Loading Databases Using Dataflow Parallelism

Tom Barclay‡, Robert Barnes‡, Jim Gray †, Prakash Sundaresan$
Digital Equipment Corporation, San Francisco Systems Center

Abstract: This paper describes a parallel database load prototype for Digital's Rdb database product. The prototype takes a dataflow approach to database parallelism. It includes an *explorer* that discovers and records the cluster configuration in a database, a *client* CUI interface that gathers the load job description from the user and from the Rdb catalogs, and an *optimizer* that picks the best parallel execution plan and records it in a *web* data structure. The web describes the data *operators*, the *dataflow rivers* among them, the binding of operators to processes, processes to processors, and files to discs and tapes. This paper describes the optimizer's cost-based hierarchical optimization strategy in some detail. The prototype executes the web's plan by spawning a *web manager* process at each node of the cluster. The managers create the local *executor* processes, and orchestrate startup, phasing, checkpoint, and shutdown. The execution processes perform one or more operators. Data flows among the operators are via *memory-to-memory streams* within a node, and via web-manager multiplexed tcp/ip streams among nodes. The design of the *transaction* and *checkpoint/restart* mechanisms are also described. Preliminary measurements indicate that this design will give excellent scaleups.

1. Introduction: The Parallel Imperative

Technology and economic trends encourage us to build computers as processor-arrays, disk-arrays, tape-arrays, and communication-line arrays. We call such an array, a *cluster*. Clusters are a challenge to program. Current programming languages and techniques are geared to step-by-step algorithms. Clusters require parallel algorithms. Both the scientific and commercial communities are struggling to develop such new programming styles.

Some applications, like file service or online transaction processing, have natural parallelism. The applications consist of many small jobs operating against a common database. Over the last decade we have learned how to scale up such applications so that processor and disk arrays can service a hundred thousand users (clients) [Serlin]. The unifying concept has been the notion of a *transaction*: an atomic unit of work that executes independently of concurrently executing tasks -- giving the programmer the ACID properties (atomicity, consistency, isolation and durability). This allows programmers to write step-by-step algorithms, without concern for concurrency issues that parallel execution creates.

There have been notable success in building systems that execute a single large database tasks on a cluster. Teradata [Teradata], the Japanese 5th Generation project [Kitsuregawa 1, 2], the University of Wisconsin [DeWitt 1], and Tandem [Englert] demonstrate batch scaleup to large clusters. Oracle, Informix, NCR, Sybase and IBM all have ambitious parallel databae efforts underway.

Parallel online transaction processing systems are commonplace Parallel database systems have not had comparable success. They were ahead of their time -- the imperative for processor and disk arrays is just arriving. The technology of terabyte disk farms and 100 processor arrays not only *allows* parallel database access, the technology *requires* parallel data access. Hundred dollar per gigabyte disks allow very large online databases. These databases must be accessed in parallel. Scanning a terabyte at single-disk speed or single-processor speed will take days. Parallel access gives speedups of 100x or 1000x – turning a one-year task into an eight-hour job.

Figure 1 diagrams a cluster – a hundred-processor, hundred-tape, thousand-disk system. We believe such computers will cost less than a million dollars within ten years. Users will have so many components that one cannot program the individual processors and disks individually -- rather the system must automatically decide where to place data and computation within the cluster.

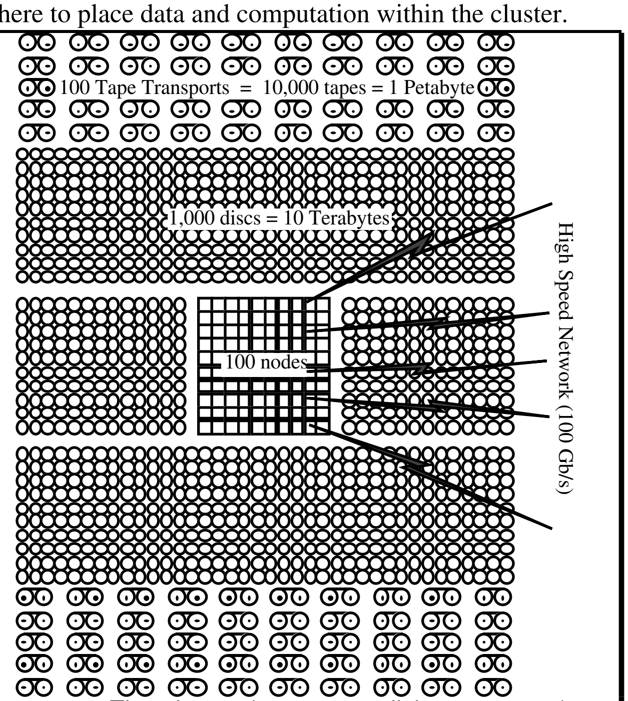

Figure 1: The cluster (processor, disk, tape array) we designing for. We believe such clusters will be the typical way large servers are built in the future. Each node has a few (less then 10) processors sharing a common memory of a gigabyte or so. Next in the storage hierarchy is a pool of discs, each served by a processor. Tape robots form the base of the storage hierarchy. All the components of the cluster have a high-speed interconnect (GB/s point-to-point) and a slower external network.

Current address:
‡ Microsoft, One Microsoft Way, Redmond, WA 98052-6399. {tbarclay, rbarnes}@microsoft.com.
† 310 Filbert St., S.F., CA 94133. gray@crl.com.
$ Informix, 921 SW Washington St. # 670, Portland, OR 97205. psundaressan@informix.com.

How can we program a cluster? The database community has adopted a *dataflow* approach to describing and implementing parallel algorithms. In this approach, data resides in files or databases on many disks, tapes, or other memory devices. Algorithms search subsets of this data, and either deposit their answers on target storage devices, or return their answers to an array of application programs. A dataflow algorithm is described as a directed graph.

Graph nodes, called *operators*, are sequential programs. Each operator reads its input record streams (dataflows), transforms the data, and produces one or more sequential output record streams. The operator is programmed as a purely sequential program in a conventional programming language (COBOL or FORTRAN or C). The edges of the graph show the dataflows among operators. Storage (disks, tapes) and application programs are data sources and sinks in the graphs.

The simplest dataflow graph is a *pipeline*, in which each operator takes in a stream of data, operates on it, and then passes it downstream (see Figure 2). Pipeline parallelism gives modest speedups because the pipeline is rarely very long: a pipeline of four operators gives at most a four-fold speedup. Partitioning the data streams and cloning each operator gives *partition parallelism*. If the data is partitioned among a thousand disks, a thousand-fold partitioning can give a thousand-fold speedup. Partition parallelism has huge payoffs -- especially as technology gives more and more disks and processors per dollar.

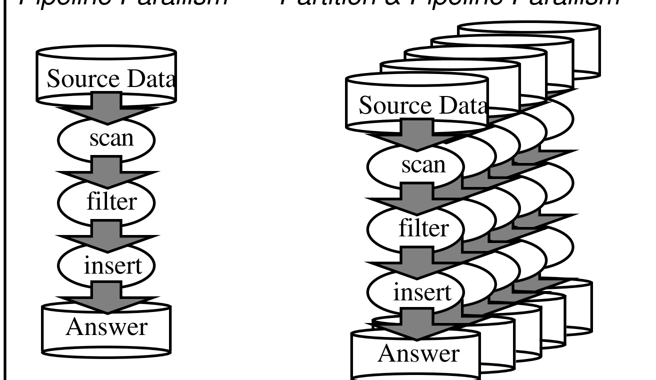

Figure 2: Two kinds of parallelism. (1) *Pipeline parallelism* has a sequence of operators operating concurrently, processing a single data stream. (2) *Partition parallelism* clones each operator of the pipeline and splits the data streams into many disjoint streams. Each of these streams is fed to a different operator. The pipeline in the left figure gives at most a three-fold speedup, the partition at the right gives at most a fifteen-fold speedup (3x5).

Relational databases are ideally suited to a dataflow approach. Relations are uniform collections of data. Relational operators consume one or more relations and produce a new relation. Certain operators like GROUP-BY and SORT do not produce pure relations, but they do produce uniform data streams. So relational operators are naturally pipelined, and the data streams are easily partitioned.

Figure 2 is a little vague. Each scan operator can certainly read a single input stream. But, what if the output records of a particular filter operator are destined for different insert operators. For example, what if the filter is a sort operator. Then "high" records should go to "high" insert operators and "low" records should go to "low" insert operators so that the concatenation of the resulting file partitions is indeed a sorted file.

The Gamma and Volcano systems developed a way to transparently partition data streams among operators. We refined those ideas with the following terminology. A data stream is partitioned when either or both of the source and destination operators are cloned to get partition parallelism. If the source is cloned N ways and the destination is cloned M ways, then there are $N \times M$ streams. We call the resulting set of data streams a dataflow *river*. Rivers are analogous to Gamma's split tables [DeWitt 1] and Volcano's *exchange* operators [Graefe].

Figure 3 shows a dataflow with the source operators partitioned two ways and the sink operators partitioned three ways. Each source operator dumps records into the river and each sink operator takes records out of its partition of the river. Each is unaware that the river is partitioned into six streams.

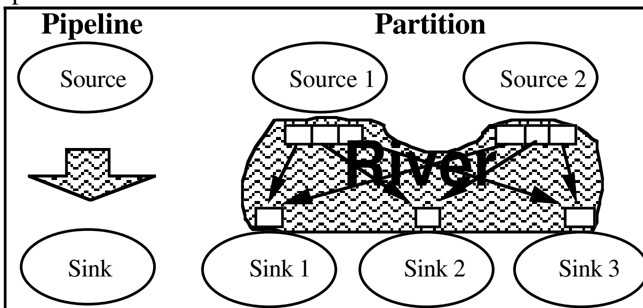

Figure 3: When operators are cloned, dataflows are partitioned. The partitioned dataflow, called a *river*, is composed of many point-to-point data *streams*. Source operators put records into the river. The river has a *split table* at each source that designates which sink operator (stream) the record should be sent to. Splits can be based on record key ranges, key hashes, on round-robin, or can replicate records. Sink operators have *merge* boxes that combine all incoming streams into a single stream.

River partitioning is based on a *split-table*. All the streams of a river have the same split table. As the name suggests, when a record is inserted into a river, the river program uses the split table to pick a destination stream for the record. The river program first extracts field values from the record. Then it compares these values to values in the split table to pick a destination stream. The split table can be a range-partitioning, a hash partitioning, a round robin, or even a replication (in which input records are sent to all sink operators).

This discussion gives a sense of the database community's approach to parallelism. Almost every database vendor has a parallelism project based on these ideas. All believe that parallel database systems will be a major trend over the next decade. In 1992 we started an advanced-development project to adapt known parallel database techniques to Rdb, Digital Equipment Corporation's database system. We were surprised to find that considerable research and innovation was needed to apply the techniques we thought were well understood. We encountered issues that we had never seen discussed before. This paper documents most of these issues and the approaches we took to them.

2. Goals and Approach

Our initial approach was to adapt the techniques used by Teradata, Gamma, and Tandem to Digital's Rdb. We wanted to build the infrastructure to execute parallel programs, and to build some utilities using that infrastructure. The implementation was to be portable among several operating systems (OpenVMS, OSF/1, and NT). This was a four-person effort separate from the Rdb development group -- so we focused on building a prototype that could later be integrated with Rdb. The prototype did not modify Rdb, but rather ran as an application. That integration is now in progress.

We designed for processor clusters accessing many disks and tape drives. Each node of the cluster is a shared-memory multi-processor with a large RAM memory. Nodes of the cluster are interconnected by a high-bandwidth (multi-gigabit/second) interconnection network. Each processor can directly access a subset of the disks and tapes. Typically, disks and tapes are *served* by one node to other nodes. A node indirectly accesses a device by sending requests to the device's server node.

We wanted to apply parallel techniques to the problem of loading data from disk or tape into an SQL database. This is a fixed problem and so is much simpler than planning an arbitrary database query. On the other hand, data loading exercises all the components of our infrastructure. It has to pick a load plan, assign processes to processors, scratch files to disks, start and monitor the processes executing the plan, deal with failures, and provide an operator interface to control and observe the load operation. This infrastructure is equally useful to a parallel query executor.

Simply stated, a data load task copies a collection of data records into a disk-resident SQL database. Several requirements are implicit in this description:

- **Heterogeneous:** The input stream may have multiple record types and the target may be multiple tables. The target tables may be partitioned to disjoint storage areas or clustered together in common storage areas.
- **Diverse-Input Media**: The data source may be a process, a file, or a table. If it is a file, the data may be on disk or tape.
- **Data conversion:** The input data format is probably different from the table format (data types). The data must be converted to the target data types (i.e., ASCII to IEEE float).
- **Integrity Checks**: The input data may have errors. Erroneous data is sent to a *reject file* and diagnostics are sent to a *message file*.
- **Clustering and Partitioning:** The table may be partitioned among many storage devices -- either for capacity or bandwidth. Each storage area is defined by a partitioning criterion, and each has a rule for clustering related records. For efficiency, the load operation must partition the input data and then sort each partition into clustered order.
- **Indexing**: The records of the table typically have secondary indices (hash, B-tree, R-tree, signature,....). These indices must be updated to reflect the new records.

Our main focus was on the use of parallelism to accelerate load operations. But speed is not the only requirement. The load operation is expected to have the following properties.

- **Automatic**: Once the load is specified, the details of allocating space and performing the load should be automatic.
- **Incremental**: It should be possible to load additional data into a pre-existing table, not just load data into a new table.
- **Online**: Especially for incremental loads, other applications may need read-write access to the table while the load is proceeding.
- **Scaleable**: The load should be able to handle very large jobs. Terabyte loads will be common.
- **Monitored:** The system administrator wants to inquire about the load status, cancel the load, suspend it, resume it, or change the load rate.
- **Restartable**: If there is a failure during the load, the operation should be restartable, with no loss of data integrity and with minimal loss of work.
- **Portable**: The design should be portable to a verity of commodity operating systems (NT, UNIX, OpenVMS).

Our prototype focused on a single-table load supporting process, file (disk or tape), or table sources. B-tree and hashed base tables and indices are supported. All load phases except table definition are automated. Incremental and online load is supported by combining transactions with a checkpoint-restart facility.

Figure 4 diagrams the load data flow for a clustered base table and three indices. The legend explains the flow.

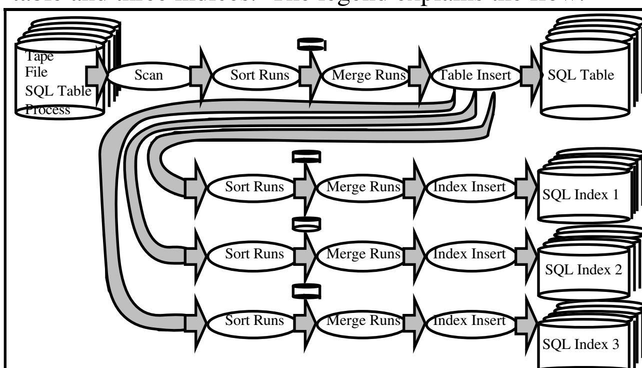

Figure 4: The dataflow graph of a single-table database load operation. Input records must be sorted by the clustering key in order to build the base table -- otherwise the insert operation would do random disk IO and would run 100-times slower. The base table has three secondary indices in this example. The index records are of the form: (alternate-key, record_id). One cannot construct the index record until the base table record has been placed and its record_id assigned by the database system. The table insert operator builds these index records. Each index load has a dataflow graph similar to table-load graph. If the base table or indices are hashed, then a hash operator (MakeDBkey) must be inserted prior to the sort step in the flow, so that records can be sorted in hash order. Each node of the graph can be cloned for partition parallelism.

You might think that Figure 4 allows a seven-fold pipeline speedup. After all, the pipeline is seven deep and the three index-build steps can proceed in parallel. In fact, the sort-merge steps are *blocking*: merge cannot start until sort-run generation has completed. The little disks above the arrows in Figure 4 indicate these blocking flows. No pipeline in Figure 4 is deeper than three operators. The computation actually consists of three *phases*:

(1) Scan the input stream and build base table runs.
(2) Sort the runs, insert the base table records, and generate the index records. Sort these index records into runs.
(3) Merge the index records insert them into the indices.

Pipeline speedup will be less than three-fold on this job. Partition-parallelism must be used to get ten-fold and hundred-fold speedups.

The parallel loader automatically builds and executes parallel versions of such graphs. It picks the appropriate graph and the appropriate degree of parallelism for each phase. The plan is constrained by the input and output data formats and by the cluster size and speed. The goal is to find and execute the fastest plan satisfying these constraints.

Our specific performance goal was to load a Wisconsin terabyte in a day. The load can use the cluster described in Figure 1. A Wisconsin terabyte is based on the Wisconsin Benchmark [DeWitt 2] and consists of:

- A four billion record base table. Each record is 208 bytes and has 18 integer and character fields. This table occupies about 800GB.
- Three B-tree indices on three integer fields. One of the indices is "clustering", the others are secondary indices. Each index has 4 billion entries, and is about 60GB. Together they are about 180GB.

We calculate that this job would take over a 150 days if run on Rdb without using parallelism. Our goal was a 150x speedup by using a combination of parallelism and improved algorithms.

3. System Structure

Load requests are defined by commands to a character or graphical user interface (GUI or CUI) process called the *client*. The commands describing the load job are passed to an *optimizer* program that picks a parallel execution plan for the job.

The optimizer has three inputs:

- The user's description of the input data size, source, and format.
- The SQL database and table definition of the target table and its indices.
- A definition of the cluster's hardware and software as found by a cluster explorer.

Based on these parameters the optimizer picks a dataflow graph, a degree of parallelism for each operator, and a binding of operators to processes in the cluster. The plan is chosen to minimize the elapsed execution time of the load job.

The execution plan is expressed as a data structure called a *web*. The web can be displayed in human-readable form, but its purpose is to define the execution plan to the parallel execution environment. For debugging purposes the CUI can construct and edit webs. This allowed us to manually program the execution environment before the optimizer was fully functional. It also allowed us to benchmark the optimizer's webs against hand-built ones.

Once a web has been picked, a *web manager* process is forked to execute the web. The first manager forks web managers at each other node of the cluster. Each web manager in turn forks a local set of *executor* processes to perform the web operations at the local node. The web managers communicate among one another using tcp/ip. Within a node, all interprocess communication is via a stream interface built on shared memory to eliminate memory copies.

The executor processes examine their part of the web and create an execution thread for each operator. Operator threads initialize themselves by opening their input and output rivers, files, and databases. Thereafter, each thread executes by reading input data from input rivers or files, processing data, and then writing output data to output rivers or inserting the data into an SQL table. The execution rate is limited only by the speed of each operator and by the rate at which data can flow through the rivers.

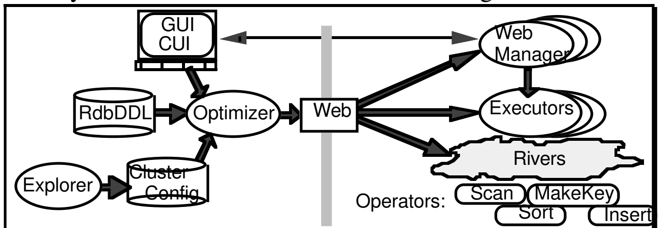

Figure 5: The parallelism prototype has a planning phase shown at the left and an execution phase at the right. The optimizer generates a parallel plan (web) based on the job definition, the database definition, and the cluster configuration. The client starts the plan execution by forking a web-manager. It in turn forks web managers in each other cluster node. The web managers fork executors at their nodes. The executors perform their part of the web by creating a thread per operator. The operators communicate via the river system.

If the load is incremental, the executors commit their updates every few seconds -- this prevents resources from being locked for very long periods, but has a high cost in logging and forcing premature index updating. In any case, the web managers maintain a checkpoint-restart mechanism that allows the web to restart from a recent point and continue the computation. The restart mechanism is designed to assure that each record is loaded exactly once.

3.1. CUI-GUI: The User Interface

Suppose the system administrator has a terabyte of data to be loaded into the system. The data could come from local disk files or tapes, or it could arrive via high-speed communication lines.

The administrator invokes a client process and describes the input data. He defines

(1) the input record format in a field-by-field manner giving its name and type,
(2) the names of the data sources be they files or tapes,
(3) if the input is from tape, then the approximate number of records on each tape,
(4) the name of the target database and table, and
(5) if the target table does not already exist, the logical definition of the table including column names, types, indices, comments, constraints, and triggers.

Ideally, the parallel database utility would do the rest. It would do the physical database design for the target table, pick a load plan, and execute it. Our prototype does not do the physical design. The system administrator must define the table and indices. In addition he must partition it among the discs. Looking at Figure 1 the administrator has to pick the thousand storage-area partitions and assign the storage areas to the thousand individual disks. Clearly this physical design process should be automated by a program that looks at the configuration database (Figure 6) and spreads the data among disks with enough capacity to hold the data and carry the data traffic. The physical database design program should create storage areas to hold the data, and then connect these storage areas to the table definition by extending the table partitioning criterion. These storage areas could be allocated by a greedy algorithm that simply spread them among disks proportional to the free space on each disk. Alternatively, the distribution could be based on the current "heat" of each disk, preferentially placing data on the coldest (least used) disks. To restate, we wanted to do this, but did not have time to implement it. Rather, in our prototype, the human user did physical database design.

The loader did automate all steps beyond physical database design. The optimizer reads the source and target data definitions, plans data conversion, data sorting, data loading, and then index building. The optimizer produces a web. This step takes a few seconds. The web is passed to the web manager processes for execution.

Once the web is executing, the client can monitor the execution by peeking at the per-web shared memory at each node. The prototype client has a primitive character interface to monitor the execution -- but it is still quite useful. A graphical interface would be nicer. The user can stop or cancel the web by issuing commands to the client.

3.2. The Explorer: Discovers Cluster Configuration.

The first task in programming or managing a cluster is to *explore* the cluster and record the configuration and capacity of each node and device. Our explorer runs atop the operating system and builds an SQL database describing each node, disk, tape, and describing how they interconnect (See Figure 6).

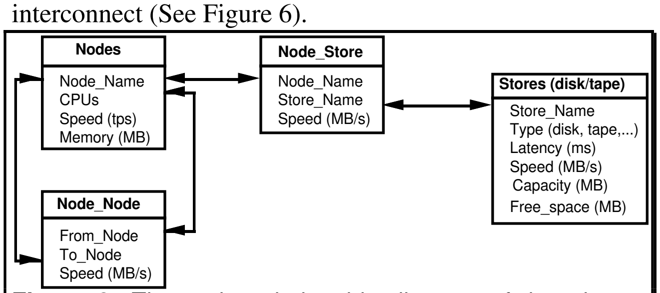

Figure 6: The entity-relationship diagram of the *cluster configuration database* built by the explorers executing at each node. The Nodes table describes the processors and memory at each node. The Stores table describes the speed, capacity, and free space of each disk and tape robot. The Node_Store and Node_Node tables record the point-to-point connectivity and bandwidth between directly-connected nodes and disks and among nodes.

An explorer process is launched at each cluster node. Each explorer examines the size and speed of the processors (by running simple benchmarks and by asking the operating system). It also benchmarks the accessible disks by reading and writing them and it records how much free space each disk has. It then benchmarks the speed of the interconnect between this node and other members of the cluster (again by running simple benchmarks). The results of all these experiments are recorded in the cluster-wide configuration database shown in Figure 6. The explorer runs occasionally (once a day) to update this database with current statistics.

The actual configuration database includes more detailed information about the nodes and stores. In particular, it fences off some stores and processors that are not to be used by the parallel executor. It could record how busy each disk and tape is and avoid using busy devices for temporary results.

The explorer was very successful. Few people know what is in their cluster and how full or fast it is. The explorer code is operating-system specific, but the resulting database is generic. The only difficulty we had was in building the Node_Node table. First, it is difficult to discover the cluster wiring diagram. We had to use many heuristics. Second, rating the connection speed between N nodes is a $2N^2$ problem. Our solution will not scale to very large processor clusters.

3.3. Optimizer: Picks a Plan

The optimizer picks a good plan for a specific load task and generates a web describing the plan. The optimizer begins by reading the SQL table definition from the database, and the cluster configuration from the explorer's SQL database (see Figure 6). This information, combined with the user's input data definition defines the task and the constraints on the execution plan.

The optimizer's goal is to find a parallel plan that will fit on the cluster and that will give the fastest possible load time. The optimizer assumes the load task is the only job on the system, and that all the (not-fenced) devices and processors and all the not-used storage are available for the job.

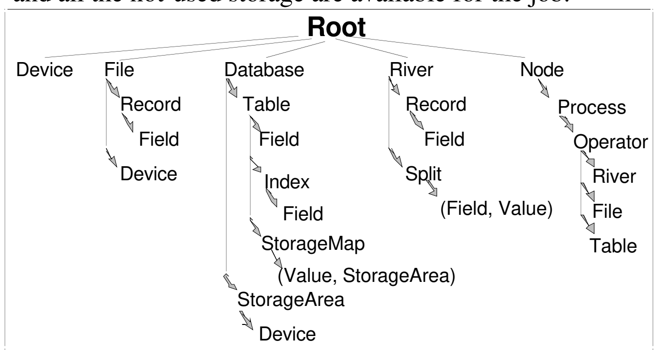

Figure 7: The web schema. The device (disk, tape,...) and node (processor, memory) data come from the explorer. Input and output file information comes from the CUI. The optimizer generates scratch file information. Each file has a record descriptor and resides on a set of devices. The database information comes from the SQL schema. Each database has a set of tables and storage areas. Each table has a set of fields, indices, and storage maps. Each storage map maps a table-range to a storage area. Each river has a record definition and a split table. Each node has a set of processes. Each process has a set of operators that read and write rivers, files and tables. The optimizer picks an appropriate phasing, process, operator, and river structure.

Even though a data load is a simple `INSERT-SELECT` statement, parallel load optimization is complex problem. There are many issues to consider, such as selecting appropriate types and numbers of operators, grouping operators into processes, partitioning work among operators, placing processes at nodes, allocating memory for operators, picking devices required by operators for scratch and log files, etc.

The load optimization problem is computationally intractable. The simpler problem of optimally assigning a set of n tasks with given CPU requirements to a set of p processors is known as the *processor-scheduling* problem [Garey & Johnson]. It is NP-Complete in the number of tasks n. The processor-scheduling problem is one component in the load optimization problem. So the load optimization problem is at least as hard. Consequently, we used a combination of analytical reasoning and heuristics to reduce the search space.

Related Work

Hong [Hong] looked at the question of optimization for a shared-memory multi-processor environment. He advocated the idea of *two phase* optimization. The first phase produces the best sequential plan for a given query without regard for parallelism. The second phase subsequently produces the best parallelization of this sequential plan. Hong presents arguments and experimental evidence to show that this is a good optimization strategy for an SMP environment.

Hasan and Motwani [Hasan, Motwani] examined the tradeoff between communication costs and parallelism. They present analytical techniques to identify *worthless parallelism* where the communication costs associated with parallelism outweigh gains from parallel execution. Their techniques eliminate many plans that are provably sub-optimal.

Optimization Strategy: Four Phases

Our optimizer is a *Cost-based Hierarchical* with four decision steps. The Optimizer enumerates various plans by making different choices at each step. The first step consists of choosing a *template* for the plan. Second, the degree of parallelism is decided for the plan. The third step does process placement, memory allocation, and device selection. The fourth step evaluates the cost of the resulting plan. The least-cost plan is ultimately chosen. We now discuss each of these steps in greater detail.

(1) Pick a template: A *template* is a blueprint for a parallel plan. It contains a high-level description of a sequential plan along with a specification, showing the dataflows, the binding of operators to processes, and showing how the plan may use partition parallelism. Templates are similar to Hong's best serial plans, except that they (1) show the process and data flow splits, (2) make parallelism explicit, and (3) the optimizer may consider multiple templates.

Analytic techniques, including those described in [Hasan, Motwani], restrict the choice of templates. Reasoning about worthless parallelism helps decide to co-locate operators in a single process. For example, if the output of each Merge operator feeds into a single corresponding InsertTable operator, co-locating these operators in a single process type is best. Operators may also have widely varying characteristics: ScanFile is extremely CPU-light while InsertTable is extremely CPU-heavy. Picking different degrees of parallelism for these two operators requires that they be in different processes. This *bottleneck* analysis prescribes cloning ratios among operators: a single fast upstream operator may be able to drive five downstream operators.

(2) Pick degree of parallelism: The second optimization step transforms a template into a partitioning plan. A partitioning plan specifies the parallelism degree of each template process. Different parallel plans are obtained by cloning each process a specified number of times. The Optimizer deduces a maximum degree of cloning for each process type. It then iterates through the search space generating plans with varying numbers of processes of each type up to the maximum. For example, if there are 100 input files to be scanned and the target table has 200 storage areas, the optimizer considers between 1 and 100 Scan processes and between 1 and 200 Sort-Insert processes. The cloning degree of the source and sink of a template river in turn imply a specific partitioning of each river. For each partitioned plan, the optimizer considers many process placements, memory allocations, and device selections for individual operators.

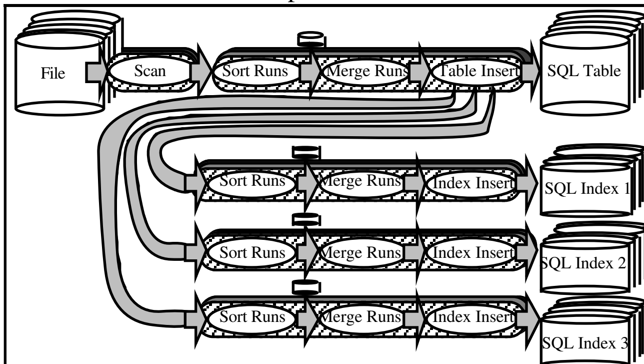

Figure 8: The template for a load plan showing operator-process bindings, rivers, and showing a two-pass sort. Each index build is done by a separate group of processes. This template also shows possible partition parallelism.

Considering every combination of the process cloning degrees, p, leads in the 100-source 200-sink case to a search space with 100 x 200 = 20,000 cases. This space may be searched more efficiently by realizing that many of these 20,000 cases are quite similar and therefore need not all be considered individually. For example, while there is a substantial difference between plans containing 1, 2, 3, 4, or 5 Scan processes, there is not much difference between having, say, 80, 81, 82, 83 or 84 InsertTable processes. The search space can be dramatically reduced by only considering partition values that result in each Scan process scanning a distinct number of files. A similar heuristic is applied for Insert processes and the number of storage areas they insert into. This leads to a reduction in the search space from $O(\Pi m_i)$ to $O(\sqrt{\prod m_i})$ where m_i is the maximum partitioning of each template process type. In the example, the heuristic reduces the search space from 20,000 cases to less than 600. This *square-root heuristic* is a special case of the idea that the optimizer need only examine plans that are significantly different.

(3) Place processes and data in cluster: The third step in the optimization process places processes at nodes, selects devices for logs and sort scratch files, and allocates memory for operators such as Sort and SQL engines. A simple heuristic chooses devices: each Sort operator's scratch files are co-located with the corresponding InsertTable's storage areas. Each InsertTable operator also co-locates its log files on the target disk. These heuristics are based on a performance analysis that indicates that scratch file IO and log IO nicely overlap with the corresponding database IO. These heuristics dramatically reduce the search space and also ensure that no one disk becomes a bottleneck.

Process placement and memory allocation decisions are relatively easy when the configuration is either a single SMP node or a shared-disk cluster of similar SMP nodes. In these cases, the configuration's symmetry enables the Optimizer to spread the work evenly across the nodes, and distribute memory evenly across the Sort operators on a node. For the template shown in Figure 8 the Optimizer would evenly distribute the Scan processes among the nodes and then evenly distribute the Insert processes. If, in addition, each Insert process receives input from a single Scan process, then reasoning about worthless parallelism directs the Optimizer to co-locate each Scan process and all its related Insert processes on the same node.

The optimization problem is computationally harder for asymmetric configurations. If different nodes have different speeds and different amounts of memory, then it is no longer straight-forward to distribute the work evenly among the nodes. Again, if particular devices are accessible only from particular nodes, the process placement and device selection steps become inter-related and more complicated. Thus, for asymmetric configurations, one either *pretends* to have symmetry and uses simple techniques, or searches the space. Given the exponential size of the search space, randomized algorithms such as *simulated annealing* seem to be the only alternative to presumed symmetry.

A shared-nothing cluster of similar nodes where each disk is "owned" by some node that servers that disk still possesses significant symmetry. The optimization strategy used for shared disk-clusters may be extended to this case by first placing all processes that have device affinity near (one of) their desired devices -- a greedy algorithm. The remaining processes are then placed in the emptiest nodes unless they in turn have an affinity to processes. The cost function discards placements that are infeasible or sub-optimal.

(4) Estimate plan cost: The first three steps produce a fully specified plan. The fourth optimization step estimates the plan's elapsed execution time. The Optimizer's goal is to find the plan with the minimum estimated elapsed time. Blocking operators divide the execution of a plan into natural, non-overlapping phases. The optimizer estimates the cost of a multi-phase plan as the sum of the phase elapsed times.

The plan's estimated elapsed time during a phase is the maximum of the elapsed times for each of the processors, devices, networks, or processes during that phase. Cost functions are associated with each operator and with each type of river (intra-process, inter-process, and inter-node). During the cost evaluation phase, it is convenient to think of rivers as operators. Cost evaluation proceeds in a source-to-sink fashion. The cost functions for the operators and rivers are used to calculate the elapsed times for the processors, devices networks, and processes.

Operator and river cost functions are multi-dimensional: they specify the processing, I/O, memory, and communications cost to process a data unit. The cost or an operator is obtained as a maximum of a number of terms. Special note is taken of any asynchronous I/O and asynchronous network transmission costs in estimating elapsed times -- these times overlap with execution and so the cost is the maximum of the three, rather than the sum. Some operators such as Sort have a discontinuous costs (one pass or two). Available memory and disk space are modeled as constraints. These constraints often eliminate plans involving one-pass sorting.

Complex cost functions make global analytical reasoning extremely hard. They force an enumeration-evaluation approach to optimization. Analysis reduces the number of plans that are evaluated.

The combined use of templates, the square-root reduction, and the placement heuristics, works well. Optimization for SMP and symmetric shared-disk clusters was quick. Planning a load for a 4-node (24-processor SMP), 400 storage area (disk), 24 input file (tape) system involved generating and evaluating about 400 plans. It took less than a second to pick a plan.

4. Parallel Execution

4.1. Execution Environment: Overview

Webs are executed by a collection of *executor* processes spread among nodes of the cluster. Each node has a *web manager* that creates the executors at that node and performs node-wide services for the web. The web manager allocates a node-wide shared memory segment, manages intra-node communication, coordinates startup, phasing, checkpointing, and shutdown, and monitors performance.

The web assigns each executor a set of operators to execute. Each executor can be thought of as a multi-threaded process: one thread for each operator in the process. Executors all run the same program. That program locates the executor's part of the web and initializes the rivers and operators specified by the web for that process. Startup is interpretive, but after a second or two the web is initialized and the operators execute at the raw machine speed.

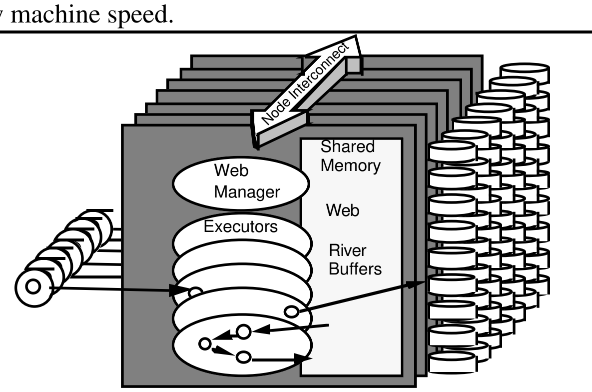

Figure 9: The parallel execution environment. Each node has a web manager. The web describes a set of operators and their bindings to processes. The web manager examines the web and creates a shared memory region containing the web and the river/steam buffers for communication among operators and processes at that node. It creates the executor processes that perform the web operations. Operators communicate via rivers. Inter-node communication is via sessions tcp/ip streams among web mangers.

Each operator is programmed as a sequential operation with three phases: (1) initialization, (2) execution, and (3) termination. The initialization phase opens the operator's input and output rivers, opens or creates input and output files, and attaches to the database if necessary. The execution phase reads the input rivers or files, operates on the records, and produces a dataflow stream that flows to an output river, file, or table. When the input streams dry up (when all records have been processed), the operator terminates by closing the input and output rivers. This simple model is complicated by checkpointing, error handling, and transaction commit.

Rivers carry dataflows among operators. Rivers within a node just pass data via shared memory. This communication is especially fast within a process, because user-level threads dispatch very quickly and because there are often no splits or merges within a process. Flows among processes at a node (an SMP) may involve splits and merges but still avoid extra memory copies by using shared memory. Intra-node rivers use operating system process waits and dispatches that add overhead. Flows among nodes involve tcp/ip and are much more expensive. The relative costs of these three kinds of flows are 1:2:40 in the no-copy case and 1:2:12 in the one-data-copy case where either producer or consumer must move the data in memory. We expected to replace tcp/ip with a much more efficient cluster-communication protocol based on reflective memory.

4.2 Startup: Processes, Rivers, Operators

After the optimizer picks a plan and records it as a web file, the client process forks a web manager on the local node. This first web manager, called the *root*, coordinates the web's execution. The root web manager reads the web file and forks a web manager for each other cluster node used by the web. The fork passes the web file name and the root's tcp/ip socket number. Each subsidiary web manger reads the web, gets a tcp/ip address and sets up a communication session to the root. The root collects these socket names and broadcasts the resulting directory. Now, each web manager knows the addresses of all others and can contact the ones it needs to talk to. One web manager needs to talk with another if the web specifies a dataflow between execution processes in their two nodes.

At the same time, each web manager allocates and initializes a shared memory area at the node. This area holds the web and the river buffers for all operators at the node. The segment also holds the performance meters and other node-wide information. The web manager then forks that node's local execution processes as specified by the first phase of the web.

Each executor first attaches to the shared memory segment. It reads its part of the web from shared memory. Based on this it, initializes the rivers and then initializes each operator bound to the process. It then begins operator execution. When all it's operators have completed, the executor process notifies the web manager that the phase is complete. If this its last phase, the executor terminates. When all phases are complete, the web managers signal the root and a job completion file is written with summary statistics.

4.3 The River System - Startup and Execution

The river system is a key part of the design. Initially, we considered using a tcp/ip session for each stream among processes. This idea was short lived for two reasons:

Polynomial Explosion: A thousand scanners feeding a thousand sort-merge-insert operators would require a million tcp/ip sessions. We had to do something to cut down this polynomial explosion.

tcp/ip Performance: The tcp/ip implementation we used was expensive. It cost six million instructions to transfer one megabyte of data within the node and eight million instructions to transfer a megabyte between two nodes. Writing data to a shared disk and reading it back is ten times faster than using tcp/ip. The cpu cost of a message is approximately $f + m \times b$ where f is the fixed cost, m is the per-byte cost, and b is the message size in bytes. For memory-to-memory (same-node) requests, f is about 3,000 instructions and m is about 6 instructions per byte. For LAN transfers, f is approximately 3,000 instructions and m is approximately 8 instructions per byte.

Performance problems with tcp/ip are legendary. A one-instruction-per-bit-sent is typical of commercial LAN/WAN communications protocol stacks. It makes them un-usable for dataflow computing. Our solution to these problems was to use tcp/ip as little as possible and to look forward to the day that we can eliminate it. We did the following:

RIO: Create a new communications protocol that lends itself to fast implementations. The protocol allows operators to exchange data streams with no extra memory copies. This protocol, called RIO (for river IO), maps to a memory-to-memory protocol (MIO).

MIO (Memory-to-memory streams): MIO is used for communication within an SMP. Eventually, MIO can be extended to distributed memory and reflective memory hardware clusters [DASH]. MIO uses SIO for off-node communication.

SIO (stream IO on the LAN): SIO is for node-to-node communication based on tcp/ip until it can be replaced with a standard high-performance cluster protocol.

Multiplex Sessions on Web Manager tcp/ip sessions: We ameliorated the polynomial-explosion problem by only opening tcp/ip sessions among web managers. The tcp/ip session between the web managers at two nodes multiplexes all traffic between operators at those two nodes. This cuts the polynomial explosion from a million to less than five thousand sessions in a hundred-node cluster. A three-level multiplexing scheme would be needed to cut the polynomial explosion for massive clusters (thousands of nodes).

MIO-SIO is a uni-directional session-oriented protocol involving `open()`, `get_buffer()`, `send_buffer()`, and `close()` routines. There is only one extra call: `free_buffer()` that indicates to MIO-SIO that the buffer has been consumed. The semantics of MIO-SIO `send_buffer()` and `get_buffer()` are unusual. Once a buffer is sent, the producer can no longer access it. `get_buffer()`, when invoked by a producer returns an empty buffer, while `get_buffer()` invoked by a consumer gets a full buffer. Figure 10 illustrates many of these ideas.

4.4 The RIO Protocol and Protocol stack

MIO-SIO flow control is simple: Each stream has a budget of two or three buffers. When a producer exceeds its budget, its `get_buffer()` requests stall. If a downstream operator stalls, all upstream operators will stall until the consumer catches up. A consumer process may have wait for input and a producer may have to wait for a consumer to free a buffer. On the VMS operating system, process waits and wakeups were implemented with mailboxes and asynchronous system traps (ASTs.) Threads and ASTs also exist on NT. On UNIX, threads and the event-signal mechanism would be used.

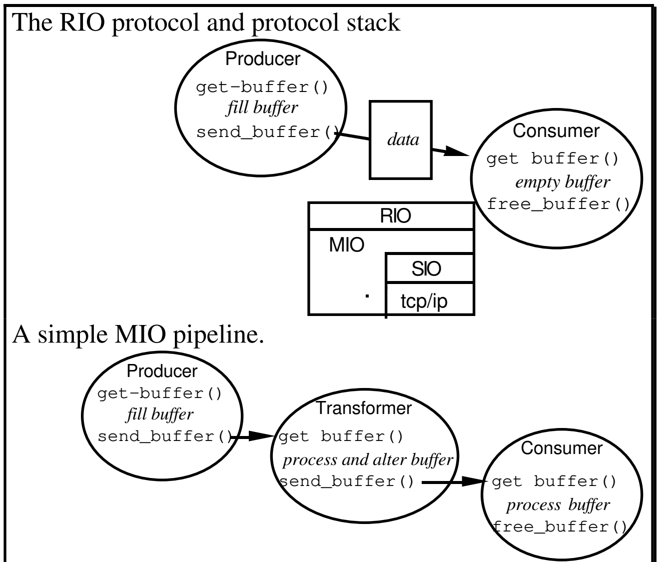

Figure 10. The RIO interface is built on a memory-to-memory transport (MIO) for communication within a process or SMP. MIO uses a stream interface (SIO) for node-to-node communication. SIO buffers flows via MIO to the local web manager that then uses tcp/ip to flow the data to the destination node web manager (see Figure 11.) The MIO-SIO protocol is suited for a zero-copy memory-to-memory implementation. A producer gets a buffer from MIO-SIO and fills it. When the buffer is full, the producer sends the buffer. The consumer gets a buffer, empties it, and then frees it. If the two processes share memory, there are no extra memory copies. The diagram at the bottom left shows a pipeline using the MIO-SIO interface. Rather than freeing the buffer, the middle process modifies it and passes it down stream. This is useful for data validation and data conversion operations. River merge operators also work in this pipeline fashion. The river system, RIO, implements a record-at-a-time interface on top of SIO and MIO. It also manages steam merges and splits.

Multiplexing data streams on top of a few tcp/ip sessions creates a three-hop inter-node communications protocol. When an operator at one node wants to send data to an operator at another node, the data first flows to the local web manager (via MIO), then node-to-node (via tcp/ip), and then from the destination web manager to the destination operator (via MIO).

This scheme is practical because it costs about 12,000 instructions to send a 32KB buffer through MIO, while it costs over 250,000 instructions to send it via tcp/ip. So the two extra MIO hops add less than 10% to the overhead.

RIO is a record-at-a-time interface programmed atop MIO-SIO. RIO also implements the river system's record split and merge operators. The call interface to rivers is `open_river()`, `get_record()`, `put_record()`, `close_river()`. In addition, a bulk interface allows an operator to deal with the records in a buffer as a batch.

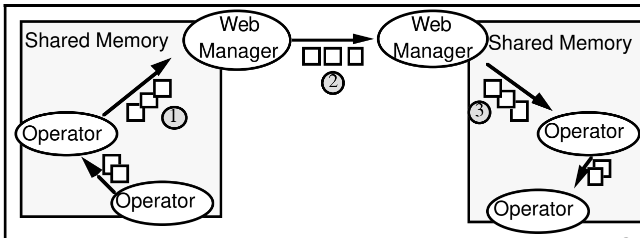

Figure 11: The three-hop message protocol used by MIO-SIO to reduce the polynomial explosion. LAN streams and rivers travel via MIO (1) to the local web manager that then multiplexes them over a tcp/ip sessions SIO (2) to other web managers. MIO is used at the destination to deliver the data to the destination operator. Since MIO is a zero-copy message protocol, it adds less than 10% to the message cost. A three-level multiplexing scheme would be needed for clusters much larger than 100 nodes (600 processors).

The `get_record()`,`put_record()` requests are pointer oriented so that no extra data copies are needed -- ideally, they can pipeline data through the operators without extra data copies. Only when a record is transformed, or split into one of several data buffers is the record actually copied. This means that some operators get the zero-copy performance indicated in Table 12. We believe the MIO fixed cost could be reduced by a factor of three by more careful programming -- it should be less than the tcp/ip fixed cost.

Table 12: Cost of sending 1MB in 32KB chunks from operator to operator (measured in thousands of instructions.) Operators attempt to minimize data copies within the RIO system, so both zero-copy and 1-copy time are relevant.

transfer method	fixed cost / 32KB (k ins)	per byte cost (ins)	total cost/MB (k ins)
read/write disk	5	0.1	250
intra-process MIO, 0- copy	7	0.0	200
intra-process MIO, 1-copy	7	0.5	700
inter-process MIO, 0-copy	12	0.0	400
inter-process MIO, 1-copy	12	0.7	1,100
local-tcp/ip	4	6.0	6,000
LAN tcp/ip	5	8.0	8,150
LAN RIO 0-copy	29	8.0	9,000
LAN RIO 1-copy	29	8.7	9,700
LAN RIO 2-copy	29	9.4	10,400

4.5. Executor and Operator Structure

We considered having just one process at each node and using our own thread mechanism to execute the operators at that node. This approach would have reduced operating system overheads but has several drawbacks. First, the database system we were using is not thread-safe. That is, it lacks an asynchronous call interface, and it associates transactions with the process rather than with the thread. This is a typical problem with database systems. So each database operator needs to have a separate operating systems task. Second, we find that many thread packages are not much faster than the OpenVMS process mechanism. Using operating system processes, exploits multiprocessing and has little cost on VMS. In essence,

VMS processes are our threads and the global shared memory is our one-process address space.
All the papers we have read on parallel database systems have focused on the query problem. In queries, data is pulled from the database into the application. The application calls for a set of records, and this translates to calls to upstream data sources. This is a *pull architecture* where downstream operators pull (call for) data from upstream operators. Pull works very well if there is a single data sink. As diagrammed in Figure 8, data loading, does not have a single data sink. That diagram shows four data sinks, the base table and three index tables. In multiple-sink dataflows, a push architecture with flow control is needed. That is, each operator works independently and attempts to keep its output buffers full. Each operator pulls from its inputs and pushes to its outputs. The operator blocks when there are no inputs, or when the two or three output buffers are full.
Orthogonal to the data flow, it is occasionally necessary to commit a transaction or checkpoint the current state of the web. These issues require that the operator thread poll for such events or that it provide a callback to perform such actions.
These issues led us to implement a simple thread package for operator execution. Each operator is implemented as a set of entry points:

op_start(): Initialize the operator, allocating storage, opening rivers, tables, and files.

op_do(): Read a buffer-load (about 32KB) of data from the upstream rivers, process it, and generate target data.

op_end(): Flush any internal data downstream and close all rivers.

op_commit(): Prepare to commit all work (see the next section)

op_checkpoint(): A checkpoint is being taken, record any data you will need at restart (see the next section)

All operators follow this template. We implemented operators to read and write files, tables, and tapes. We also implemented sort and hash operators. Having implemented one operator, it was relatively easy to implement others. We are confident that sophisticated users can copy these templates and construct other operators such as hash-join, aggregates, and cross-tabs.

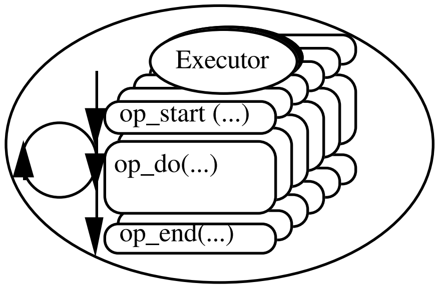

Figure 13: The flow within an executor process is to call the `op_start()` entry of each operator belonging. Then the executor calls any `op_do()` operation that has an input buffer. When all operations return a completion code, the executor calls `op_end()` for all operations and then terminates the process. Each `op_do()` step processes one or more input buffers and passes them downstream to the next river and operators.

4.6. Transactions and Checkpoint/Restart

Thus far, we have ignored errors. What if a record fails an integrity check, or a unique-key check, or a referential-integrity check? What happens if a transaction cannot commit? What happens if a process or processor fails? These are all difficult questions that have no easy answers. Solving these problems is a fundamental part of any parallel database system design.
Bad records are the simplest problem. If a record contains data that violates an integrity constraint (e.g., `birth_year between 1850 and CURRENT`), then the record can be sent to an error river (called the *sewer*) along with a diagnostic. These rejected records can be handled later. Both the data conversion and the database insert code may discover bad records. For incremental loads, integrity checks are enforced at each insert or are deferred to commit, but they are checked almost immediately. This makes error handling relatively straightforward. The details of handling bad records are much more complex for batch loads that create indices and check referential integrity after the base table has been loaded. The techniques are still fairly obvious. If a record violates a constraint, it is deleted and sent to the sewer.
How does dataflow interact with transactions? The simple answer is: "Not well." There are no obvious transaction boundaries in the dataflow beyond the whole transaction -- that is an old-master new-master batch model. Unfortunately, to allow incremental loads of databases, no part of the database should be locked for more than a few seconds. This means that incremental loads must commit every few seconds.
In the Rdb/VMS system, transactions are associated with processes. Using the distributed transaction mechanism, it is possible to associate a transaction with several processes. This has relatively high cost, so we adopted the following simpler approach. Every few thousand insert operations (every few seconds), an `InsertTable` process commits. It does this by calling the op_commit() entry point of each operator to allow it to flush any buffers and make any integrity checks it needs to make. In particular, this causes any deferred index updates to be sorted, and performed in batch. Once each operator has signaled willingness to commit, the process commits. (This requires that all corresponding index inserts of this process must be performed by this process).
If the commit is successful, fine. If the commit fails, then there is a problem - some record violated an integrity constraint. Now, the `InsertTable` process must re-insert every record of the failed transaction, carefully inserting each record and sending bad records to the sewer. This requires that the insert-table operator keep a copy of each such record. This can be done by retaining the input river buffers and committing when a certain number of buffers have accumulated. The InsertTable operator has a callback to either discard the input buffers (successful commit) or to carefully reinsert the records and then commit (failed commit case).
We implemented this simple transaction mechanism. We designed but did not implement checkpoint-restart mechanisms. Performance numbers reported here include the transaction overhead for UNDO-REDO logging needed to make incremental loads work.

Process and processor failure are much more difficult problems. What if a hundred processors and a thousand disks have been working for a day, and some one of them fails? Much work would be lost by restarting from scratch. If the load is incremental, then UNDO-REDO logging is operating and the database is consistent. The only problem is that some records may be lost (not inserted in the database.) If the process were recreated and its input data streams were reset to that point, then the process could re-execute the needed data processing.

Checkpoint-restart is the standard technique for masking failures of long-running batch operations. The root web manager initiates a checkpoint approximately every hour. In this case, a failure will lose and redo only a half hour of work on average.

Checkpoints and restarts are coordinated by the root web manager. It first requests all web managers to quiesce all their executors. Each executor calls its operators asking them to prepare to checkpoint. The idea is that data sources stop reading records, data consumers empty their input rivers, flush any internal sate, and fill their output buffers. Eventually, all executors are in their outer loops, all transactions are committed and most rivers are empty. Now each operator is called to checkpoint its state in a node-local checkpoint file. In addition all rivers checkpoint their non-empty buffers and their current stream sequence numbers. When this is complete, the web starts the dataflow again.

If a node fails, each node restarts from the most recent successful checkpoint. The operators and rivers are reinstantiated, the computation proceeds from that point forward.

Much of this technology was pioneered in the 1960's by batch processing systems and is recorded in checkpoint-restart manuals from that era. We found two novel issues.

Blocking Operations. Most operations (like scan, make-key, and insert) have almost no state. Certain operations like aggregates, join, sort, and merge have a great deal of state and have scratch files. They must record this state at each checkpoint. Even when they complete, they cannot delete their state until the next checkpoint completes (this suggests that phase changes should trigger checkpoints). This is similar to triggering a checkpoint each time a tape ends.

Idempotence: It is important that records not be inserted twice. Restarting at an early point in the data stream might place a record in the database twice. The idempotence problem is fairly easily solved: when a transaction commit, it writes a "high-water-mark" in the database, recording the identifier or sequence number of the highest inserted record. At restart, the insert operator reads this high-water-mark, and discards all records prior to that as they appear in the input stream. Only records beyond the high water mark are inserted in the input stream. This is called the MiniBatch technique in Gray and Reuter [Chapter 5].

Determinism: A second execution forward from a given state may not generate records in the same order. Records coming from a sequential device (tape or disk) and records coming from a transformer will be in the same order on re-execution, but records coming from a river-merge operation may be permuted if the buffers arrive from the sources in different order. If a river merges data from sources A and B, the first execution might present a pair of record buffers in the order A1 A2 A3 B1 B2 B3, while a second execution might present them in the order A1, B1, A2, B2, A3, B3, or any other of the sixteen possible permutations that preserve the order of the A and B sub-sequences. We have found only two solutions to this problem. (1) If the operator is order insensitive, that is the operator's output is independent of the input order, then one can ignore this problem. Sort and aggregate operators have this property. (2) If the operator is order-sensitive (e.g., Insert), then it must insist on a deterministic ordering of the input buffers so that the river merge operation will produce the same record steam independent of the timing of the buffers arriving from multiple data sources. Buffer sequence numbers are the standard way to manage this logic.

Fault-tolerance for these long-running batch operations seems ripe for rediscovery of old techniques, and invention of some new ones. We did not implement our ideas, and so cannot tell how successful they would have been.

5. Performance Measurements

Recall that our goal was to load a Wisconsin terabyte in a day using a hundred-processor thousand-disk cluster. Most of our development and performance results were done on a much more modest one-node, two-processor, eight-disk system in our laboratory. We were unable to get access to a large cluster before the project ended. We had occasional access to a 24-processor 400-disk cluster.

The hardware was a DEC System 4000 -620 which is a dual-processor 150Mhz Alpha RISC processor. Each processor is rated at 100SPECints, but SPECint is a cache-local benchmark. Measurements of memory intensive applications typical of database systems indicate the machine executes about 40 million instructions per second. It had 256MB of main memory and eight 2 MB/s SCSI discs on four controllers.

We tested the load of a million-record hash-structured base table with no secondary indices. The source data was stored on separate disk files. All loads were done as incremental loads with undo logging and locking enabled. Transactions committed every 50,000 inserts. Turning transactions off cuts the IO and cpu cost in half.

The Rdb load utility is the base case. It is a single-process that reads an input source and inserts data into the target table. This single process (1) scans the input, (2) formats it as a database record, (3) constructs the hash key for each record, (4) sorts the records into hash order, (4) inserts the records in hash order, and (5) commits occasionally.

Next a single-process web was measured. It emulates the loader but uses a batch-insert interface to Rdb insert developed for us by the Rdb team. This simple change, allowing InsertTable() to insert 100-records at a time rather than making a separate call per insert reduced cpu time by 40%. In fact, we found a 20:6:2:1 ratio between dynamic SQL (PREPARE-EXECUTE the INSERT statement), static SQL, (EXEC SQL INSERT...), a low level. (DSRI) interface, and this vector-oriented interface. The Rdb load utility is now converted to this vector interface.

Table 14: Elapsed and cpu time to load a million records from file to SQL table using the Rdb load utility, a 1-process web, a simple pipeline, and then a 2-way and 5-way cloning of the sort-insert process. All tests were run on a dual-155Mhz SMP DEC Alpha 4000 system.

Form of Load	elapsed (sec)	cpu (sec)
Rdb Load	1785	1097
1 Process web	1217	685
1 Scan 1 Sort-Insert	1205	685
1 Scan 2 Sort-Insert	889	717
1 Scan 5 Sort-Insert	565	766

Next pipeline parallelism overlaps input scanning the with sorting. A parallel web of a scan process feeding a hash-sort-merge-insert process had almost no effect on either elapsed time or cpu time. That is because the hash-sort-merge-insert process is IO bound. This incidentally shows that pipeline parallelism is often not a giant improvement.

The next two experiments cloned the hash-sort-merge-insert process two and then five ways. These experiments show that the cpu time rises slightly (about 10%) with the parallel execution and consequent extra scheduling, but that the elapsed time drops by 25% and then 54%. The 1x5 case has only 70% cpu utilized. With more discs, the system could have been configured as a 1x8 web. We believe this web would have a 15% cpu increase but 4x reduction in elapsed time to less than 400 seconds. This would correspond to a 2500 record/second load rate for this node. The index load time is a second phase that runs about four times faster than the initial load since the data volume is four times less.

By using six-way 200 MHz Alpha processors, and four times as many disks, the load rate should reach 10,000 records per second. That translates to 173 GB/day. By extrapolating from Table 13 and 15, one estimates that a hundred processor thousand-disk system structured as sixteen SMP nodes would be able to load 2.4 TB per day. The traffic in and out of each node would be a modest 2MB/s which is well within FDDI and ATM bandwidths.

These extrapolations assume that there are no bottlenecks in the design. We analyzed the design and believe that no bottlenecks exist. We were able to try some very simple webs in a small cluster, and did not see any bottlenecks. Unfortunately, the project ended before we could try the software on a really large cluster.

6. Summary and Conclusions

Dataflow parallelism is the most promising approach to parallelize database operations. The prototype we built automates much of the parallel database loading task. An explorer discovers the cluster configuration, and an optimizer picks the minimum-plan to perform the load. Physical database design and placement remain to be done.

Our system execution model was novel. There is a web execution monitor at each node of the cluster. It sets up a collection of processes executing at that node. Operators within a node can use a high-performance zero-copy memory-to-memory stream transport. Streams among nodes are multiplexed over tcp/ip sessions to reduce the polynomial explosion.

Connecting transactions with the dataflow web allows incremental and online loads, but raises interesting issues related to transaction failure. Adding checkpoint-restart to database dataflows is an area we designed but did not implement. It will be interesting to see how these ideas work in the product.

In the end, the parallel data load utility demonstrated good speedup due to parallelism. Unfortunately, staff and budget constraints prevented us from testing the prototype on a large cluster. Despite that, we believe the approach to be sound and believe that these ideas will be widely used in database future products.

7. Acknowledgments

The Rdb team, especially Louis Dimino, Jay Feenan, Steve Hagan, Paul Mackin, Rabah Mediouni, Ian Smith and Peter Spiro helped us with advice, encouragement, code, and equipment. They were a key part of this effort, and are now moving the prototype into their product.

8. References.

[DASH] M. Heinrich, et. al., "The Performance Impact and Flexibility of the Stanford FLASH Multiprocessor, " 6th ASPLOS, Oct. 1994.

[DeWitt 1] D. DeWitt, et. al., "GAMMA - A High Performance Dataflow Database Machine", Proc. 12th VLDB, Chicago, Sept. 1986.

[DeWitt 2] D. DeWitt, "The Wisconsin Benchmark, Past, Present, and Future", in *The Benchmark Handbook for Database and Transaction Processing Systems*. 2nd ed., Morgan Kaufmann, San Mateo 1993.

[Englert] S. Englert, "Performance Benefits of Parallel Query Execution and Mixed Workload Support in NonStop SQL Release 2", Tandem Systems Review, V.6.2, Oct 1990, pp. 12-23.

[Garey & Johnson] M.R. Garey, D.S. Johnson, *Computers and Intractability,* W.H. Freeman, 1979.

[Graefe] Graefe, G., "Query Evaluation Techniques for Large Databases," ACM Computing Surveys, **V. 25**.2, pp: 73-170, June, 1993.

[Gray & Reuter] J. Gray, A. Reuter, *Transaction Processing Concepts and Techniques*. Morgan Kaufmann, San Mateo, 1992.

[Hasan, Motwani] W. Hasan and R. Motwani, "Optimization Algorithms for Exploiting the Parallelism-Communication Tradeoff in Pipelined Parallelism," Proc. 20th VLDB, Santiago, pp. 36-47 Sept. 1994.

[Hong] W. Hong, *Parallel Query Processing Using Shared Memory Multiprocessors and Disk Arrays,* Ph.D. Thesis, U.C. Berkeley, 1992.

[Kitsuregawa 1] M. Kitsuregawa, H. Tanaka, T. Moto-ka, "Application of Hash to Database Machine and Its application," *New Generation Computing*, 1,1 pp.63-74, Springer Verlag, 1983

[Kitsuregawa 2] M. Kitsuregawa, Yasushi Ogawa, "Bucket Spreading Parallel Hash : A New Robust Parallel Hash Join Method for Data Skew in the Super Database Computer(SDC)." Proc. 16th. VLDB, pp.59-70. 1990.

[Serlin] O. Serlin, "The History of the TPC", in *The Benchmark Handbook for Database and Transaction Processing Systems*. 2nd ed., Morgan Kaufmann Publishers, San Mateo 1993.

[Teradata] *Teradata DBS Concepts and Facilities for the NCR System 3600*, AT&T GIS, Dayton Ohio, Jan 1994.