Query Processing with Heterogeneous Resources (Technical Report)

Tobias Mayr
Cornell University
4104 Upson Hall
Ithaca, NY 14853
mayr@cs.cornell.edu

Philippe Bonnet
Cornell University
4122 Upson Hall
Ithaca, NY 14853
bonnet@cs.cornell.edu

Johannes Gehrke
Cornell University
4108 Upson Hall
Ithaca, NY 14853
johannes@cs.cornell.edu

Praveen Seshadri[1]
Cornell University
4130 Upson Hall
Ithaca, NY 14853
praveen@cs.cornell.edu

ABSTRACT

In emerging systems, CPUs and memory are integrated into active disks, controllers, and network interconnects. Query processing on these new multiprocessor systems must consider the heterogeneity of resources among the components. This leads to the more general problem of how to deal with performance heterogeneity in parallel database systems.

We study database query processing techniques that increase the leverage of heterogeneous resources. We show that the traditional algorithms used in shared-nothing parallel databases fail to utilize non-uniform resources. Uniform resource usage across non-uniform components leads to resource bottlenecks.

We describe a class of new execution techniques that balance the usage of system resources using non-uniform intra-operator parallelism. We show that these techniques improve performance on heterogeneous architectures by allowing trade-offs between the various resources. Traditional techniques are subsumed as a special case for symmetric architectures.

We show a formal model that maps out the new execution space of alternative processing techniques. A simplified cost model allows analytic performance evaluation of the alternative techniques. The proposed new execution paradigm is an extension of the classical dataflow paradigm.

1 Introduction

This section motivates and explains the problems that arise for database query processing in environments with active components. We describe the technological trends that motivate this paper and how these new technologies should be modeled from the viewpoint of database query processing. We point out the problem of traditional processing techniques and describe our contribution to the solution.

1.1 Motivations

The performance demands on database systems grow with increasing data volumes and processing workloads. The standard approach to building scalable database systems uses off-the-shelf computing components, attached to a fast interconnect, with "shared-nothing" parallel query processing techniques [DG92,D+90,B+90,S86]. But the hardware architectures underlying this approach are changing: Due to continued cost and size reduction of CPUs and memory, processing power is becoming a cheap commodity available on every system component, like disk drives, storage controllers and network interconnects. The emerging class of system architectures consisting of such "active" components, which each contribute their processing power, holds great promise for highly scalable systems [G+97,G+98,KPH98,RGF98, AUS98,UAS98,HM98].

As an environment for query processing, such architectures differ from traditional parallel architectures in the heterogeneity of the involved resources. Processing is not confined to the servers – it can happen on all active components of the system, e.g., disks, storage controllers and clients. The utilized platforms vary widely in terms of processing power, disk I/O rate, and communication bandwidth.

The next subsection shows how to model systems with active components from the viewpoint of relational database query processing.

1.2 Modeling the New Environments

Our goal is to find an abstract model for the new architectures that reflects all aspects that are relevant for query processing. This will allow us to recognize the shortcomings of traditional parallel processing techniques in these new environments.

[1] Praveen Seshadri is currently on leave from Cornell University and employed at Microsoft Corporation.

Because of our focus on the heterogeneity of resources across different components, each individual resource will be modeled with its specific bandwidth. Each site consists of several such resources, and all sites are connected by a shared interconnect, which also corresponds to a resource. Figure 1 shows this structure. In this example, a site consists of the resources processor, disk and networking. The networking bandwidth corresponds to the site's specific bandwidth limitations for inter-site communication, while the interconnect represents the bandwidth limitations on the accumulated communication between all sites.

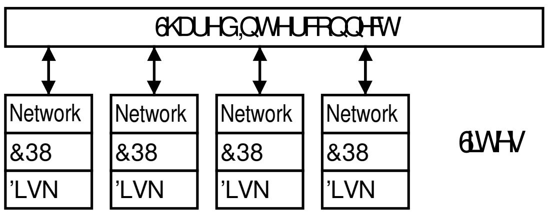
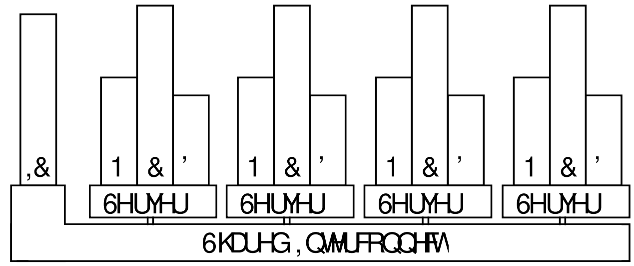


Figure 1: Resource Model

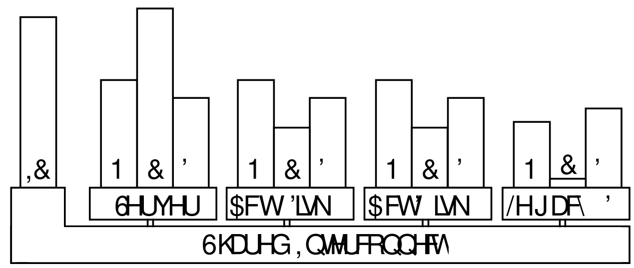
This bandwidth-centric model can represent a broad class of real-life systems. As examples, consider shared-nothing parallel systems, systems with active disks and systems with network attached storage. Figure 2 shows instantiations for these systems in our resource model.

What distinguishes the new architectures that we want to discuss from classical ones? Our concern is that the resources are not *uniform* across the sites of the system: Uniformity means that the different resources are present in the same proportion on each site. Figure 2a) shows an example with uniform resources. Figure 2b) and 2c) are examples for non-uniform resources: In both cases the server has relatively more processing power, while other sites are stronger in either their networking or the disk bandwidth.

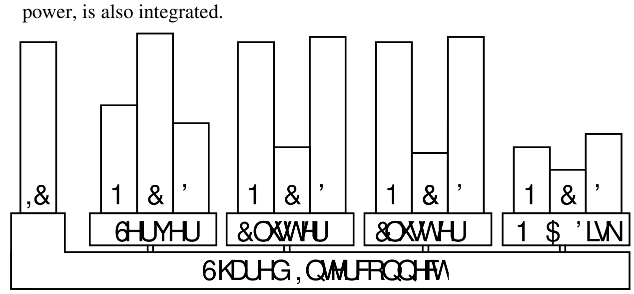
With uniform resources, different sites can be fully characterized by simply giving their relative capacity – they are not distinguished by the proportion in which their resources are available. But the new architectures that we consider here do not allow this abstraction, the model has to represent each resource individually. The next section visualizes the problems of traditional techniques in this new model.



(a) A shared-nothing cluster consists of symmetric processing units each with disks and network access. A high bandwidth interconnect serves as a connection between the components.



(b) This active disk system has two active disks, each with a moderately powerful processing units. An older legacy disk, with little processing power, is also integrated.



(c) This system consists of a server, two clusters of disks with processing power on their controllers, and an active disk that is directly attached to the network.

Figure 2: Example Architectures

1.3 Problems of Existing Techniques

In the traditional approach, the primary way to distribute workload across the sites of a parallel system is the use of *intra-operator parallelism* [D+86]. A relational operation is executed identically on different subsets of the data that are located on the different sites. The sizes of the different subsets are balanced so that the overall execution time is minimized. Figure 3 shows such a balanced execution. No site and no single resource is dominating the execution time as a bottleneck.

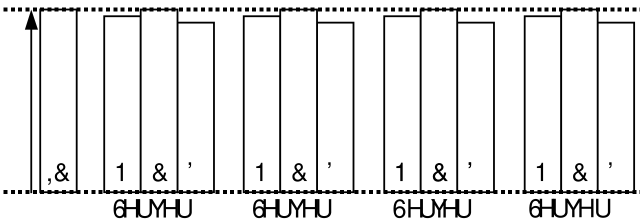


Figure 3: Classical Parallel Execution on the System of Figure 2a)

The existing techniques assume that the resources are distributed uniformly across the sites[2]. This can be seen from the uniform resource usage of these techniques: On each site the same operation is executed, using each site's individual resources in the same proportion.

But for non-uniform resources, balancing the local amounts of data on the sites does not prevent individual resources from being overutilized – forming a bottleneck, while others are underutilized.

Figure 4 shows an example: While the resource usage of the operation is near optimal for the server, it leads to unbalanced use of the resources on the other components – even after adjusting the workloads to have balanced execution times across the sites..

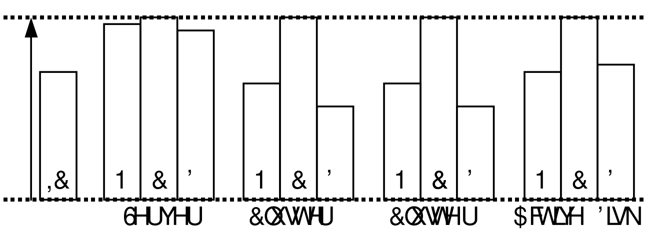


Figure 4: Traditional Execution on the System of Figure 2b)

The problem is that we can only vary the workload per site, not per resource. To leverage the heterogeneous resources it would be necessary to adapt for each site not only the workload size, but also the processing of that workload. The resource usage will be adapted to each site's specific resource availability only after adapting the processing techniques for each site. This paper presents an execution paradigm that allows such adaptivity.

[2] Gamma [D+90] introduced diskless sites as a special case, but did not treat non-uniformity in general.

1.4 Contribution

Based on the realization that the existing processing techniques fail to leverage heterogeneous resources, we propose an extension to the classical data-flow paradigm. Our extension allows the adaptation of the workload processing to each site's specific resource situation. We present techniques that introduce tradeoffs between the individual resources in the extended data-flow paradigm. These techniques demonstrate but do not exhaust the possibilities of the proposed extension to the paradigm.

Our contribution is based on a formal execution model for pipelined operator execution. The goal of this model is to fully reflect the inherent flexibility of the data-flow paradigm, thus allowing adaptation of the execution to heterogeneous resource situations. The benefits of the model are the description of the *execution space* and the *costing* of this space. The execution space is the set of all possible ways in which to execute a given pipeline of operators on a given architecture. It shows all possibilities to adapt the resource usage of execution to the given resource availability of the system. The costing of these possibilities allows us to analyze the expected performance benefits.

Our techniques focus on *intra-operator parallelism* for two reasons:

1. The benefits of pipeline and independent parallelism are very limited compared to intra-operator parallelism. The degree of parallelism achieved is limited by the length of the pipeline and the number of independently parallel subplans.
2. For heterogeneous resources, the adaptation of intra-operator parallelism appears to be the main challenge, while the other forms of parallelism adapt easier[3].

Our contribution is the first step towards parallel database systems that leverage the heterogeneous resources available on active system components. We confined ourselves to the adaptation of operator execution, and left independent and pipeline parallelism for future work. Facing the complex execution space that we propose, resource-adaptive

[3] Work has been done on the effects of inter-query parallelism [C+88,RM95]

query optimization will be challenging. We focused in this work exclusively on the exploration of the execution space, which in our view forms the necessary base for future work on optimization.

2 The Traditional Approach

This section explains the problems that non-uniform resources pose to traditional intra-operator parallelism. The traditional approach attempts to process data uniformly, applying the same algorithms to the data on different sites in parallel [D+90,C+88,DG90,GD93]. For heterogeneous resources, this results in bottlenecks: Certain resources are overloaded and slow down overall execution, while other resources are idle. The solution to this problem is presented in Section 2, where we describe how the idle resources can be used to relieve the overloaded ones.

Subsection 2.1 establishes a basic understanding of the traditional data flow paradigm for intra-operator parallelism. Subsection 2.2 shows this paradigm's limited adaptivity to the underlying resource situation, and points out the resulting bottleneck problem.

2.1 Data Flow

In the classical data flow approach [DG90,DG92], parallelism is achieved by executing the same operation in parallel on multiple sites. On each site, only the locally present data, called the site's *partition*, is processed.

Some operations, like joins or aggregates, cannot be correctly executed on arbitrary subsets of the data. An equality join, for example, has to process all tuples that are equal on the join column together. Data that could possibly be combined by an operation have to be collocated in the same partition, that is, on the same site.

For this reason, the 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 data streams between each pair of involved sites: Every site *splits* its existing partition according to the new partitioning, and sends each fragment to its new location. Every site receives such fragments from all other sites and *merges* them to form its new partition.

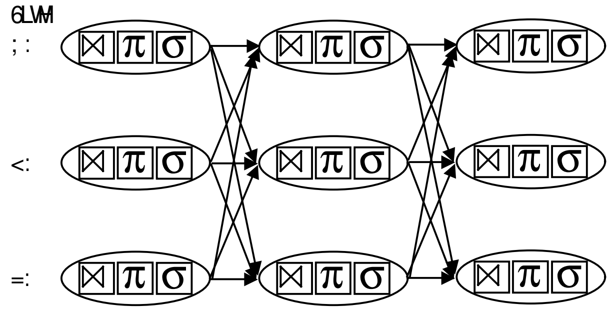


Figure 5: The Classical Data Flow Paradigm

Figure 5 shows this data flow for a pipeline of three operations, with two interleaved repartitionings. The operations are SPJ operators, each consisting of a join, a selection, and a projection. It is assumed that the data are initially distributed so that tuples that might be joined in the first operation are collocated on one site. The two repartitionings will establish adequate distributions for the other two joins.

Besides the collocation of related tuples, repartitionings allow the adjustment of the data volumes that are processed by each site. This is called *workload balancing*. The size of the partitions is optimal if the overall execution time is minimized[4]. This is the case if all sites need the same amount of time to process their workload. If certain sites would need more time than others, execution time could be reduced by distributing some of their workload among the idle sites. For example, Figure 4 shows the result of balancing the workload across the sites of the architecture in Figure 2b). Because of the better resources of the server, workload has been moved from the other sites to the server to achieve equal execution times on all sites and thus to minimize overall execution time.

To determine the execution time of a site 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 site. Every site processes its workload with its effective bandwidth. The next subsection explains in how far this limits performance.

[4] We ignore the issues of overheads for full declustering [C+88] as well as the effects of inter-query parallelism [C+88,RM95,MD93].

2.2 The Limitations of Workload Balancing

For non-uniform resources, traditional techniques will optimize utilization only insofar as no site will be underutilized entirely. Its bottleneck resource will always be utilized for the full execution time. Figure 4 shows how all sites are busy for the same time, because each site's bottleneck resources are utilized for that time. But other resources will be underutilized.

In general, why would some resources of a site be underutilized, while others are fully utilized? In the traditional approach, the same algorithms are executed on each site, leading to the same resource usage on all sites. However, the resources available in environments with active components will be highly non-uniform – they are part of fundamentally different hardware components, like disks, controllers and interconnects. Adjustment of the partition size only leads to proportionally higher or lower usage of all resources on a site. The problem is that the proportion of available resources is different for each site. Thus, the mentioned local bottlenecks are inevitable.

In the non-uniform case, our focus has to be on the underutilization of single resources not on that of whole sites. In Figure 4, the resource usage of the executed operation matches the available bandwidth of the resources only for the server. Every of its resource is fully utilized during the execution time. On the active disk sites, most of the resources are underutilized because these sites are simply very different from the server. The available and unused bandwidth of these resources should be leveraged to relieve the bottleneck resources and thus reduce the overall execution time.

To achieve this, different processing needs to happen on sites with different resources. Sites that have strong CPUs, like servers, should do CPU intensive tasks, while sites with relatively more disk bandwidth should be used mainly on this resource. The classical approach is based on the idea that the same operation is executed on all sites[5]. As we have seen, workload adaptation does only avoid underutilized sites, not underutilized resources.

[5] The simple adaptation of choosing different implementations for the same operation on different sites is limited by the fact that the operation mostly dictates the resource usage.

On clusters of identical components, for which the classical approach was developed, the traditional approach can succeed in fully utilizing every available resource. But in active environments the available resources are a byproduct of a variety of necessary hardware components and thus are inevitably heterogeneous. New techniques are needed to leverage these newly available resources for scalable, faster query processing.

3 New Processing Techniques

Our goal is to use the available bandwidth on underutilized resources to reduce the usage on the bottleneck resources. We achieve this goal by migrating the processing of certain tasks between sites. These tasks have a specific resource usage, which is removed from one site and applied to another. In contrast to workload balancing, where data is migrated, the migration of processing itself leads to a change in the usage of the individual resources on the involved sites. Workload balancing only attacks the problem of site bottlenecks, while our techniques can resolve local bottlenecks on each site.

We can migrate processing by realizing the full flexibility inherent in the data flow paradigm. The paradigm must be extended to maximize its flexibility, which allows adaptive query processing on heterogeneous resources. For that, we identify all scopes at which processing of subsets of the data is possible during the data flow and allow different choices of processing for each of these scopes.

Subsection 3.1 describes our new execution framework as an extension of the classical data flow paradigm. Subsection 3.2 describes a collection of techniques that realize some of the tradeoffs possible in the new framework. Section 4 develops the contents of this section to a formal framework.

3.1 New Execution Framework

Consider the data flow scheme shown in Figure 6: It shows all opportunities to execute algorithms on the data. We speak of the *execution scope* of an algorithm, consisting of the place and the timing of the execution, and the set of processed data. We use the partitions and data streams between sites as available data sets. Places are the sites of the system and possible timings are the stages of the pipeline, subdivided into five different phases that we now introduce into the data flow paradigm.

Say we have n sites, then for each stage of the pipeline and for each site the execution scopes are:

1) On the n fragments of the partition incoming on the data streams from the n sites. We call this the *incoming* phase.
2) During the merging of these fragments into one partition. We call this the *merging* phase.
3) On the whole partition on the site. We call this the *merged* phase.
4) During the splitting of the partition into the fragments outgoing during the following repartitioning. We call this the *splitting* phase.
5) On the n fragments of the partition outgoing on the data streams to the n sites. We call this the *outgoing* phase.

Figure 5 shows the five phases of each stage and the execution scopes within each phase. Each ellipses in the figure corresponds to an independent execution scope. Bold ellipses correspond to the execution scopes that are part of the original data flow paradigm. They form a subset of the scopes in the extended paradigm.

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

Our motivation was to migrate processing between sites to vary the usage of individual resources. The extended paradigm allows, roughly, the following options:

- Operations can be executed differently on the data streams between sites during repartitionings.
- Splitting partitions into data streams and merging them can happen differently on different sites.
- Operations can be executed differently on the partitions on different sites.

The first two options allow us to migrate parts of an operation's execution from site to site, while the last one allows a limited adaptation of the execution for a specific site.

The next subsection shows concretely how the flexibility of the extended paradigm can be used to leverage non-uniform resources.

3.2 Non-Uniform Execution Techniques

The problem that we are trying to resolve is that certain resources form the bottleneck of execution, while others are underutilized and partially idle. This problem is caused by the fact that the same operation has to be executed on sites with very different resource availability. Our proposed solutions fall into three different categories:

- Migration of processing: We migrate algorithms that use certain resources from sites that overutilize these resources to sites that underutilize them.
- Additional processing: We introduce additional processing, like compression, which trades off some resources against others.
- Alternative processing: We use alternative implementations of the same operations in different resource environments.

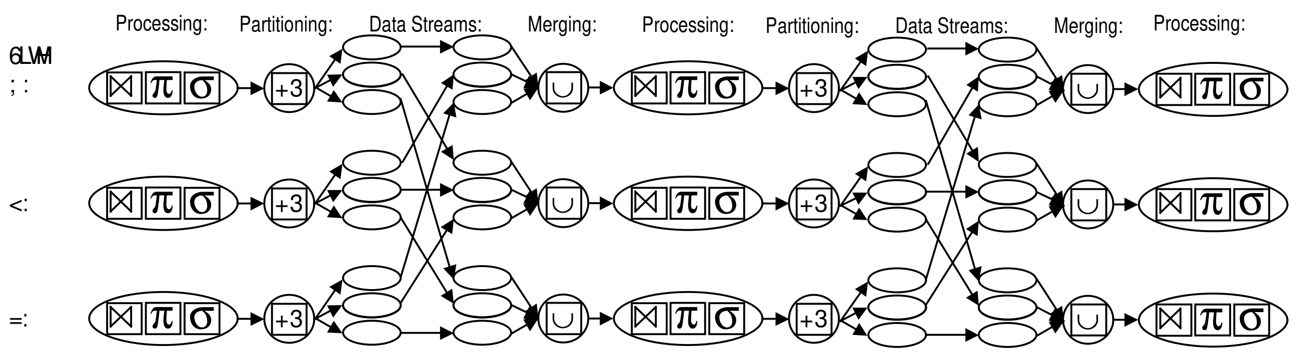


Figure 6: The Extended Dataflow Paradigm

We present techniques from all three areas, while our focus is on the first one, which allows the greatest improvements over the traditional approach[6].

The formal model presented in Section 4 will allow us to map out the complete execution space, showing all possible ways to apply given operations to data on a given architecture. The techniques presented in this section point out important parts of the execution space, but are by no means exhausting.

3.2.1 Migrating Operations

Considering the operations in Figure 6, we realize that only the joins have to be executed on each partition as a whole. Selections and projections can also be correctly executed on each of the fragments of the partitions that are sent out to other sites. 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.

We migrate operations along the data streams by applying them on the sending site for some streams and on the receiving site for others. Figure 7 illustrates this for a simple case, where selections and projections are migrated away from one of the sites. When the streams are merged on the receiver sites, the operations must have been applied to all of them.

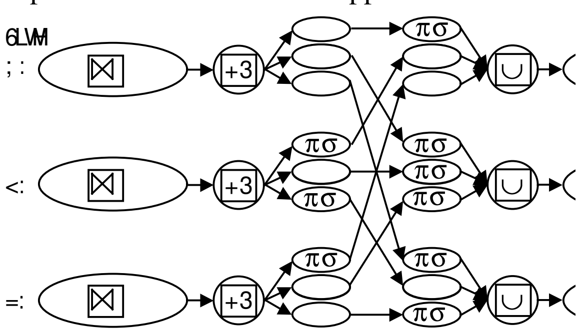


Figure 7: Migrating Operations

For each pair of sites, this technique gives us the decision if, on the data exchanged during the following repartitioning, the operation should be applied on the first or on the second site. This will be of benefit if the resources used by the operation are overutilized by exactly one of the two sites.

3.2.2 Migrating Join Preparation

Only selections and projections can freely be moved between the sites during repartitioning. Joins have to happen on the full partition between repartitionings. The reason is that joins have to be executed on each partition as a whole. Executed separately on fragments of the partition, not all possibly combinable tupels would be combined.

Nevertheless, the fragments on incoming data streams can be prepared on their source sites. 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 sites 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 site 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. Also, in many cases, the merging of incoming data streams has to be aware of the preprocessing. Streams that were not preprocessed on other sites, have to be preprocessed immediately before the merge.

3.2.3 Migrating Repartitioning Preparation

The last two subsections discussed how to migrate selections, projections, and parts of the join. The other major work done between repartitionings 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.

The splitting itself 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 sites.

3.2.4 Selective Compression

This technique trades off processing bandwidth on a pair of sites against the network bandwidth between the sites. The three techniques presented earlier migrated work that consumed resources local to the

[6] 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 be a tradeoff between the amount of underutilized resources used and the amount of utilized resources freed.

execution site. If they affected the network load at all, they increased it.

Since the resources are distributed non-uniformly, not all sites have the same processing bandwidth available for data compression. Compression and decompression can be applied on the partition fragments sent to other sites during repartitioning. Thus the decision about compression can be made on a site to site base, utilizing only the underutilized resources to relieve the network.

3.2.5 Alternative Algorithms

Our initial observation, that uniform processing over non-uniform resources leads to bottlenecks, can guide us to two complementary solutions:

- On different sites, do different parts of the query processing: Concentrate parts of the execution where the needed resources are available. This has been done in the first three subsections.
- On different sites, do the query processing in different ways: Pick an implementation of the required operation whose resource usage matches availability. This is the topic of this subsection.

There are usually many different implementations for a given operation that has to be processed in parallel on multiple sites. Implementations can be chosen for each site 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. This is why we expect the migration techniques, presented in Sections 3.2.1, 3.2.2, and 3.2.3, to be more powerful.

4 Formal Execution Model

This section formalizes the extension to the data flow paradigm, resulting in a definition of the new execution space and, based on it, a cost model.

The execution space is the set of all possible ways in which given relational operations can be processed by a given system. The execution space of our extended data flow paradigm will be a superset of that of the traditional one. Our claim is that for non-uniform architectures there are executions that are elements of the extended but not of the traditional space and that have better performance than any of the traditional executions. The reason for this is that they allow improved leverage of otherwise underutilized resources and thus reduced execution time.

Based on the execution space, we will model the cost of every execution in terms of overall execution time. This model allows us to compare different executions in terms of their expected performance. Also, such a cost model is the base for the design of optimization algorithms that search for optimal solutions within the execution space.

4.1 System Architecture

We want to model all features of the execution environment that we deem relevant for our execution space and cost model. The chosen abstraction should not hide any execution alternatives and it should reasonably reflect all execution constraints as costs. Accordingly, every involved component will be modeled as a full-fledged site allowing data processing in any form. Each site is modeled by individual bandwidths for a generic set of resources, which allows us to constrain data processing through the specific bandwidth settings of a site. The specific requirements of active environments and the corresponding contributions of our techniques are only reflected in models that have multiple resources with independent bandwidth[7] on each site.

To establish the components of an architecture, similar to the examples in Figure 2, we define a set of sites, of resources per site, and of shared resources.

Let each of the following be a given set of identifiers:

- *Sites* = *{x,y,z,…}* (Components of the architecture)
- *SiteResTypes* = *{ p, d, n,…}* (Resources present on each component, e.g., processor, disk, network access)
- *SharedResTypes* = *{ic, …}* (Resources shared among all components, e.g., the interconnect)

Sites is the set of all components or sites of the architecture. Each site has individual instances of the resource types in *SiteResTypes*. Additionally, all sites

[7] Independent bandwidth means that the proportion between the bandwidths on each site are not necessarily constant across all sites.

share a single instance of each resource type in *SharedResTypes*.

Based on these given sets we define the following naming conventions:

- $ResTypes = SiteResTypes \cup SharedResTypes$
- $SiteRes = \{r_x : r \in SiteResTypes, x \in Sites\}$
  (Set of resource instances present on the components)
- $SharedRes = SharedResTypes$
  (Set of shared resource instances, one per type)
- $Res = SiteRes \cup SharedRes$
  (Set of all resource instances)
- $ResType : Res \rightarrow ResTypes$
  For $r_x \in SiteRes : ResType(r_x) = r$
  For $r \in SharedRes : ResType(r) = r$
  (Type of a resource)
- $ResSite : SiteRes \rightarrow Sites$
  For $r_x \in SiteRes : ResSite(r_x) = x$
  (Site on which a resource is located)
- For $r \in SiteResTypes$: $R = \{r_x, r_y, r_z, \ldots\}$
  (Set of all instances of a site resource)
- $ResOfSite : Sites \rightarrow 2^{SiteRes}$
  For $x \in Sites$: $ResOfSite(x) = \{r_{x'} \in Res : x = x'\}$
  (Set of all resource instances on a site)

This gives us the set of resource instances as the shared resources together with the combinations of given sites with given site resource names. In Figure 2, the set of sites consists of the four clusters of columns on the right, while the columns in the clusters correspond to the site resources. The single column on the left is the only shared resource.

We will assign a bandwidth to every one of these resources, expressing the amount of data processed per time unit[8]. Let the following be a given mapping from resources to their bandwidths:

- $BW: Res \rightarrow [0;\infty[$

Bandwidth expresses the amount of data that can be processed during a given time period, relative to the processing algorithms resource usage. Usage will be defined in Section 4.3.

[8] The units in which data volumes and time are measured are unimportant for the development of the model. Only the ratios between the involved bandwidths are relevant to determine the relative performance of different processing strategies.

For example, $BW(p_x)=2*BW(p_y)$ implies that the same algorithm executed on the same amount of data would utilize the processor resource on site x twice as long as on site y. If the resource usage is $RU(a,p)$ (see Section 4.3), then the execution time would be $RU(a,p)/BW(p_x)$ on site x. The value of BW for a resource corresponds to the height of the corresponding column in resource graphs like Figure 2.

Resources are not exclusively used by algorithms. Shipping data between sites during repartitionings will utilize some of the resources. For this reason we identify local and shared resources that are utilized whenever data is sent or received by a site. While the shared resources are always used, the local resources are only used for communications of their specific site.

Let the following be given sets:

- $SharedComResTypes \subseteq SharedResTypes$
  (Shared resource types that incur cost for communication)
- $LocalComResTypes \subseteq LocalResTypes$
  (Local resource types that incur cost for communication)
- $ComRes = \{r \in Res:$
  $ResType(r) \in SharedComResTypes \vee$
  $ResType(r) \in LocalComResTypes\}$
  (Resource instances that incur cost for communication)

Section 4.6 will detail how communications and the execution of algorithms will affect the execution cost. As example, let d amount of data be sent by site x, with n and ic being a local and a shared resource. Then $d/BW(n_x)$ is incurred on resource n_x and $d/BW(ic)$ on resource ic.

Some caveats are in place, regarding the simplicity of the presented abstractions. Our model focuses completely on data throughput and does not reflect any latency. The solutions that we propose for the problems of traditional techniques are based on leverage of idle bandwidth. We simplified the presentation by focusing on this performance component.

It could be argued that our resource model is to simplistic in that a resource is either used only by one site or shared by all sites. More complex models could allow resources shared by a subset of the components, like a local interconnect. Again, simplicity of the presentation motivated our choice.

Algorithms are executed on a site at a specific time on a specific subset of the local data. The next section refines our model to express this scope of execution.

4.2 Execution Scopes

Figure 5 shows the possible scopes of execution for an algorithm on the defined architecture as part of a pipeline. Execution of algorithms is possible during the different phases of the pipeline on the different subsets available on a site. Each of the ellipses in Figure 5 forms a separate execution scope.

As explained in Section 3.1, each stage of the pipeline is subdivided into five independent phases, each of which forms execution scopes in combination with the available data sets in that phase. During the incoming and outgoing phases, on each site there is one dataset per incoming respectively outgoing data stream. That is, one set for each pair of sites. During the merging, the merged and the splitting phase, there is only one relevant data set per site, to which algorithms can be applied.

Let *nStages* be the number of stages in the pipeline. *Stages* has to be a finite set that is linearly ordered by '$\subseteq$'. We simply take natural numbers as names for stages[9]:

- $Stages = \{0,1,\ldots, nStages\}$
- For $x,y \in Stages$: $x \subseteq y \Leftrightarrow x \leq y$

We observed, that within each stage there are five possible execution phases. We need a naming convention for these phases. We call *phase types* the abstract phases that will happen in every stage, while a *phase* is a concrete instance within a specific stage.

- $PhaseTypes = \{Incoming, Merging, Merged, Splitting, Outgoing\}$
  (Identifiers for phase types, independent of stages)
- $Phases = \{ p_s : p \in PhaseTypes, s \in Stages \}$
  (Set of phase instances across all the stages)

The following are naming conventions for relevant subsets of *Phases* :

- For $s \in Stages : Phases_s = \{ p_{s'} \in Phases : s' = s \}$
  (Phases in the *n*th stage of the pipeline)
- $Incoming = \bigcup_{s \in Stages} \{Incoming_s\}$
  (Set of phase instances of a certain type across all stages)
- $Merging = \bigcup_{s \in Stages} \{Merging_s\}$
- $Merged = \bigcup_{s \in Stages} \{Merged_s\}$
- $Splitting = \bigcup_{s \in Stages} \{Splitting_s\}$
- $Outgoing = \bigcup_{s \in Stages} \{Outgoing_s\}$

Each phase has to be combined with a data set to form an execution scope. This happens for the merging, merged and splitting phases simply by picking the site of execution. For the incoming and outgoing phases, we also have to pick a subset on the chosen site, by picking the source or destination site of the in- or out-bound data stream. Thus, each execution scope is a combination of a phase with one respectively two sites:

Execution scopes for algorithms during the five phases:

- $WhileIncoming = Incoming \times Sites \times Sites$
  (Incoming streams on the first site, coming from the second site)
- $ForMerging = Merging \times Sites$
  (Merging of all streams on a specific site)
- $WhileMerged = Merged \times Sites$
  (Processing of the merged data on a site)
- $ForSplitting = Splitting \times Sites$
  (Splitting of the data into the data streams on a site)
- $WhileOutgoing = Outgoing \times Sites \times Sites$
  (Outgoing streams on a site, directed to the second site)

The sites in the incoming and outgoing tuples are not in the direction of the stream's flow. The first site is always the site on which the data is located, while the second site, if present, is the remote source or the target site of the data. The following are notational conventions related to the given definitions.

- $ExecScopes = WhileIncoming \cup ForMerging \cup WhileMerged \cup ForSplitting \cup WhileOutgoing$
  (Set of all execution scopes)
- $Site: ExecScopes \rightarrow Sites$
- Let $(p,s) \in ForMerging \cup WhileMerged \cup ForSplitting$: $Site(p,s) = s$
- Let $(p,s,s') \in WhileIncoming \cup WhileOutgoing$: $Site(p,s,s') = s$
  (Site of an execution scope)

[9] Our very generic definition would alternatively allow for sequences of stages, in which new stages could be inserted by the optimizer. In that case, natural numbers would be inadequate identifiers.

The following section shows how to populate execution scopes with algorithms.

4.3 Algorithms

The application of relational operations on a data set is modeled as the execution of algorithms at specific execution scopes within the pipeline. According to the different signatures of the execution scopes – merge of multiple streams, processing of a single stream, splitting into multiple streams – there are three different kinds of algorithms:

- *Merge:* An algorithm that processes multiple data sets as inputs and that produces a single result, for example, a simple union of the inputs.
- *Standard:* An algorithm that works on a single input data set, producing a single output. Examples are a sort, a projection, or a filter operation. Only standard algorithms can be executed in sequence.
- *Split:* An algorithm that works on a single input data set and that produces multiple result sets. An example is a hash partitioning of the data.

Algorithms are characterized through their resource usage and their effect on the data volume. The usage in combination with the available bandwidth and the processed data volume determines the execution time. Every algorithm has linear resource usage in terms of the shared and local resources. The usage is modeled as a number that, divided over the corresponding bandwidth, determines the execution time per data item.

The results of an algorithm's processing can have a different size than the inputs. In our model, the result size is linear in the size of the input. Associated with every algorithm is a *resizing factor* that reflects this linear relation between in- and output. For multiple in- or outputs, there is a separate resizing factor for each processed or produced data set.

We begin by defining the sets of available algorithms:

- *Let StdAlg, SplitAlg, and MergeAlg, be given sets of disjoint algorithms.*

Resource usage is defined for each algorithm with respect to every single resource type. Usage is defined for resource types and not for resources, because for multiple resource instances of the same type the resource usage should be the same. The cost of an algorithm on different sites only differs if the available bandwidth is different.

- $RU: (StdAlg \cup SplitAlg \cup MergeAlg) \times ResType \rightarrow [0;\infty[$
  (Resource usage of the algorithms)
- $RF: StdAlg \cup MergeAlgorithms \cup SplitAlgorithms \times Sites \rightarrow [0;\infty[$
  (Resizing factors of the algorithms)

For split algorithms, resizing happens with respect to each in- and output separately. For example, a split s sends $RF(s,x)$ of its input to site x, it produces $|Sites|$ separate outputs of the accumulated size $\sum_{x\in Sites} RF(s,x)$ times the input size. The size of a merge's output is $RF(m)$ times the sum of its inputs.

Since standard algorithms can be executed in sequence, the definitions of resource usage and of resizing are extended for sequences of standard algorithms. We write $[X]$ for the set of sequences over a given set X. For $sx \in [X]$, we write $Length(sx)$ for the length of sx, and sx_n for the nth element of sx ($1 \leq n \leq Length(sx)$). We also use set notation on sequences to mean the set of a sequence's elements, eg, $sx_i \in sx$.

- $RU: [StdAlg] \rightarrow [0;\infty[$
  For $seq \in [StdAlg]$, $rt \in ResType$:
  $RU(seq, r) = \sum_{1\leq i \leq Length(seq)} (\Pi_{1\leq j<i} RF(alg_j)) * RU(alg_i, rt)$
  (Resource usage for a sequence of algorithms)
- $RF: [StdAlg] \rightarrow [0;\infty[$
  For $seq \in [StdAlg]$:
  $RF(seq) = \Pi_{1\leq i\leq Length(seq)} RF(seq_i)$
  (Resizing for a sequence of algorithms)

With this we established sequences of algorithms as an extension of the set of algorithms. We can now identify *StdAlg* with the one-element sequences in *[StdAlg]* and use the latter whenever standard algorithms can be applied. The next section details how algorithms are applied in the execution scopes of the last section.

4.4 Execution Space

The proposed extended data flow paradigm consists of the combination of the execution scopes with the algorithms that are executed on them. Every such combination is a way to process the data on the given architecture. The traditional dataflow paradigm consists of a subset of the possible combinations.

This section defines the *extended execution space* consisting of all possible combinations.

An execution maps each execution scope onto the algorithms that are executed in that scope. We combine five mappings, one for each type of execution scope, to represent this. The mappings have different ranges, depending on the kind of algorithms that can be executed. Our execution space is the set of all combinations of such mappings.

- $ExecSpace =$
  $(WhileIncoming \rightarrow [StdAlg]) \times$
  $(ForMerging \rightarrow MergeAlgorithms) \times$
  $(WhileMerged \rightarrow [StdAlg]) \times$
  $(ForSplitting \rightarrow SplitAlgorithms) \times$
  $(WhileOutgoing \rightarrow [StdAlg])$

As an example, consider the execution shown in Figure 5. Each scope, shown as an ellipsis, is mapped onto the algorithms that are shown inside the ellipsis. As a convention, we will use the name of an execution as the symbol for each of its mappings. If the shown execution is called *ex*, we would write $ex(Incoming_1, s_1, s_2) = [sel_1, proj_1]$ and $ex(Merging_1, s_1) = stdMerge$.

The extended execution space, named *ExecSpace* above, is the space of all executions possible in our model. It represents the extended data-flow paradigm that this paper proposes. The size of this space is behemoth: Even if only one algorithm should be applied on the data streams of a single repartitioning, there are $2^{(n^2)}$ possible ways to combine early and late executions for n sites. Sophisticated optimization techniques will be needed to find close to optimal executions in such a space.

4.5 Data Distribution

This section formalizes an abstract concept of data distributed across the components of the system. The structure or semantics of the processed data is not necessary to demonstrate our techniques. A set of data that processed by an algorithm is simply represented as a specific amount of data. Consistent with bandwidth, usage and time, data amounts are measured by positive numbers without explicit units.

We start with the given initial distribution of data across the sites.

- *Let* $IDD: Sites \rightarrow [0;\infty[$ *be a given mapping from sites to their initial data volume.*
  (Initial Data Distribution)

Based on such a distribution and on a given execution, we can determine the data amounts for all execution scopes. This data distribution, expressing the amount of data that is processed as input in each scope, is represented by the following mapping.

- $DD: ExecScopes \rightarrow [0;\infty[$
  (Data Distribution)

The first pipeline stage will need too be defined different than later ones, because it reflects the initial data distribution instead of distributions of earlier stages.

Let $x,y \in Sites$*:*

- $DD(Incoming_0, x, y) = 0$
  (In the first stage, nothing is received)
- $DD(Merging_0, x) = 0$
  (Nothing is merged)
- $DD(Merged_0, x) = IDD(x)$
  (This reflects the initial data distribution)
- $DD(Splitting_0, x) = IDD(x) * RF(ex(Merging_0, x))$
  (The effect of the operation in *Merged* on the data)
- $DD(Outgoing_0, x, y) = IDD(x) *$
  $* RF(ex(Merging_0, x)) * RF(ex(Splitting_i, x), y)$
  (The combined effects from *Merged* and *Splitting*)

We compute the data volume that has to be processed at each execution scope. It depends on the initial data distribution and on the resizing that happens later. The data is resized by all algorithms that are executed on it. Splitting algorithms divide the data in independently resized fragments, while merging algorithms unite such fragments, resizing them uniformly. All phases are defined in terms of preceding phases.

Let $x,y \in Sites,\ s \in Stages,\ s \neq 0$*:*

- $DD(Incoming_s, x, y) =$
  $DD(Outgoing_{s-1}, y, x) * RF(ex(Outgoing_{s-1}, y,x))$
  (The data resulting at the other end of the data stream)
- $DD(Merging_s, x) =$
  $\Sigma_{y \in Sites}\ (DD(Incoming_s, x, y) *$
  $RF(ex(Incoming_s, x, y)))$
  (All data from incoming data streams)

- $DD(Merged_s, x) =$
  $DD(Merging_s, x) * RF(ex(Merging_s, x))$
  (All data after merging)
- $DD(Splitting_s, x) =$
  $DD(Merged_s, x) * RF(ex(Merged_s, x))$
  (All data on the site, after local processing)
- $DD(Outgoing_s, x, y) =$
  $DD(Splitting_s, x) * RF(ex(Splitting_s, x), y)$
  (The fraction that is sent to the specific target)

Thus the algorithms in every execution scope have to process the resized data processed in the last execution scope. In the case of a split, the resizing depends on the site of the follow-up scope. In the case of a merge, the data of multiple preceding scopes are relevant and are resized together.

This section determined the data amounts involved in a given execution. Based on this, the next section will determine its cost.

4.6 Execution Costs

Section 4.4 mapped out *ExecSpace*, the space of all possible executions in our new framework. This section will evaluate the alternative executions by estimating their costs in terms of overall execution time. As a result we can compare plans of our extended model with those of the traditional space (see Section 2.1).

The cost is constituted by the costs of each algorithm on each site's resources. It is influenced by the resource usage of the algorithm, by the resource availability on the execution site, and by the amount of data processed in the particular execution scope. Thus, we get utilization times for each algorithm and each resource. Multiple utilization of the same resource happens sequential and adds up, while the utilization of different resources happens in parallel and shows as the maximum utilization time of all resources. The resulting cost is a real number in $[0;\infty[$ without unit. Its unit is omitted, analogously to the omitted units of bandwidth (see Section 4.1) and data volume (see Section 4.5).

We will define the cost of an execution $ex \in ExecSpace$ in three steps: First, we define the cost per scope $es \in ExecScopes$ and per resource $r \in Res$, called $Cost(ex,es,r)$:

- *If* $r \notin Shared \vee r \in ResOfSite(Site(es))$:
  $Cost(ex, es, r) =$
  $DD(es) * RU(ex(es), ResType(r)) \;/\; BW(r)$
  else $Cost(ex, es, r) = 0$

Then, we define the cost per resource $r \in Res$ as the sum over all the scopes that affect that resource plus the incurred communication costs – $Cost(ex, r)$:

- *If* $ResType(r) \in SharedComResTypes$:
  $Cost(ex, r) =$
  $\sum_{es \in ExecScopes} Cost(ex,es,r) +$
  $\sum_{es \in WhileIncoming} DD(es)/BW(r)$
  If $ResType(r) \in LocalComResTypes$:
  $Cost(ex,r) =$
  $\sum_{es \in ExecScopes} Cost(ex,es,r) +$
  $\sum_{(Incoming_S,x,y) \in ExecScopes \wedge x=Site(r) \wedge x \neq y} DD(es) +$
  $\sum_{(Outgoing_S,x,y) \in ExecScopes \wedge x=Site(r) \wedge x \neq y} DD(es)$

Finally, we define the overall cost as the maximum of the costs on the resources – $Cost(ex)$:

- $Cost(ex) = MAX_{r \in Res}\, Cost(ex,r)$

We use one symbol, Cost, for the three cost functions with different domains. The cost of execution is the maximum of the times that the single resources need to finish. To finish, each resource has to sequentially serve in each execution scope on its site. An algorithm's cost is its resource usage divided over the resource bandwidth times the amount of data.

This cost model, complicated as it may seem, is the result of numerous simplifications. It does not reflect any concurrency overheads, latencies, sequential per-task overheads, or resource conflicts. These very real complications were left out to allow a focus on the data flow pipeline with its execution scopes.

5 Example: Migrating Workload along Data Streams

This section exemplifies the use of the formal model by analyzing the effects of one of the techniques that we propose. We will present a simple example that serves to demonstrate the features of the model and its role in analyzing new execution techniques. It is important to keep in mind that the techniques discussed in Section 3, among them our example, do not exhaust the possibilities that are presented as the execution space defined in Section 4.4.

For our example, we consider a join with a consecutive filter operation that is executed in parallel on the sites of a given system. Because the filter involves expensive computations, the combined operation is CPU bound on all the sites. Formally, $p \in SiteResTypes$ being the CPU, j and f being the algorithms executing the join and the filter :

$$p = Max_{rt \in SiteResTypes} (RU([j,f],rt) / BW(rt_x))$$

for all $x \in Sites$. The fraction that is maximized, resource usage over bandwidth, is the execution cost for the operation on a specific resource, relative to the processed amount of data.

When balancing the workload across the sites of the system, the optimizer can only attempt to balance the utilization times, minimizing the execution time of the whole system. Balancing can only be optimal for a single resource, as in our case the bottleneck resource p. The fraction of the overall data that should be processed on a site x is $BW(p_x) / \sum_{y \in Sites} BW(p_y)$.

The resulting workloads are imbalanced with respect to other resources that are distributed in different proportions across the sites. Consider sites that are active disks. The bandwidths of their processors will be much weaker in proportion to their other resources than that of server sites. Assume that the processor of an active disk xa is ten times slower than the processor on a server xs, while their disk I/O is similar, i.e., $BW(p_{xa})=0.1*BW(p_{xs})$ and $BW(d_{xa})=BW(d_{xs})$. This implies that the utilization of the active disk is at most a tenth of that of the server's disk:

$$BW(p_{xa})/\sum_{y\in Sites} BW(p_y) * RU([j,f],d)/BW(d_{xa}) =$$
$$0.1*BW(p_{xs})/\sum_{y\in Sites} BW(p_y)* RU([j,f],d)/BW(d_{xs})$$

Consequently, the active disks main resource, d_{xs}, is utilized for less than 10% of the execution, because workload balancing can only account for a single 'weakest' resource, p_{xs} in our case.

Clearly, other, more adaptive techniques are needed. We would like to move processor intensive tasks away from the active disks, relieving their CPU bottleneck. As a result, the amount of data processed on the disk could be increased, reducing overall execution time. We can achieve this goal using the task migration technique. Consider the traditional execution $ex \in Exec$ of the query, $i \leq Stages$ being the pipeline stage and $x,y \in Sites$ arbitrary:

$$ex(Incoming_i,x, y) = ex(Outgoing_i,x, y) = [\,]$$
$$ex(Merged_i, x) = [\, j, f\,]$$
$$ex(Merging_i, x) = union \quad ex(Splitting_i, x) = partition$$

(*union* forms the union of its inputs; *partition*[10] splits its input in preparation for the join in the next stage)

As a first step, we realize that the filter does not need to be executed on the partition as a whole. It can also be applied on its fragments, before sending them to other sites[11]. This movement from the $Merged_i$ phase to the $Outgoing_i$ phase does not change the overall costs, as long as the sum of the resizing factors of *partition* is 1:

$\sum_{y\in Sites} RF(partition,y)=1.0$. This reflects the fact that the overall amount of data is the same before and after the partitioning.

As a second step, we realize that the data processed in $(Outgoing_i, x, y)$ are the same as in $(Incoming_{i+1}, y, x)$ because these phases are the two ends of the same data stream. This allows us to delay the application of f to the data of the stream until after the shipping of the data:

$ex(Outgoing_i,x, y) = [\,]$ and
$ex(Incoming_{i+1},y, x) = [f]$

This affects the resource usage on x, y, and on the communication resources. The latter are affected because the selectivity of the filter is lost on the shipped data: $DD(Incoming_i, y, x) = DD(Outgoing_i,x, y)$, instead of $DD(Incoming_i, y, x) = RF(f) * DD(Outgoing_i,x, y)$. The table in Figure 8 presents the change in costs per resource as a consequence of delaying f between x and y (ex' is the modified execution). The effect on communication resources is additional to the other effects.

	$Cost(ex', r) - Cost(ex, r)$
$r \in ResOfSite(x)$	$- RU(f, ResType(r)) * DD(Outgoing_i, x, y) / BW(r)$
$r \in ResOfSite(y)$	$+ RU(f, ResType(r)) * DD(Outgoing_i, x, y) / BW(r)$
$r \in SharedRes$	$+ 0$
additionally, if $r \in ComRes$	$+ DD(Outgoing_i, x, y) * (1-RF(a))$

Figure 8: Effects of Migrating the Operation

[10] We simplify by not costing these operations.

[11] Certainly this will involve overheads in terms of the separate, additional processing phase. For now, we ignore these overheads and leave their exploration to future work with actual prototype systems.

Site x is relieved of exactly the specific resource usage of the filter algorithm, which is instead added to site y. The effect is different, though, since the bandwidths of the resources on both sites are different. The costs are in inverse proportion to the resources' bandwidths. Moving CPU load from a site with slow CPU to a site with strong CPU will add less cost to the latter than it removed from the former. The effect on shipping the data corresponds to the amount by which the data would have been reduced.

If we delay processing on all data streams, the filter is simply applied immediately before the next join. This, as delaying it on none of the streams, corresponds to a traditional execution. Migrating the filter task allows us an individual choice for each data stream between the source and the target site. For $n = |Sites|$, there are n^2 independent choices and $2^{(n^2)}$ combinations of such. Searching for (near-)optimal executions among these possibilities is future work.

Returning to the example, the techniques can be used to relieve the active disks of the CPU workload that comes with the filter operation. On any data stream connecting an active disk and a server site, the filter will be delayed to the server process. This reduces the usage on the disks' bottleneck resource relative to its other resources. As a consequence, more data can be processed on the site within the same amount of time. The additional workload can be taken from the servers which received additional CPU workload. The benefit of this corresponds to the ratio of disk versus server CPU bandwidth. Combining the effects from Figure 8 with the assumption that the disk's CPU bandwidth is a tenth of the server's, we get:

$$RU(f, r) * DD(Outgoing_i , x, y) / BW(r_d) = RU(f, r) * DD(Outgoing_i , x, y) / (0.1*BW(r_s)) = 10 * RU(f, r) * DD(Outgoing_i , x, y) / BW(r_s)$$

This means that moving the tasks to the server only adds a tenth of the utilization time to the server compared to what was gained on the disks. The migrating of tasks is complemented by a rebalancing of workload in the reverse direction. The migration adds utilization time to one resource while removing it from another in a favorable proportion. Workloads have to be rebalanced to take this into account.

This concludes our example.

6 Related Work

Traditional approaches to query processing in parallel shared-nothing database systems assume a more or less uniform architectural model [DG90, DG92, C+88, D+90, GD93]. Accordingly, they do not explicitly model non-uniform resources, as we do. The same resources are available on each component of the system[12]. We described, the underlying approach – the classical data-flow paradigm – in Section 2. In the following, we survey existing systems in their relation to our approach. In a later subsection we discuss related work that focuses on specific aspects of query processing.

6.1 Existing Parallel Systems

Heterogeneous resource environments were not a focus in either of the discussed database systems. We will thus simply try to outline the specific techniques that each system contributed to what we termed the traditional approach. River, the last system in this section is a generic parallel processing environment, not specialized for relational query processing. We include it because it discusses dynamic fluctuations of resource availability, a topic closely related to the static heterogeneities that are central to this paper.

6.1.1 Gamma

Gamma was built between 1984 and 1989 at the University of Wisconsin, Madison, as a highly parallel database prototype [D+90]. Architecturally, Gamma is based on a shared-nothing architecture [S86]. It followed the much earlier DIRECT project [D79], which used shared memory and centralized control and thus had very limited scalability [D+90].

Gamma's key concepts are horizontally partitioned relations, hash-based parallel algorithms and dataflow scheduling. Horizontal partitioning, also known as declustering, targets the leverage of the accumulated I/O bandwidth. Gamma allows round robin, hashed and range partitioning. Round robin[13] across all nodes is the standard for query results that are relations[14].

[12] The join sites of the simple hash join [SD89] do not need to have disks. An early version of Gamma [D+90] integrates disk-free sites as a special case.

[13] Round Robin was characterized as a strategy that minimizes locality and such skew, as compared to value based partitioning schemes [C+88].

[14] Dewitt et al. saw this as a major design flaw in retrospect. See Bubba's heat of a relation as a better alternative [C+88].

Clustered and non-clustered indexes are allowed orthogonal to the employed partitioning scheme.

The query scheduler uses the partitioning information in the query plan to distribute operators on a subset of the sites, for example based on the intersection of a predicate and the partition ranges. The generation and execution of plans follows traditional relational techniques [SA80,A+76]. Left-deep trees with pipelining of not more than two joins are used.

On the relevant subset of sites, operators are executed locally on the data received from other sites. Their output is partitioned through different types of split tables [D+86] that relate the tuples to their outgoing streams.

A centralized scheduler that coordinates the execution of a query initiates processes for each operator on each site through local dispatchers. Build inputs to a join are scheduled concurrently with the join build phase, but complete before the probe inputs are initiated to run concurrently with the join probe phase. Consuming operations later in the pipeline are always initiated before earlier, producing operations. Scans and selects are operations without input streams while store operations have no output streams.

Gamma allows simple scans and selects, both executed at the relevant subset of sites where the relation is initially located. Predicates are executed as compiled native code.

Equijoins are by default executed as hybrid hash joins [SD89], which involves two split tables: The partitioning split table separates the joined relations into logical buckets that each fit into the aggregate memory of the components. The joining split table is used to separate the tuples of each bucket into the partitions that will be joined on the components.

Aggregate functions are computed in two phases: Each component computes local, partial results. Then the tuples are repartitioned on the 'group by' column. The results for each group can then be computed locally on its site.

Gamma uses chained declustering [HD90] as a replication scheme to cope with site failures. See [B81,CK89] for alternatives and improvements to chained declustering.

[D+92] treats the problem of workload skew with Gamma as a test bed. Hash-based partitioning leads to load imbalances during further processing (for the effects on Gamma's join algorithms, see [SD89]). Weighted range partitioning with replication of subsets of repeated values is proposed. Adequate ranges are determined by sampling the involved data. Virtual processor scheduling (similar to the 'data cells' of [HL90]) produces many small partitions instead of a single large one per processor. These partitions can be migrated between components to mitigate join product skew.

6.1.2 Bubba

[C+88] sets out to find some compromise between minimizing the amount of total work and optimizing the load balance across the sites. Data partitioning and parallel execution increase the total work by introducing overheads. But avoiding these overheads leads to underutilization of sites due to imbalanced execution on one or a few sites. Analogously, our approach tries to increase the balancing of processing across the individual resources and eventually a compromise between the introduced overheads and the gained balance has to be found. For Bubba, the benefit of minimizing overall work is the availability of processing capabilities for other queries, independent, or dependent parallelism[15]. In contrast to Bubba's limited declustering, Gamma and Teradata used full declustering. This was motivated by their focus on single transaction performance, which disregarded multi-query parallelism. Earlier work [LKB87] that did consider multi transaction workloads recommended full declustering for all but very high numbers of parallel transactions. [C+88] finds that less than full declustering outperforms full and no declustering.

Bubba's shared nothing architecture is quite similar to that of GAMMA [B+90, D+90]. The main difference is Bubba's focus on optimal data placement while Gamma simply relies on full declustering. [C+88] suggest, but does not employ, a composite workload that consists of weighted workloads for the different resources, like CPU and disk. This already recognizes the problem that we are treating in the more critical context of non-uniform resources. Partitioning the workload according to the locality of usage of a specific resource could be seen

[15] Our approach assumes, for the time being, that other forms of parallelism cannot make good use of the isolated underutilized resources that our techniques are designed to consume.

as a limited alternative to our approach: Data which is accessed by transactions of a specific resource usage is placed on sites with availability of the corresponding resources.

6.1.3 Paradise

Paradise was started in 1993 to combine object-oriented techniques from the EXODUS project [C+86] and parallelization techniques from the GAMMA project [D+90]. The application was the emerging area of Geographic Information Systems (GIS) with their large data volumes and complex data types. We focus here on the parallel aspects, described in [P+97].

Paradise focuses on new parallelization techniques especially for geo-spatial workloads, like spatial partitioning, parallelism for individual objects, and complex aggregates. Underlying are the parallel techniques of GAMMA.

Operators communicate via streams, following the push model from the leaves of a query plan up to the root. Streams allow flow-control to regulate the processing speed of different operators. Split streams are used to partition data sets for parallel processing. The different stream types are transparent to the operators.

Large objects are accessed following the pull model: A separate operator on the source node is started which serves selective pull requests from the consumer node. This avoids the shipment of unnecessary data, but it introduces overheads for the separate operator, and it generates random disk seeks.

Another project involving parallel geo-spatial data processing was MONET [BQ96].

6.1.4 Volcano

Volcano [G90,GD93,G94] integrates the parallelism into extensible query processing systems. Because new data types, functionality, and relational operators should be added in a simple manner, parallelism has to be transparent to these extensions. Another goal of Volcano is architectural independence, which also prohibits parallelism to be pervasive in the design of the system. Volcano's answer is to focus all mechanisms that are necessary to introduce different forms of parallelism into one relational operator, called the 'exchange' operator.

Earlier systems, like Gamma and Bubba, failed to completely separate parallelism issues from the implementation of the parallelized operators [G90]. Volcano proposes an operator model that introduces parallelism into query plans in the form of the 'exchange' operator. This operator separates the flow of control in a pipeline by introducing two processes instead of one. This allows concurrency between the two parts of the pipeline, before and after the exchange operator. The exchange operator can also be used to partition its input data set and run independent versions of another operator on each of the fragments, introducing intra-operator parallelism. In a third variation, the exchange operator is used to allow independent (bushy) parallelism: Each of the independently executed subplans is extended by an exchange operator that runs it in a separate process.

The underlying architectures of the Volcano system are shared memory and shared disk architectures, as well as hybrids. In contrast to Gamma and Bubba, shared nothing architectures are not employed. Nevertheless, the ideas embodied by Volcano – separation of parallelism and functionality, uniformity of operator interfaces and extensible optimizer design – seem to apply as well to shared nothing systems.

6.1.5 River

River [A+99,A99] introduces techniques that deal with performance skew – dynamic fluctuations in the availability of resources. Due to various reasons, components in a parallel system develop performance failures, which can reduce the available bandwidth of some of their resources dramatically. River introduces two techniques that make systems robust against these failures: Graduated declustering and distributed queues.

Chained declustering [HD90,B81,CK89] is a replication scheme that ensures functionality in the case of component failures. Graduated declustering adapts this technique to deal gracefully with performance failures. While this alleviates performance skew on the producer site, distributed queues adapt the data flow for skew on the consumer site. Both techniques are based on adapting the flow between the different components of the system, depending on their actual processing rates.

Flow control does not easily apply to parallel join processing because data is partitioned semantically. Depending on the value of the joined attribute, data is placed on a specific site. Adapting this partitioning dynamically was explored in the context of skew handling (see Section **Error! Reference source not found.**). River was used to implement query processing by using its techniques for non-join operations, like scans and writes [A99].

River's flow control dynamically changes the workload balancing between different components. The techniques that we propose are based on static information and actually change the resource usage, not only the amount of processed data per site.

6.2 Other Related Work

This section discusses contributions to specific problems in the area of parallel processing in their relation to our work.

6.2.1 Algorithms

Alternative algorithms implementing common relational operations have been explored in [SD89]. Performance is examined under certain resource constraints, like insufficient memory, and robustness with respect to performance skew.

[SN95] proposes parallel aggregation algorithms where aggregation and repartitioning are intermixed. The *repartitioning* algorithm repartitions the raw data and computes the aggregates at the target nodes. The *two-phase* algorithm first computes a local aggregate at the source nodes, then repartitions the locally aggregated data and finally merges local aggregates at the target nodes. The two-phase algorithm trades increased processing on the source node for reduced network traffic. Our approach would suggest to precompute aggregates only on sites with available resources, analogously to join preparation (see Section 3.2.2).

6.2.2 Workload Balancing

[RM95] examines how workloads should be balanced dynamically in a multi-query environment. The degree of parallelism – the number of nodes – and the placement of the computation – the choice of nodes – both depend on the existing workload of already running queries. Different resources, CPU, disk or memory, suggest different tradeoffs. Differently than this paper we focused on the more fundamental problem of balancing the execution of a single query in a setting with heterogeneous resources.

Very influential work on data placement based on the 'heat' of the data – its access frequency – was presented as part of the Bubba system [C+88] (see Section 6.1.2). The results suggest that relations should be spread across part of the available sites, with the degree of declustering depending on their heat. Other systems [T87,T88,D+90] find near-linear scaleup for declustering relations across as many sites as possible. This seeming discrepancy of results, between partial and full declustering is based on different workloads. While Bubba examined a workload consisting of many different transactions, the other studies focused on the idealized situation of processing a single query. As explained in Section 6.1.2, the benefits of partial declustering are only realized through pipeline, independent, or multi-query parallelism.

Most systems use replication in one or another form. Gamma uses chained declustering [D+90,HD90], Tandem mirrored disks [B81], and Teradata interleaved declustering [CK89]. River [A+99] introduces graduated declustering as a performance robust improvement of chained declustering. River proposes distributed queues as a flow control that allows the dynamic placement of data according to the availability of the data consumers. Unfortunately, this does not apply to imbalances during value-based partitioning, a problem that is called redistribution skew [WDJ91].

In an ideal uniform system, optimal performance is achieved with a perfectly balanced load (i.e., identical amount of data on each processing node) [HL90]. In a slightly different context, [BVW96] shows that, in an identical architecture, minimal response time is obtained when the loads of all servers are equal.

Among our assumptions is the uniform distribution of data with respect to the values used in hash partitioning. Without this assumption, data skew poses a major problem for workload balancing. Hash functions with low skew are discussed in [CW79]. [WDJ91] describes and distinguishes redistribution skew from join product skew. Improved hash functions only improve the former, and they cannot deal with skew due to duplicate values [D+92]. [HL90] proposes partition tuning by reassigning data cells from overflow to underflow partitions dynamically. [HL91] discusses specializations of join algorithms based on partition tuning. [D+92] proposes different algorithms for different degrees of skew, measured on a small sample of the data. [MD97] simulates different strategies and shows how changing technology trends change the involved performance tradeoffs.

Dynamic scheduling and load balancing techniques have been developed to face the problems introduced by skewed data distributions, or by the concurrent execution of multiple queries [HL91,MD93, RM95]. These techniques either propose new join algorithms (repartitioning data to balance the load) or adjust the number of processing nodes and select the actual processing nodes based on CPU and memory utilization. The techniques we propose for trading bandwidth utilization across the various components of a system can be seen as a complement to these load-balancing techniques.

6.2.3 Active Storage

Existing work on active storage addresses general architectural issues [G+98,UAS98,KPH98,HM98], studies programming models [AUS98], and evaluates the benefits for specific applications, like data mining [RGF98]. So far, relational query processing has not been a focus in this new environment.

Work on storage systems [G+99,LT96] and on file systems [G+97,TML97] that integrate active storage, suggests that leveraging processing capabilities close to the data allows large performance benefits. Our expectation is, that leveraging these capabilities for higher-lever applications like relational query processing will have even higher benefits.

7 Conclusion

We identified the problem that heterogeneous resources pose for classical parallel query processing techniques. Heterogeneous resources, as present on active storage and active networks, extend the capabilities of a system in a non-uniform manner. Using traditional intra-operator parallelism to distribute operations uniformly across the available components will lead to severe underutilization of the resources of the new components.

As an alternative we propose to extend the classical data-flow paradigm by recognizing, splits, merges, incoming and outgoing data streams as available execution scopes for data processing. This allows us to make independent choices for each data stream between a pair of sites. The execution of specific algorithm can be migrated towards sites which have the required resources available.

We formalized the proposed extension to the classical paradigm by defining the space of possible executions of given algorithms on a given architecture. Our cost model allows us to estimate the performance gains of the extended space over the subsumed classical execution space.

Our future work will examine the actual underutilization in a non-uniform prototype system, the performance gains possible through our techniques, and the introduced overheads. Optimization of declarative queries for parallel execution in this extended paradigm will be an interesting challenge. Finally, the automatic administration of systems consisting of heterogeneous hardware platforms, the declustering of relations and the choice of a replication scheme will be a very relevant contribution.

Bibliography

[A+76] M.Astrahan, et al.: System R: A Relational Approach to Database Management. ACM Transactions on Database Systems, Vol.1, No. 2, June 1976, pp.97-137.

[A+99] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David A. Patterson, Katherine A. Yelick: Cluster I/O with River: Making the Fast Case Common. IOPADS 1999: 10-22

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

[AUS98] Anurag Acharya, Mustafa Uysal, Joel H. Saltz: Active Disks: Programming Model, Algorithms and Evaluation. ASPLOS 1998: 81-91

[B+90] Haran Boral, William Alexander, Larry Clay, George P. Copeland, Scott Danforth, Michael J. Franklin, Brian E. Hart, Marc Smith, Patrick Valduriez: Prototyping Bubba, A Highly Parallel Database System. TKDE 2(1): 4-24. 1990.

[B81] Andrea J. Borr: Transaction Monitoring in ENCOMPASS: Reliable Distributed Transaction Processing. VLDB 1981: 155-165

[BQ96] Peter A. Boncz, Wilko Quak, Martin L. Kersten: Monet And Its Geographic Extensions: A

Novel Approach to High Performance GIS Processing. EDBT 1996: 147-166

[BVW96] Yuri Breitbart, Radek Vingralek, Gerhard Weikum: Load Control in Scalable Distributed File Structures. Distributed and Parallel Databases 4(4): 319-354 (1996)

[C+86] Michael J. Carey, David J. DeWitt, Daniel Frank, Goetz Graefe, M. Muralikrishna, Joel E. Richardson, Eugene J. Shekita: The Architecture of the EXODUS Extensible DBMS. OODBS 1986: 52-65

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

[CK89] George P. Copeland, Tom Keller: A Comparison Of High-Availability Media Recovery Techniques. SIGMOD Conference 1989: 98-109

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

[D79] David J. DeWitt: Query Execution in DIRECT. SIGMOD Conference 1979: 13-22

[D+86] David J. DeWitt, Robert H. Gerber, Goetz Graefe, Michael L. Heytens, Krishna B. Kumar, M. Muralikrishna: GAMMA - A High Performance Dataflow Database Machine. VLDB 1986: 228-237

[D+90] David J. DeWitt, Shahram Ghandeharizadeh, Donovan A. Schneider, Allan Bricker, Hui-I Hsiao, Rick Rasmussen: 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

[DG90] David J. DeWitt, Jim Gray: Parallel Database Systems: The Future of Database Processing or a Passing Fad? SIGMOD Record 19(4): 104-112 (1990)

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

[G90] Goetz Graefe: Encapsulation of Parallelism in the Volcano Query Processing System. SIGMOD Conference 1990: 102-111

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

[G+97] Garth A. Gibson, David Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, Jim Zelenka: File Server Scaling with Network-Attached Secure Disks. SIGMETRICS 1997: 272-284

[G+98] Garth A. Gibson, David Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, Jim Zelenka: A Cost-Effective, High-Bandwidth Storage Architecture. ASPLOS 1998: 92-103

[G+99] NASD Scalable Storage Systems. USENIX99, Extreme Linux Workshop, Monterey, CA, June 1999.

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

[HM98] Mark Heinrich and Rajit Manohar. Active Fabric: An Architecture for Programmable, Scalable I/O Subsystems. Cornell Computer Systems Lab Technical Report CSL-TR-1998-990, October 1998

[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

[KPH98] Kimberly Keeton, David A. Patterson, Joseph M. Hellerstein: A Case for Intelligent Disks (IDISKs). SIGMOD Record 27(3): 42-52 (1998)

[LKB87] Miron Livny, Setrag Khoshafian, Haran Boral: Multi-Disk Management Algorithms. SIGMETRICS 1987: 69-77

[LT96] Edward K. Lee, Chandramohan A. Thekkath: Petal: Distributed Virtual Disks. ASPLOS 1996: 84-92.

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

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

[P+97] Jignesh M. Patel, Jie-Bing Yu, Navin Kabra, Kristin Tufte, Biswadeep Nag, Josef Burger, Nancy E. Hall, Karthikeyan Ramasamy, Roger Lueder, Curt Ellman, Jim Kupsch, Shelly Guo, David J. DeWitt, Jeffrey F. Naughton: Building a Scaleable Geo-Spatial DBMS: Technology, Implementation, and Evaluation. SIGMOD Conference 1997: 336-347

[RGF98] Erik Riedel, Garth A. Gibson, Christos Faloutsos: Active Storage for Large-Scale Data Mining and Multimedia. VLDB 1998

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

[S86] Michael Stonebraker: The Case for Shared Nothing. Database Engineering Bulletin 9(1): 4-9, 1986.

[SA80] Patricia G. Selinger, Michel E. Adiba: Access Path Selection in Distributed Database Management System. ACM SIGMOD 1979, p.23-34, Boston, MA, USA, June 1979.

[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

[SN95] Ambuj Shatdal, Jeffrey F. Naughton: Adaptive Parallel Aggregation Algorithms. SIGMOD Conference 1995: 104-114

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

[T88] The Tandem Performance Group: A Benchmark of NonStop SQL on the Debit Credit Transaction (Invited Paper). SIGMOD Conference 1988: 337-341.

[TML97] Chandramohan A. Thekkath, Timothy Mann, Edward K. Lee: Frangipani: A Scalable Distributed File System. SOSP 1997: 224-237

[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