in grids', *Future Generation Computer System*, Vol. 24, pp.342–353.

Mandal, A., Kennedy, K., Koelbel, C., Marin, G., Mellor-Crummey, J., Liu, B. and Johnsson, L. (2005) 'Scheduling strategies for mapping application workflows onto the grid', *Proceedings, 14th IEEE International Symposium on High Performance Distributed Computing*, pp.125–134.

Özsu, M.T. and Valduriez, P. (2011) *Principles of Distributed Database Systems*, 3rd ed., New York.

Paton, N.W., Buenabad-Chavez, J., Chen, M., Raman, V., Swart, G., Narang, I., Yellin, D.M. and Fernandes, A.A. (2009) 'Autonomic query parallelization using non-dedicated computers: an evaluation of adaptivity options', *The VLDB Journal*, Vol. 18, pp.119–140.

Quiane-Ruiz, J-A., Lamarre, P. and Valduriez, P. (2009) 'A self-adaptable query allocation framework for distributed information systems', *The VLDB Journal*, Vol. 18, pp.649–674.

Silva, V.F.V.D., Dutra, M.L., Porto, F., Schulze, B., Barbosa, A.C. and De Oliveira, J.C. (2006) 'An adaptive parallel query processing middleware for the grid: research articles', *Concurrency and Computation: Practice and Experience*, Vol. 18, pp.621–634.

Smith, J. and Watson, P. (2005) 'Fault-tolerance in distributed query processing' in IDEAS05', *9th International Database Engineering and Application Symposium*, pp.329–338.

Smith, J. and Watson, P. (2007) 'Failure recovery alternatives in grid-based distributed query processing: a case study, knowledge and data management in grids', in Talia, D., Bilas, A. and Dikaiakos, M.D. (Eds.): *Knowledge and Data Management in Grids*, Springer, USA, pp.51–63.

Soe, K.M., Nwe, A.A., Aung, T.N., Naing, T.T. and Thein, N.L. (2005) 'Efficient scheduling of resources for parallel query processing on grid-based architecture', *6th Asia-Pacific Symposium on Information and Telecommunication Technologies (APSITT 2005)*, pp.276–281.

Taylor, N.E. (2008) *Recovery from Node Failure in Distributed Query Processing*, Technical Report, University of Pennsylvania, Department of Computer and Information Science.

Venugopal, S., Buyya, R. and Winton, L. (2006) 'A grid service broker for scheduling e-science applications on global data grids: research articles', *Concurrency and Computation: Practice and Experience*, Vol. 18, pp.685–699.