Leveraging Non-Uniform Resources for Parallel Query Processing

Tobias Mayr
IBM Almaden
tmayr@us.ibm.com

Philippe Bonnet
University of Copenhagen
bonnet@diku.dk

Johannes Gehrke
Cornell University
johannes@cs.cornell.edu

Praveen Seshadri
Microsoft
pravse@micosoft.com

Abstract

Clusters are now composed of non-uniform nodes with different CPUs, disks or network cards so that customers can adapt the cluster configuration to the changing technologies and to their changing needs. Such modular clusters challenge parallel databases. The load balancing techniques used by existing parallel databases partition data across a set of nodes that all run the same relational operations. We show in this paper that this form of load balancing is ill suited for modular clusters because running the same operation on different subsets of the data does not fully utilize non-uniform hardware resources. We propose and evaluate new load balancing techniques that blend pipeline parallelism with intra-query parallelism. We consider relational operators as pipelines of fine-grained operations that can be located on different cluster nodes and executed in parallel on different data subsets to best exploit non-uniform resources. Our new techniques thus partition both operations and data across cluster nodes. They can be incorporated in the architecture of existing parallel database systems. We present an experimental study that confirms the feasibility and effectiveness of the new techniques in a parallel execution engine prototype based on the open-source DBMS Predator.

1. Introduction

Now that clusters are commonplace, customers want to progressively scale-up their configurations. Not all customers are ready to wait until their cluster is outdated to replace it by a newer, larger one (as it has been the case in the research community for years). Typically, customers want to add (or remove) computing or storage capacity as their business evolves and as technology changes.

A flexible and cost-effective way to adapt a cluster configuration to changing needs as well as changing technology is to assemble a cluster as a collection of non-uniform nodes with different CPUs, disks or network cards. Dell for instance supports modular clusters of non-uniform nodes with their *modular blades* [GO2].

In this paper, we recognize that modular clusters challenge one of the key assumptions that were made when parallel databases were designed, namely that all cluster nodes are uniform[1]. We propose new load balancing techniques that leverage non-uniform resources in a modular cluster. Before we describe the problem we tackle in this paper and our contributions, let us briefly review essential parallel query processing notions.

1.1. Parallel Query Processing

Parallel databases are popular now that scalability has become a key issue[2]. This is particularly true for very large data warehouses (the top results of the on-line analytical processing benchmark TPC-H are obtained with parallel databases [TPC]).

Parallel query processing relies on the following forms of parallelism:

Independent parallelism: multiple queries are executed on different nodes in parallel. The number of queries that need to be processed at any time limits the number of nodes that can be used in parallel, i.e., the *degree of parallelism*. Only a very large number of small queries make independent parallelism scalable, for example in transaction processing applications.

Pipeline parallelism: single queries are executed by executing their operators on different nodes. Every query consists of a number of such operators that are connected with each other via a producer-consumer pipeline. Operators can be executed on separate nodes. Because different operators consume different resources, pipeline parallelism allows the balancing of resource consumption on different nodes. This form of parallelism is however limited in its degree by the number of operators employed

[1] A limited form of non-uniformity, due to diskless nodes and skew, was envisaged in Gamma [D+90]. The load balancing techniques that were devised in this context simply refine the data partitioning across the nodes, while our techniques adapt the usage of individual resources (see Related Work Section).

[2] Research on parallel database systems started in the late 80s, and nowadays all major database vendors propose a parallel version of their system.

in a query. It also poses problems matching operators to nodes.

Intra-operator parallelism, also commonly known as *dataflow parallelism*: a single relational operator is executed on several nodes on different subsets of the data. Relational operators are connected via data streams established between pairs of cluster nodes (operators such as join, aggregates, sort require data to be repartitioned across all nodes while select, project and merge can be performed on the local data subset). This form of parallelism is a very powerful solution because it leverages the fact that relational query processing is performed on large sets of uniform data that can be arbitrarily partitioned. It is virtually unlimited in its degree because the processed data sets can be partitioned into arbitrary small subsets.

Existing parallel database systems rely on these different forms of parallelism, but use them in separation. Typically, small queries are dispatched to different nodes using independent parallelism. Expensive query fragments with operators such as joins and aggregates, are parallelized on a large set of nodes using dataflow parallelism. Depending on the kind of workload (either transactional or analytical), a parallel database system either uses independent or dataflow parallelism to make query execution scalable. The use of pipeline parallelism is limited due to the small number of operators in a typical query.

Load balancing is achieved by attributing large subsets of the data to the nodes whose resource usage is low. This form of load balancing relies on data partitioning (while all nodes execute the same relational operations).

Load-balancing techniques only based on data partitioning however ignore (1) hardware heterogeneity in a modular cluster, as well as (2) the skew of resource usage and availability[3] and (3) interferences between various tasks executed on the same node (e.g., background OS services consume resources at varying rates)[4].

As a result, existing load balancing techniques fail to adapt to the resources situation on individual cluster nodes. They under-utilize resources and thus fail to avoid bottlenecks. We further discuss this problem in Section 2.

In the rest of the paper, we address the following issues:

1. How can existing load balancing approaches based on data partitioning be complemented with new techniques that allow the needed fine-grained adaptation to individual node resources in a modular cluster? We should avoid techniques that are hard to implement, involve high overheads, and do not allow adaptations for individual resources.
2. How feasible and effective are the new techniques when implemented in a prototype? We seek effective, low-overhead execution techniques that allow trade-offs in the utilization of individual resources. The techniques should be usable in existing parallel database systems without major redesigns.

1.2. Contributions

This paper shows new ways in which parallel query processing can be adapted to *non-uniform resources* – resources such as CPU, disk or network bandwidth which vary across the components of a modular cluster. The problems posed by non-uniform cluster resources have been studied for a few years (e.g., [DF+98][AP+00][BK+02]); this paper is the first to study them in the context of parallel query processing.

We focus on the execution of complex read-only queries over object-relational data that characterize data warehouse workloads. The complexity of these queries stems from the combination of relational operators, e.g., equi-joins, with expensive type-specific operations, like analysis methods over time series. Parallelism is critical to improve the execution time of such queries.

We demonstrate the limitations of load balancing techniques based on data partitioning, and we recognize that, in order to leverage non-uniform resources, it is necessary to adapt the kind of work that is performed on each cluster node in addition to adapting the amount of data that is performed.

We propose a new query execution framework that combines the scalability of dataflow parallelism with the scheduling of resources made possible by pipeline parallelism. We consider that expensive relational operators are themselves a pipeline of fine-grained operations. The data streams established across cluster nodes to support dataflow parallelism, allow these fine-grained operations to be added, reordered and moved from one cluster node to another to optimize the resource usage of individual cluster nodes. The proposed new framework combines two known form of parallelism (dataflow and pipeline) in a unique way that is far more powerful than each of them separately. Our load balancing techniques complement traditional balancing techniques based on data partitioning.

This paper makes the following contributions:

1. We show that the traditional data-flow paradigm, which executes the same operation on balanced data subsets in parallel, is limited in its adaptivity to non-uniform resources.
2. We introduce an execution paradigm in which fine-grained operations can be applied individually to data streams that transfer data between individual nodes.

[3] Resource availability varies with the local component's performance. For example, each disk delivers varying bandwidths depending on track position and data fragmentation. Network interface cards vary in the actual bandwidth depending on switch topology and scheduling. (see also [A+99])

[4] Note that the former point is a characteristic of modular clusters while the latter two points are related to dynamically changing phenomena occurring in any cluster node.

To allow for finer grained adaptations, we introduce the notion of 'virtual substreams'– identifiable subsets of a data stream that can each be processed differently.

3. We present load-balancing techniques for this new parallel query execution paradigm. For individual data streams, these techniques migrate processing between sender and receiver, add extra processing like compression, employ alternative algorithms, or reroute data to a third node.
4. We presents performance results that show the effectiveness of the proposed migration and rerouting techniques These results were obtained using a prototype environment based on the open-source DBMS Predator.

This paper is a necessary first step towards parallel database systems adapted to modular clusters. Further work is under way to understand the relative importance of the load balancing techniques we propose and to get insights on appropriate load-balancing strategies. We recognize also that our load balancing techniques introduce a great deal of complexity because they deal with individual node resources. Resource monitoring and dynamic query optimization techniques need to be developed to handle this complexity. These are great topics for future work.

2. Traditional Parallel Execution

In this Section, we show the limitations of data flow parallelism and the limitations of load balancing technique solely based on data partitioning in the context of clusters with non-uniform resources.

2.1. Data Flow Parallelism

Data Flow Parallelism is the classical form of intra-operator parallelism ([DG92]) – parallelism is achieved by executing the same operation on different data subsets in parallel. On each node, relational operators process the local *partition*, i..e, the subset of the data present on the node.

Some operations, like joins or aggregates, cannot be correctly executed on arbitrary partitions. For example, an equality join has to process together all tuples that are possibly equal on the join column. All data that could possibly be combined by an operation have to be collocated in the same partition, i.e., on the same node.

For this reason, partitions usually have to be changed between two such operations. In addition, the number and the sizes of the partitions might need readjustment [C+88,MD93,MD97,RM95]. This process of changing partitions is called *repartitioning.* It involves a data stream between each pair of involved nodes: Every node *splits* its existing partition according to the new partitioning, and sends each fragment to its new location. On the receiver side, every node receives such fragments from all nodes and *merges* them to form its new partition.

Figure 1 shows this data flow for a pipeline of three operations with two interleaved repartitionings. The operations are 'SPJ operators', each consisting of a select, a project and a join. It is assumed that the data are initially distributed so that tuples that might be joined in the first operation are collocated on one node. The two repartitionings will establish semantically correct distributions for the other two joins.

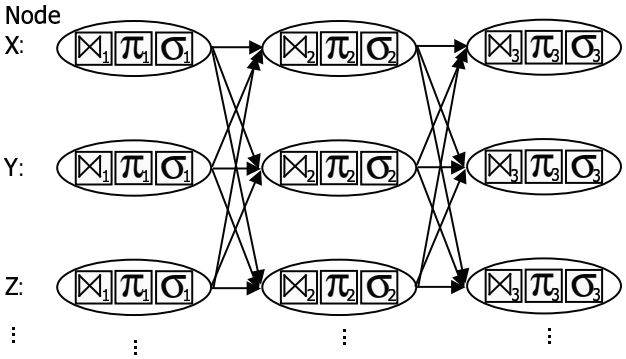


Figure 1: The Classical Data Flow Paradigm. Each node executes a select-project-join (SPJ) operator (that regroups a join, projection (Л) and selection (б))

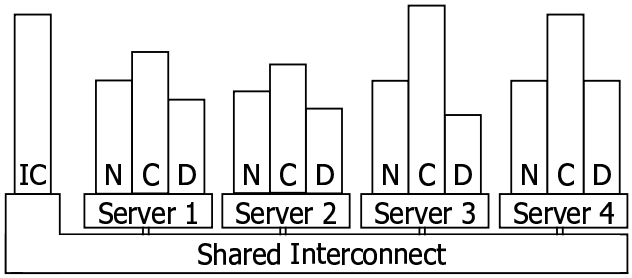
2.2. Non-Uniform Resource Environments

We use a simplified, bandwidth-centric view to discuss the resource environment of parallel query processing (for realistic models, as used to monitor and predict availability, see related work in Section 5). In our example in Figure 2, each node has CPU, Disk and Network Access resources while all nodes share their interconnect[5] - each resource's bandwidth is shown as a vertical bar. Servers 1 and 2 are of an earlier hardware generation than 3 and 4, which is why the latter have faster CPUs. Server 4's variation to 3 is *non-uniform* – each resource varies independently. For example, its disk delivers at a higher bandwidth. At the same time, its CPU is slower. We recall that non-uniform resources are due to hardware heterogeneity in a modular cluster as well as resource skew or interference.

With uniform resources, different nodes can be fully characterized by simply giving their relative capacity – they are not distinguished by the proportion in which their resources are available. But hardware differences as well as skew and interference do not allow this abstraction and as a consequence we consider each resource individually. The next section will demonstrate the problems of traditional techniques in this new environment.

[5] The networking bandwidth corresponds to the node's specific bandwidth limitations for inter-node communication, while the interconnect represents the bandwidth limitations on the accumulated communication between all nodes.

The presented model should not insinuate that resource availability could statically be predicted with any precision. Dynamic monitoring and ad-hoc adaptivity are needed to apply our techniques. Our focus here is to develop and validate query execution techniques. Resource monitoring and query optimization are topics for future work.



Simple bandwidth model with per-site resources CPU (C), disk (D), and network access (N), and one shared resource, the interconnect (IC)

Figure 2: Example Architecture. Server1 and Server 2 have uniform resources; while Server 3 and Server 4 have non-uniform resources.

2.3. Existing Load Balancing Techniques

In contrast to pipeline and multi-query parallelism, which afford flexibility in scheduling the different operations across the available nodes, data parallelism executes the same operation on all nodes. Fortunately, data repartitioning not only ensures the collocation of related records, it also allows adjusting the volumes of data processed on each node. This is called *workload balancing*. The size of the partitions is optimal if the overall execution time is minimized[6]. This is the case only if all nodes need the same amount of time to process their workload. If certain nodes would need more time than others, distributing some of their workload among the idle nodes could reduce execution time.

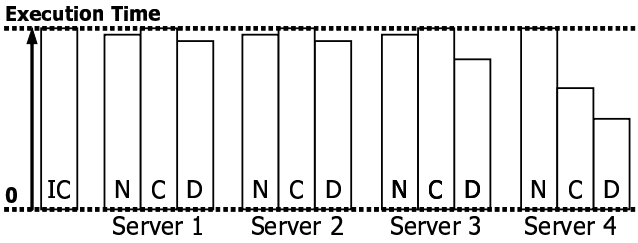


Figure 3: Traditionally Balanced Execution on the System of Figure 2

Figure 3 shows such a balanced execution: The vertical dimension represents utilization time for each resource, with the maximum utilization time being the overall execution time. It can be seen that the balancing of the local amounts of data across the nodes can only adapt to *uniform resource distribution.* For example, if all of a node's resources are uniformly reduced by 10%, as for Server 2 compared with Server 1 in Figure 2, a corresponding reduction of the workload can lead to balanced utilization. But workload balancing will not prevent the under-utilization of resources that are distributed non-uniformly, as for Server 3 and Server 4 in Figure 3. Some of their resources are underutilized because the workload does not match the local resource availability. To determine the execution time of a node with respect to the given operation, only the resource that is utilized most matters. In our bandwidth-centric view, this *bottleneck resource* dominates the execution time and its bandwidth becomes the *effective bandwidth* of the node.

The problem with workload balancing is that we can only vary the workload per node, not per resource. To fully leverage non-uniform resources it is necessary to adapt the kind of work and not only its amount.

3. A New Execution Paradigm

Consider the data flow scheme shown in Figure 4: It shows all opportunities to execute algorithms on the data as ellipses. We speak of the *execution scopes* of algorithms, viewed as a combination of the place and the timing of the execution, and the set of processed data. Possible data sets are the data partitions and the data streams between the nodes. Possible execution places are simply the nodes of the system. Possible execution times are the stages of the pipeline, subdivided into the five different phases that we introduce in the following.

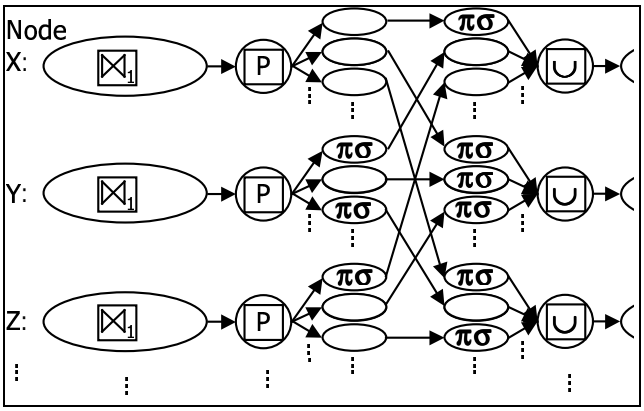


Figure 4: Migrating Operations. Each node executes a pipeline of operations equivalent to an SPJ operator (the shown operations are joins, partitionings (P), projections (π), selections (б), and unions(U)). The projection and selection operations are regrouped on node Y.

Say we have *n* nodes, then the execution scopes are, for each stage of the pipeline and for each node:

[6] In the context of this article we do not consider the overheads of full declustering [C+88] or the throughput issues in the context of multiple parallel queries [C+88,RM95,MD93].

1) The *incoming* phase: On each of the *n* received fragments of the new partition on the data streams.
2) The *merging* phase: While merging these fragments into one partition.
3) The *merged* phase: On the whole partition, after merging.
4) The *splitting* phase: While splitting the partition into the *n* outgoing fragments for the following repartitioning.
5) The *outgoing* phase: On each of the *n* fragments of the partition that go out onto the data streams.

Figure 4 shows the five phases of each stage with their execution scopes. The scopes of the merged phases are those of the original data flow paradigm and form only a subset of the scopes in the extended paradigm.

Per pipeline stage, there are $2n^2+3n$ scopes – independent opportunities to apply algorithms to parts of the data. In contrast, the traditional data flow paradigm applied algorithms identically on all nodes during the merged phase, only varying the amounts of data on each node.

Our proposed solutions fall into four different categories[7]:

- Migration of processing: We migrate algorithms that use specific resources from nodes that overutilize these resources to nodes that underutilize them.
- Additional processing: We introduce additional processing, like compression, which trades off available resources against overutilized ones.
- Alternative processing: We use alternative implementations of the same operations in different resource environments.
- Rerouting: We reroute the data transfer between certain overutilized nodes to allow processing other nodes that have available resources

We present techniques from all areas, but the focus of our experimental work will be on the first and the last one, which promise the greatest improvements over the traditional approach[8].

3.1. Migrating Operations

Considering the operations in Figure 4, we realize that only the joins have to be executed on each partition as a whole – in the merged phase. Selections and projections can also be correctly executed on each of the fragments of the partitions that are sent out to other nodes. They are not bound to any particular partitioning of the data and can be applied separately to the subsets of the partition on the outgoing data streams – i.e., they can be applied in the outgoing phase and migrate across the data streams. Moving selections and projections is relevant because they include complex type-specific methods applied to the data (e.g., user-defined functions).

We migrate operations along the data streams by applying them on the sending node for some streams and on the receiving node for others. Figure 4 illustrates this for a simple case, where selections and projections are migrated away from the upper two nodes. Once the streams are merged on the receiver nodes, the operations must have been applied to all of them. We evaluated this option in our experiments in Section 4.4.

3.2. Migrating Joins

Joins have to happen on each merged partition as a whole. If they were executed separately on fragments of the partition, not all possibly joinable tupels would be combined. Nevertheless, the incoming data can be prepared on their source nodes. For example, for a sort-merge join, the incoming fragments could already be sorted and would simply be merged when the partition is constructed. Only nodes that have available resources would sort before sending off their partitions, while others would leave the sorting to the receiver.

This technique allows *migrating part of the join* from one node to another despite of the mentioned constraints. Its applicability strongly depends on the available join algorithms. Preferably, these algorithms should be structured to allow preprocessing on parts of the data.

3.3. Migrating Data Partitioning

The last two subsections discussed how to migrate selections, projections, and parts of the join. The other operation consuming resources is the splitting of the partition into fragments for the outgoing data streams. This splitting prepares the next join, by partitioning the local subset of the data with respect to the new join column. It can be prepared by tagging all data with its future partitions. Splitting would then simply dispatch the data according to the tag. We can migrate tagging across incoming data streams to some of the sending nodes.

3.4. Selective Compression

This technique trades off CPU bandwidth on a pair of nodes against the network bandwidth between the nodes. Compression and decompression can be applied to the partition fragments sent to other nodes during repartitioning. Thus the decision about compression can be made individually for each pair of nodes, utilizing only the underutilized resources to relieve the network.

[7] The techniques discussed in this section are important points in the new execution space, but they do not exhaust this space. The formal model presented in [M+00] allows us to map out the complete space, showing all possible ways to apply given operations to data on a given architecture.

[8] The presented techniques will attempt to use underutilized resources as much as possible to reduce the usage on other resources. In the larger context of pipelined, independent and multi-query parallelism, there will actually be a tradeoff between the amount of underutilized resources used and the amount of utilized resources freed.

3.5. Alternative Algorithms

There are usually many different implementations for a given operation that has to be processed in parallel on multiple nodes. Implementations can be chosen for each node independently, as long as the partitioning of the workload before the operation and the repartitioning of the results work independent of the particular implementation. This technique finds its limitation in the variety of resource usage of different implementations of the same operation. Presumably, the operation will determine the usage to a large degree.

3.6. Rerouting

Assume an operation can be migrated on a data stream, but both involved nodes are overutilized on the relevant resources (compared to other nodes). In this case migration between the nodes leaves us only the choice between two bottlenecks. Instead, we can trade off network resources and the resources of a third node against the overutilized ones on that particular stream. This can be done through rerouting.

The sender redirects its outgoing stream to a third node that has the needed resources available. This node receives the stream, processes the problematic operation on it, and forwards it to the original receiver node. This technique is useful whenever the interconnect is underutilized and a whole group of nodes[9] is short on resources required for a certain operation.

Rerouting complements migration: the latter applies when the receiver can do the sender's work, the former applies when only a third node can do it. We evaluated this option in our experiments in Section 4.5.

3.7. Summary: Streams and Substreams

In our discussion so far, the new techniques are applicable to individual data streams between pairs of nodes. In contrast to uniform pipelining across all streams, each stream is viewed as an individual pipeline, independent in its ordering of operators, its timing of the network transfer, its additional operations, and its implementation alternatives. This results in a finer granularity for adaptations to the specific resource situation on each individual node.

In principle, it is possible to extend the techniques presented in this Section to process differently subsets of records within a stream. We call these subsets *virtual streams.* The granularity of these substreams is only constrained by the necessity for sender and receiver to identify the substream to which each record belong, e.g., using tagging or a counting mechanism.

For example, an expensive function that filters records can be applied to some records before a network transfer, to others afterwards. The choice might be motivated by the dynamically changing CPU availability on the sender and receiver nodes. The receiver node has to distinguish records that have already been processed from those on which processing has been delayed. To allow this distinction, the headers of unprocessed records could be tagged by the sender, or alternatively, the sender could interleave marker records that turn filtering on the receiver on or off. Each such mechanism introduces specific overheads that have to be weighed against the benefits of the finer granularity of resource scheduling[10]. To summarize our approach, we introduce processing adjustments on subsets of the data to adapt the resource usage on each node to its specific availability. We do this with increasingly fine granularity – on smaller and smaller subsets – by considering data in partitions on individual nodes, data in streams between pairs of nodes, and data in 'virtual substreams' of streams (on subsets of the stream data that are identifiable by both sender and receiver).

4. Experimental Evaluation

This section presents an experimental study of two load balancing techniques introduced in Section 3. This study illustrates the potential benefits of our new parallel query processing framework. We implemented a prototype and performed an experimental study of their feasibility and effectiveness.

4.1. Parallel Execution Engine Prototype

Figure 5 shows our prototype architecture: Independent instances of a non-parallel database server are running on the two depicted nodes. They execute query plans whose in- and outputs are redirected to a communication layer that exchanges the data streams through the interconnecting network.

We used the existing Predator OR-DBMS for the local execution engines and implemented the communication layer. The control mechanism is a shared client that scripts the execution for the database engines on the different nodes.

The communications layer is a simple version of a river system [A+99]. A *river* is a communications abstraction that connects programs on different nodes of a cluster through *data sources* and *sinks*. All data sent through the sinks are redistributed by the river to the different sources across the cluster nodes. Rivers encapsulate all issues of parallelism and data flow balancing in parallel programs, quite similar to the 'exchange operators' of parallel database systems [G94]. The specifics of our implementation of streams of database records in terms of the underlying OS (Windows 2000) and network

[9] This group could be the original core of a cluster that was incrementally upgraded with more powerful machines.

[10] For a similar approach, using tagging of records to allow individual ordering of join operations, compare 'Eddies' [AH00].

abstractions are described in [MG00]. The key feature of our river is that data streams can be manipulated independently: operations can for example be migrated or added individually on each single stream between two nodes.

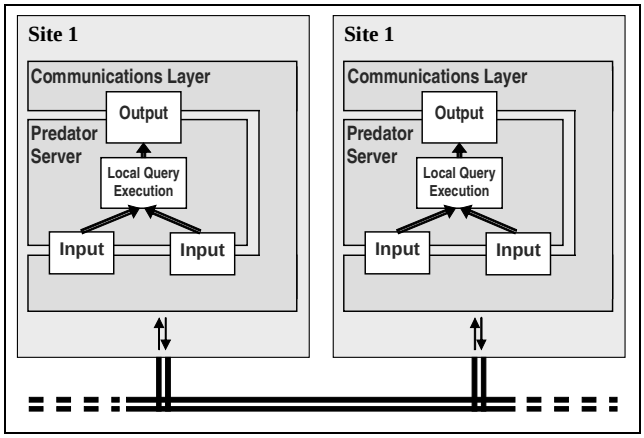


Figure 5: Architecture of the Parallel Execution Prototype

The control and monitoring of the parallel execution is based on Predator's existing client-server architecture. A special 'controller client' contacts all involved servers and sends each the necessary requests for the execution of its local fragments and for the proper connections of its sinks and sources. The client thus executes scripts that control the parallel execution across all nodes and then collects the resulting performance reports. Once initiated, the data flow between the different nodes does not require any centralized control.

4.2. Experiments

Given the described parallel execution prototype it is fairly easy to set up different scenarios of parallel query executions. We present two scenarios to explore the feasibility of the new execution techniques presented in Section 3. Our experiments cover the following two cases:

- Migration of operations to vary the usage of resources across different nodes
- Rerouting of data streams to leverage additional resources.

We present these experiments as prototypical studies of the various possibilities for adaptive techniques in our extended parallel framework. The results that we show prove the soundness and feasibility of the concepts described in Section 3, but they are certainly not exhausting the technical possibilities or the scenarios for their application.

4.3. Experimental Setup

We chose a small parallel system with a particular set of executed operations as starting point for all the following scenarios. Our focus is on the specific features of each examined technique and not on the layout of realistic, complex setups.

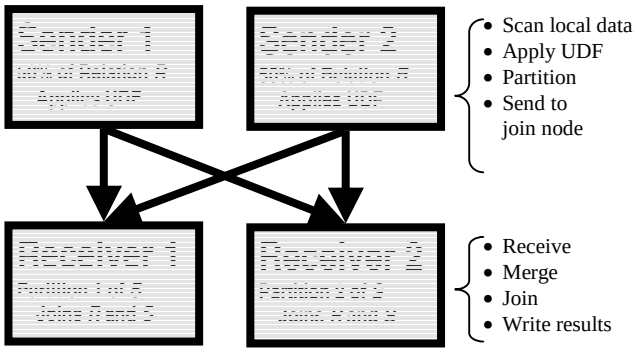


Figure 6: Experimental Setup

Figure 6 shows the basic architecture and the operations executed: A relation, *R*, of 100,000 records is distributed between two 'sender' nodes, while a second much smaller relation, *S*, is distributed between two other, 'receiver' nodes. The size of the second relation is chosen for each experiment to generate the desired join costs between the local partitions on each node. *R* is initially distributed evenly across the two sender nodes and needs to be repartitioned for the join. The resulting partitions area again balanced. A complex user defined function (UDF) has to be applied to each record before the join. In each scenario the basic setup is to apply the UDF early, i.e., on the sender nodes before transferring them to the receiver nodes. This setup expresses the assumptions that the receiver nodes are fully utilized by the join and that the initial distribution of *R* across the sender nodes is balanced with respect to the UDF application costs. Each scenario introduces deviations from these 'balance' assumptions and shows how the exemplified techniques can be employed to adapt to these deviations.

4.4. Migration of Operations

In this experiment we show how we can react to performance perturbations on an individual node by moving operations across its data streams. Figure 7 shows our experimental setup with Sender 1 and its outgoing data streams highlighted. In this experiment, the UDF cost on this sender is varied from that on Sender 2 to simulate performance skew that was not considered in the original setup of the execution.

Migration of operations allows individual decisions on each data stream to apply certain operations before or after the network transfer. In our scenario, we would delay the CPU-intensive UDF on the streams that originate from the first sender to deal with a higher CPU usage of the UDF on that node. This trades off CPU usage on Receiver 1 and 2 against usage on the overutilized Sender 1.

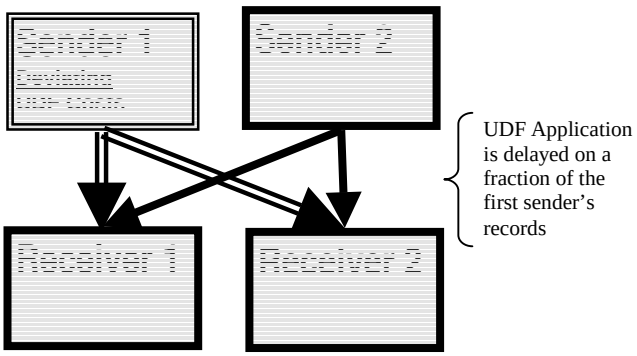


Figure 7: Migration Scenario

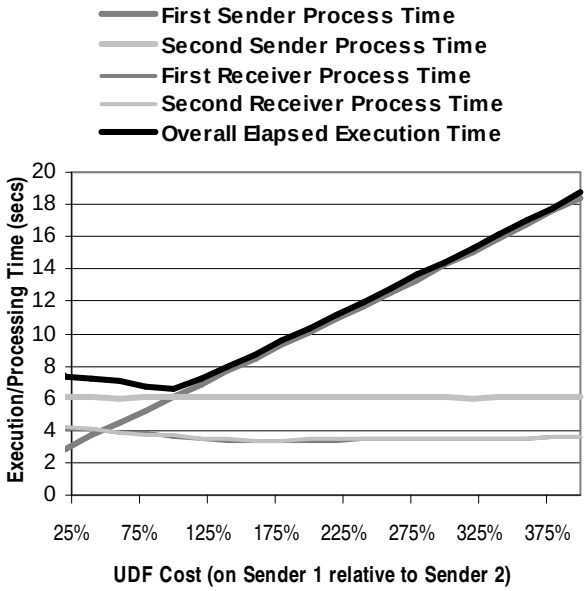


Figure 8: Effect of UDF Cost Deviation on Sender 1

We start with a graph that shows the effect of the UDF cost deviation without any adaptations. Figure 8 plots the overall execution time and the processing times on each of the nodes on the vertical axis, while the UDF cost on Sender 1 is varied along the horizontal axis. The times are shown in seconds while the UDF cost is given relative to the constant cost for the UDF on Sender 2.

It can be seen that the processing time on the first sender is linear in the UDF cost while that on the second and on both receivers is constant. The CPU cost on the sender (extended to the left of the shown graph) does not pass through the origin because there is a constant cost component involved that results from reading the records from disk and sending them to the receivers. Only at 100% the two senders' CPU costs are balanced. Before that point the overall elapsed time apparently results from the receiver CPU cost. The constant distance between the receiver and the overall time curve is explained through an additional cost component on the receiver: A large part of I/O work is done by the operating system in kernel threads and not by the measured process in either kernel or user mode. This work happens in deferred procedure calls (DPCs) that handle the completion of I/O operations.

Another interesting observation is that the elapsed time actually decreases as the utilization of the first sender increases. This could be explained by the adjustment of the rate at which data are sent to the rate at which they can be received. Sending data faster than the receiver can process them causes additional costs on the receiver due to buffer flooding. This observation is not relevant to our demonstration of the migration technique.

After 100%, the elapsed time is dictated by the first sender as the bottleneck of execution. The CPUs of the other nodes are underutilized, even considering the constant DPC overhead for the receivers. In this experiment, we attempt to leverage the underutilized receiver resources to lower the utilization of the first sender and thus lowering the overall execution time.

To do this we delay UDF application on a fraction of the records on each of the streams that originate from Sender 1. Using identical counting mechanisms for the records on the sender and on each receiver, both can identify the records that belong into this fraction. Accordingly, the sender will let them go unprocessed while the receiver will apply the UDF. If a filter operation were to drop records after the UDF application on the sender but before that on the receiver, a more sophisticated mechanism, for example tagging, would be necessary to identify the delayed records. In our setup, a receiver incurs a cost per UDF application identical to the deviating one on the first sender.

Along the vertical axis, Figure 9 shows the same times as Figure 8, while this time not the UDF cost but the *delayed fraction* is varied from 0% to 60% along the horizontal axis. The cost deviation on Sender 1 is fixed at 200% of the cost on Sender 2. The situation at 0% delayed fraction corresponds to that in Figure 8 for 200% UDF cost. For larger delayed fractions, the CPU utilization on Sender 1 decreases because more and more of the UDF applications happen on the two receivers. As the bottleneck cost on Sender 1 decreases, and with it the execution time, the CPU usage on each receiver nodes increases at half that rate. We redistribute processing from an overloaded node to two underutilized nodes.

At 34% the minimum execution time is achieved because after this point the increasing receiver utilization will also increase the execution time. The constant distance between the CPU times on the receiver and the execution time are again explained by the 'hidden' costs of network receiving, incurred in kernel threads that we do not measure. We ran these experiments for different CPU costs on Sender 1, and as expected, the observations are qualitatively the same as in the shown graph, but for higher costs shifted upwards and to the right. For any cost deviation, we can thus experimentally determine an optimal delay – which can also be confirmed by a simple

analysis of the balancing of costs on the sender and the receivers.

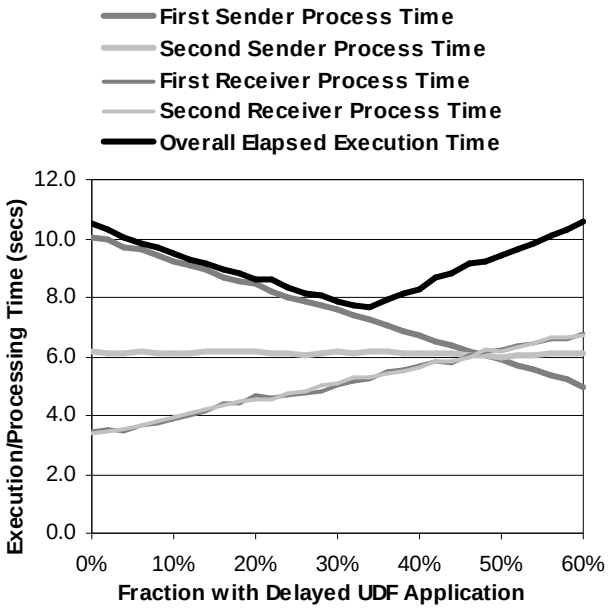


Figure 9: Effect of Delayed UDF Application for 200% UDF Cost

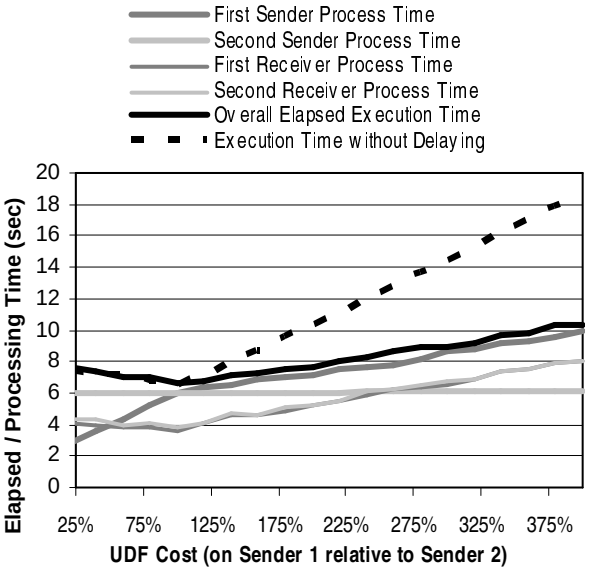


Figure 10: Increasing UDF Cost Deviation with Optimal Migration

In Figure 10, we summarize the possibilities of operator migration by varying the UDF cost along the horizontal axis while using an estimated optimal delayed fraction for each cost. In addition to the resulting execution time, we show the original execution time from Figure 8 as an interrupted line. The difference between these two lines is the benefit derived from the migration technique. It can be observed that delaying balances the cost on the first sender with the cost on each of the two receivers so that both equally affect the overall execution time and neither forms a bottleneck (again, the actual receiver cost contains constant kernel thread costs that are not shown). On the right side of the graph, beyond 250% it becomes apparent that the second sender is underutilized, as it is not part of the balancing through UDF migration. The next experiment will focus on leveraging the second sender to alleviate overload on Sender 1.

4.5. Rerouting of Data Streams

Rerouting introduces new intermediate nodes into an existing data stream to put their resources to use for operations that can be migrated along that stream. In our setup all records are rerouted, no matter what fraction of them will actually be processed on the intermediate node, which we called Sender 2. Similarly to the last scenario, we will vary the fraction of records that is 'delayed', but this time delaying leads to processing on Sender 2. Receiver 1 and Receiver 2 will not do any processing.

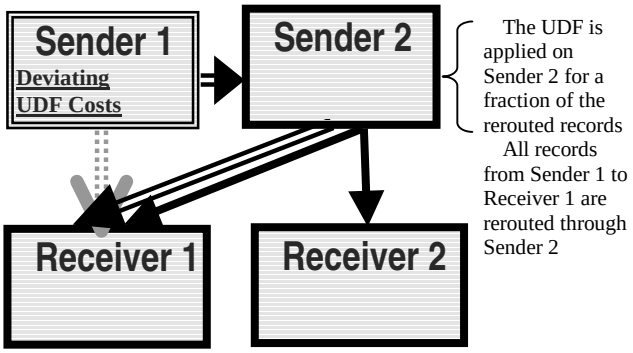


Figure 11: Rerouting Scenario

To emphasize the specific benefits of rerouting, we use a different scenario than the one in the last section. The cost of the UDF on Sender 1 is again being modified, while the UDF on Sender 2 is this time cheaper in comparison to the join costs on the receivers. This simply means that Sender 2 is more apt to relieve the bottleneck Sender 1 than the receivers, which were used in the last scenario. Figure 12 shows the effect of UDF cost deviation on the first sender: Along the x-axis we increase the UDF cost on Sender 1 relative to that on Sender 2. We plot the processing times on the four components and the overall execution time as in Figure 8. A key feature in this modified scenario is that the stream between Sender 1 and Receiver 1 is rerouted through Sender 2. This happens for all records, even when no processing is delayed and everything is processed on Sender 1, as here in Figure 12. Sender 2 is affected in its CPU usage by this rerouting: We plotted the usage excluding rerouting as an interrupted line, running at about 80% of the overall usage on Sender 2. It can be observed that the processing time on Sender 1 dominates the overall execution time as it becomes higher than the execution time on the receiver nodes.

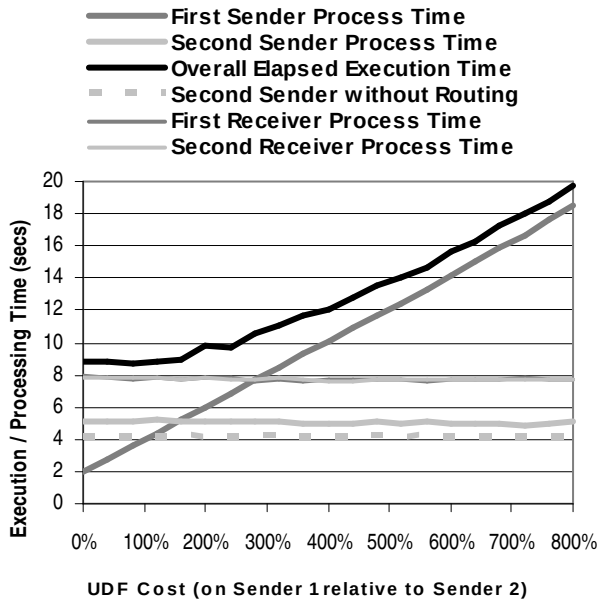


Figure 12: Effect of UDF Cost Deviation on Sender 1

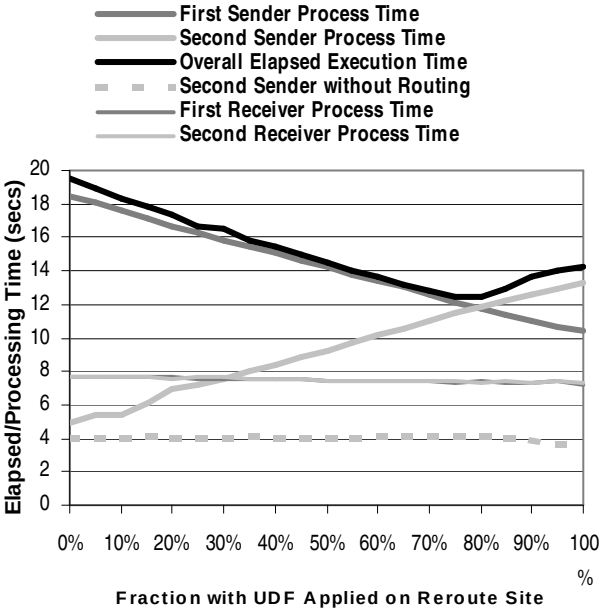


Figure 13: Effect of Delayed UDF Application for 800% UDF Cost

Figure 13 shows the execution and processing times for an arbitrary fixed UDF cost of 800%. The fraction of records that is processed on the reroute node Sender 2 is varied along the horizontal axis. We observe that, while the receivers are this time not affected, the cost on the bottleneck Sender 1 is reduced while the rerouting costs (above the fragmented line) on Sender 2 increases at the same rate. Below 80% processing on the reroute node this reduces the execution time because Sender 1 is the bottleneck. Beyond that point Sender 2 becomes a new bottleneck, increasing the execution time.

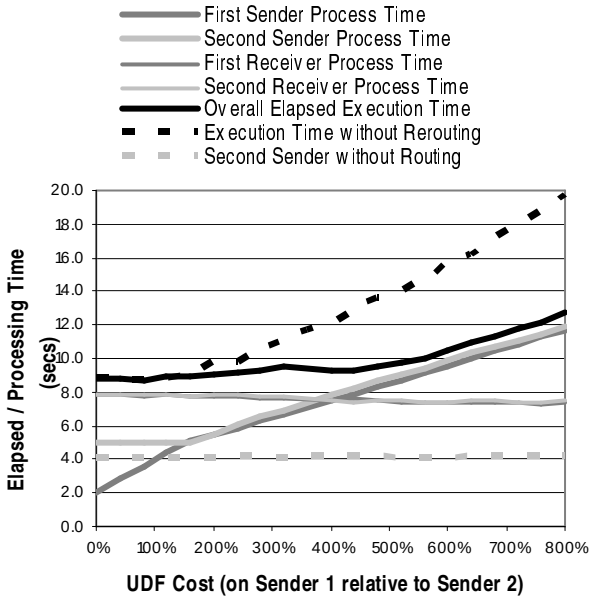


Figure 14: Increasing UDF Cost Deviation with Optimal Rerouting

With similar experiments the optimal reroute fractions can be determined for various UDF costs. Our experiments confirm the analytical result that *(C1-C2) / UC1* is the optimal fraction, where *C1* is the overall cost on Sender 1, *C2* that on Sender 2, and *UC1* the UDF costs on Sender 1. *C1* is constituted by the basic sending cost and *UC1*, *C2* by the same basic cost, the rerouting cost and *UC2*. *UC1* is 800% of *UC2* in the example above. The formula *(C1-C2) / UC1* determines the fraction that the overhead of Sender 1 forms relative to the overall UDF cost on Sender 1. Actually, only half of the UDF cost is on the stream to Receiver 1 and thus reroutable, but this factor is neutralized by the fact that only half of the overhead fraction should be rerouted to balance both costs: *(C1-C2) / UC1 = ((C1-C2) / (UC1/2))/2*. We used these analytical and experimental results to optimally balance an increasing UDF cost using rerouting, analogously to what we did for migration in Figure 10. The results are shown in Figure 14. Again, the benefit of rerouting can be seen as the difference between the original execution time, shown as an interrupted black line, and the adapted execution time, shown as a thorough black line.

4.6. Discussion

These experiments illustrate the potential benefits of the load balancing techniques we have introduced in Section 3. Further work is needed to evaluate their actual impact and their relative importance. Questions remaining to be answered include: When is data rerouting an appropriate

strategy? What are the kinds of resource skews we can expect (will we regularly reach resource skews of 800%)? What is the impact of our load-balancing techniques on the overall performance of the system? These are topics for future work.

5. Related Work

In parallel systems, *data placement* is crucial for workload balancing. The processed relations have to somehow be partitioned across the parallel nodes of the system. Gamma [D+90, GD90] uses 'full declustering': relations were distributed evenly across all nodes. In contrast, Bubba [B+90] tried to find the optimal data placement. [C+88] considered the 'heat' of the data, i.e., their access frequency. Depending on their heat, relations should be spread across only a subset of the nodes. Other systems [T87, D+90], examining multi-query loads, find near-linear scaleup for declustering relations across as many nodes as possible. Our focus is on the execution of a single query without the benefits of locality of data access. Consequently, the queries in our prototype access the fully declustered data uniformly.

[D+92, MD93, MD97, SD89] examine the problem of *workload skew* within Gamma. Skew can emerge from hash-based partitioning, from joins, and from duplicate values [WDJ91]. Virtual processor scheduling [D+92] (similar to the 'data cells' of [HL90, HL91]) uses many small partitions per processor, some of which can be migrated between nodes. [RM95] examines how workloads should be balanced dynamically in a multi-query environment. We complement both static and dynamic workload balancing with new fine-grained per-resource tradeoffs.

Researchers [JM98,NM99] have explored the *parallelization of user-defined functions* because purely relational techniques are unsatisfying for object-relational systems. The focus is on aggregate UDFs that require a specific input ordering and that allow special forms of partitioning of data streams into 'windows' (the granularity of processing of parallel clones).

Dynamic self-regulation of the *data flow* in general parallel systems is the goal of the River approach [A+99,A99]. Rivers deal with performance skew – dynamic fluctuations in the availability of resources – and are based on previous work on robustness [HD90,CK89]. This kind of flow control unfortunately does not apply readily to parallel join processing because the joined data are partitioned semantically. Depending on the value of the joined attribute, data is placed on a specific node. Rivers were successfully applied to simpler operations, like scans and writes [A99].

DataCutter [BK+02] models data-intensive applications as sets of filters. Data streams connect filters possibly located on different cluster nodes. This modular application architecture allows for a combination of dataflow and pipeline parallelisms; possibly filters can be moved to adapt to individual node ressources. We apply similar ideas to parallel query processing in general.

Abacus [AP+00] considers the skew of resource usage and availability in a client-server application. They propose a run-time system that partitions data-intensive functions between client and server depending on the resources available. Our load-balancing technique based on operation or join migration corresponds to the static placement of an operation in Abacus. Note that our notion of migration could be extended to cover the dynamic migration of running operations in the spirit of Gardens [BK+00] or Emerald [J89].

Recent work examines tools that allow automatic cost/benefit predictions of task placement in clusters [K+01]. Although this work targets scientific workloads based on message passing [OMP] and not on data streams, similar approaches could be applied to determine the workload distribution for parallel query processing. Orthogonally, our techniques add dynamic adaptivity and increased flexibility in the placement of parallel tasks.

6. Conclusion and Future Work

This paper examined the parallel execution of complex queries on systems with non-uniformly available resources. Asymmetry of the available resources and their usage stems from skew, interference, and hardware asymmetry. The traditional workload balancing techniques cannot adapt the usage of individual resources. To complement them, we introduced a new framework that treats the data streams that repartition data from node to node (and their virtual substreams) as individual pipelines of operators. Within this framework, we were able to propose adaptivity techniques for individual resources on a fine granularity. Conceptually, our new framework combines the variability of pipelined parallelism with the scalability of intra-operator parallelism.

We examined the feasibility and effectiveness of the new framework in the context of a parallel execution engine prototype based on the Predator OR-DBMS and a newly implemented communication layer. Our experience was that the new framework fits well into existing parallel architectures and our measurements show that new techniques can provide the adaptivity that is needed for non-uniform resource environments.

Future work includes a complete implementation of our parallel query execution framework on a modular cluster. This work has started at DIKU in collaboration with Dell. Such an implementation will allow us to conduct an extensive performance study. Other areas of future work concern the dynamic resource monitoring and query optimisation techniques necessary to develop a full fledged system.

Bibliography

[A+99] R. H. Arpaci-Dusseau, et al.: Cluster I/O with River: Making the Fast Case Common. IOPADS 1999: 10-22.

[A99] R. H. Arpaci-Dusseau: Performance Availability for Networks of Workstations. PhD Thesis, Univ. of California at Berkeley 1999.

[AP+00] K. Amitri, D.Petrou, G.Ganger and G.Gibson. Dynamic Function Placement for Data-intensive Cluster Computing. Usenix 2000.

[B+90] Haran Boral, et al.: Prototyping Bubba, A Highly Parallel Database System. TKDE 2(1): 4-24. 1990.

[BK+00] Ashley Beitz, Simon Kent and Paul Roe. Optimising Heterogeneous Task Migration in the Gardens Virtual Cluster Computer. Proc. 9th Heterogenous Computing Workshop (HCW 2000).

[BK+02] M.Beynon, T.Kurc, U.Catalyurek, A.Sussman and J.Saltz. Efficient Manipulation of Large Datasets on Heterogeneous Storage Systems'. HCW 2002. April 2002

[C+88] George P. Copeland, William Alexander, Ellen E. Boughter, Tom W. Keller: Data Placement In Bubba. SIGMOD Conference 1988: 99-108

[CW79] J. Lawrence Carter, Mark N. Wegman: Universal Classes of Hash Functions STOC 1977: 106-112

[D+90] David J. DeWitt, et al.: The Gamma Database Machine Project. TKDE 2(1): 44-62 (1990).

[D+92] David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider, S. Seshadri: Practical Skew Handling in Parallel Joins. VLDB 1992: 27-40

[DF+98] T. Decker, M.Fischer, R.Luling and S.Tschoke. A Distributed Load Balancing Algorithm for Heterogeneous Parallel Computing Systems. Proc of the 1998 Int. Conf. on Parallel and Distributed Processing Techniques and Applications.

[DG92] David J. DeWitt, Jim Gray: Parallel Database Systems: The Future of High Performance Database Systems. CACM 35(6): 85-98 (1992)

[G94] Goetz Graefe: Volcano - An Extensible and Parallel Query Evaluation System. TKDE 6(1): 120-135. 1994.

[G02] Randy Groves. Gaining Flexibility with Modular Blades Architectures. Dell Power Solutions. August 2002.

[GD90] Shahram Ghandeharizadeh, David J. DeWitt: A Multiuser Performance Analysis of Alternative Declustering Strategies. ICDE 1990: 466-475.

[GD93] Goetz Graefe, Diane L. Davison: Encapsulation of Parallelism and Architecture-Independence in Extensible Database Query Execution. TSE 19(8): 749-764 (1993)

[HD90] Hui-I Hsiao, David J. DeWitt: Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines. ICDE 1990: 456-465

[HL90] Kien A. Hua, Chiang Lee: An Adaptive Data Placement Scheme for Parallel Database Computer Systems. VLDB 1990: 493-506

[HL91] Kien A. Hua, Chiang Lee: Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning. VLDB 1991: 525-535

[J89] Eric Jul. Migration of light-weight processes in Emerald. Operating Systems Technical Committee Newsletter, 3(1):25--30, 1989.

[JM98] M. Jaedicke and B. Mitschang. On parallel processing of aggregate and scalar functions in objectrelational dbms. In Proc. of ACM SIGMOD, 1998.

[K+01] Dimitrios Katramatos et al.: Developing a Cost/Benefit Estimating Service for Dynamic Resource Sharing in Heterogeneous Clusters: Experience with SNL Clusters. IEEE CCGrid 2001

[M+00] T. Mayr, P. Bonnet, J. Gehrke, P. Seshadri: Query Processing with Heterogeneous Resources. Technical Report TR00-1790, Cornell University, March 2000.

[MD93] Manish Mehta, David J. DeWitt: Dynamic Memory Allocation for Multiple-Query Workloads. VLDB 1993: 354-367

[MD97] Manish Mehta, David J. DeWitt: Data Placement in Shared-Nothing Parallel Database Systems. VLDB Journal 6(1): 53-72 (1997)

[MG00] Tobias Mayr, Jim Gray: Performance of the 1-1 Data Pump. See http://www.research.microsoft.com/~gray/River

[NM99] Kenneth W. Ng, Richard R. Muntz: Parallelizing User-Defined Functions in Distributed Object-Relational DBMS. IDEAS 1999: 442-445.

[RM95] Erhard Rahm, Robert Marek: Dynamic Multi-Resource Load Balancing in Parallel Database Systems. VLDB 1995: 395-406

[S86b] Michael Stonebraker: The Case for Shared Nothing. Database Engineering Bulletin 9(1): 4-9, 1986.

[SD89] Donovan A. Schneider, David J. DeWitt: A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment. SIGMOD Conference 1989: 110-121

[T87] Tandem Database Group: NonStop SQL: A Distributed, High-Performance, High-Availability Implementation of SQL. HPTS 1987: 60-104

[TPC] Top Ten TPC-H Results by Performance. http://www.tpc.org/tpch/results/tpch_perf_results.asp

[UAS98] M.Uysal, A.Acharya, J.Saltz: An Evaluation of Architectural Alternatives for Rapidly growing Datasets: Active Disks, Clusters, SMPs. Technical Report TRCS98-27. University of California at Santa Barbara. 1998.

[WDJ91] Christopher B. Walton, Alfred G. Dale, Roy M. Jenevein: A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins. VLDB 1991: 537-548