PARALLEL DATABASE TECHNOLOGY

Kien A. Hua

School of Computer Science
University of Central Florida
Orlando, FL 32816-2362
Topics

**PROCESS STRUCTURE**

<table>
<thead>
<tr>
<th>QUERY OPTIMIZATION (4)</th>
<th>* Parallelizing Query Optimization Techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXECUTOR (3)</td>
<td>* Parallel Algorithms</td>
</tr>
<tr>
<td>STORAGE MANAGER (2)</td>
<td>* Data Placement Strategies</td>
</tr>
<tr>
<td>HARDWARE (1)</td>
<td>* Parallel Architectures</td>
</tr>
</tbody>
</table>

**COMMERCIAL SYSTEMS (5)**
A database structure is a collection of *tables*.

Each table is organized into rows and columns.

The persistent objects of an application are captured in these tables.
Relational Operator: **SCAN**

\[
\begin{align*}
\text{SELECT} & \quad \text{NAME} \\
\text{FROM} & \quad \text{EMPLOYEE} \\
\text{WHERE} & \quad \text{ADDR} = \text{"Orlando"};
\end{align*}
\]

<table>
<thead>
<tr>
<th>EMP#</th>
<th>NAME</th>
<th>ADDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0002</td>
<td>Jane Doe</td>
<td>Orlando</td>
</tr>
<tr>
<td>0005</td>
<td>John Smith</td>
<td>Orlando</td>
</tr>
</tbody>
</table>

\[
\begin{array}{|c|c|c|}
\hline
\text{EMP#} & \text{NAME} & \text{ADDR} \\
\hline
0002 & Jane Doe & Orlando \\
0005 & John Smith & Orlando \\
\hline
\end{array}
\]

\[
\begin{align*}
&\text{SELECT (ADDR=Orlando)} \\
&\text{SCAN} \\
&\text{PROJECT(NAME)}
\end{align*}
\]

<table>
<thead>
<tr>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jane Doe</td>
</tr>
<tr>
<td>John Smith</td>
</tr>
</tbody>
</table>
Relational Operator: JOIN

SELECT * 
FROM EMPLOYEE, PROJECT 
WHERE EMP# = ENUM 

EMPLOYEE:

<table>
<thead>
<tr>
<th>EMP#</th>
<th>NAME</th>
<th>ADDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0002</td>
<td>Jane Doe</td>
<td>Orlando</td>
</tr>
</tbody>
</table>

PROJECT:

<table>
<thead>
<tr>
<th>ENUM</th>
<th>PROJECT</th>
<th>DEPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0002</td>
<td>Database</td>
<td>Research</td>
</tr>
<tr>
<td>0002</td>
<td>GUI</td>
<td>Research</td>
</tr>
</tbody>
</table>

EMP_PROJ:

<table>
<thead>
<tr>
<th>EMP#</th>
<th>NAME</th>
<th>ADDR</th>
<th>PROJECT</th>
<th>DEPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0002</td>
<td>Jane Doe</td>
<td>Orlando</td>
<td>Database</td>
<td>Research</td>
</tr>
<tr>
<td>0002</td>
<td>Jane Doe</td>
<td>Orlando</td>
<td>GUI</td>
<td>Research</td>
</tr>
</tbody>
</table>
Hash-Based Join

**Examples:**

- $0 \mod 4 = 0$
- $4 \mod 4 = 0$
- $1 \mod 4 = 1$
- $5 \mod 4 = 1$
- $2 \mod 4 = 2$
- $6 \mod 4 = 2$
- $3 \mod 4 = 3$
- $7 \mod 4 = 3$
Bucket Sizes and I/O Costs

Bucket B does not fit in the memory in its entirety. Bucket A must be loaded several times.

Bucket B(1) fits in the memory. A(1) needs to be loaded only once.
Speedup and Scaleup

The ideal parallel systems demonstrate two key properties:

1. **Linear Speedup**:

   \[
   \text{Speedup} = \frac{\text{small\_system\_elapsed\_time}}{\text{big\_system\_elapsed\_time}}
   \]

   **Linear Speedup**: Twice as much hardware can perform the task in half the elapsed time (i.e., speedup = number of processors.)

2. **Linear Scaleup**:

   \[
   \text{Scaleup} = \frac{\text{small\_system\_elapsed\_time\_on\_small\_problem}}{\text{big\_system\_elapsed\_time\_on\_big\_problem}}
   \]

   **Linear Scaleup**: Twice as much hardware can perform twice as large a task in the same elapsed time (i.e., scaleup = 1.)
Barriers to Parallelism

- **Startup:**

  The time needed to start a parallel operation (thread creation/connection overhead) may dominate the actual computation time.

- **Interference:**

  When accessing shared resources, each new process slows down the others (hot spot problem).

- **Skew:**

  The response time of a set of parallel processes is the time of the slowest one.
The Challenge

- The ideal database machine has:

  1. a single infinitely fast processor,
  2. an infinitely large memory with infinite bandwidth

  \[\rightarrow\] \textit{Unfortunately, technology is not delivering such machines!}

- The challenge is:

  1. to build an infinitely fast processor out of infinitely many processors of finite speed, and
  2. to build an infinitely large memory with infinitely many storage units of finite speed.
Performance of Hardware Components

- **Processor:**
  - Density increases by 25% per year.
  - Speed doubles in three years.

- **Memory:**
  - Density increases by 60% per year.
  - Cycle time decreases by $\frac{1}{3}$ in ten years.

- **Disk:**
  - Density increases by 25% per year.
  - Cycle time decreases by $\frac{1}{3}$ in ten years.

**The Database Problem:** The I/O bottleneck will worsen.
Hardware Architectures

(a) Shared Nothing (SN)

(b) Shared Disk (SD)

(c) Shared Everything (SE)

Shared Nothing is more scalable for very large database systems
Shared-Everything Systems
Shared Disk Architecture

Communication Network

Cross Interrogation for a small change to a page. Processing units interfere each other even they work on different records of the same page.
Hybrid Architecture

- SE clusters are interconnected through a communication network to form an SN structure at the inter-cluster level.

- This approach minimizes the communication overhead associated with the SN structure, and yet each cluster size is kept small within the limitation of the local memory and I/O bandwidth.

- Examples of this architecture include Sequent computers, NCR 5100M and Bull PowerCluster.

- Some of the DBMSs designed for this structure are the Teradata Database System for the NCR WorldMark 5100 computer, Sybase MPP, Informix Online Extended Parallel Server.
Parallelism in Relational Data Model

- **Pipeline Parallelism**: If one operator sends its output to another, the two operators can execute in parallel.

\[
\text{INSERT INTO C} \\
\text{SELECT *} \\
\text{FROM A, B} \\
\text{WHERE A.x = B.y;}
\]

- **Partitioned Parallelism**: By taking the large relational operators and partitioning their inputs and outputs, it is possible to turn one big job into many concurrent independent little ones.

\[
\text{INSERT INTO C} \\
\text{INSERT} \\
\text{JOIN} \\
\text{SCAN} \\
\text{SCAN} \\
\text{A} \\
\text{B}
\]

\[
\text{INSERT} \\
\text{INSERT} \\
\text{INSERT} \\
\text{JOIN} \\
\text{SCAN} \\
\text{SCAN} \\
\text{A0} \\
\text{A1} \\
\text{A2} \\
\text{B0} \\
\text{B1}
\]
Merge & Split Operators

- **Merge operator** combines several parallel data streams into a simple sequential stream.
- **Split operator** is used to partition or replicate the stream of tuples.
- With split and merge operators, a web of simple sequential dataflow nodes can be connected to form a parallel execution plan.
Data Partitioning Strategies

Data partitioning is the key to partitioned execution:

- **Round-Robin**: maps the $i$th tuple to disk $i \mod n$.
- **Hash Partitioning**: maps each tuple to a disk location based on a hash function.
- **Range Partitioning**: maps contiguous attribute ranges of a relation to various disks.
Comparing Data Partitioning Strategies

- **Round Robin Partitioning:**
  
  **Advantage:** *simple*
  
  **Disadvantage:** *It does not support associative search.*

- **Hash Partitioning:**
  
  **Advantage:** *Associative access to the tuples with a specific attribute value can be directed to a single disk.*
  
  **Disadvantage:** *It tends to randomize data rather than cluster it.*

- **Range Partitioning:**
  
  **Advantage:** *It is good for associative search and clustering data.*
  
  **Disadvantage:** *It risks execution skew in which all the execution occurs in one partition.*
Horizontal Data Partitioning

Query 1: Retrieve the names of students who have a GPA better than 2.0.

⇒ Only P2 and P3 can participate.

Query 2: Retrieve the names of students who major in Anthropology.

⇒ The whole file must be searched.
Multidimensional Data Partitioning

Advantages:

- **Degree of parallelism is maximized** (i.e., using as many processing nodes as possible)
- **Search space is minimized** (i.e., searching only relevant data blocks)
Query Types

**Query Shape:** The shape of the data subspace accessed by a range query.

**Square Query:** The query shape is a square.

**Row Query:** The query shape is a rectangle containing a number of rows.

**Column Query:** The query shape is a rectangle containing a number of columns.
Optimality

• A data allocation strategy is **usage optimal** with respect to a query type if the execution of these queries can always use all the PNs available in the system.

• A data allocation strategy is **balance optimal** with respect to a query type if the execution of these queries always results in a balance workload for all the PNs involved.

• A data allocation strategy is **optimal** with respect to a query type if it is usage optimal and balance optimal with respect to this query type.
Coordinate Modulo Declustering (CMD)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

**Advantages:** Optimal for row and column queries.

**Disadvantages:** Poor for square queries.
Hilbert Curve Allocation Method (HCAM)

Advantages: Good for square range queries.

Disadvantages: Poor for row and column queries.
General Multidimensional Data Allocation (GMDA)

<table>
<thead>
<tr>
<th>Row 0</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 1</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Row 2</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Row 3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>Row 4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Row 5</td>
<td>7</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>Row 6</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Row 7</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Row 8</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

Check row

\[ N \] is the number of processing nodes

**Regular Rows:** Circular left shift \( \lfloor \sqrt{N} \rfloor \) positions.

**Check Rows:** Circular left shift \( \lfloor \sqrt{N} \rfloor + 1 \) positions.

**Advantages:** optimal for row, column, and small square range queries \((|Q| < \lfloor \sqrt{N} \rfloor^2)\).
Handling 3-Dimensional Case

A cube with $N^3$ grid blocks can be seen as $N$ 2-dimensional planes stacked up in the third dimension.

**Algorithm 3D_GMDA:**

1. We compute the mapping for the first rows of all the 2-dimensional planes by considering these rows collectively as forming a plane along the third dimension. We apply the 2D_GMDA algorithm to this plane, except that the shifting distance is set to $[\sqrt[3]{N}]^2$.

2. We apply the 2D_GMDA algorithm to each of the 2-dimensional planes using the shifting distance $[\sqrt[3]{N}]$ and the first row already computed in Step 1.
Handling Higher Dimensions: Mapping Function

A grid block \((X_1, X_2, \ldots, X_d)\) is assigned to PN \(GeMDA(X_1, X_2, \ldots, X_d)\), where

\[
GeMDA(X_1, \ldots, X_d) = \left[ \sum_{i=2}^{d} \frac{X_i \cdot GCD_i}{N} \right] + \sum_{i=1}^{d} (X_i \cdot Shf\_dist_i) \pmod N,
\]

\(N = \text{number of PNs},\)

\(Shf\_dist_i = \left[ \sqrt{N} \right]^{i-1}, \text{ and}\)

\(GCD_i = \gcd(Shf\_dist_i, N).\)
# Optimality Comparison

<table>
<thead>
<tr>
<th>Allocation scheme</th>
<th>Optimal with respect to row queries</th>
<th>Optimal with respect to column queries</th>
<th>Optimal with respect to small square queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>HCAM</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>CMD</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>GeMDA</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Conventional Parallel Hash-based Join: Grace Algorithm

JOIN

BUCK TUNING

JOIN

BUCK TUNING

JOIN

BUCK TUNING

JOIN

BUCK TUNING

JOIN

BUCK TUNING

JOIN

DATA TRANSMISSION

HASH

R1 S1

R2 S2

R3 S3

R4 S4

PN1 PN2 PN3 PN4

Join matching bucket pairs

A bucket after merging

Merge buckets to fit memory space

Transmit tuples to their hash bucket

A hash bucket

Hash tables into buckets

Hash
The Effect of Imbalanced Workloads

Number processors = 64
I/O = 64 X 4 MBytes
Communication = 64 X 4 MBytes
Partition Tuning: *Largest Processing Time (LPT) First Strategy*
Naive Load Balancing Parallel Hash Join (NBJ)
Adaptive Load Balancing Parallel Hash Join (ABJ)

Local buckets are collected to their destined PN, based on “bin packing,” to form the local partitions.

Each PN hashes its local tuples into local buckets.

Workload is balanced among the PN’s throughout the computation.

What if the partitioning is skew initially?
Tuple Interleaving Parallel Hash Join (TIJ)

• Workload is balanced among the PN’s throughout the computation
• Better to do partition tuning before bucket tuning
Simulation Results

ABJ continues to suffer the initial skew after the hash phase resulting in a performance worse than even that of GRACE under a severe initial skew condition.
Sampling-based Load Balancing (SLB) Join Algorithm

- **Sampling Phase**: Each PN loads a small percentage of its tuples into memory, and hash them into a large number of in-memory hash buckets (hash on the join attribute)

- **Partition Tuning**: The coordinating PN applies “bin packing” to the in-memory buckets to determine the optimal bucket allocation scheme (BAS)

- **Split Phase**:
  - The in-memory buckets are collected to their destined PN in accordance with the BAS to form the initial partial buckets
  - Each PN loads the remaining tuples and forwards them to their destined hash buckets (“bin packing” not needed)

- **Join Phase**: Each PN performs the local joins of respectively matching buckets
nCUBE/2 Results:
SLB vs. ABJ vs. GRACE

- The performance of SLB approaches that of GRACE on very mild skew conditions, and
- it can avoid the disastrous performance that GRACE suffers on severe skew conditions.
Pipelining Hash-Join Algorithms

- **Two-Phase Approach:**

  - **Advantage:** requires only one hash table.
  - **Disadvantage:** pipelining along the outer relation must be suspended during the build phase (i.e., building the hash table).

- **One-Phase Approach:** As a tuple comes in, it is first inserted into its hash table, and then used to probe that part of the hash table of the other operand that has already been constructed.

  - **Advantage:** pipelining along both operands is possible.
  - **Disadvantage:** requires larger memory space.
Aggregate Functions

• An SQL aggregate function is a function that operates on groups of tuples.

**Example:**

```sql
SELECT department, COUNT(*)
FROM Employee
WHERE age > 50
GROUP BY department
```

• The number of result tuples depends on the selectivity of the GROUP BY attributes (i.e., `department`).
Centralized Merging

<table>
<thead>
<tr>
<th>Coordinator</th>
<th>Department</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PN1</th>
<th>Department</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PN2</th>
<th>Department</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PN3</th>
<th>Department</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>

Partition 1

Partition 2

Partition 3
Distributed Merging

PN1

<table>
<thead>
<tr>
<th>Department</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td>20</td>
</tr>
</tbody>
</table>

PN2

<table>
<thead>
<tr>
<th>Department</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>17</td>
</tr>
<tr>
<td>7</td>
<td>13</td>
</tr>
</tbody>
</table>

PN3

<table>
<thead>
<tr>
<th>Department</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>14</td>
</tr>
</tbody>
</table>

This scheme incurs less data transmission compared to next technique.

A counter is created for each department. This strategy uses more memory.
Repartitioning

<table>
<thead>
<tr>
<th>PN1</th>
<th>Department</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PN2</th>
<th>Department</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>13</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PN3</th>
<th>Department</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>14</td>
<td></td>
</tr>
</tbody>
</table>

This scheme incurs more data transmission compared to next technique.

No need to write the new partitions to disks.

A counter is created for each department. This strategy uses more memory.

This scheme incurs less data transmission compared to next technique.
Performance Characteristics

- **Centralized Merging Algorithm:**
  
  **Advantage:** It works well when the number of tuples is small.
  
  **Disadvantage:** The merging phase is sequential.

- **Distributed Merging Algorithm:**
  
  **Advantage:** The merging step is not a bottleneck.
  
  **Disadvantage:** Since a group value is being accumulated on potentially all the PNs the overall memory requirement can be large.

- **Repartitioning Algorithm:**
  
  **Advantage:** It reduces the memory requirement as each group value is stored in one place only.
  
  **Disadvantage:** It incurs more network traffic.
Coventional Aggregation Algorithms

- **Centralized Merging (CM) Algorithm:**
  - **Phase 1:** Each PN does aggregation on its local tuples.
  - **Phase 2:** The local aggregate values are merged at a predetermined central coordinator.

- **Distributed Merging (DM) Algorithm:**
  - **Phase 1:** Each PN does aggregation on its local tuples.
  - **Phase 2:** The local aggregate values are then hash-partitioned (based on the GROUP BY attribute) and the PNs merge these local aggregate values in parallel.

- **Repartitioning (Rep) Algorithm:**
  - **Phase 1:** The relation is repartitioned using the GROUP BY attributes.
  - **Phase 2:** The PNs do aggregation on their local partitions in parallel.

**Performance Comparison:**

- CM and DM work well when the number of result tuples is small.
- Rep works better when the number of groups is large.
Adaptive Aggregation Algorithms

- **Sampling Based (Samp) Approach:**
  - CM algorithm is first applied to a small Page-oriented random sample of the relation.
  - If the number of groups obtained from the sample is small then DM strategy is used; Rep algorithm is used otherwise.

- **Adaptive DM (A-DM) Algorithm:**
  - This algorithm starts with the DM strategy under the common case assumption that the number of group is small.
  - However, if the algorithm detects that the number of groups is large (i.e., memory full is detected) it switches to the Rep strategy.

- **The Adaptive Repartitioning (A-Rep) Algorithm:**
  - This algorithm starts with the Rep strategy.
  - It switches to DM if the number of groups is not large enough (i.e., number of groups is too few given the number of seen tuples).

**Performance Comparison:**

- In general, A-DM performs the best.
- However, A-Rep should be used if the number of groups is suspected to be very large.
Implementation Techniques for A-DM

• **Global Switch:**

  – When the first PN detects a memory full condition, it informs all the PNs to switch to the Rep strategy.

  – Each PN first partitions and sends the so far accumulated local results to PNs they hash to. Then, it proceeds to read and repartition the remaining tuples.

  – Once the repartitioning phase is complete, the PNs do aggregate on the local partitions in parallel (as in the Rep algorithm).

• **Local Switch:**

  – A PN upon detecting memory full stops processing its local tuples. It first partitions and sends the so far accumulated local results to the PNs they hash to. Then it proceeds to read and repartition the remaining tuples.

  – During Phase one, one set of PNs may be executing the DM algorithm while other are executing the Rep algorithm. When the latter receives an aggregate value from another PN, it accumulates it to the corresponding local aggregate value.

  – Once all PNs have completed their Phase 1, The local aggregate values are merged as in the DM algorithm.
A-DM: Global Switch

Switching to Rep = Step 1 + Step 2

Step 1: Prepare for Repartitioning
Step 2: Apply Repartitioning to the remaining tuples
A-DM: Local Switch

- Can only handle some of the departments
- Still handle all departments
- Distributed Merging
- Only PN1 switches to the Rep scheme
- All PNs start with Distributed Merging
SQL (Structured Query Language)

An SQL query:

```
SELECT ENAME
FROM EMPLOYEE, WORKSON, PROJECT
WHERE PNAME = 'database' AND
PNUM = PNO AND
ENO = ENUM AND
BDATE > '1965'
```

- SQL is *nonprocedural*.
- The Compiler must generate the *execution plan*.
  1. Transforms the query from SQL into *relational algebra*.
  2. Restructures (optimizes) the algebra to improve performance.
Relational Algebra

### Relation T1

<table>
<thead>
<tr>
<th>ENAME</th>
<th>SALARY</th>
<th>ENUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrew</td>
<td>$98,000</td>
<td>005</td>
</tr>
<tr>
<td>Casey</td>
<td>$150,000</td>
<td>003</td>
</tr>
<tr>
<td>James</td>
<td>$120,000</td>
<td>007</td>
</tr>
<tr>
<td>Kathleen</td>
<td>$115,00</td>
<td>001</td>
</tr>
</tbody>
</table>

### Relation T2

<table>
<thead>
<tr>
<th>ENUM</th>
<th>ADDRESS</th>
<th>BDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>Orlando</td>
<td>1964</td>
</tr>
<tr>
<td>003</td>
<td>New York</td>
<td>1966</td>
</tr>
<tr>
<td>005</td>
<td>Los Angeles</td>
<td>1968</td>
</tr>
<tr>
<td>007</td>
<td>London</td>
<td>1958</td>
</tr>
</tbody>
</table>

- **Select**: Selects rows.
  \[
  \sigma_{\text{SALARY} \geq 120,000}(T1) = \{(Casey, 150000, 003), (James, 120000, 007)\}
  \]

- **Project**: Selects columns.
  \[
  \pi_{\text{ENAME, SALARY}}(T1) = \{(Andrew, 98000), (Casey, 150000), (James, 120000), (Kathleen, 115000)\}
  \]

- **Cartesian Product**: Selects all possible combinations.
  \[
  T1 \times T2 = \{(Andrew, 98000, 005, 001, Orlando, 1964),
  (Andrew, 98000, 005, 003, New York, 1966),
  \cdots,
  (Kathleen, 150000, 001, 005, Los Angeles, 1968),
  (Kathleen, 115000, 001, 007, London, 1958)\}
  \]

- **Join**: Selects some combinations.
  \[
  T1 \bowtie T2 = \{(Andrew, 98000, 005, Los Angeles, 1968),
  (Casey, 150000, 003, New York, 1966),
  (James, 120000, 007, London, 1958),
  (Kathleen, 115000, 001, Orlando, 1964)\}
  \]
Transforming SQL into Algebra

An SQL query:

```sql
SELECT ENAME
FROM EMPLOYEE, WORKSON, PROJECT
WHERE PNAME = 'database' AND
      PNUM = PNO AND
      ENO = ENUM AND
      BDATE > '1965'
```

This query tree (procedure) will compute the correct result. However, the performance will be very poor. ➞ *needs optimization*!
Optimization Strategies

**GOAL:** Reducing the sizes of the intermediate results as quickly as possible.

**STRATEGY:**

1. Move SELECTs and PROJECTs as far down the query tree as possible.

2. Among SELECTs, reordering the tree to perform the one with lowest selectivity factor first.

3. Among JOINs, reordering the tree to perform the one with lowest join selectivity first.
Example: Apply SELECTs First

Canonical Query Tree

SELECT Clause

WHERE Clause

FROM Clause

After Optimization
Example: Replace \( \sigma - \times \) by \( \bowtie \)

**Before Optimization**

\[
\begin{align*}
\pi_{\text{ENAME}} & \rightarrow \\
\sigma_{\text{PNUM} = \text{PNO}} & \\
\times & \\
\sigma_{\text{ENUM} = \text{ENO}} & \sigma_{\text{PNAME} = 'database'} \\
\times & \\
\sigma_{\text{BDATE} > '1965'} & \text{WORKSON} \\
\text{EMPLOYEE} & \text{PROJECT}
\end{align*}
\]

**After Optimization**

\[
\begin{align*}
\pi_{\text{ENAME}} & \rightarrow \\
\bowtie_{\text{PNUM} = \text{PNO}} & \\
\bowtie_{\text{ENUM} = \text{ENO}} & \sigma_{\text{PNAME} = 'database'} \\
\sigma_{\text{BDATE} > '1965'} & \text{WORKSON} \\
\text{EMPLOYEE} & \text{PROJECT}
\end{align*}
\]
Example: Move PROJECTs Down

**Before Optimization**

```
\begin{array}{c}
\text{ENAME} \\
\downarrow \\
\text{PNUM} = \text{PNO} \\
\downarrow \\
\text{ENUM} = \text{ENO} \\
\downarrow \\
\sigma_{\text{BDATE} > '1965'} \\
\text{ENAME, PNO} \\
\downarrow \\
\text{ENUM} = \text{ENO} \\
\downarrow \\
\sigma_{\text{PNAME} = 'database'} \\
\text{PROJEC\text{T}} \\
\end{array}
```

**After Optimization**

```
\begin{array}{c}
\text{ENAME} \\
\downarrow \\
\text{PNUM} = \text{PNO} \\
\downarrow \\
\text{ENAME, PNO} \\
\downarrow \\
\text{ENUM} = \text{ENO} \\
\downarrow \\
\text{ENAME, ENUM} \\
\downarrow \\
\sigma_{\text{BDATE} > '1965'} \\
\text{ENAME, PNO} \\
\downarrow \\
\text{ENO, PNO} \\
\downarrow \\
\sigma_{\text{PNAME} = 'database'} \\
\text{PROJEC\text{T}} \\
\end{array}
```
Parallelizing Query Optimizer

Relations are fragmented and allocated to multiple processing nodes:

\[\rightarrow\] Besides the choice of ordering relational operations, the \textit{parallelizing optimizer} must select the best sites to process data.

\[\rightarrow\] The role of a \textit{parallelizing optimizer} is to map a query on a global relations into a sequence of local operations acting on local relation fragments.
Parallelizing Query Optimization

Parallelizing Optimizer:

- Parallelizes the relational operators.
- Selects the best processing nodes for each parallelized relational operator.
Parallelizing Example

Fragments:

\[ E_1 = \sigma_{ENO \leq E3}(E) \]
\[ E_2 = \sigma_{E3 < ENO \leq E6}(E) \]
\[ E_3 = \sigma_{ENO > E6}(E) \]
\[ G_1 = \sigma_{ENO \leq E3}(G') \]
\[ G_2 = \sigma_{ENO > E3}(G') \]

Query: 

\[
\text{SELECT * FROM E, G WHERE E.ENO = G.ENO}
\]

(1) Sequential query tree

(2) Data Localization

(3) Distributing \( \bowtie \) over \( \cup \)

(4) Eliminate useless JOINs

(5) Select the best processing node for each fragment operator
Parallelizing Query Optimization

1. Determines which fragments are involved and transforms the global operators into fragment operators.

2. Eliminates useless fragment operators.

3. Finds the “best” ordering of the fragment operators.

4. Selects the best processing node for each fragment operator and specifies the communication operations.
Prototype at UCF

- A prototype of a shared-nothing system is implemented on a 64-processor nCUBE/2 computer.

- Our system was implemented to demonstrate:
  - GeMDA multidimensional data partitioning technique,
  - dynamic optimization scheme with load balancing capability, and
  - competition-based scheduling policy.
System Architecture

Create/Destroy Tables

Results

SQL Queries

PRESENTATION MANAGER

Global Schema

Fragment Schema

QUERY TRANSLATOR

QUERY EXECUTOR

Operator Routine...

Operator Routine

LOAD UTILITY

STORAGE MANAGER

Database

Database

Interprets the execution tree and calls the corresponding operator routines to carry out the underlying operations.

Provides a sequential scan interface to the query processing facilities.
Software Components

- **Storage Manager**: This component manages physical disk devices and schedules all I/O activities. It provides a sequential scan interface to the query processing facilities.

- **Catalog Manager**: It acts as a central repository of all global and fragment schemas.

- **Load Utility**: This program allows the users to populate a relation using an external file. It distributes the fragments of a relation across the processing nodes using GeMDA.

- **Query Translator**: This component provides an interface for queries. It translates an SQL query into a query graph. It also caches the global schema information locally.

- **Query Executor**: This component performs dynamic query optimization. It schedules the execution of the operators in the query graph.

- **Operator Routines**: Each routine implements a primitive database operator. To execute an operator in a query graph, the Query Executor calls the appropriate operator routine to carry out the underlying operation.

- **Presentation Manager**: This module provides an interactive interface for the user to create/destroy tables, and query the database. The user can also use this interface to browse query results.
Server Class

Three operator servers “simultaneously” share the computing resources of PN2.

- Process 1 to run “PowerPoint”
- Process 2 to run a browser

PN1

PN2

Windows
A logical server is software running in a process capable of performing a certain basic database operation (e.g., SCAN) to support query processing.
Each operator server is associated with a processing node.

**Advantage:** Fair.

**Disadvantage:** System utilization is not maximized.
Competition-based: Potential drawbacks

With some planning, the following schedule is better:
Planning-based Scheduling

- Advantage: Better system utilization
- Disadvantage: Less fair
Hardware Organization

IFP: Interface Processor
ACP: Access Processor

- Catalog Manager, Query Manager and Scheduler processes are run on IFP’s.
- Operator processes are run on ACP’s.
### Structure of Operator Processes

- The output is demultiplexed through a split table.

- When the process detects the end of its input stream,
  - it first closes the output streams and
  - then sends a control message to its coordinator process indicating that it has completed execution.

<table>
<thead>
<tr>
<th>Hash Value</th>
<th>Destination Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(Processor #3, Port #5)</td>
</tr>
<tr>
<td>1</td>
<td>(Processor #4, Port #6)</td>
</tr>
<tr>
<td>2</td>
<td>(Processor #5, Port #8)</td>
</tr>
<tr>
<td>3</td>
<td>(Processor #6, Port #2)</td>
</tr>
</tbody>
</table>
Example: Operator and Process Structure

- **Query Tree:**

  ![Query Tree Diagram]

  - Executing Join on PN3 and PN4

- **Process Structure:**

  ![Process Structure Diagram]

  - Use "B" tuples to probe the hash tables
  - Build hash tables for relation A
A storage manager provides the primitives for scanning a file via a sequential or index scan.

A storage manager provides the primitives for scanning a file via a sequential or index scan.

**COMPiled QUERY**

**COMPiled QUERY**

**STORAGE MANAGER**

**STORAGE MANAGER**

**OPERATOR METHODS**

**OPERATOR METHODS**

**Contains code for each operator in the database access language.**

**Maintains an active scan table that describes all the scans in progress.**

**Maps file names to file ID’s, manages active files, searches for the page given a record.**

**Manages a buffer pool.**

**Manages physical disk devices, performs page-level I/O operations.**

**Database**

**Database**

**ACCESS METHODS**

**ACCESS METHODS**

**ACCESS METHODS**

**STORAGE STRUCTURES**

**STORAGE STRUCTURES**

**STORAGE STRUCTURES**

**BUFFER MANAGEMENT**

**BUFFER MANAGEMENT**

**BUFFER MANAGEMENT**

**PHYSICAL I/O**

**PHYSICAL I/O**

**PHYSICAL I/O**

**A data entry**

**A data entry**

**Search Key**

**Search Key**

**ACCESS METHOD**

**ACCESS METHOD**

**ACCESS METHOD**

**A data record**

**A data record**

**Record ID**

**Record ID**

**Record ID**

**Record ID**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Transaction Processing

The consistency and reliability aspects of transactions are due to four properties

- **Atomicity**: A transaction is either performed in its entirety or not performed at all
- **Consistency**: A correct execution of the transaction must take the database from one consistent state to another
- **Isolation**: A transaction should not make its updates visible to other transaction until it is committed
- **Durability**: Once a transaction changes the database and changes are committed, these changes must never be lost because of subsequent failure
### Transaction Manager

- **Lock Manager:**
  - Each local lock manager is responsible for the lock units local to that processing node.
  - They provide *concurrency control*.

- **Log Manager:**
  - Each local log manager logs the local database operations.
  - They provide *recovery services*.
Two-Phase Locking Protocol

- Any schedule generated by a concurrency control algorithm that obeys the 2PL protocol is serializable (i.e., the isolation property is guaranteed).

- 2PL is difficult to implement. The lock manager has to know:
  1. the transaction has obtained all its locks, and
  2. the transaction no longer needs to access the data item in question. (so that the lock can be released).

- Cascading aborts can occur. (because transactions reveal updates before they commit)
The lock manager releases all the locks together when the transaction terminates (commits or aborts).
Wait-for Graph

- If a transaction reads an object, the transaction depends on that object version.

- If the transaction writes an object, the resulting object version depends on the writing transaction.

**The transaction execution sequences**

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Operation</th>
<th>Object Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>READ</td>
<td>(o,1)</td>
</tr>
<tr>
<td>T2</td>
<td>WRITE</td>
<td>(o,2)</td>
</tr>
</tbody>
</table>

**The dependency graphs**

- **READ → WRITE dependency**
- **WRITE → READ dependency**
- **WRITE → WRITE dependency**

---

**Wait-for Graph**

Transaction T₉ is waiting for transaction T₅.
Handling Deadlocks

- **Detection and Resolution:**
  
  - Abort and restart a transaction if it has waited for a lock for a long time.
  
  - Detect cycles in the wait-for graph and select a transaction (involved in a cycle) to abort.

- **Prevention:**

  If $T_i$ requires a lock held by $T_j$,

  * If $T_i$ is older $\implies T_i$ can wait.
  
  * If $T_i$ is younger $\implies T_i$ is aborted and restarted with the same timestamp.

There can never be a loop in the wait-for graph. Deadlock is not possible !!
When a transaction is blocked, it sends a special probe message to the blocking transaction. The message consists of three numbers: the transaction that just blocked, the transaction sending the message, and the transaction to whom it is being sent.

When the message arrives, the recipient checks to see if it itself is waiting for any transaction. If so, the message is updated, replacing the second field by its own TID and the third one by the TID of the transaction it is waiting for. The message is then sent to the blocking transaction.

If a message goes all the way around and come back to the original sender, a deadlock is detected.
Two-Phase Commit Protocol

To ensure the atomicity property, a 2P commit protocol can be used to coordinate the commit process among subtransactions.

- **Coordinator**: it originates the transaction.
- **Agent**: it executes a subtransaction on behalf of its coordinator.

---

**Diagram: Two-Phase Commit Protocol**

1. **Prepare**: Coordinator initiates the protocol.
2. **Ready or Abort (Voting)**: Agents vote on whether to commit or abort.
3. **Commit or Abort**: Agents either commit or abort.
4. **ACK**: Agents acknowledge the decision.
Recovery

• An entry is made in the local log file at a processing node each time one of the following commands is issued by a transaction:

  – begin transaction

  – write (insert, delete, update)

  – commit transaction

  – abort transaction

• Write-ahead log protocol:

  – It is essential that log records be written before the corresponding write to the database.

    **If log entry wasn’t saved before the crash, corresponding change was not applied to database!**

  – If there is no commit transaction entry in the log for a particular transaction, then that transaction was still active at the time of failure and must therefore be undone.

    **This is to ensure the atomicity property**
It may have over 1,000 processors and many thousands of disks.

Each relation is hash partitioned over a subset of the AMPs.

Near-linear speedup and scaleup on queries have been demonstrated for systems containing over 100 processors.
A tournament tree

Globally sorted

Locally sorted

PN_1
145
168
172
...

PN_2
137
141
170
...

PN_3
140
158
174
...

PN_4
139
142
169
...

2, 5, 7, 8, 9, ... 122, 136
Teradata DBC/1012: Distribution of Data

- The fallback copy ensures that the data remains available on other AMPs if an AMP should fail.
- In the following example, if AMPs 4 and 7 were to fail simultaneously, however, there would be a loss of data availability.

![Primary Copy Area: DSU/AMP 1: 1, 9, 17 Fallback Copy Area: DSU/AMP 2: 2, 10, 18 DSU/AMP 3: 3, 11, 19 DSU/AMP 4: 4, 12, 20]

- Additional data protection can be achieved by "clustering" the AMPs in groups.
- In the following example, If both AMPs 4 and 7 were to fail, all data would still be available.
Tandem systems run the applications on the same processors as the database servers.

Relations may be range partitioned across multiple disks.

It is primarily designed for OLTP. It scales linearly well beyond the largest reported mainframes on the TPC-A benchmarks.

It is three times cheaper than a comparable mainframe system.
Challenges Facing Parallel Database Technology

- Multimedia applications require very large network-I/O bandwidth.

- A parallel parallelizing query optimizer will be able to consider many more execution plans.

- Design parallel programming languages to take advantage of parallelism inherent in the parallel database system.

- We need new data partitioning techniques and algorithms for non-relational data objects allowed in SQL3, e.g., graph structures.