Parallel Querying with Non-Dedicated Computers

Vijayshankar Raman Wei Han Inderpal Narang
IBM Almaden Research Center
650 Harry Road, San Jose CA 95120
{ravijay,whan,narang}@us.ibm.com

Abstract

We present DITN, a new method of parallel querying based on dynamic outsourcing of join processing tasks to *non-dedicated*, *heterogeneous* computers. In DITN, partitioning is not the means of parallelism. Data layout decisions are taken outside the scope of the DBMS, and handled within the storage software; query processors see a "Data In The Network" image. This allows *gradual scaleout* as the workload grows, by using non-dedicated computers.

A typical operator in a parallel query plan is Exchange [7]. We argue that Exchange is unsuitable for non-dedicated machines because it poorly addresses node heterogeneity, and is vulnerable to failures or load spikes during query execution. DITN uses an alternate *intra-fragment* parallelism where each node executes an independent select-project-join-aggregate-group by block, with no tuple exchange between nodes. This method cleanly handles heterogeneous nodes, and well adapts during execution to node failures or load spikes.

Initial experiments suggest that DITN performs competitively with a traditional configuration of dedicated machines and well-partitioned data for up to 10 processors at least. At the same time, DITN gives significant flexibility in terms of gradual scaleout and handling of heterogeneity, load bursts, and failures.

1 Introduction

Parallel query processing [4] has evolved from being a research idea (*e.g.,* Gamma, XPRS [5, 8]) to being a standard feature provided by most DBMS vendors (*e.g.,* Tandem, Teradata, Oracle, Informix XPS, and DB2 [20, 22, 14, 2]). The parallelism in these systems is highly scalable, with vendors reporting good speedup with even 1000s of parallel nodes [4].

Traditionally, parallel query systems have been classified as shared nothing (SN), shared memory (SMP), and shared disk (SD) [19]. But a common characteristic of all these three types is that they rely on *dedicated* processors, pre-configured and pre-assigned for the parallel query task.

For SMP systems (and SMP nodes in SD/SN systems), this coupling is done in hardware. In SD systems, the compute nodes are often connected by specialized interconnects to a shared storage. SN systems are more loosely coupled due to a cluster-architecture. But the compute nodes still need to be dedicated for the parallel query task because the data is pre-partitioned across the compute nodes (typically by hash or range of a join column value). In general, both SN and SD systems rely on partitioning as the primary means of parallelizing expensive query operations such as hash joins.

In contrast to these dedicated styles of parallel querying, the modern trend towards *grid computing* emphasizes *non-dedicated* computers. The advantage is that multiple applications can share machines, and so the enterprise need not over-provision for the peak load of any single application. For example, desktop workstations are used primarily during office hours.

The DITN project at IBM Almaden is exploring parallel query processing over such non-dedicated computers. There are many challenges:

- We cannot rely on pre-partitioned data if we want to exploit transiently available nodes.
- Non-dedicated nodes can have volatile loads and may stall, so dynamic load-balancing is important.
- Grid nodes are often highly heterogeneous in CPU power. For example, a personal workstation and a server may be available simultaneously.
- Failures/Addition: Nodes can be added to or unplugged from grid at any time. *E.g.,* a workstation may leave the grid as soon as the user logs in.

While meeting these challenges, DITN's goal is *not to beat the performance of a dedicated parallel configuration with well-partitioned data.* Instead DITN is designed to perform competitively, but at a much lower cost, because of gradual scaleout and by exploiting non-dedicated compute nodes. In addition, DITN is

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

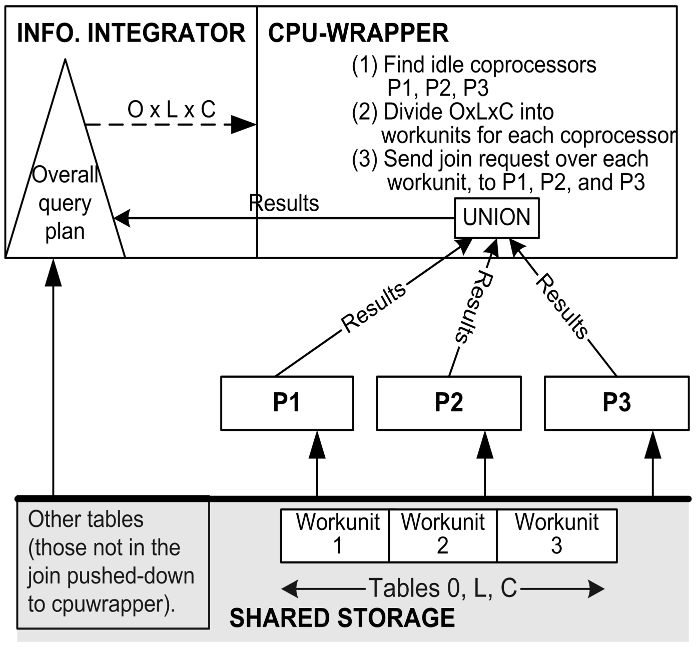

INFO. INTEGRATOR
CPU-WRAPPER
(1) Find idle coprocessors P1, P2, P3
(2) Divide OxLxC into workunits for each coprocessor
(3) Send join request over each workunit, to P1, P2, and P3
O x L x C
Overall query plan
Results
UNION
Results
Results
Results
P1
P2
P3
Other tables (those not in the join pushed-down to cpuwrapper).
Workunit 1
Workunit 2
Workunit 3
Tables 0, L, C
SHARED STORAGE

Figure 1: Architecture of DITN

designed to dynamically adapt to heterogeneity, load bursts and flaky node availabilities, and perform better than traditional DBMSs in such circumstances.

1.1 DITN Architecture

DITN is built as a prototype on top of IBM's Websphere Information Integrator (I.I.).

Figure 1 shows the architecture of DITN. Its main virtualization component is a *cpuwrapper* module for I.I. I.I. uses wrappers (as in, *e.g.,* [9, 16]) to obtain a relational access interface to heterogeneous data sources. During its dynamic-programming optimization phase, I.I. repeatedly tries to "push down" various plan fragments to the wrappers, and chooses the extent of the push down so as to minimize the overall plan cost. The cpuwrapper is a special kind of wrapper that wraps not data sources, but compute nodes. We call these compute nodes *co-processors.*

I.I views the cpuwrapper as a wrapper over a single data source, "GRID". I.I tries to push down various plan fragments to the cpuwrapper, but the only ones that the cpuwrapper accepts are select-project-join fragments, with aggregation and group by (SPJAG). Other, more complex fragments are returned back to I.I. as not pushdownable to the "GRID" source, and performed at the I.I. node itself. *E.g.,* in Figure 1, the *OLC* join alone is pushed down to the cpuwrapper; the rest of the query plan is done by I.I.

When I.I. pushes down SPJAGs to the cpuwrapper, the cpuwrapper executes them by *outsourcing* the work to the co-processors (P1, P2, P3 in the example). The essence of this outsourcing is to (a) identify idle co-processors using a grid load monitor, (b) logically split the input tables into work-units, (c) rewrite the overall SPJAG as a union of SPJAGs over the work-units, and (d) execute these SPJAGs in parallel on the co-processors. In step (d), the cpuwrapper detects and handles dynamic load-bursts and failures, by reassigning delayed work-units to alternative co-processors.

Outline of the Paper: The rest of the paper elaborates on this process, by describing three key aspects of DITN. First, DITN relies on a shared storage architecture to communicate with the co-processors. This is described in Section 2. Then, Section 3 describes how DITN breaks up the overall SPJAG into work-units. Section 4 describes the runtime mechanics of DITN — how the cpuwrapper orchestrates the parallel execution of work-units in a load-resilient fashion, and how the co-processors perform their work-units. We present an experimental evaluation of this technique in Section 5. We discuss related work in Section 6 and conclude with pointers to future work in Section 7.

2 DITN Storage Architecture

In traditional parallel DBMSs, tables are physically partitioned (or replicated, if the table is very small) across the cluster nodes. In order to exploit non-dedicated compute nodes, DITN does away with this idea of pre-partitioning data. Instead, the cpuwrapper treats the co-processors as CPU-only, with *Data* residing (logically) *In The Network (DITN)*, on a shared storage system. In our implementation, this is a storage-area network, virtualized by IBM's TotalStorage SAN file system (SANFS).

Briefly, the SAN consists of a cluster of disks connected by a high-speed network. The SANFS stripes files across these disks to provide a single file system image, with the combined scan bandwidth and random I/O throughput (due to multiple disk heads) of all these disks.

DITN stores each table directly as a single file on the SANFS. The work-unit allocated to each co-processor is to do a join of a contiguous segment from each file. The co-processors access these input files directly from the SANFS, without communication to the cpuwrapper (we assume an uncommitted read semantics is okay for the queries).

Traditionally, the DBMS relies on partitioning to get parallelism in I/O, and to reduce network traffic. Due to striping, DITN gets I/O parallelism directly from the file system. As regards network traffic, our experimental analysis suggests that in modern networks, the network bandwidth is much larger than the bandwidth of query processing operators like sort, join, and even scan (see Section 2.1 below). So for moderate scales of parallelism, the network is not a bottleneck.

DITN derives two advantages by not physically partitioning tables on disks:

Operator	Throughput	Query
SCAN	7.69MB/s	select sum(R.a) from R
SORT	0.48MB/s	select max(13-7*R.a) from R
ODBC-SCAN	0.32MB/s	select sum(R.a) from R-NN

Table 1: Throughput of query operators (R is a local table, R-NN is a remote table accessed via ODBC)

- The cost for adding (or removing) a node is low, and so DITN can use transiently available machines. DITN can also gradually grow or shrink the system as the load varies.
- We are not forced to grow CPUs and disks in lock-step. We can independently scale the CPUs or disks, depending on which is the bottleneck.

2.1 CSV File-based communication

A key assumption behind DITN is network transfer costs are a minor part of query processing cost. But in order to realize these fast transfer rates, we need to use a file-based data communication scheme. We explain this with a micro-benchmark experiment.

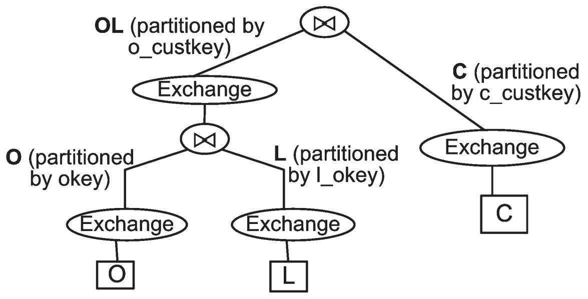
We run queries that aggregate records from a table R with a single integer column "a". The queries are designed so that their cost is dominated by a single operator. We run this query with R table of different sizes, fit a linear curve, and thereby calculate an operator *throughput*, defined as the number of bytes that operator can process in a second. Table 1 lists some operators and their observed throughputs. Note first that all these throughputs are *much lower* when compared to network throughputs – in a LAN point-to-point bandwidths of 2 or 4 Gbps are common, and aggregate bandwidth can reach 10 Gbps with modern switches and backplanes. This large difference between query operator and network throughputs is the justification for not co-locating data with compute node (via partitioning) – the network can ship data faster than query operators can process them.

But an important qualifier to this reasoning is that DITN should transfer work-units to the co-processors at the full network bandwidth. Realizing this is tricky. For example, the ODBC scan throughput is only 0.32MB/s – this is far less than the ideal throughput of 7.69MB/s that is theoretically achievable (the scan throughput is 7.69MB/s, and the network bandwidth in this experiment is 1 Gbps, so we should be able to scan remote data at 7.69MB/s).

Our solution to this problem is to keep the work-units as comma separated value files, rather than as database tables. So when a co-processor accesses its work-unit, it directly reads a corresponding byte range from the CSV file for each input table.

3 Intra-Fragment Parallelism

Having described the DITN storage architecture, we now turn to work allocation. The task is to break up the SPJAG over tables in the shared storage into SPJAGs over work-units that can be done on the co-processors.



⋈
OL (partitioned by o_custkey)
Exchange
⋈
O (partitioned by okey)
L (partitioned by l_okey)
Exchange
Exchange
O
L
C (partitioned by c_custkey)
Exchange
C

Figure 2: Exchange-based plan for OxLxC

Traditionally, work allocation is parallel query plans is done through Exchange [7] operators (also known as TQs [2]). Both base tables and intermediate results are split according to the join column as part of query execution, and each node gets to operate on a table subset with particular hash value (or range) of the join column. Figure 2 shows an example plan.

The problem with this work allocation approach is that Exchange assumes significant homogeneity and predictability of CPU speeds on the co-processors.

- First, since tuples are shipped between plans, Exchange is very vulnerable to load spikes and failures at even a single compute node.
- Second, all nodes must run the same query plan. This is inefficient if the nodes have different CPU speeds, memory, etc. For example, a node with more memory may want to do a 1-pass hash-join while another with less memory does a merge join.
- Tuple shipping between plans also typically forces each node to run the same release of the same DBMS, which is restrictive over a grid.

To avoid these limitation, we outsource not at the operator level but rather at the join fragment level. We adopt a zero-communication, "embarrassingly parallel" method of splitting a SPJAG: we send the entire SPJAG to each co-processor, to be run independently. To divide the work, we divide the *input* into work units.

Given a join over tables $T_1, T_2, \dots T_k$, we logically divide each input into partitions $T_i = T_i^1 \cup T_i^2 \cup \cdots \cup T_i^{p_i}$. The join becomes a cross-product $(T_1^1 \cup \cdots \cup T_1^{p_1}) \times \cdots \times (T_k^1 \cup \cdots \cup T_k^{p_k})$. Each component of this cross-product is assigned to a separate co-processor, so we use $m = \Pi_{i=1}^k p_i$ co-processors in total. The I.I. node performs a top-level union of the results from each co-processor.

This division of the cross-product into work-units, and the assignment of work-units to processors, must be done carefully. In order to minimize the overall query response time, larger tables must be broken down into more logical partitions, and larger work units must be assigned to faster nodes. The cpuwrapper performs this optimization by modeling the query response time as a function of the work unit sizes and the node speeds, and solving a linear program to minimize the expected response time. Before describing this, we briefly discuss aggregation and group by.

Aggregate pushdown:

Besides joins, it also useful to push down aggregates and group bys, because they can substantially reduce the amount of data that needs to be communicated back from the co-processors to the cpuwrapper.

We use this simple transformation that is applicable for aggregates such as SUM and COUNT:

$AGG(GROUP\ BY(\cup_{i=1}^{n} R_i) = SUM(GROUP\ BY(\cup_{i=1}^{n} AGG(GROUP\ BY(R_i))))$

This transformation can also be used for AVG, STDEV, etc. by writing them in terms of these aggregates (*e.g.,* AVG = SUM/COUNT).

We apply this transformation by pushing down the group by and aggregation into the co-processor (so that each co-processor does the full SPJAG over its work-unit), and then doing a top-level group by and aggregation at the I.I. to combine the aggregates from each co-processor.

Figure 3 illustrates this aggregate pushdown.

.

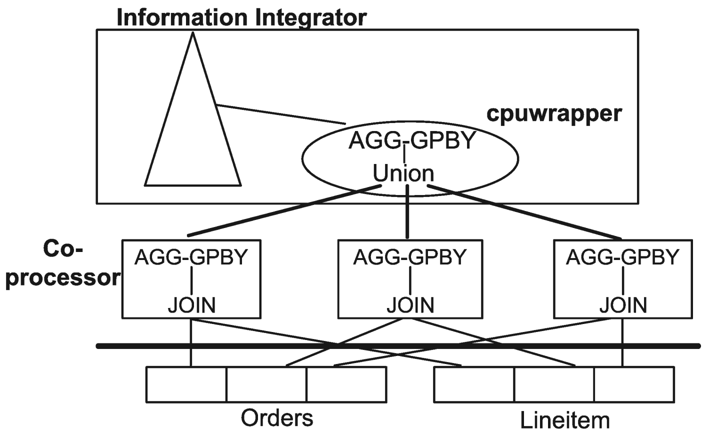


Figure 3: Query Plan for DITN $O \bowtie L$

3.1 Work Allocation Strategies

The cpuwrapper has two work allocation decisions:

- it divides the join cross-product space into work units
- it allocates work units to co-processors

The total work that needs to be performed in evaluating a join can be visualized as a hyper-rectangle corresponding to the cross-product of the input relations. Figure 4 illustrates this for a two table join. The total join work corresponds to the volume inside this hyper-rectangle, because each value in the cross-product must be determined to lie either within or outside the join result.

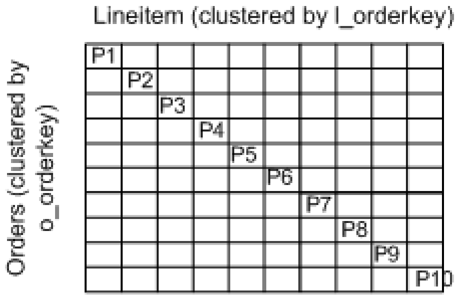

Lineitem (clustered by l_orderkey)
Orders (clustered by o_orderkey)
P1
P2
P3
P4
P5
P6
P7
P8
P9
P10

Figure 4: Exchange-Style Work Allocation for $O \bowtie L$

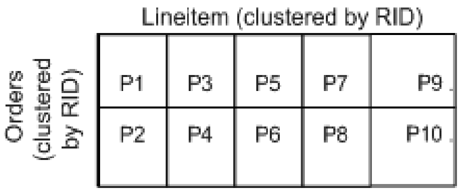


Figure 5: DITN Work Allocation for $O \bowtie L$

In the Exchange style of work allocation, careful partitioning ensures that the only work units that need to be evaluated are the *diagonal work units* – when the tables are laid out according to the partitioning. For more than two-table joins, there may not be a single layout of the tables that eliminates off-diagonal work; so we may need to perform exchange on the intermediate results in the query plan. This tuple exchange is intrinsically vulnerable to failures and stalls, because reassigning the intermediate tuples that have been sent to a stalled node involves recomputing those tuples from scratch.

We have studied two alternative styles of dividing up the cross-product space into work units that avoid these problems:

- Divide by RIDs (Record IDs), as per the existing table clustering.
- Divide by RIDs, with dynamic partitioning of the largest two tables.

Input Division by RID

The simplest method of dividing the inputs is to use their existing clustering on disk, and divide the input files by RID, as in Figure 5. This corresponds to dividing the logical input tables by their RIDs.

Thus each T_i^j is a contiguous set of tuples, as identified by their RID. The advantage is that each co-processor processes a contiguous set of tuples from the input table, which provides substantial I/O efficiencies for join algorithms that scan their inputs (*e.g.,* hash join, merge join, clustered index join).

The cpuwrapper can implement this RID based splitting in many ways. If the data is stored in regular database table format, we can specify a direct predicate on the RID, or a predicate on the clustering column, to the query sent to each co-processor. For our Apache-Derby co-processor, RID-based splitting is even easier since the tables can be stored as CSV files. We simply divide up the file byte range. Our co-processor uses a file wrapper that takes in a byte range over a CSV file, and returns a tuple cursor. Note that the byte offsets chosen by the cpuwrapper need not align with record boundaries. The file wrapper searches in the vicinity of the byte offset to determine the closest record delimiter.

The drawback with this method is the extra work needed to perform the off-diagonal work units. Given a fixed number of co-processors, the presence of these additional work units forces the cpuwrapper to make each work-unit larger, and thereby increases the response time for performing the join.

Dynamic partitioning the largest two tables

A simple optimization avoids the off-diagonal work units for the two largest tables that are being joined. Before executing the SPJAG, the cpuwrapper explicitly re-partitions the two largest tables (that are connected by an equi-join predicate) by sorting (or hash-partitioning) on the join column connecting them. Then we can divide these two tables by the join-column value, rather than RID.

This is like doing an Exchange on the 2 largest tables. But we *do not ship tuples to the co-processors – data remains in the SANFS*. So each node can run its own plan, and if one node fails or stalls, we can re-allocate its work-unit to another node.

Since we avoid exchanging tuples in the middle of query plans, this method can only partition each table along one column. So the exchange-style work allocation will theoretically perform better than this method for joins involving more than two tables. For example, in a TPC-H equijoin of Lineitem (L), Orders (O), and Customers (C), this method partitions Lineitem and Orders by orderkey. But, unlike Exchange, it cannot re-partition the $Lineitem \bowtie Orders$ result on custkey for the join with Customers.

This is the tradeoff we make for the flexibility of heterogeneity and load resilience. Our experiments suggest that this performance loss is negligible with this optimization, because the largest tables tend to dominate the cost of join processing (Section 5).

We now flesh out the RID-based work allocation scheme, to decide how to split each input table and allocate work units to the co-processors.

3.1.1 Symmetric Work Allocation Scheme

We first devise a work allocation scheme assuming that the co-processors are equally powerful, *i.e.,* they take identical time to join inputs of identical size. We then generalize this in Section 3.1.2.

We assume in this paper that the number m of co-processors to use is specified as a policy parameter by the application developer or database administrator (we discuss future work to automatically estimate m in Section 7).

Given a number m of equally powerful co-processors, the optimization problem for outsourcing SPJAG processing to co-processors is to determine,

- the number of partitions p_i to split each table T_i into, and
- the size of each partition $t_{i,j} = |T_i^j|, 1 \leq i \leq k, 1 \leq j \leq p_i$

Note: We represent the sizes $t_{i,j}$, $|T_i|$ and $|T_i^j|$ in bytes rather than tuples, for ease of analysis.
Let $JoinCost(x_1, x_2, \ldots, x_k)$ be the cost of processing a work-unit of size $x_1 \times x_2 \times \cdots x_k$ at a co-processor. Then, the response time (RT) is given by (d_i is an index into the work-units of i'th table):

$$RT = \max_{d_i \in [1,p_i] \forall 1 \leq i \leq k} (JoinCost(t_{1,d_1}, t_{2,d_2}, \ldots t_{k,d_k}))$$

It is easy to see that the response time is minimized when the work units are of equal size along each dimension.

Theorem 1 *For a fixed $p_1, p_2, \ldots p_k$, assuming that JoinCost is monotonically increasing in each of its arguments, RT is minimized when $t_{i,j_1} = t_{i,j_2}\ \forall 1 \leq i \leq k, 1 \leq j_1, j_2 \leq p_i$.*

Proof Sketch: Since *JoinCost* is monotonic, minimizing RT is equivalent to minimizing, for each dimension $1 \leq i \leq k$, the largest partition $\max_j(t_{i,j})$. This happens when partitions are equal-sized along each dimension. □

By this theorem, RT = $JoinCost(|T_1|/p_1, |T_2|/p_2, \ldots |T_k|/p_k)$. Our next task is to find $p_1, p_2, \ldots p_k$.

The only restriction on $p_1, p_2, \ldots p_k$ is that they be positive integers, and that $\Pi_{i=1}^{k} p_i = m$. To minimize RT we ideally want to know the exact form of the *JoinCost* function. Unfortunately we are dealing with remote, possibly heterogeneous join processors. So DITN currently assumes that *JoinCost* is a symmetric polynomial function of its inputs, with non-negative coefficients:

$$\begin{aligned} JoinCost = {} & \lambda_1 \textstyle\sum_{i \in [1,k]} (|T_i|/p_i) + \\ & \lambda_{1,2} \textstyle\sum_{i_1, i_2 \in [1,k]} (|T_{i_1}||T_{i_2}|/(p_{i_1} p_{i_2})) + \\ & \lambda_2 \textstyle\sum_{i_1 \in [1,k]} (|T_{i_1}|^2/(p_{i_1}^2)) + \quad \cdots \\ & \text{where } \lambda_1, \lambda_{1,2}, \lambda_2, \ldots \geq 0. \end{aligned}$$

This is a fairly general cost model which accounts for most of the typical join algorithms used – *e.g.,* for hash and merge joins the linear terms dominate, in nested loop joins the quadratic terms matter, etc. Since we don't know the internal details of the co-processors, we assume that the cost is symmetric in the input sizes.

Theorem 2 *Assuming that JoinCost is a symmetric polynomial function of its arguments, with non-negative coefficients, RT is minimized when* $|T_1|/p_1 = |T_2|/p_2 = \cdots |T_k|/p_k$.

Proof Sketch: Since *JoinCost* is symmetric, all the terms that are equivalent under transposition of input arguments (*e.g.,* { $(|T_1|/p_1)^2(|T_2|/p_2)^2(|T_3|/p_3)$, $(|T_1|/p_1)(|T_2|/p_2)^2(|T_3|/p_3)^2$, $(|T_1|/p_1)^2(|T_2|/p_2)(|T_3|/p_3)^2$ }) must have the same, non-negative coefficient. The product of these symmetric terms is some power of the product of the inputs (in the previous example, $(|T_1|/p_1)^5(|T_2|/p_2)^5(|T_3|/p_3)^5$), and hence a constant — because $m = \Pi_{i=1}^{k} p_i$. The result follows by AM-GM inequality. □

Thus, the optimal work allocation is to have an equal split within each table, and to make the partitions of all tables be approximately equal. For example, in a TPC-H database, a large table like Lineitem should be split into more segments than a smaller table like Orders or Customer.

Impact of dynamic partitioning optimization: The optimization described above is easily generalized to handle the case where the cpuwrapper partitions the two largest tables (say T_1, T_2) according to their join column.

We have $p_1 = p_2$, and Theorem 2 generalizes to $T_2/p_2 = T_3/p_3 = \cdots = T_k/p_k$. The partitioning of T_1 and T_2 allows us to make each p_i larger (and each work-unit smaller) because we now have only $k-1$ terms in the product ($m = \Pi_{i=2}^{k} p_i$).

The only constraint is that in general we will not be able to achieve $|T_1|/p_1 = |T_2|/p_2$. We resolve this by choosing the number of partitions for the larger table (say p_1) according to Theorem 2, and setting the other number (p_2) to be the same. We can use standard skew-avoidance techniques (like [23]) to ensure that all partitions within T_1 and T_2 are of approximately the same size.

Join vs Data-Access response time
This RT that we have optimized for is the time for performing the join processing within each work unit. But this operation goes in parallel with the data access from the storage software. So, the overall response time for the SPJAG fragment is:

$$\max(JoinCost(\ |T_1|/p_1, |T_2|/p_2, \ldots |T_k|/p_k), \quad \xi(|T_1|/s_1 + |T_2|/s_2 + \cdots)) \qquad \textbf{(1)}$$

where s_i stands for the degree of striping of the tables, and ξ is the scan cost per byte.

Notice that the data access cost is independent of how we partition the input tables. For an I/O bound workload, this cost will dominate. But the DBA can scale up such a workload without using more co-processors, by adding more disks to the storage pool.

3.1.2 Asymmetric Work Allocation Scheme

When the co-processors are not identical, they have different *JoinCost* functions. We solve for this situation by first doing the work allocation symmetrically, as in Section 3.1.1, and then adjusting the allocation to account for heterogeneity.

Consider the $Orders \bowtie Lineitem$ query and assume that we have arrived with the symmetric work allocation of Figure 5. If the co-processors $P1 \ldots P10$ have different JoinCost functions, each work-unit will complete at a different time. So our strategy is to consider adjacent work-units (like P1 and P2) that have different expected response times, and move the edge shared by them so that the we increase the size of the work-unit with the faster co-processor, and decrease the size of the work-unit with the smaller co-processor.

This process can be generalized as follows.

Suppose that o_1, o_2 are the sizes of the partitions of Orders, $l_{1,1}, \ldots l_{1,5}$ are the sizes of the partitions of Lineitem along the row corresponding to o_1, and $l_{2,1}, \ldots l_{2,5}$ are the sizes of the partitions of Lineitem along the row corresponding to o_2 — note that with asymmetric co-processors we must consider such non-grid like work allocations also. Now, $RT = \max_{i_1 \in [1,2] i_2 \in [1,5]} JoinCost_j(o_{i_1}, l_{i_1,i_2})$, where $JoinCost_j$ is the cost function of the processor allocated to each work-unit.

To solve this, we assume that JoinCost's are linear symmetric functions of the input sizes: $JoinCost_j(x,y) = \lambda_j(x+y)$, where λ_j is a cost calibration factor that the cpuwrapper learns from a grid load monitor.

This leads to a system of linear in-equations:
$RT \geq \lambda_j(o_{i_1} + l_{i_1,i_2})\ \forall\ 1 \leq i_1 \leq 2, 1 \leq i_2 \leq 5$
$\sum_{i_2=1}^{5}(l_{i_1,i_2}) = |Lineitem|\ \forall\ 1 \leq i_1 \leq 2$
$\sum_{i_1=1}^{2}(o_{i_1}) = |Orders|$
$o_{i_1} \geq 0, l_{i_1,i_2} \geq 0\ \forall\ 1 \leq i_1 \leq 2, 1 \leq i_2 \leq 5$

We solve this linear program to find the partition size o_1, o_2, and $l_{1,1}, \ldots l_{2,5}$. This method is easily generalized to joins of more than 2 tables as well.

There are two current restrictions of our method.

The first restriction is that we need to pick an allocation of co-processors to work-units (*i.e.,* the λ_js) beforehand, arbitrarily. One afterward do we choose the work-unit sizes for each co-processor. This can be sub-optimal, but relaxing it (*e.g.,* by adding a 0-1 indicator variable that denotes whether a particular processor is allocated to a particular work-unit) will make the problem integer programming. We have found this pre-allocation to work well in our initial experiments.

Second, we adjust the partition sizes according to a specific ordering of the tables. For example, in the Orders ⋈ Lineitem query, we adjust the Lineitem partition sizes for each Orders partition, and adjust the Orders partition sizes only overall. So the Orders partition for P1 and P2 must be of the same size. The ordering of the tables we choose is from largest to smallest. Our intuition is that we need the most flexibility in choosing work-unit sizes only for the largest table.

4 DITN Runtime

We have seen how the cpuwrapper divides an SPJAG into work-units for each co-processor. We now discuss how these work units are executed, as part of the overall query execution.

During query execution, the I.I. views the cpuwrapper as a source that serves the overall SPJAG result. I.I. uses a typical iterator model of query execution, where it makes an open() call to the cpuwrapper, and then calls fetch() repeatedly to fetch more tuples, and finally invokes a close().

The implementation of these three calls is fairly simple. At open() the cpuwrapper chooses the idle co-processors, allocates the work-units and dispatches the SPJAG requests to the co-processors. The close() call is directly passed on to the co-processors.

During fetch(), the cpuwrapper asynchronously fetches and unions results from each co-processor. Results from different co-processors can be interleaved in any manner because the cpuwrapper promises only an unordered, union semantics to I.I. We now elaborate on two aspects of this fetch() process: how the cpuwrapper handles load bursts (Section 4.1), and how the co-processor is implemented (Section 4.2).

4.1 Handling Load Bursts and Failures

The essential characteristic of a non-dedicated co-processor is transiency. The co-processor can be removed from the grid at any time, and its load can spike without notice, when a higher priority job arrives.

In a system that uses Exchange, the nodes to which an Exchange routes tuples are chosen once. If any of the nodes slows down or becomes unavailable during query execution, there is no adaptation, unless we recompute the entire subtree under the Exchange.

DITN tackles load bursts and failures as follows. During the fetch() call, the cpuwrapper keeps track of when each co-processor finishes its work-unit (by tracking EOFs). If one co-processor, say A, has not finished for a long time, the cpuwrapper waits a certain fraction of time f after all the co-processors have finished. At this point the cpuwrapper times out, and assumes that this co-processor A has either failed or is too slow. So it chooses the fastest co-processor B among the ones that have finished, and reassigns A's work-unit to B. The cpuwrapper picks up the result from whichever of the two co-processors finishes first, and cancels the processing being done by the other co-processor by closing its cursor.

The worst case for this algorithm is where both A and B finish the work simultaneously; the cpuwrapper has ended up re-using A unnecessarily. This is a factor of $(m+1)/m$ penalty when the query uses m co-processors. But this penalty is paid *only* when A has a significant slowdown.

In contrast, a fault-tolerant Exchange such as FLUX [17] pays a factor of 2 penalty, whether there is a fault or not. This is not a perfect comparison, because FLUX is designed for a general continuous query environment where the input data is a stream that may not be replayable from an earlier point. Moreover, we have delegated the fault-tolerance at the storage system level to the storage software; there is a cost for this (*e.g.,* RAID mirroring cost). But we believe the storage systems have perfected and commoditized fault-tolerance, and can provide it much more cheaply than an exchange based, "software" mirroring.

4.2 Lightweight Derby Join Processor

There are three key design goals in selecting a query processor to use for joining the work units at each co-processor.

First, it should be easy and automatic to install, so that the switching cost to exploit a new compute node is kept low. Many DBMSs have very complex installation procedures that are very hard to automate.

Second, it should have low footprint when not in use.

Third, it should be able to access remote data at the full network bandwidth. Recall from Section 2.1 that ODBC has a huge overhead, and slows down the data access bandwidth substantially.

We have experimented with two options to use as our query processor at the co-processor nodes. The simplest solution is to use I.I. itself, since it is a federated query processor and can access remote data sources efficiently. But it is too heavyweight and hard to install.

The alternate solution that we adopt is to use a hugely trimmed-down version of the Apache Derby open source DBMS. Observe that we only need the join and aggregation feature from the DBMS, and it need not store any data – so we need no updates, no complex SQL, no data manager etc. This allows us to reduce the co-processor software to under 15000 lines of code, and a single jar file.

We wrote a file wrapper for this co-processor so that it can access comma-separated-value files. The advantage of accessing files rather than ODBC is that file transfer is done by the file system, and it does not slowdown the access throughput like ODBC does.

5 Performance Results

The central tradeoff that DITN makes is to sacrifice partitioning in return for two benefits: (a) load and failure resiliency, and (b) graceful scaleout – the ability to gradually add nodes as the workload grows. We now present a preliminary experimental evaluation of this tradeoff. We quantify the loss in performance due to imperfect partitioning, and the benefits of load and failure resiliency. Quantifying the benefit of graceful scaleout will involve a cost analysis of clusters versus SANs. We defer this to future work.

Our experiments are based on a prototype implementation of DITN in Websphere Information Integrator (I.I). We use a cluster of 10 Pentium 4 workstations with 1GB memory each, running Windows XP and our trimmed down version of the Apache-Derby query processor. Our SAN had 12 disks connected by a 1 Gbps ethernet, and running the IBM TotalStorage SAN File System.

We compare query performance using three techniques: data in the network (DITN), DITN with the optimization that partitions the largest two tables involved in an equijoin by the join column (DITN2PART), and a partitioning-based parallel query plan (PBP). To make an apples-to-apples comparison, all three techniques are implemented on the same SAN itself, and all three techniques used the Apache Derby query processor for co-processor.[1]

Our experiment used a TPC-H database (scale factor 1GB). We use a variety of SPJAG queries (Table 2) in our experiments, ranging from a two table join of Orders and Lineitem (Q1), to a 6-way join (Q3). We use these rather than the actual TPC-H queries because we want to study the parallelization of their SPJAG portions in depth.

DITN and DITN2PART were implemented as described previously. In DITN, all the tables are clustered by their primary keys, whereas in DITN2PART the largest two tables (usually Orders and Lineitem) are correctly clustered on their join column. During query execution, the cpuwrapper logically splits each input table into segments (as discussed in Section 3), and then invokes the join at the co-processors on each work-unit.

Ideally we would implement PBP using a query plan with Exchange. But Apache Derby does not implement an Exchange operator. We solve this problem by "doing the exchange for free", as follows. For each query, we do a perfect partitioning of each of the input tables, tailored to that particular query, *before* the query execution begins.

In our response time numbers, we do not count the cost of this pre-partitioning – neither the cost of the partitioning the two largest tables in the join for DITN2PART, nor the cost of partitioning all the tables in the join for PBP. Thus our results skew the performance numbers of DITN2PART and PBP. This is not perfect, but is necessary because Apache Derby does not have an Exchange operator. However this imperfection is ok, because we favor PBP more than we favor DITN2PART. The point to note is that we are only overestimating the loss in performance of DITN and DITN2PART due to imperfect partitioning.

[1]In contrast, IBM's shared disk [10] and shared nothing [2] use specialized query processors, and a different interconnection topology: the shared disk system uses a sysplex, and the shared nothing system uses a cluster of servers with disks attached to the servers.

Our goal is to validate the following hypotheses:

- The imperfect partitioning and execution of off-diagonal work-units does not slow down DITN and DITN2PART by too much, even in a homogeneous cluster with dedicated nodes (Section 5.1).
- When the nodes are not dedicated, DITN and DITN2PART perform better. In Section 5.2 we study the specific issue of load bursts and failure at the non-dedicated nodes.
- When the nodes are of heterogeneous processing power, DITN and DITN2PART perform better than PBP (Section 5.3).

Query	Joins and Group By involved
Q1	$O \bowtie L$ group by o_orderpriority
Q2	$S \bowtie O \bowtie L$ group by s_suppname
Q3	$S \bowtie O \bowtie L \bowtie C \bowtie N \bowtie R$ group by n_name

Table 2: Queries Used (O=Orders, L=Lineitem, C=Customer, S=Supplier, N=Nation, R=Region)

5.1 Response-Time Speedup

We start by measuring the loss in performance of DITN and DITN-OPT, when compared to PBP.

Figure 6 shows the speedup obtained for the two-table join (Q1) as we increase the number of co-processors. The graph shows that DITN performs similarly to PBP even as we increase the number of co-processors. DITN2PART performs identically to PBP because both the exact same plan, on the same data layout.

At 10 co-processors, the response time of PBP is 8.06s and that of DITN is 14.60s. It is instructive to compare this slowdown of DITN (about 81%) with what is predicted by Equation (1) of Section 3.1.1. With 10 co-processors, Lineitem is split into 5 partitions and Orders into 2 partitions. So the cost ratio of DITN to PBP is:

$$\frac{\max(JoinCost(|O|/2, |L|/5), \xi(|O|/10 + |L|/10))}{\max(JoinCost(|O|/10, |L|/10), \xi(|O|/10 + |L|/10))}$$

From experimentation, we have found JoinCost for this simple query (it is a sort-merge join) to be linear in the combined size of the inputs. Considering the CPU cost clauses alone, the ratio works out to $\frac{|O|/2+|L|/5}{|O|/10+|L|/10} = 2.6$.

The actual overhead, about 1.811 is lesser because the I/O costs are identical in both methods and we are not able to realize full I/O parallelism with respect to CPU.

Figures 7 and 8 show similar speedup obtained for queries Q2 and Q3. Again, we see that DITN is not that much worse than PBP, while DITN2PART performs almost identically. For these two queries, DITN2PART and PBP run different query plans. In this case; PBP carefully partitions all the tables of the join, whereas DITN2PART only partitions Orders and Lineitem. Nevertheless, the remaining tables besides Orders and Lineitem are small enough that the overhead of DITN2PART is very small.

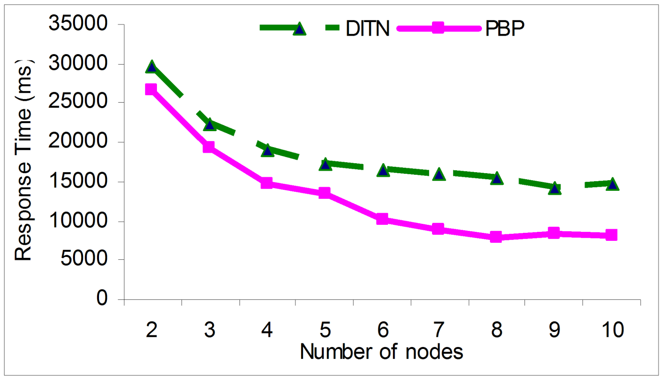


Figure 6: Response Time Speedup for Q1: join of OL

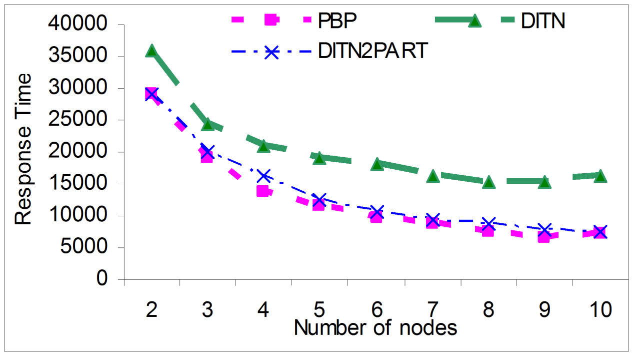


Figure 7: Speedup for query Q2: join of SOL)

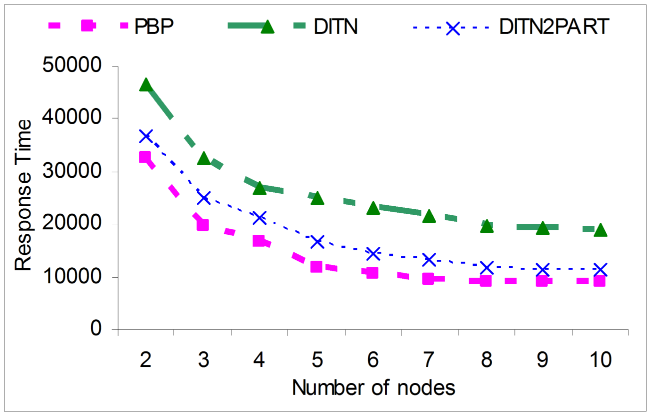


Figure 8: Speedup for Q3: join of SOLCNR

5.2 Load and Failure Resiliency

This experiment demonstrates the effectiveness of intra-fragment parallelism in adapting to load variations over time, as well as to node failures.

As discussed in Section 4.1, there are two kinds of adaptation to load variations. The first kind is to adapt dynamically, when one or more of the chosen nodes has a significant slowdown, stall, or failure during join execution. The second kind is to adapt by contacting a load monitor just before the SPJAG starts running, to find the best nodes to allocate work-units to. We now present two experiments corresponding to these two kinds of adaptation.

5.2.1 Resilience During Query Execution

First we run query Q1 over a cluster of 5 co-processors, with the tables striped across 5 disks. Immediately after the query begins execution (*i.e.,* right after the cpuwrapper has sent the SQL for the work-units to the co-processors), we impose an artificial CPU load on one co-processor[2].

Figure 10 compares the query response time of DITN and DITN2PART to that of PBP, for various values of the imposed CPU load. The x-axis plots the fraction of CPU utilization taken up by the artificially imposed load — the larger this fraction, the longer that co-processor will take to perform its work-unit.

The PBP curve shows a response time that continually rises with the degree of the load burst. In the case that a node suddenly becomes unavailable (i.e., the imposed cpu load is 100%,) the PBP system must wait until that node comes back up again. In the figure this is indicated by the response time growing out of bounds when the imposed load nears 100%.

In comparison, the cpuwrapper chooses an alternative node for the failed/slowed co-processor, after waiting 50% longer after all the other 4 co-processors have finished. In this experiment, the time-out period for DITN happens to be 28.6 seconds, and occurs whenever the imposed loads is at least 60%. So the entire query always finishes within about 46 seconds. In fact, the cpuwrapper overcomes the performance drawback of doing the off-diagonal work units, and starts to beat the perfectly partitioned system, whenever the load burst is at least 70%. DITN2PART performs even better. It closely follows PBP's curve until the imposed load hits 60%, and then starts benefiting from adapting to the alternative node. The upper-bound response time for DITN2PART is within 34 seconds.

5.2.2 Across-Query Load Adaptation: Dangers of Static Partitioning

To test load adaptation across queries, we re-run the same query as in the above experiment, but run it

[2]We impose a definite CPU load using the CPU Load simulator tool from Commlinx Solutions.

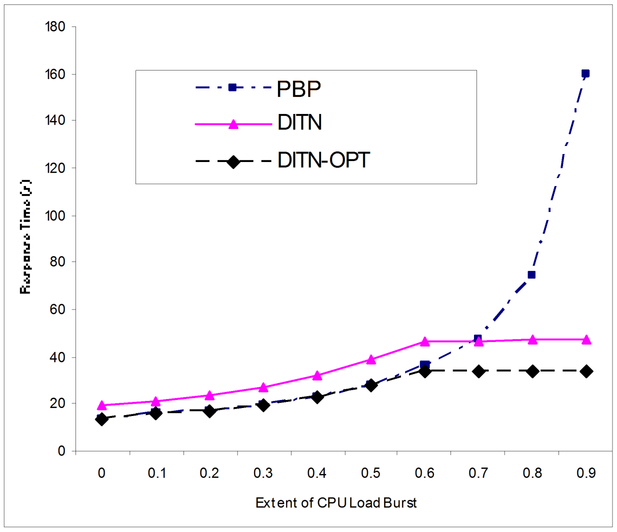


Figure 10: Load Resiliency during Query Execution

several times in sequence, retaining the artificially imposed load once it has been imposed. *i.e.,* during one particular query iteration we impose a load of 75%, and retain that imposed load throughout. The previous experiment shows how the cpuwrapper adapted during that particular query iteration. This experiment shows how the cpuwrapper learns of this load from the load monitor, and completely avoids picking that node for subsequent iterations.

The left graph of Figure 9 compares the response time of each query iteration, for DITN, and for a PBP system where the two input tables are statically partitioned per their join columns. During the first two iterations, PBP does better than DITN because it avoids the off-diagonal work units. In the 3rd and 4th iteration, when the node is loaded, both PBP and DITN response time almost quadruples. But DITN is able to recover subsequently by adapting to another available node on the grid. PBP continues to use the busy node, causing the response time to remain high.

Notice that this problem will not go away simply by implementing a load-balancing Exchange (such as [18]). Since the tables have been statically partitioned on their join columns, the query plan will not even contain an Exchange!

5.3 Heterogeneity

This experiment investigates how cpuwrapper performs in a heterogeneous environment. We run a the query over a cluster that initially has two fast nodes. As the size of the cluster grows, we adds more slow nodes into the cluster. For example, a cluster of 5 nodes includes 2 fast nodes and 3 slow ones.

Figure 11 shows how cpuwrapper with asymmetric work allocation makes use of slow nodes to improve query response time. It compares the total response

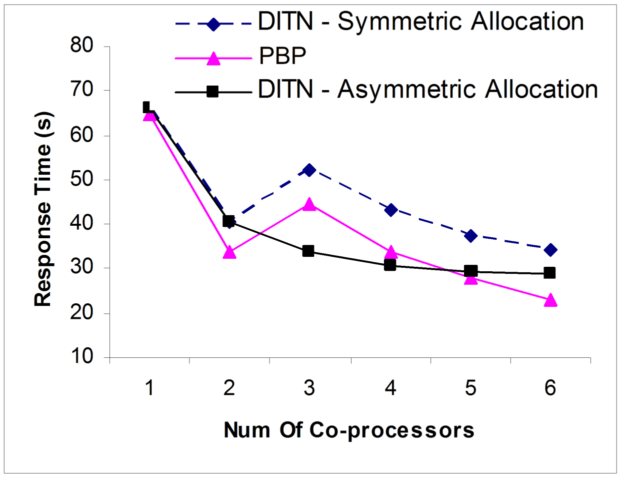


Figure 11: Handling Heterogeneity With Asymmetric Work Allocation

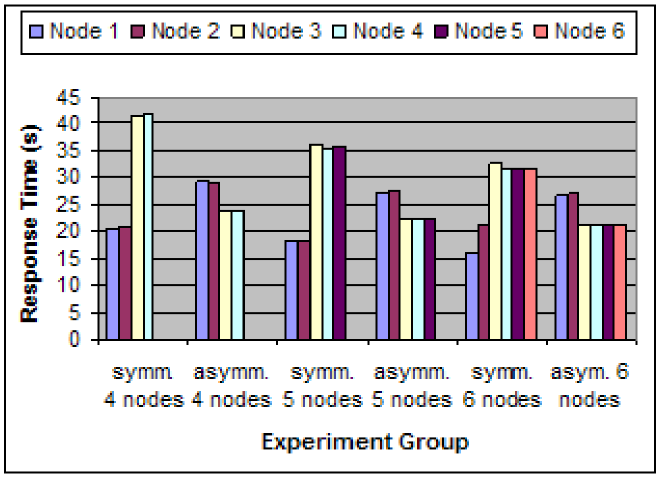
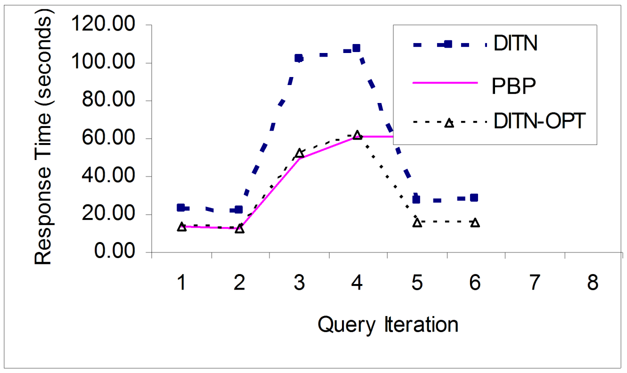


Figure 12: Individual Response Times

time taken by DITN with asymmetric work allocation to DITN with symmetric work allocation and PBP. The x-axis is the number of co-processors (nodes) in the cluster. For both DITN and PBP with symmetric work allocation, we see inclusion of slow nodes might introduce a sharp surge in total response time because the slow nodes need to spend much more time in the same amount of work load. They become destructors that cost longer response time than not including them. On the other hand, asymmetric allocation prevents slow nodes from being a bottleneck. The response time for asymmetric allocation decreases gradually as the cluster sizes increases. DITN with asymmetric allocation performs better than DITN with symmetric allocation at any cluster size. It can even beat PBP with symmetric allocation when the cluster size is 3 and 4.

Figure 12 demonstrates cpuwrapper can balance loads well across grid nodes. The y-axis is the re-



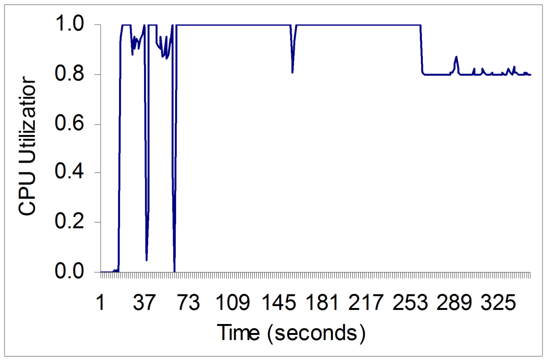


Figure 9: Load Resiliency across queries

sponse times taken at each individual node and the x-axis shows clusters in each query execution. For symmetric allocation, we see a big discrepancy between fast nodes and slow nodes. That means we have not given fast nodes enough work while slow nodes become the bottleneck for accepting too much work load. On the other hand, asymmetric allocation reduces the discrepancy. Interestingly, we have observed that the response times at fast nodes are longer than at slower nodes in asymmetric allocation, indicating we might have given too much work to the fast nodes. This behavior attributes to our assumption that JoinCosts are linear symmetric functions of the input table sizes in Section 3.1.2. That assumption favors fast nodes since it omits the polynomial components in JoinCost. However, favoring fast nodes makes sure adding a slow node can always improve the total response time.

6 Related Work

There have been numerous projects on parallel query processing in the last two decades. [4] is a good introduction to this field. Most of them (Gamma, Bubba, Tandem, Teradata, Informix XPS, DB2) [5, 3, 20, 22, 2]) use the shared-nothing [19] approach. [13] is an extensive critique that discusses many of the points that we raised in the introduction, in more detail.

There was quite a bit of interest in shared disk systems in the mid 80s and early 90s DITN is similar in spirit to the Shared Disk approach of the IBM Sysplex and the Digital VaxCluster [10]. These systems focus on transaction processing and distributed locking, rather than with parallelization of query processing. Indeed, [4] cite concurrency control as one of the two problems with the shared disk architecture (the other is network scalability). The IBM S/390 Parallel Query Processing system [10] does perform parallel query processing over a sysplex. But it parallelizes sort and scan operations, and relies on tuple shipping between co-processors to perform joins. Therefore it has the same limitations with respect to heterogeneity and non-dedicated processors.

Oracle Parallel Server is a system with a mix of shared nothing and shared disk characteristics. Hash joins are done as in shared nothing systems, by partitioning the data beforehand and joining each partition on a separate processor. Index nested loop joins and scans rely on parallelism from the storage system; the scans rely on parallelism via storage system striping, and the index lookups rely on the multiple disk-heads to achieve speedup in random access.

Another aspect of intra-fragment parallelism is the way the cpuwrapper adapts to load differences across the co-processors, and to load bursts and unavailability within each individual co-processor. This is related to the literature on load-balancing and skew-balancing either at the beginning of Exchange [15, 23, 11, 12] or even during Exchange [18]. The main advantage of our method over these is that by avoiding Exchange completely, we are not vulnerable to static partitioning, as the experiment of Section 5.2.2 showed.

The only related work on fault tolerant exchange that we are aware of is FLUX [17], where the Exchange routes each tuple to two nodes. This is a heavy-weight solution that needs dual redundancy even in the common case of no failures. Instead we timeout upon a failure or significant slowdown, and upon a timeout, we run the affected work-unit at another co-processor. We are able to avoid the dual redundancy of FLUX because we assume that the base tables are always available – their availability is handled by the storage system (FLUX in contrast is designed for fault-tolerance over general data streams, and cannot rely on a storage system to have a copy of the data available).

Some of the recent papers on adaptive query processing do look at distributed queries. For example, Eddies [1] and distributed Eddies [21] continually monitor the speed of query operators and use this information to adjust the query plan by changing the ways tuples are routed. In theory distributed eddies can allow heterogeneous plans at the co-processor. It will be interesting to combine our work with this.

The Polar-Star [6] system allocates query processing work to grid compute nodes using intra-operator parallelism. It takes in a plan with Exchange operators, and chooses the degree of parallelism of each Exchange operator in a cost-based fashion. The disadvantage of this approach is the heterogeneity and load/failure resiliency problems of intra-operator parallelism.

7 Conclusions and Future Work

We have presented DITN, a new style of parallel query processing that uses non-dedicated, possibly heterogeneous or even flaky compute nodes. We do not partition data across the compute nodes but instead leave it in a shared storage system, from which all co-processors access data as needed. We avoid shipping tuples between query operators by not using intra-operator parallelism. Instead we use an intra-fragment parallelism where entire join and aggregation blocks are outsourced to co-processors, using a federated DBMS.

Because the data is not perfectly partitioned, DITN does more work than a traditional partitioning-based parallel query system. Experiments over small scales (upto 10 nodes) suggest that this disadvantage is small, especially with an optimization to dynamically re-partition the two largest tables of the join.

But in return for this performance loss, DITN gets several advantages in situations where the co-processors have variable loads, can fail abruptly, or are heterogeneous. Moreover, DITN allows gradual scale-out of the parallel query processor using non-dedicated nodes.

One obvious area for future work is to see how far DITN will scale. Parallel DBMSs (especially of the shared-nothing variety) scale upto 100s and even 1000s of nodes. We need further investigation to determine up to what scale is the "sweet spot" of DITN.

Longer term, we want to explore push down of additional database operations into storage systems. For example, DITN-2PART dynamically partitions the largest two tables of a join. Instead, each co-processor can specify a predicate on the file open call to the base table. The storage software can accept this predicate from each co-processor, combine them into a de-multiplex operation, and push it into the network switch of the SAN (assuming a programmable switch).

An assumption in this paper is that a number m of co-processors to use is given by the user, and that DITN can pick the fastest m co-processors on the grid. But in practice a user does not care about m – he cares about the QoS (*e.g.,* response time) that his workload gets. It would be useful if DITN can take a QoS goal, and automatically determine a suitable subset of the co-processors that can meet it best.

Acknowledgments: We thank Garret Swart for many interesting discussions on this project.

References

[1] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*, 2000.

[2] C. Baru et al. DB2 Parallel Edition. *IBM Systems Journal*, 34(2), 1995.

[3] H. Boral et al. Prototyping Bubba: A Highly Parallel Database System. *TKDE*, 2(1), 1990.

[4] D. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *CACM*, 35(6), 1992.

[5] D. J. DeWitt et al. The Gamma database machine project. *TKDE*, 2(1), 1990.

[6] A. Gounaris, R. Sakellariou, N. Paton, and A. Fernandes. Resource scheduling for parallel query processing on grids. In *GRID*, 2004.

[7] G. Graefe. Encapsulation of parallelism in the volcano query processing system. In *SIGMOD*, 1990.

[8] W. Hong and M. Stonebraker. Optimization of Parallel Query Execution Plans in XPRS. In *PDIS*, 1991.

[9] V. Josifovski et al. Garlic: A New Flavor of Federated Query Processing for DB2. In *SIGMOD*, 2002.

[10] F. Lin. Understanding DB2 OS/390 Parallelism. www.ibm.com/software/data/db2/zos/index.html.

[11] H. Lu and K. L. Tan. Dynamic and Load-balanced Task-Oriended Database query Processing in Parallel Systems. In *EDBT*, 1992.

[12] M. Mehta and D. DeWitt. Managing Intra-Operator Parallelism in Parallel Database System. In *VLDB*, 1995.

[13] M. G. Norman, T. Zurek, and P. Thanisch. Much Ado About Shared-Nothing. *SIGMOD Record*, 25(3), 1996.

[14] *Oracle Database Data Warehousing Guide for 10g.* 2003.

[15] E. Rahm and R. Marek. Dynamic Multi-Resource Load-Balancing in Parallel Database Systems. In *VLDB*, 1995.

[16] M. T. Roth and P. M. Schwartz. Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In *VLDB*, 1997.

[17] M. Shah, J. Hellerstein, and E. Brewer. Highly Available, Fault-Tolerant Dataflows. In *ICDE*, 2003.

[18] M. Shah, J. Hellerstein, S. Chandrasekaran, and M. Franklin. Flux: An Adaptive Parititioning Operator for Continuous Query Systems. In *ICDE*, 2003.

[19] M. Stonebraker. The case for Shared Nothing. In *HPTS*, 1985.

[20] Nonstop sql whitepapers. http://www.tandem.com/prod_des/nstmmppd/nstmmppd.html.

[21] F. Tian and D. DeWitt. Tuple routing strategies for distributed eddies. In *VLDB*, 2003.

[22] A. Witkowski et al. NCR 3700 – the next-generation industrial database computer. In *VLDB*, 1993.

[23] J. Wolf, D. Dias, P. Yu, and J. Turek. An effective algorithm for paralellzing hash joins in the presence of data skew. In *ICDE*, 1991.