The Gamma Database Machine

Instructor: Anastassia Ailamaki
http://www.cs.cmu.edu/~natassa

Citation
IEEE TKDE 2(1): 44-63 (March 1990)

Detailed outline
- Introduction / History
  - Hardware
  - Software architecture
  - Query processing
  - Performance evaluation
History

- DIRECT 1977 - 84
  - early database machine project
  - showed parallelism useful for db apps
- Flaws curtailed scalability
  - shared memory
  - central control of execution
- Gamma 1984 - 92

Key Ideas

- Shared-nothing
- Hash-based parallel algorithms
- Horizontal partitioning ('declustering')

Gamma Hardware (v2.0)

- iPSC/2 Intel hypercube
- 32 x386 processors
- 8MB of memory
- 330MB Maxtor drive / node (45KB cache)
- Routing modules
  - 2.8 Mb/s
  - full duplex, serial, reliable
Gamma v2.0

- OS: NOSE
- multiple, lightweight processes with shared memory
- Entire DB in one NX/2 process

Detailed outline

- Introduction / History
- Hardware
- Software architecture
  - Query processing
  - Performance evaluation

Storage Organization

- Horizontal partitioning (user selectable)
  - round-robin; hashed; range partitioned
  - all relations, on all units
  - method of partitioning: recorded in catalog
- Indexes created automatically on all partitions
- Clustered index not necessarily on partitioning attribute
- Better idea?
Storage Organization

- Horizontal partitioning (user selectable)
  - round-robin; hashed; range partitioned
  - all relations, on all units
  - method of partitioning: recorded in catalog
- Indexes created automatically on all partitions
- Clustered index not necessarily on partitioning attribute
- Partitioning relations should have been based on 'heat'

Gamma Process Structure

- there is a 'host' machine;
- and the Gamma processors

Gamma process structure

SCHEMA

Query mgr
Catalog mgr
Host
Gamma processors
Gamma process structure

- **Catalog manager**
- **Query manager**
  - one associated with each user (on host)
- **Scheduler processes**
  - coordinates multi-site queries (spread out - why?)
- **Operator processes**
  - each executes a single relational operator
Query Processing

- ad-hoc and embedded query interfaces
- standard parsing, optimization, code gen.
- left deep trees only
- hash joins only
- at most two join operators active simultaneously

Operator and process structure

- data-flow: each operator process
  - reads input stream
  - output tuples in one or more output streams:
  - split tables (partitioning, joining)

Split Table

Directs operator output to the appropriate node (e.g., by some hash value)
E.g.: Parallel Hash Join

P3, P4: diskless
P1, P2: w/disk

Fig. 6.

Detailed outline

Introduction / History
Hardware
Software architecture
Query processing
Performance evaluation
Selections

- Start selection operator on each node
- Exclusion of nodes for hash and range partitioning
- Throughput considerations
  - one page read-ahead

Joins

- Partition into buckets, join buckets
- Implemented: sort-merge, Grace, Simple, Hybrid
- Parallel Hybrid

More operations

- Aggregation - How?
More operations

- Aggregation
  - Compute partial results for each partition
  - Hash on “group-by” attribute
- Updates
  - Standard techniques
  - How to update partitioning attribute?

Concurrency Control

- 2PL
  - Granularity: ?

Concurrency Control

- 2PL
  - Granularity: file and page
  - Modes: S, X, IS, IX, SIX
  - Local lock manager and deadlock detector
  - (how?)
Concurrency Control

- 2PL
- Granularity: file and page
- Modes: S, X, IS, IX, SIX
- Local lock manager and deadlock detector
- wait-for graph
- Centralized multi-site deadlock detector

Q: How often to check for deadlocks?

A: Period halves/doubles on deadlock / no-deadlock
Recovery

- Standard WAL protocol
  - Local log manager generates log records
- One or more log processors
  - Collect these log records and write them to the disk
- Each query processor (i) determines log manager (1..m) statically (i mod m)
  - Easy rollback

Recovery

- Log manager maintains *flushed log table* (last log record flushed for each node)
- What if a node needs to force a dirty page to disk?
  - Requests *flushedLSN*
  - May need to choose another page
- Local log manager
  - Makes sure at least T buffers are available

Node Failure

- Goal #1: *availability* in spite of a processor or disk failure
- Goal #2: upon failure, *spread load* as uniformly as possible
- How to handle a (single) node failure?
Node Failure

- Chained declustering (Gamma)
  - backup copy on 'next' node
- Mirrored disk (Tandem)
- Interleaved declustering (Teradata)
  - backup copy: spread over rest of nodes

Fault Tolerance

- specifically:
  - Chained Declustering
    - Primary: i mod M; Backup: (i+1) mod M
  - Interleaved declustering
    - Divide fragments into N-1 parts
    - Store parts in all disks but the one containing primary
Fault Tolerance

- single-node failure:
  - Load redirection results in \( 1/n \) increase (why?)

Chained declustering

<table>
<thead>
<tr>
<th>Node</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>Primary Copy</td>
<td>B0</td>
<td>--</td>
<td>( \frac{1}{2} \cdot B2 )</td>
<td>( \frac{1}{2} \cdot B3 )</td>
<td>( \frac{1}{2} \cdot B4 )</td>
<td>( \frac{1}{2} \cdot B5 )</td>
<td>( \frac{1}{2} \cdot B6 )</td>
<td>( \frac{1}{2} \cdot B7 )</td>
</tr>
<tr>
<td>Backup Copy</td>
<td>( \frac{1}{2} \cdot B0 )</td>
<td>--</td>
<td>( \frac{1}{2} \cdot B1 )</td>
<td>( \frac{1}{2} \cdot B2 )</td>
<td>( \frac{1}{2} \cdot B3 )</td>
<td>( \frac{1}{2} \cdot B4 )</td>
<td>( \frac{1}{2} \cdot B5 )</td>
<td>( \frac{1}{2} \cdot B6 )</td>
</tr>
</tbody>
</table>
Fault Tolerance

- single-node failure:
  - Load redirection results in $1/n$ increase (why?)
- Chained vs Interleaved declustering
  - how about a 2-node failure?

---

Detailed outline

- Introduction / History
- Hardware
- Software architecture
- Query processing
- Performance evaluation
Experimental setup

- Wisconsin benchmark (100K, 1M, 10M tuples);
- tables: hash partitioned
- selections (1%, 10%) x (non-indexed, clustered index)
- joins
- wallclock time; speedup; scale-up

Selections

- non-indexed, 1%, 10%
  - response time
  ![Graph showing response time vs. #processors]

- non-indexed, 1%, 10%
  - speedup
  ![Graph showing speedup vs. #processors]
Selections

- clustered ind., 1%,
- clustered 10%
- non-clustered 1%

response time

#processors

- clustered 1%
- clustered 10%
- non-clustered 1%
- super-linear?
- sub-linear?
- why?

speed-up

#processors

- clustered 1%
- super-linear!

why super-linear?
- why sub-linear?
Selections - scaleup

- response time vs processors, increasing the db size
- All queries: ~constant scale-up

Joins

- A join B
  - part. = join attr
  - part. attr ! = join attr

Joins

- joinAB
  - part. = join attr
  - part. attr ! = join attr
Conclusions

- Shared-nothing architecture
- horizontal partitioning
- parallel hashing
- data-flow scheduling
- good speed-up and scale-up