The Gamma Database Machine

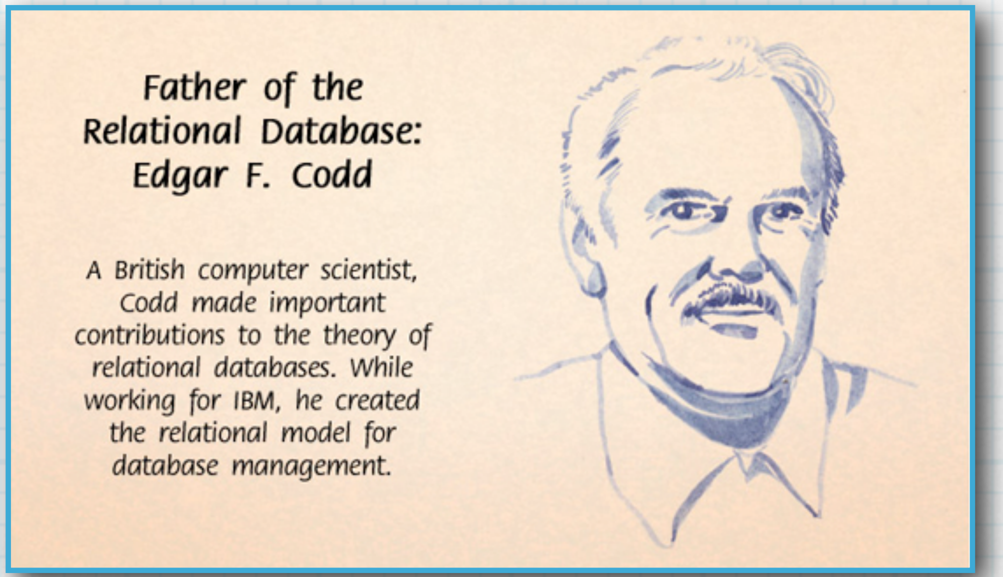
a 1990 paper from the
IEEE Transactions on Knowledge and Data Engineering

written by
David J. DeWitt, Shahram Ghandeharizadeh, Donovan A. Schneider, Allan Bricker, Hui-i Hsiao, and Rick Rasmussen

presented by
Alan G. Labouseur - alan@Labouseur.com

Gamma: the Big Idea

- Database - stores data
- Relational
  - structured data
  - tables of rows and columns
  - context turns data into information
- Supports Data Definition
- Supports Data Manipulation: CRUD



Gamma: the Big Idea

- Parallel - many processors, many disks
- Three keys to parallelism:
  1. tables are horizontally partitioned
  2. parallel hash algorithms for relational operators
  3. coordinated dataflow scheduling
- Shared-nothing architecture

The Plan

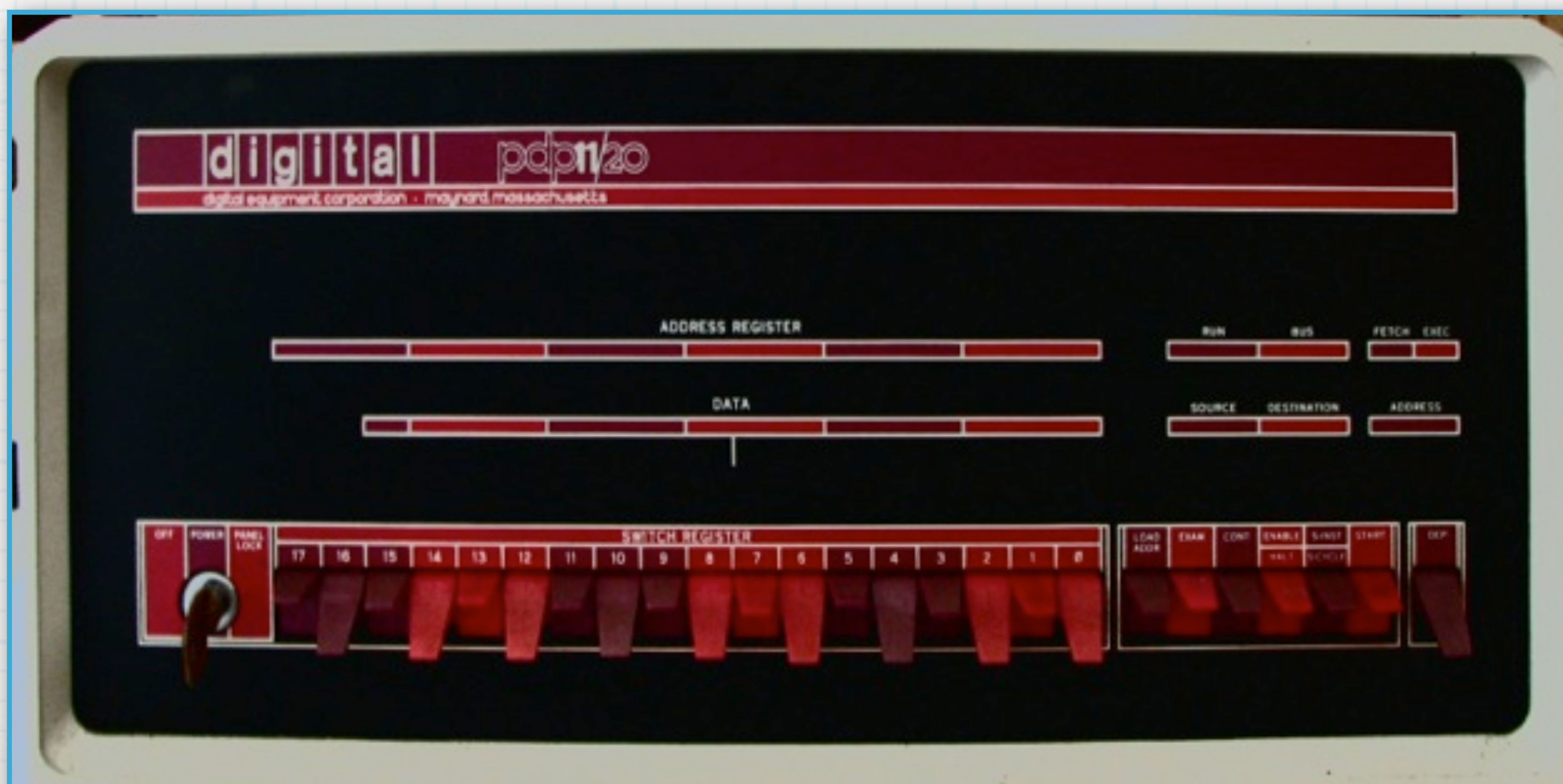
- History
- Hardware Architecture
- Software Architecture
- Query Algorithms
- Transactions
- Performance
- Summary

The Plan

- History
- Hardware Architecture
- Software Architecture
- Query Algorithms
- Transactions
- Performance
- Summary

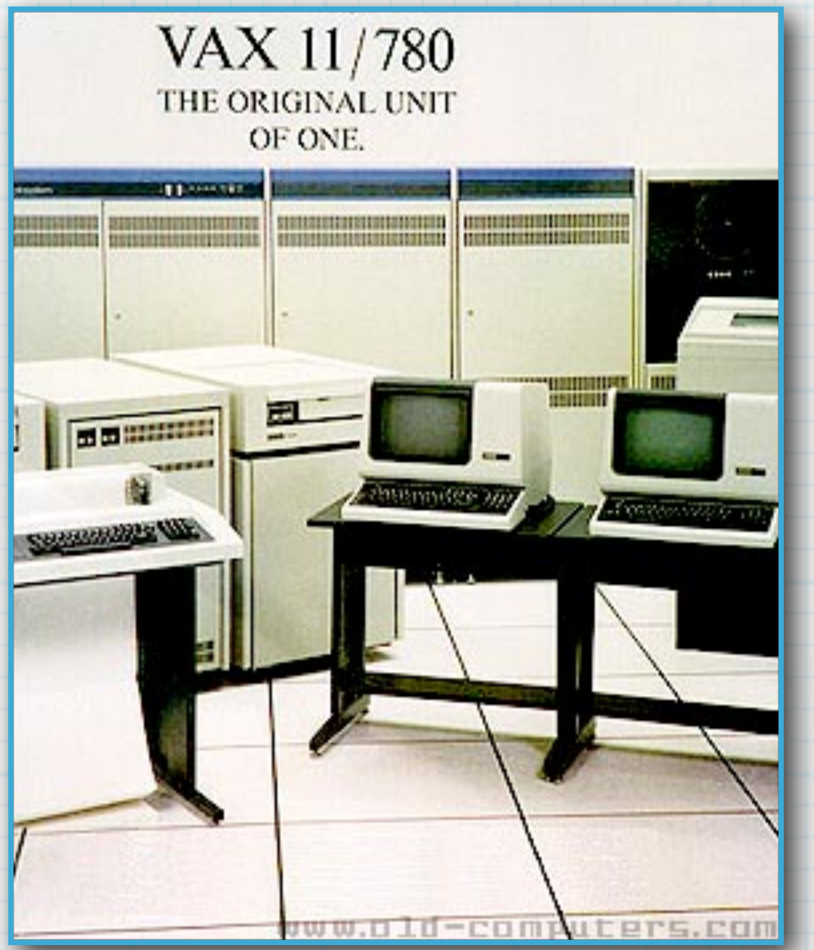
History

- Began with DIRECT (1977-1984)
  - One of the first operational parallel database systems. [2]
  - Built on the DEC PDP 11 (16-bit)



History

- *1984 - The GAMMA project began in January 1984 and ran until late 1992 at which point the code was so broken from years of patching that we gave up.*
    - David J. DeWitt on his web site [2]
- Built on a network of VAX computers (32-bit)
- Operational in 1985

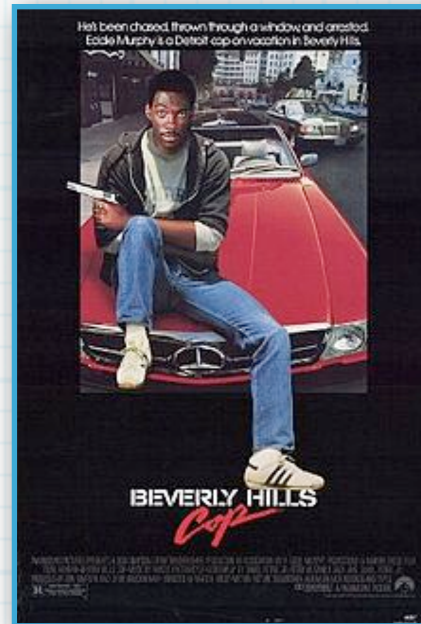


History

- *1984*

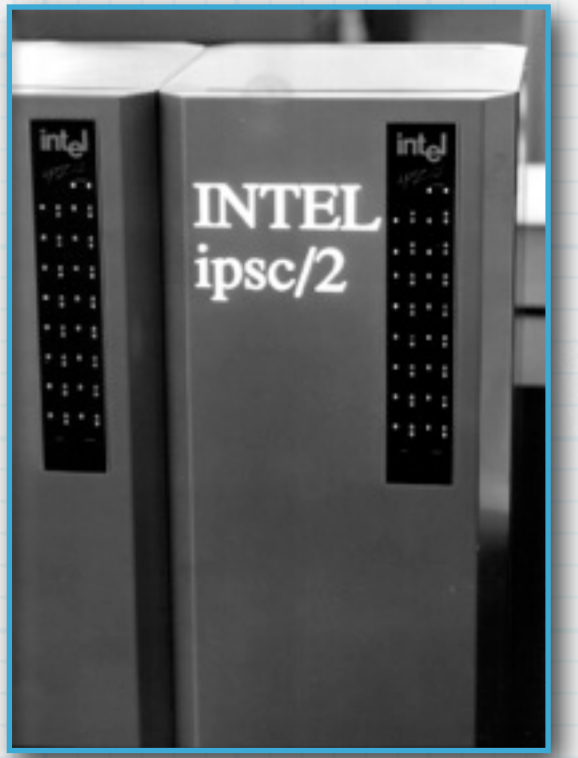
History

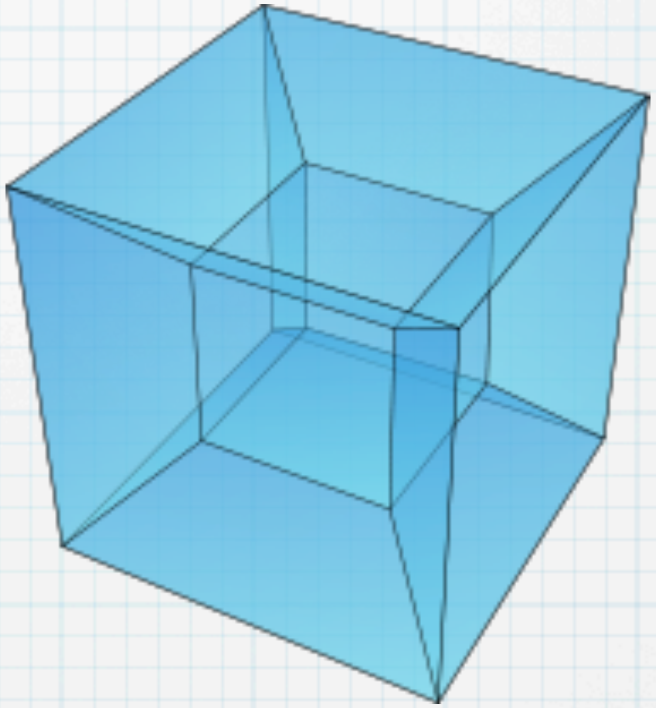
- *1984*



History

- 1988: Intel ipsc/2 hypercube - 32 i386 CPUs





- Nodes connected via VLSI routers.
  - Small messages sent as datagrams.
  - Large messages sent via circuits.

The Plan

- History
- Hardware Architecture
- Software Architecture
- Query Algorithms
- Transactions
- Performance
- Summary

Hardware Architecture

- Shared-nothing
  - All nodes are self-sufficient and independent, sharing neither disks nor memory nor CPU nor . . . anything, communicating only by sending messages. (Like people.)
- Storage is distributed among the nodes.
- Nodes are connected . . .

Hardware Architecture

- Why shared-nothing?
  - *In scalable, tunable, nearly delightful data bases, [shared-nothing] systems will have no apparent disadvantages compared to the other alternatives [shared memory, disk].* - Michael Stonebraker [3]
- This remains an excellent approach today. (Erlang, Scala with Akka, others.)
- Shared-nothing scales better than shared architectures. Why?

Hardware Architecture

- Converting from VAX to Intel uncovered previously unseen bugs in their code.
  - The VAX did not trap null pointer dereference errors.
  - The Intel 386 did. They found a number of hidden bugs.
- They also had to rewrite a lot of code because the VAX began numbering nodes at 1 while Intel began at 0.

The Plan

- History
- Hardware Architecture
- Software Architecture
- Query Algorithms
- Transactions
- Performance
- Summary

Software Architecture

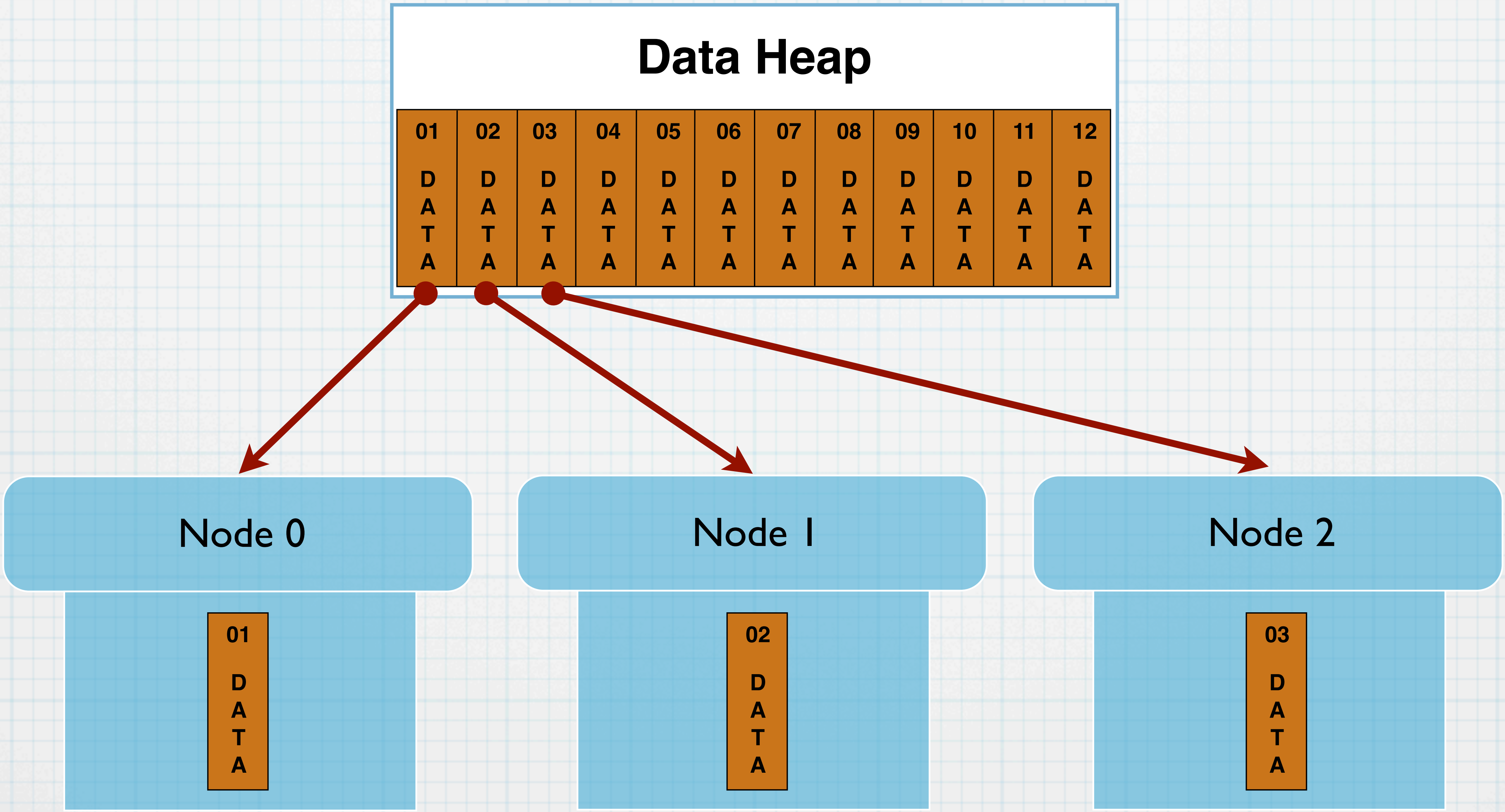
- Storage Organization
  - Tables are Horizontally Partitioned across all disks at all nodes.
    - exploits all available I/O bandwidth
  - This "declustering" (Bubba) makes parallelizing selections trivial.
    - Just send a message to each node to execute the selection operator with the passed-in parameters.

Software Architecture

- Storage Organization
  - Three declustering strategies.
    1. round robin - default method
    2. hashed - keys hashed into node ids
    3. range partitioned (“shards”)
       - Specify a range of keys for each node in a Range Table.
       - MongoDB and others do this today.

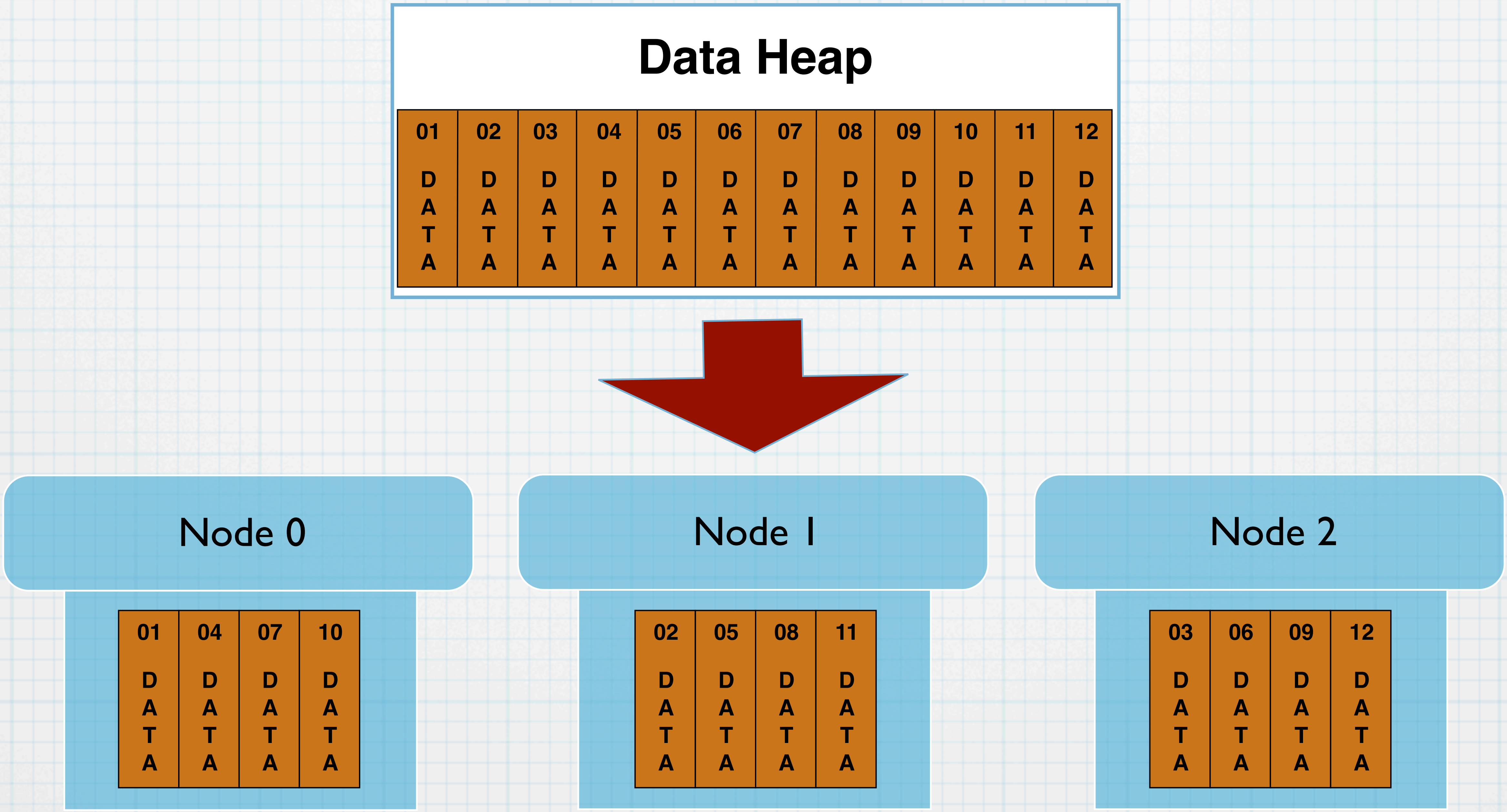
Software Architecture

- Storage Organization - Round Robin



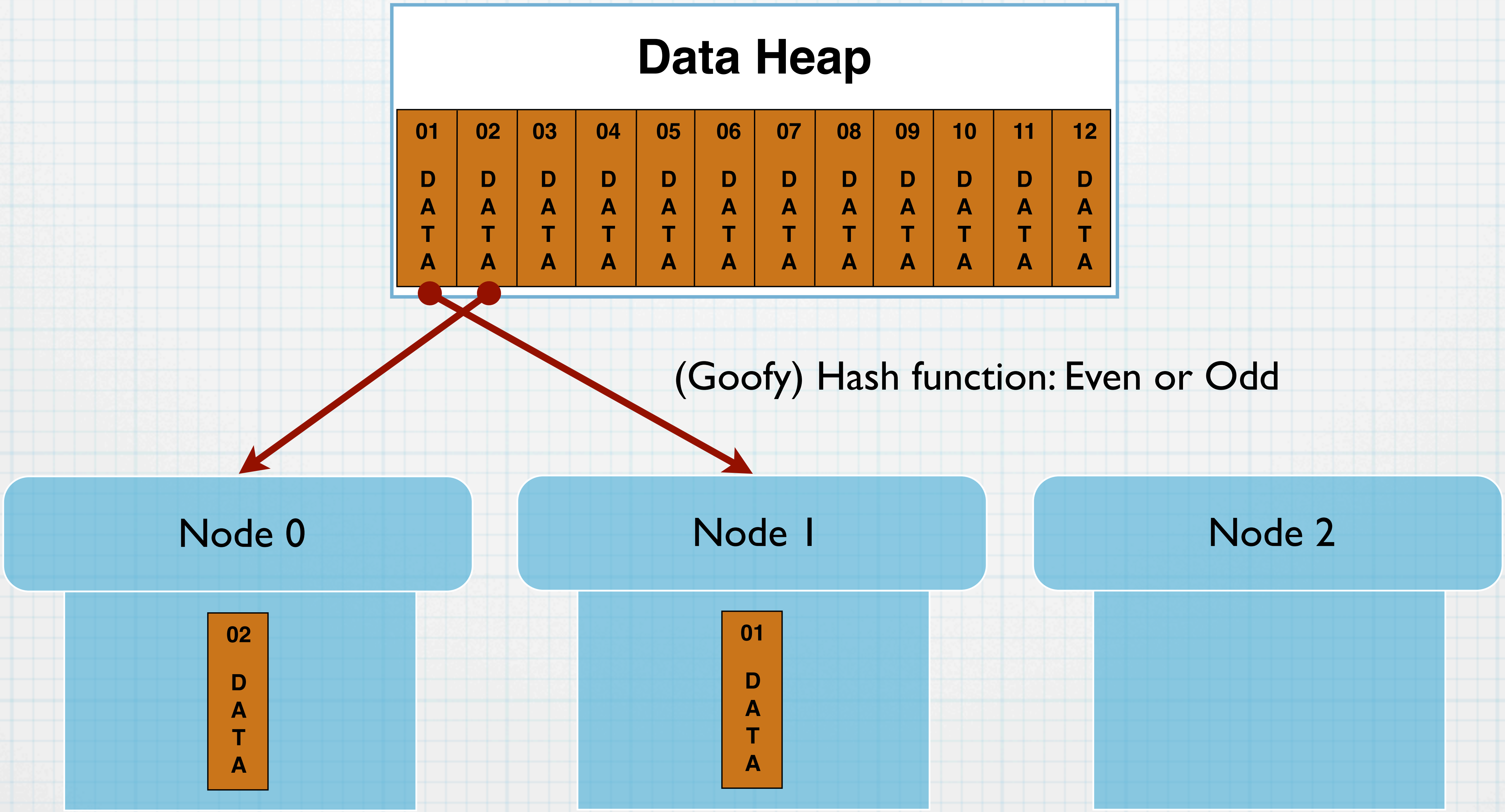
Software Architecture

- Storage Organization - Round Robin



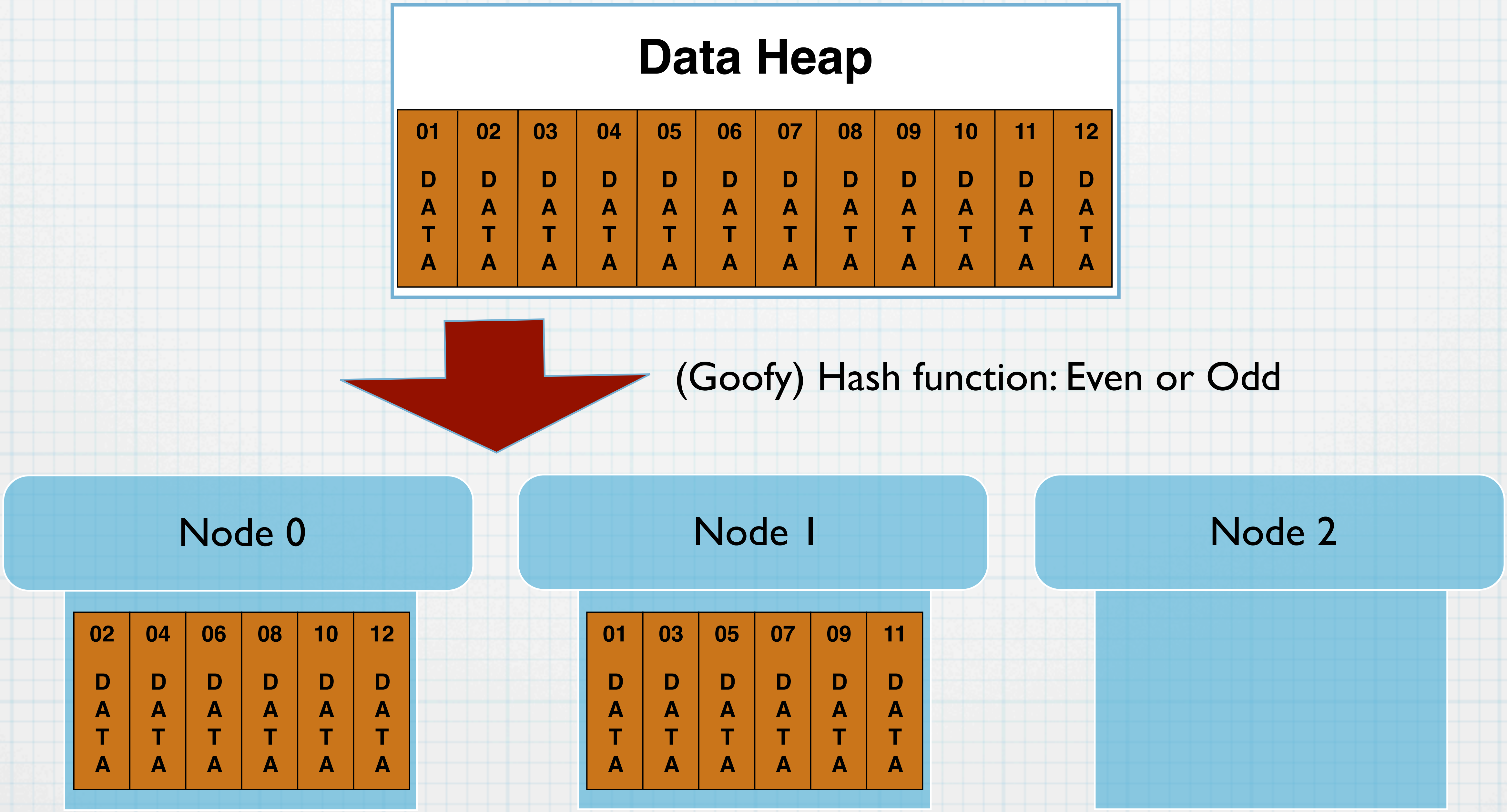
Software Architecture

- Storage Organization - Hashed



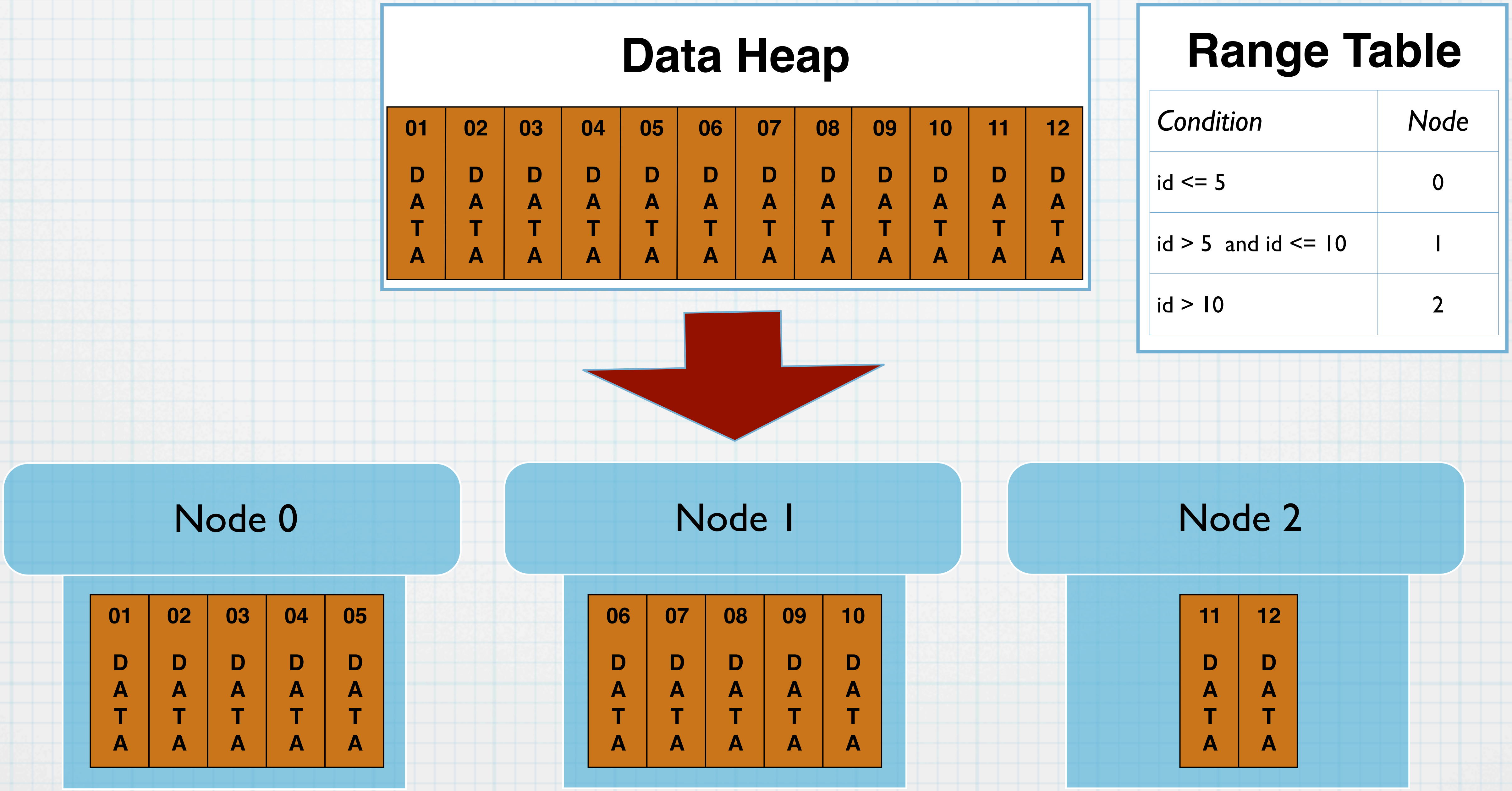
Software Architecture

- Storage Organization - Hashed



Software Architecture

- Storage Organization - Shards



Range Table

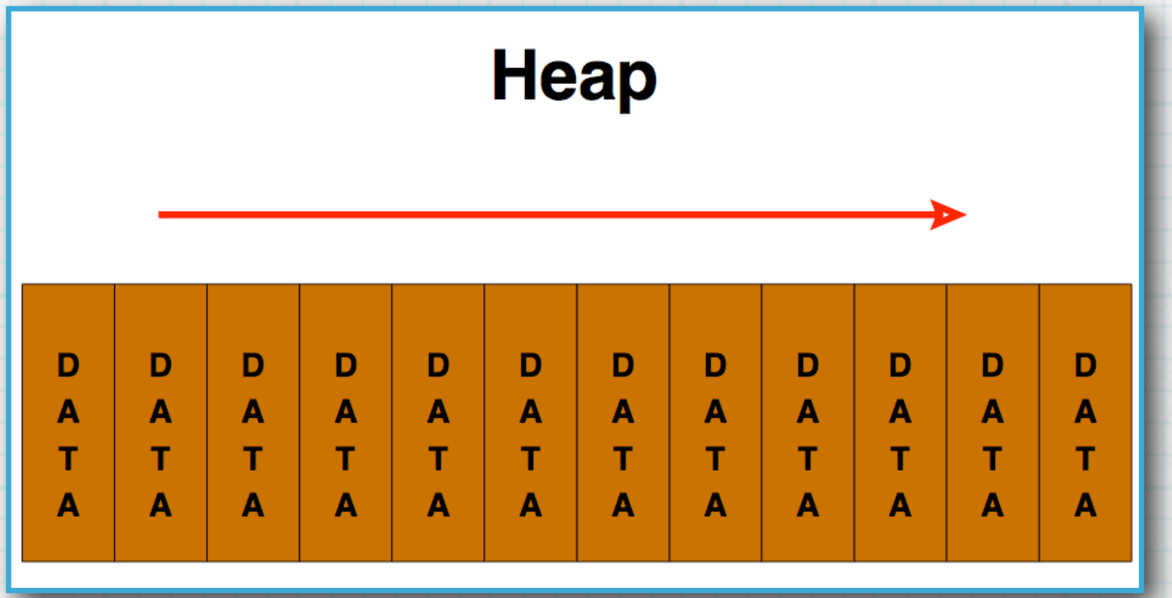
Condition	*Node*
id <= 5	0
id > 5 and id <= 10	1
id > 10	2

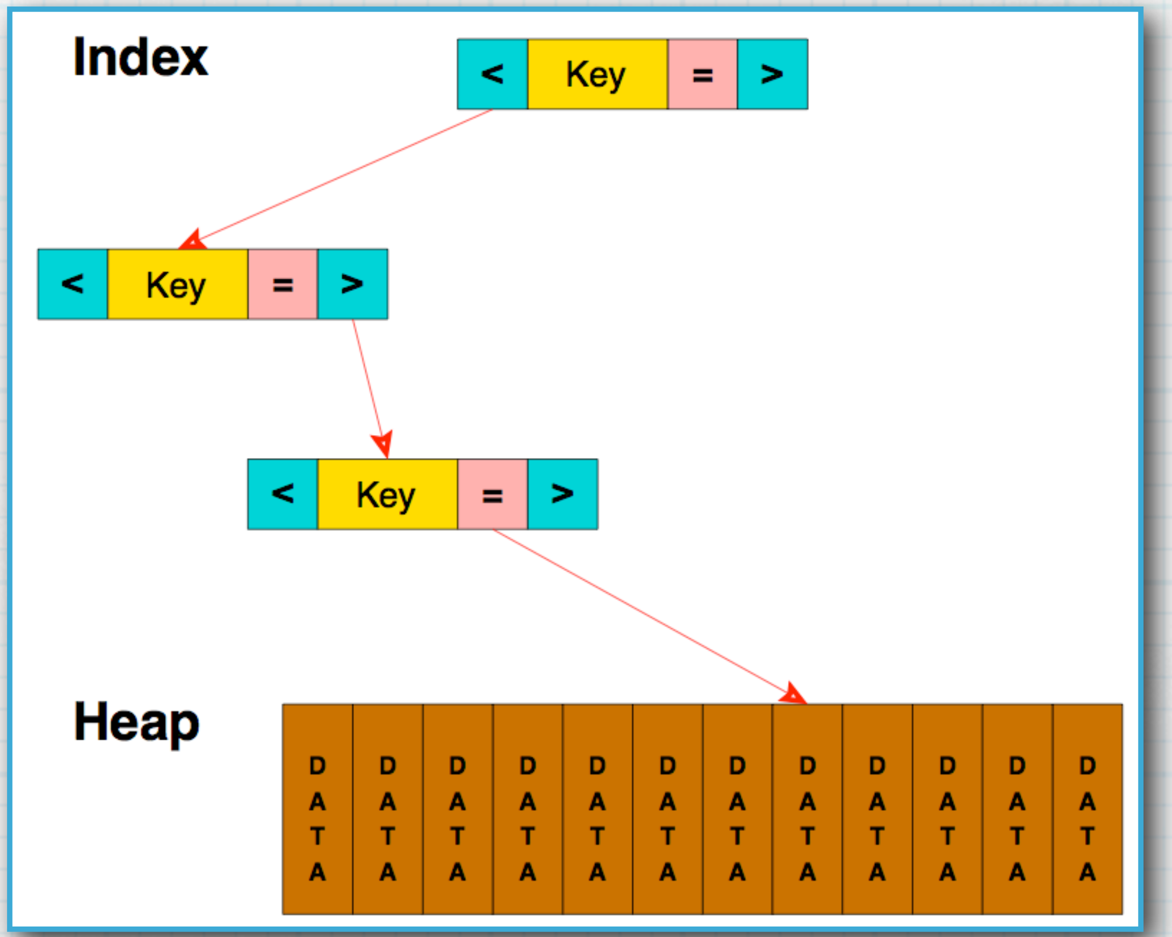
Software Architecture

- Storage Organization
  - Partition data is stored in the system catalog via the Catalog Manager.
  - This partition data is used in query optimization and planning.
  - Indexes are supported -- both clustered and non-clustered -- and are used in query optimization and planning.

Software Architecture

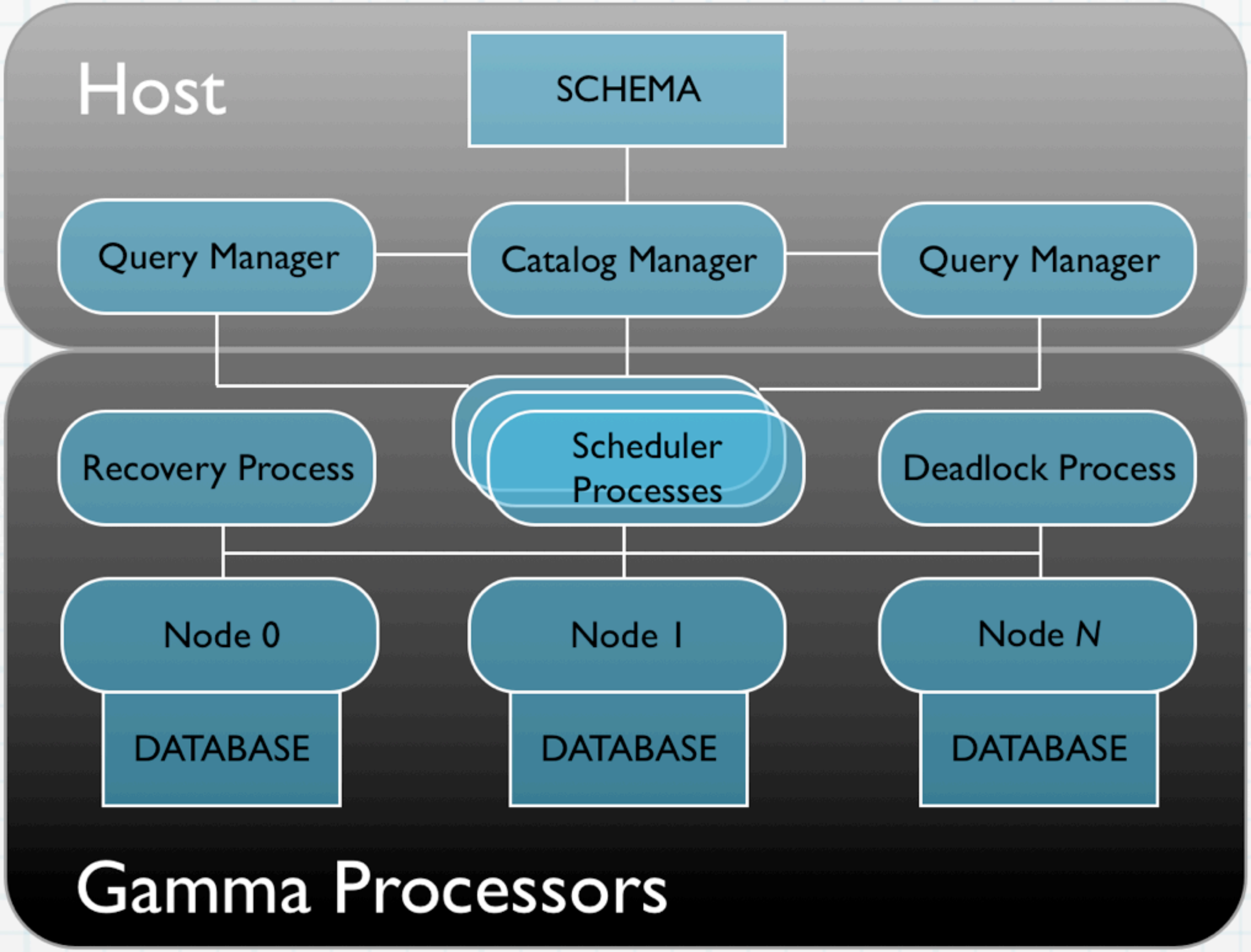
- **Indexes** [4]
  - No Index
    - Scan the data
  - With Index
    - Clustered (not pictured)
    - Non-clustered B-Tree





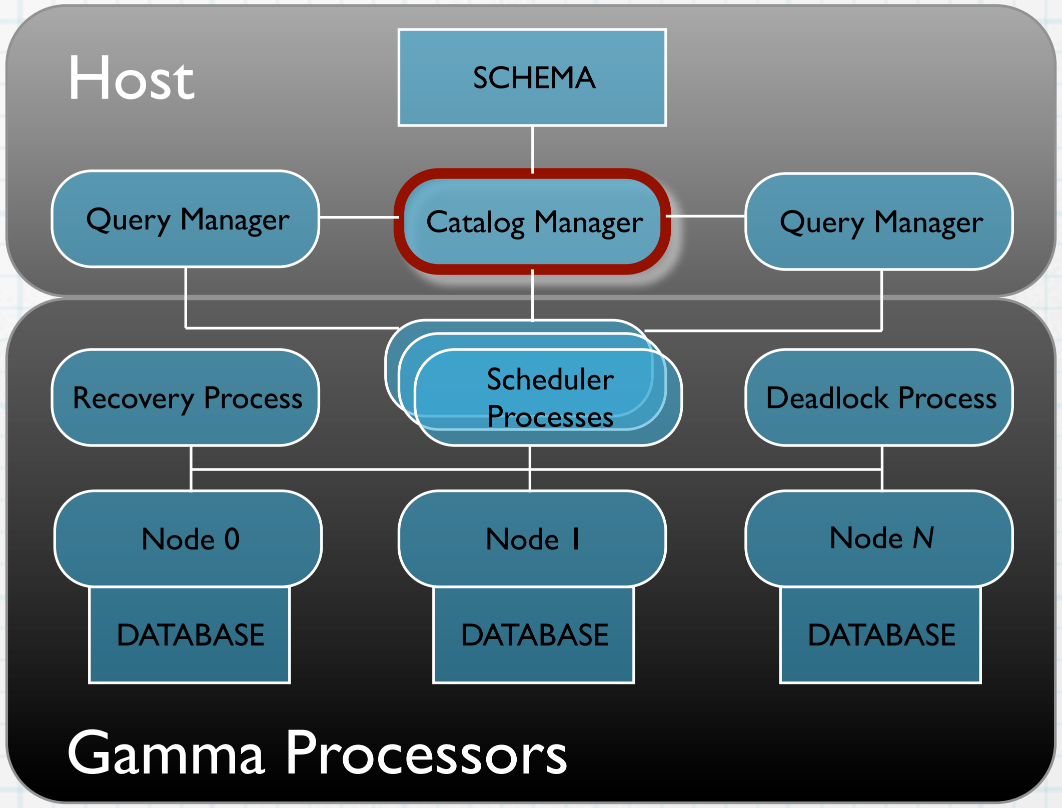
Software Architecture

- Gamma's Process Structure



Software Architecture

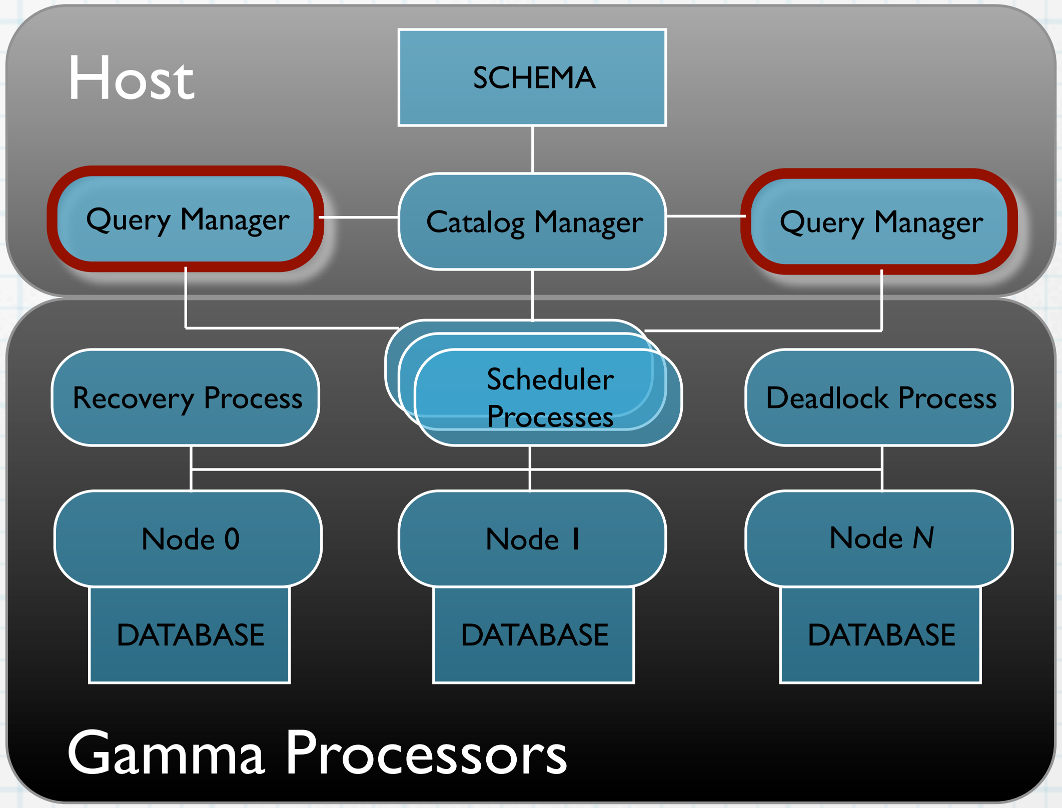
- Catalog Manager
  - Central repository for all schema and partition data.
  - Loaded when database is started.



  - Ensures consistency among cached copies elsewhere.

Software Architecture

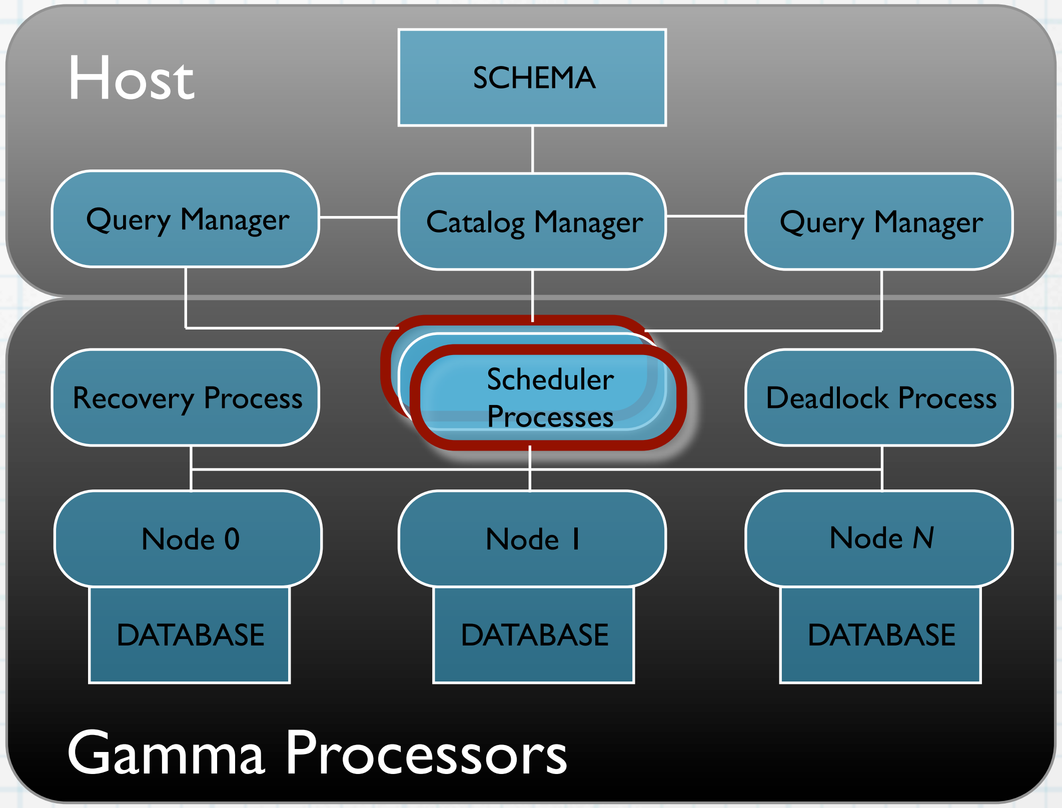
- Query Manager
  - Each user gets a Query Manager process.
  - Locally caches schema data.



  - Provides interface for ad-hoc queries
  - Performs query parsing, optimization, planning, and compilation.

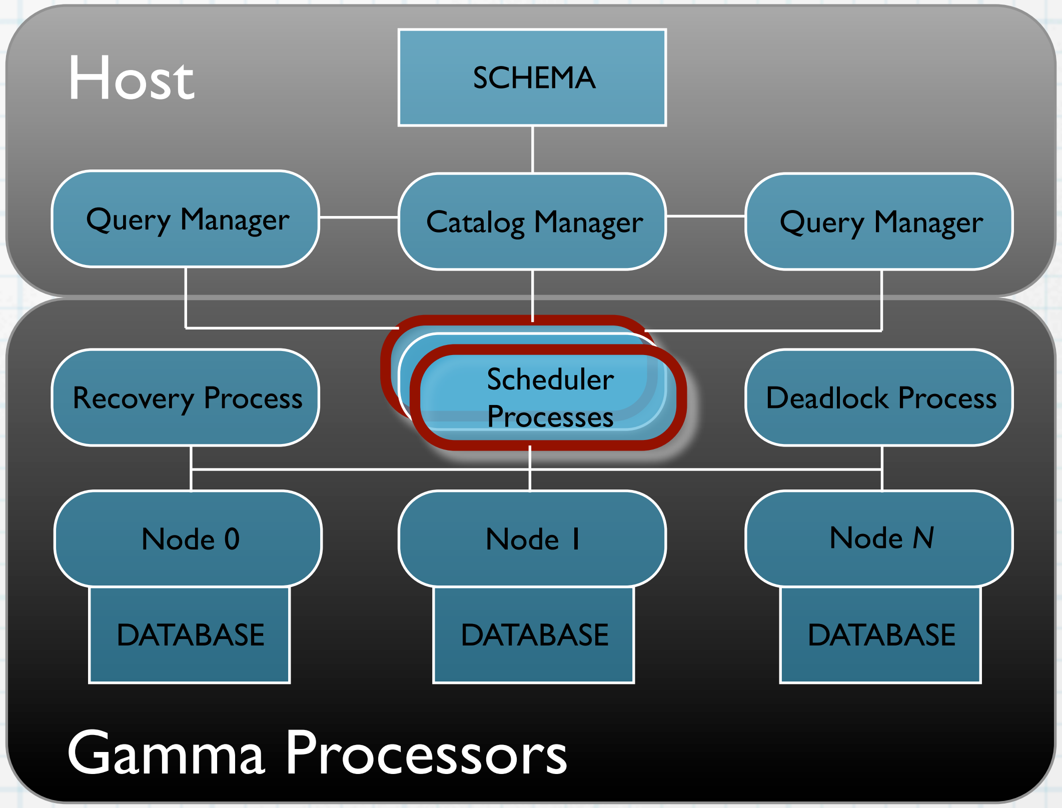
Software Architecture

- Scheduler Processes
  - Each query is controlled by a scheduler process.
  - Activates operator processes on participating nodes.
  - They can be run on any node, ensuring that none becomes a bottleneck.



Software Architecture

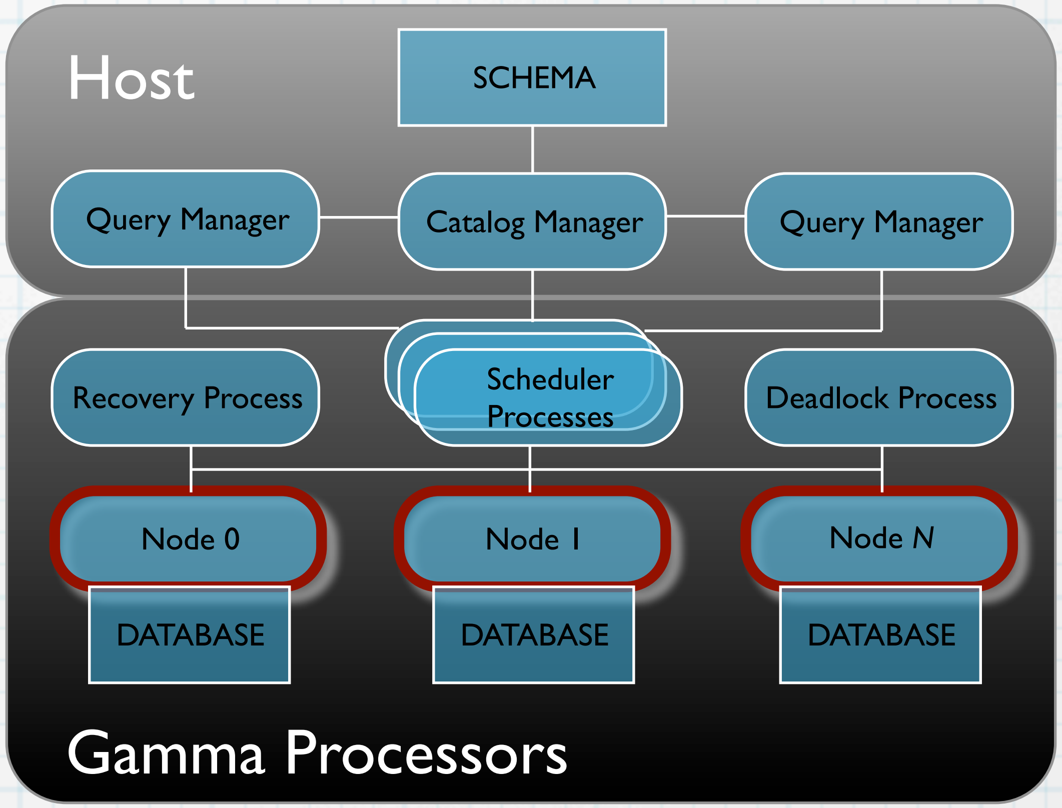
- Scheduler Processes
  - If the Query Manager/optimizer notes that a query requires only a single site it is sent to the appropriate node for execution.



  - In that case the scheduler processes are bypassed.

Software Architecture

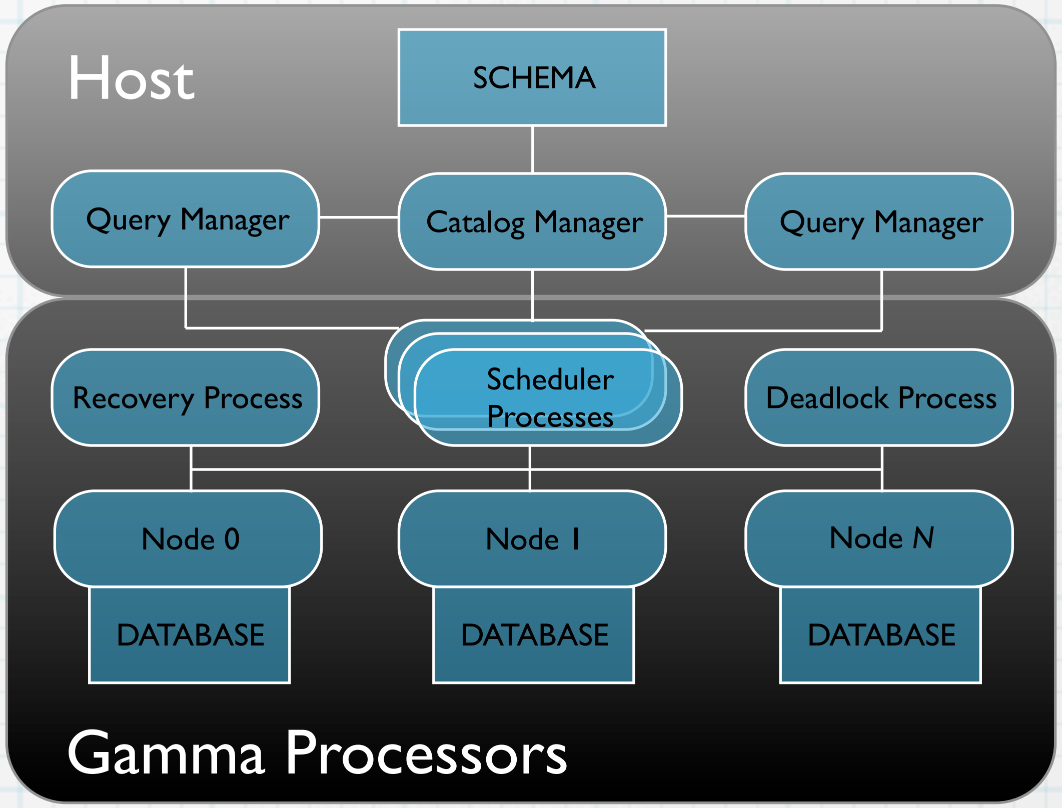
- Execution/Operator Processes
  - There is one operator process for every relational operator (select, join, etc.) in the compiled query.



  - The scheduler spreads these out over the nodes participating in the query execution.

Software Architecture

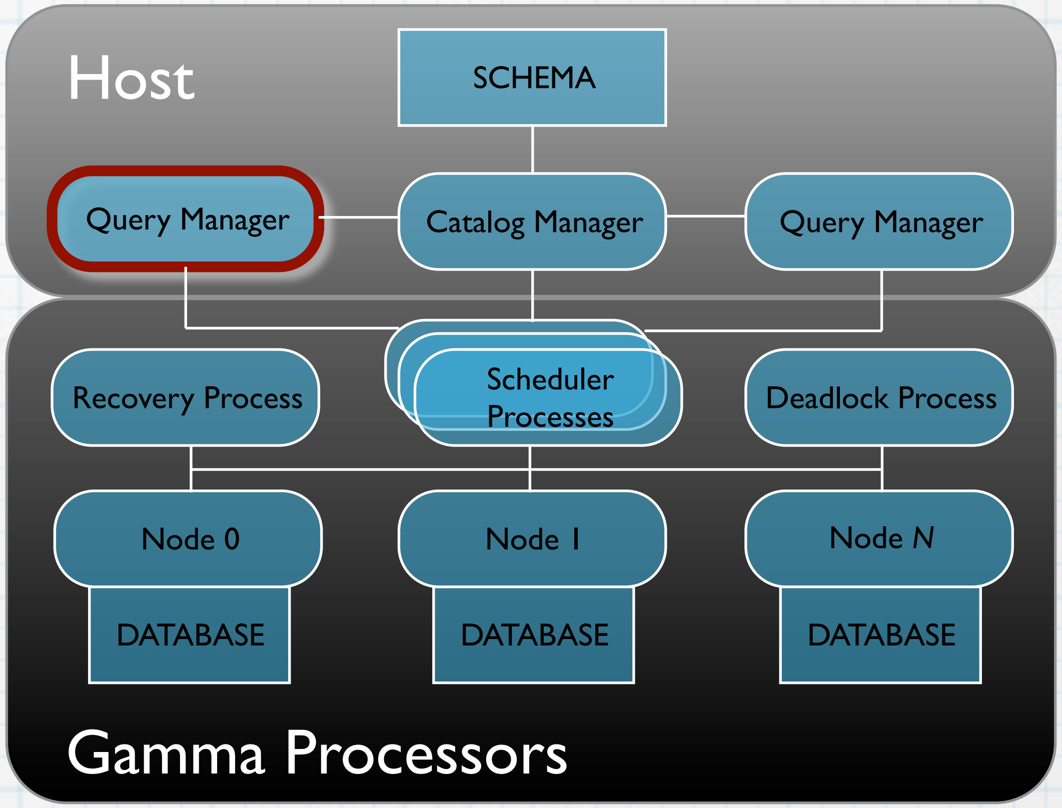
- Query Execution Overview
  - User invokes ad-hoc query interface.
  - Range of u is users
    Retrieve u.name
    Where u.clue > 0



Hey... What language is that?

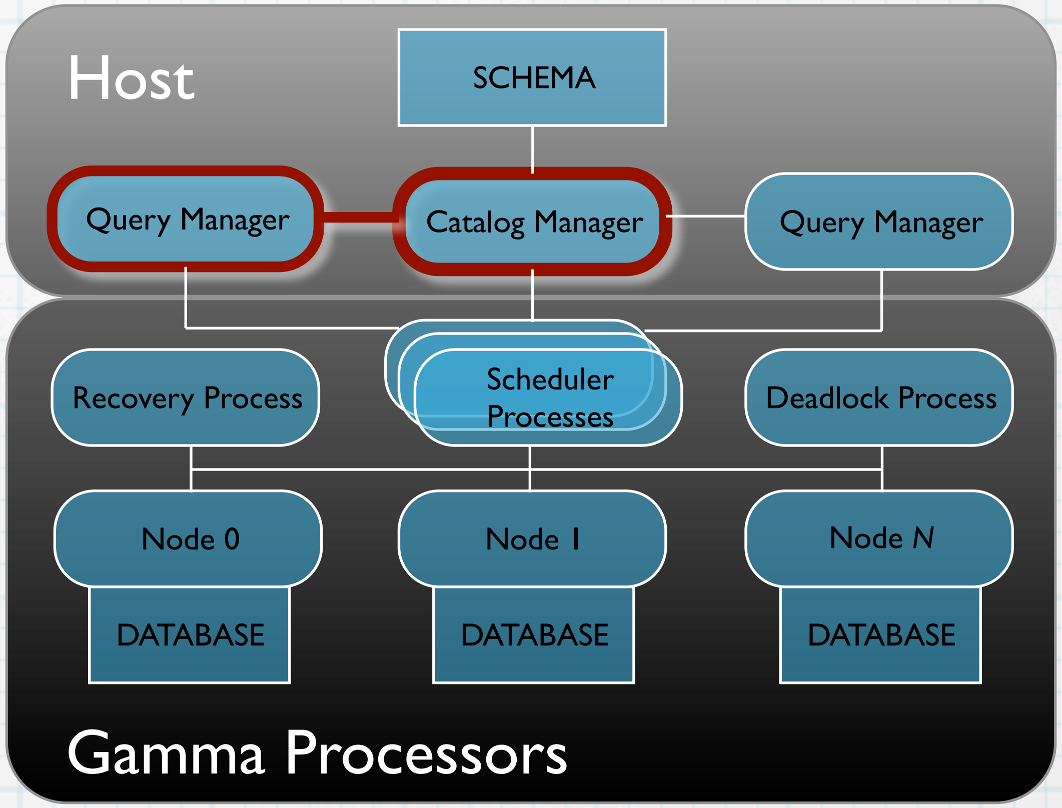
Software Architecture

- Query Execution Overview
  - A Query Manager process starts



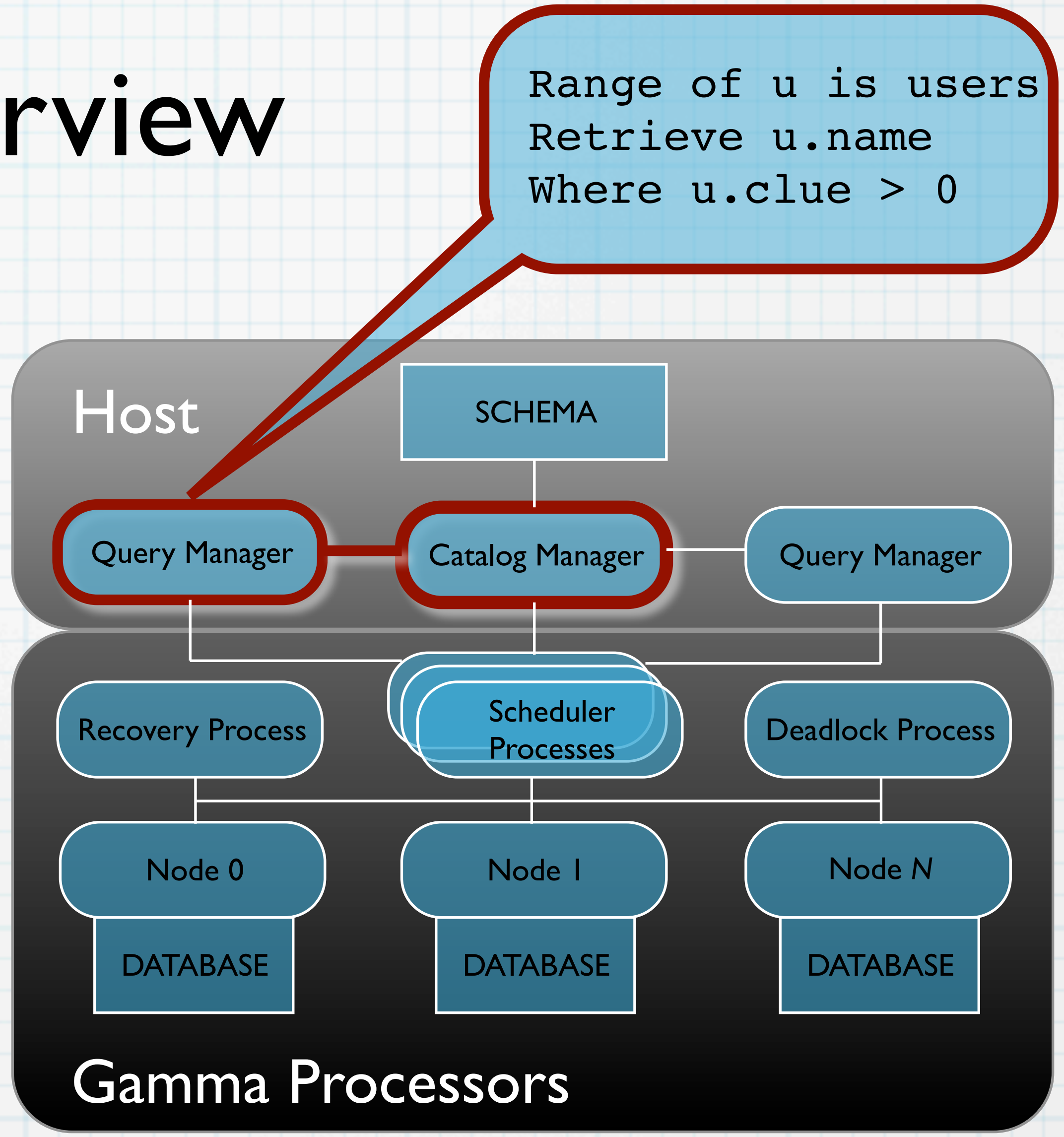
Software Architecture

- Query Execution Overview
  - A Query Manager process starts,
  - connects itself to the Catalog Manager process



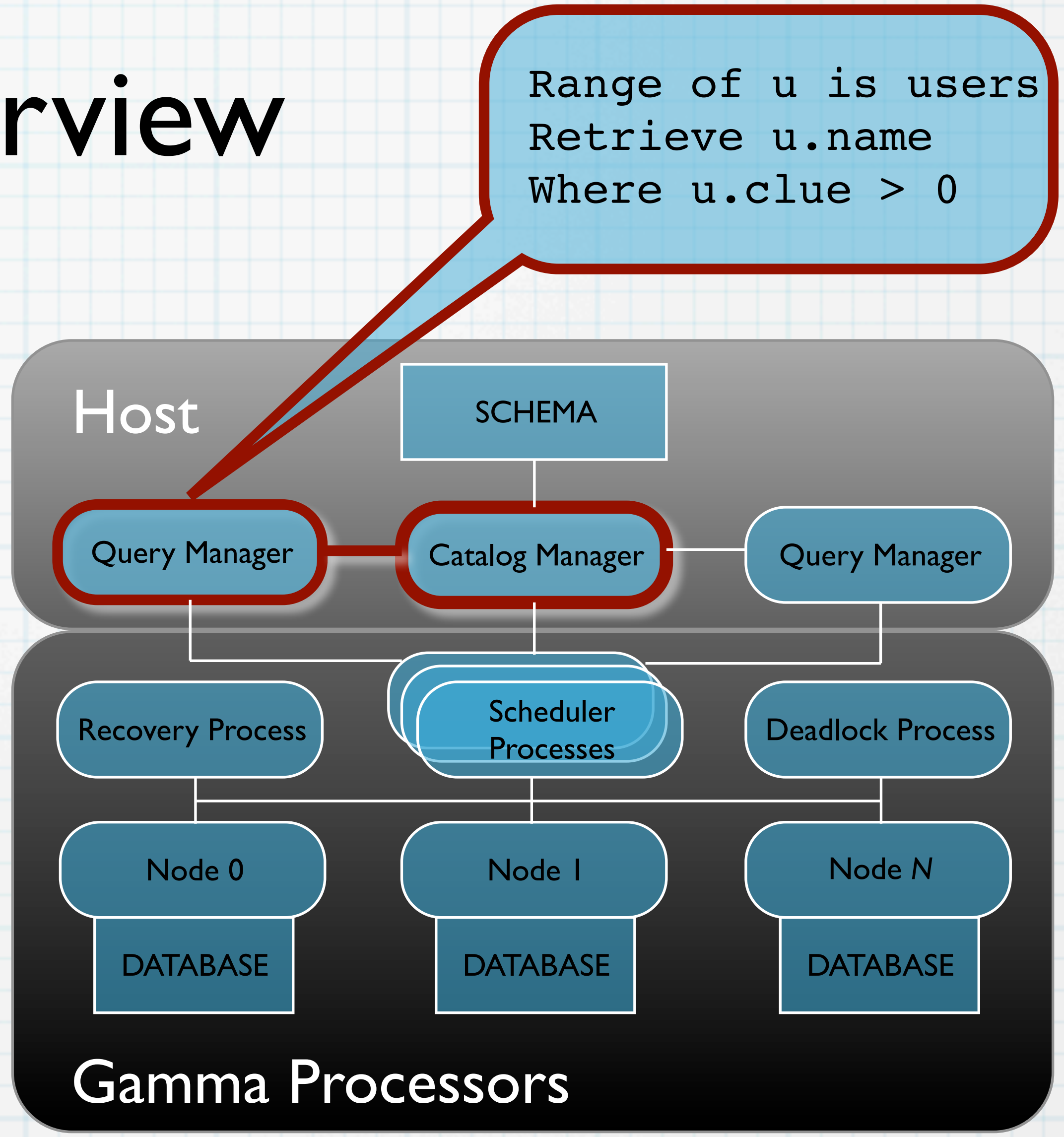
Software Architecture

- Query Execution Overview
  - A Query Manager process starts,
  - connects itself to the Catalog Manager process,
  - and gets to work on the query.



Software Architecture

- Query Execution Overview
  - The Query Manager does...
    - parsing
    - optimization
    - planning



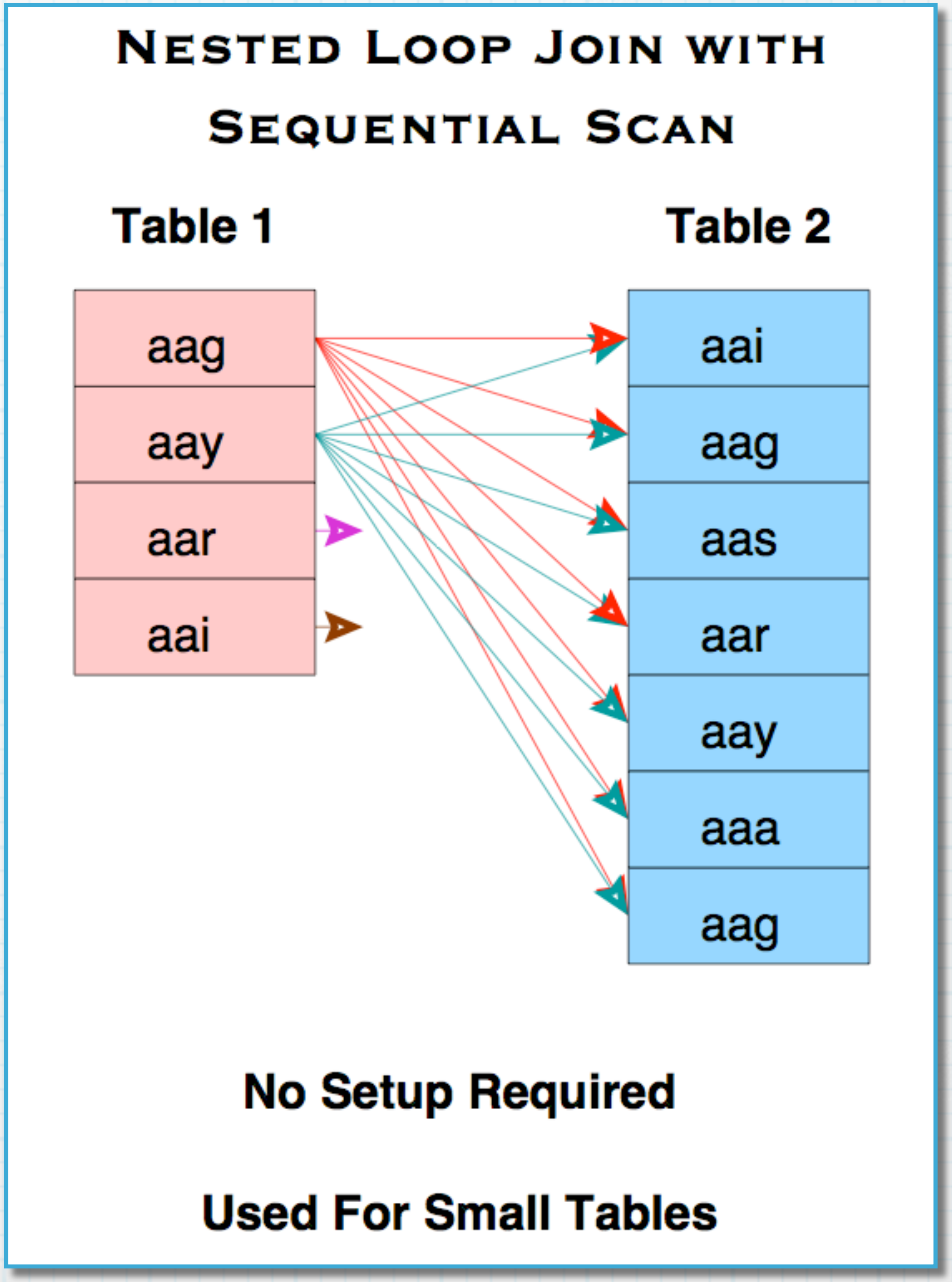
  - ... in the traditional relational ways,
  - but with only **hash-based joins**.

Software Architecture

- Aside: Three Common Join Types
  - the Nested-Loop join
  - the Merge join
  - the Hash join

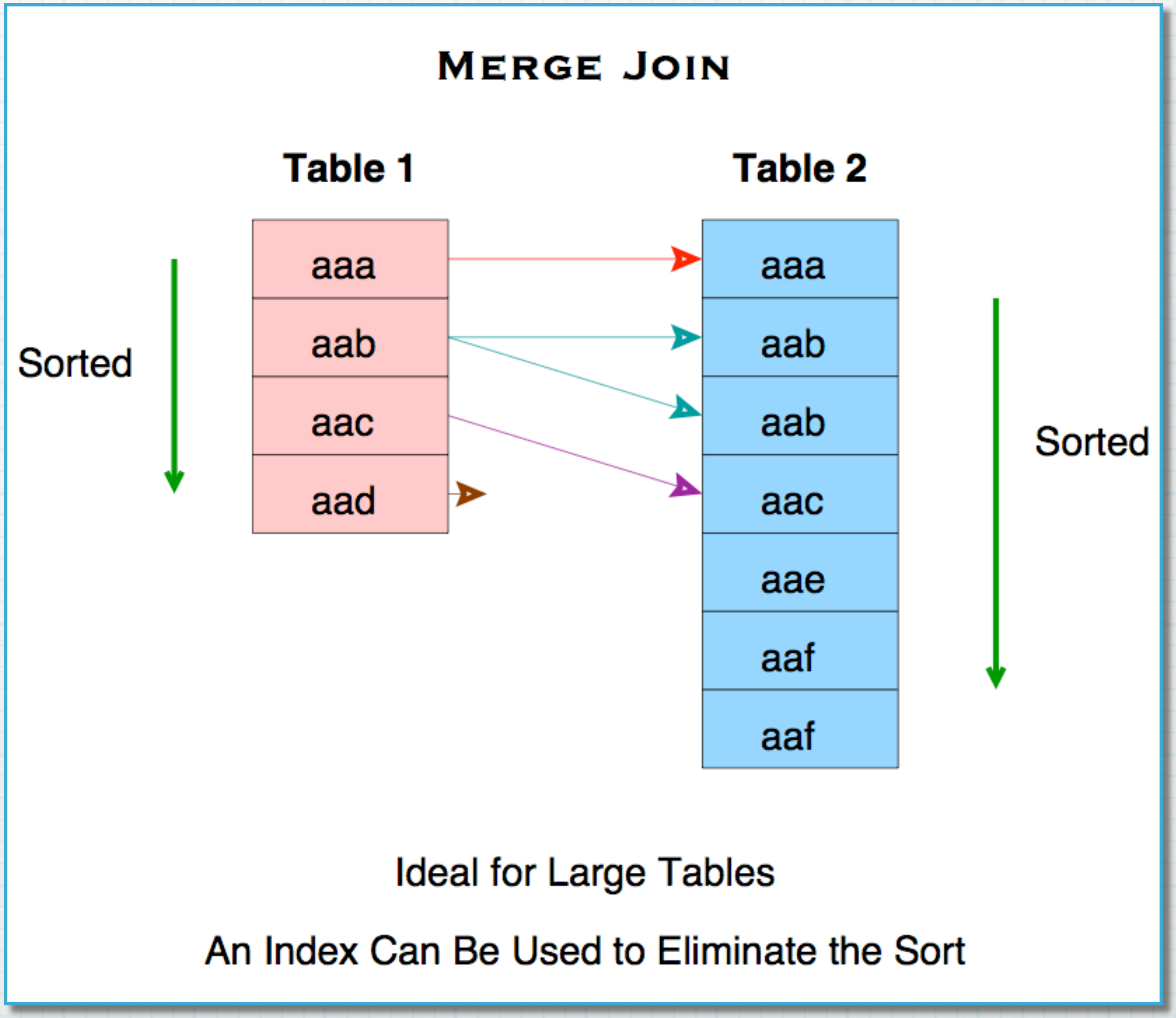
Software Architecture

- Aside: the Nested Loop Join [4]



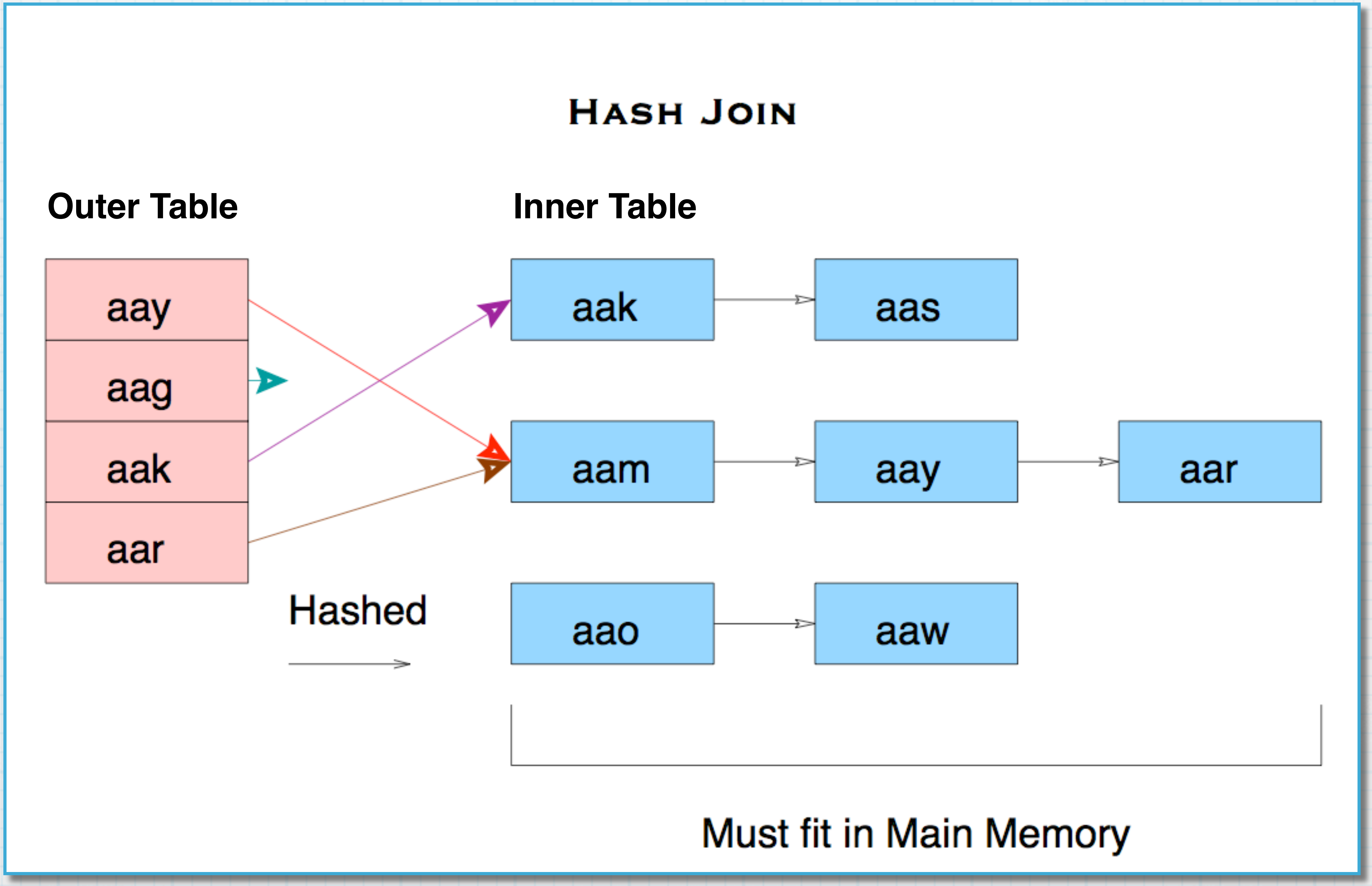
Software Architecture

- Aside: the Merge Join [4]



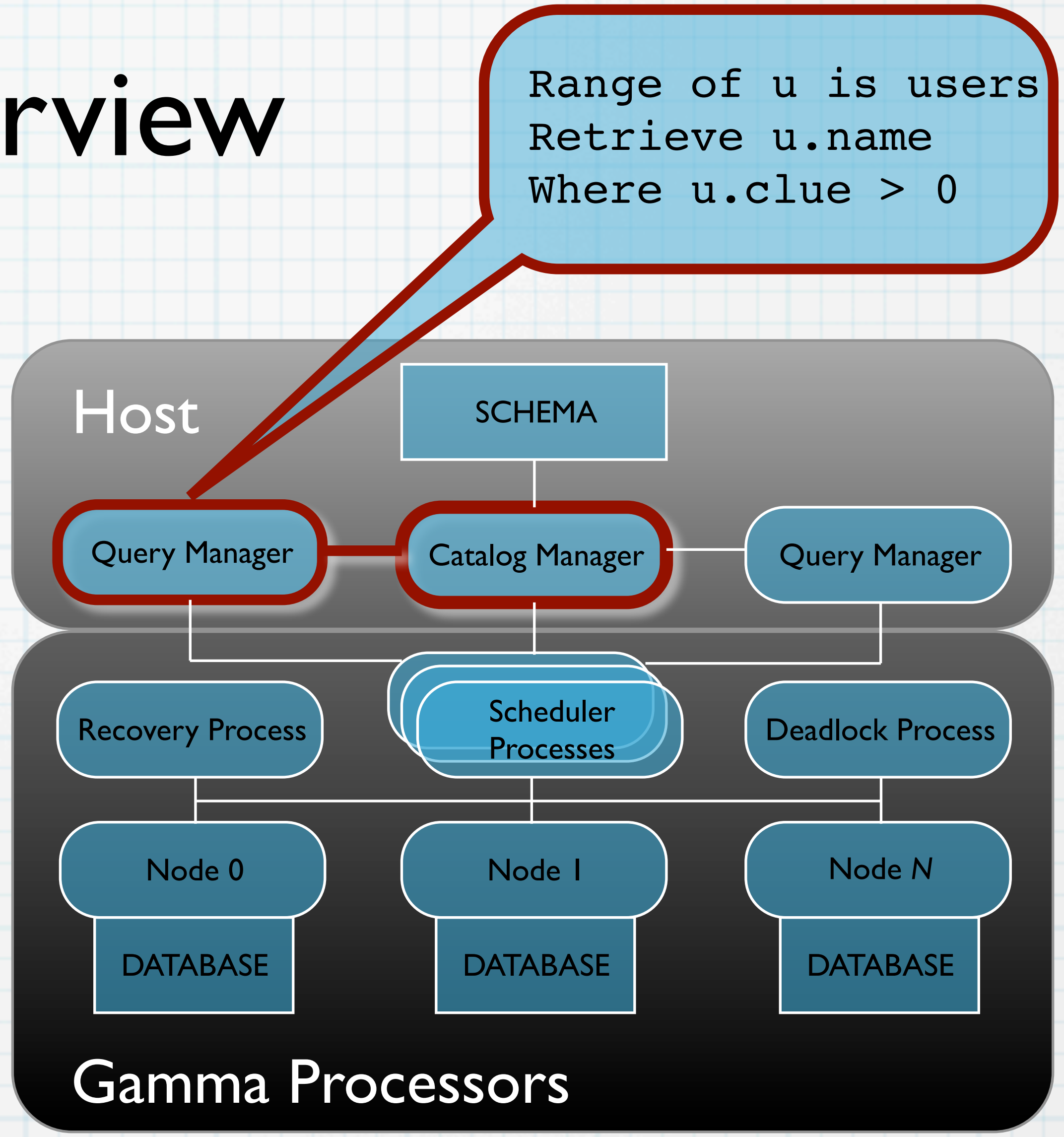
Software Architecture

- Aside: the Hash Join [4] - Gamma's Join



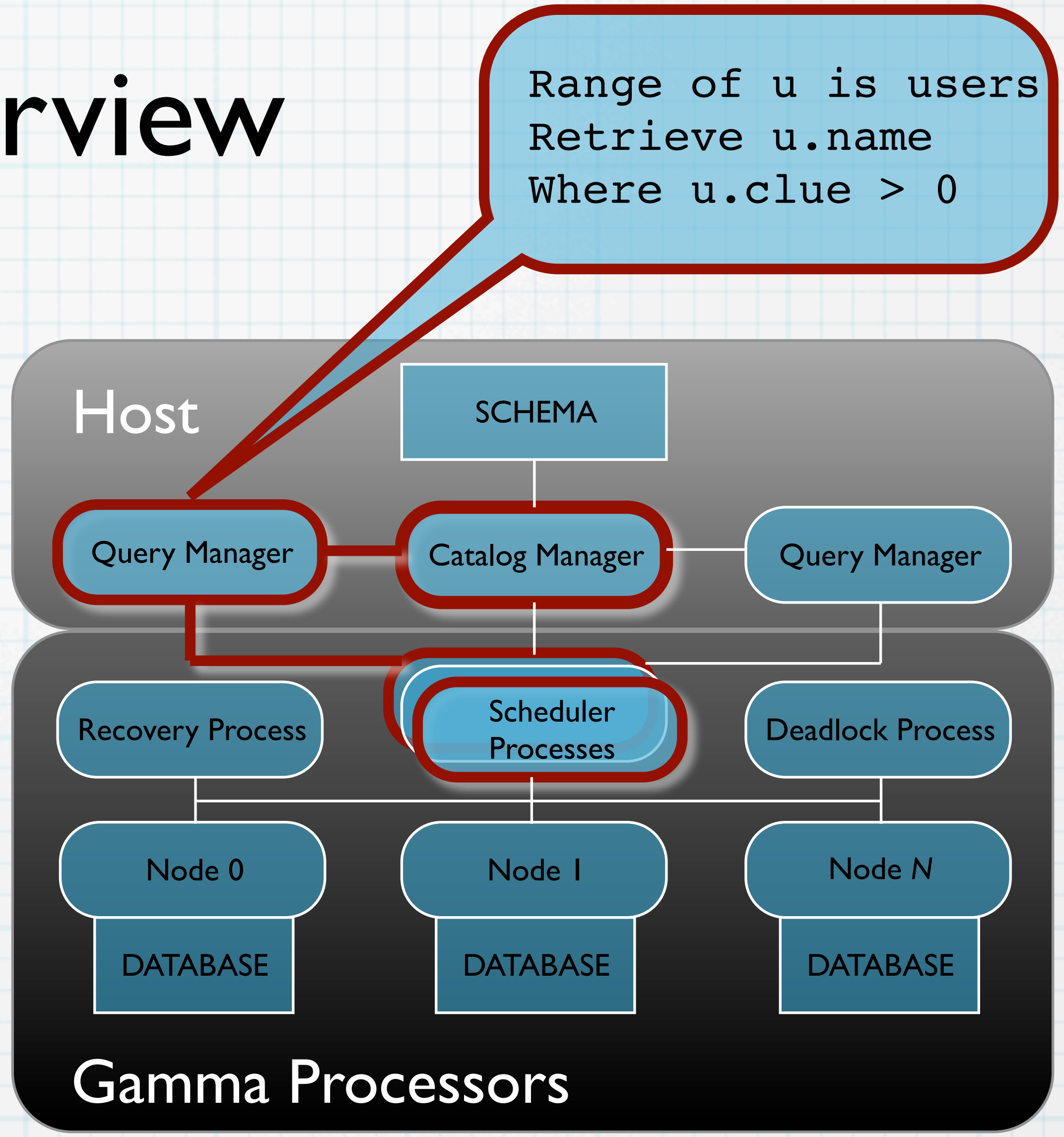
Software Architecture

- Query Execution Overview
  - The Query Manager does...
    - parsing
    - optimization
    - planning
  - ... in the traditional relational ways,
  - but with only **hash-based joins**.



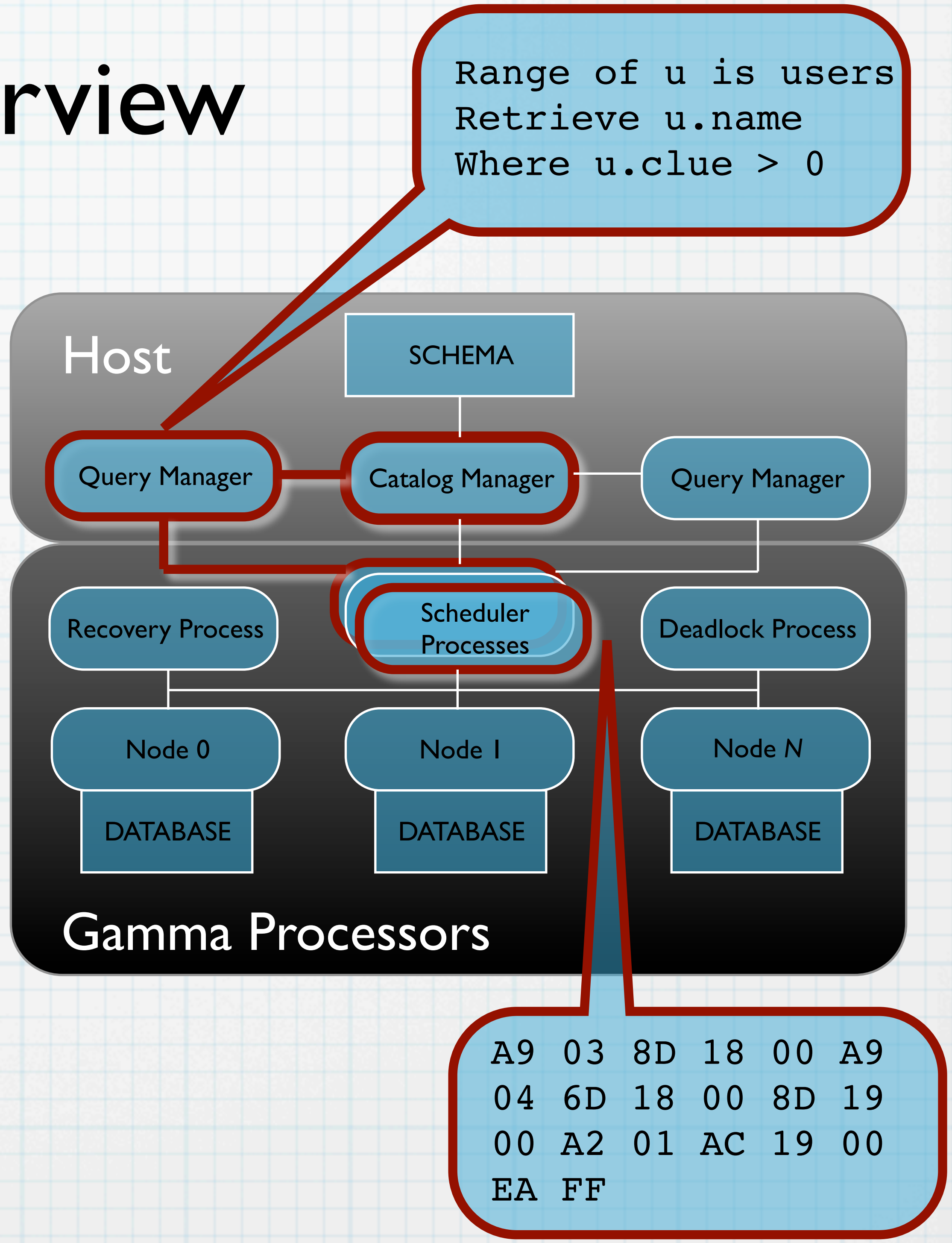
Software Architecture

- Query Execution Overview
  - Now the Query Manager connects to an idle scheduler



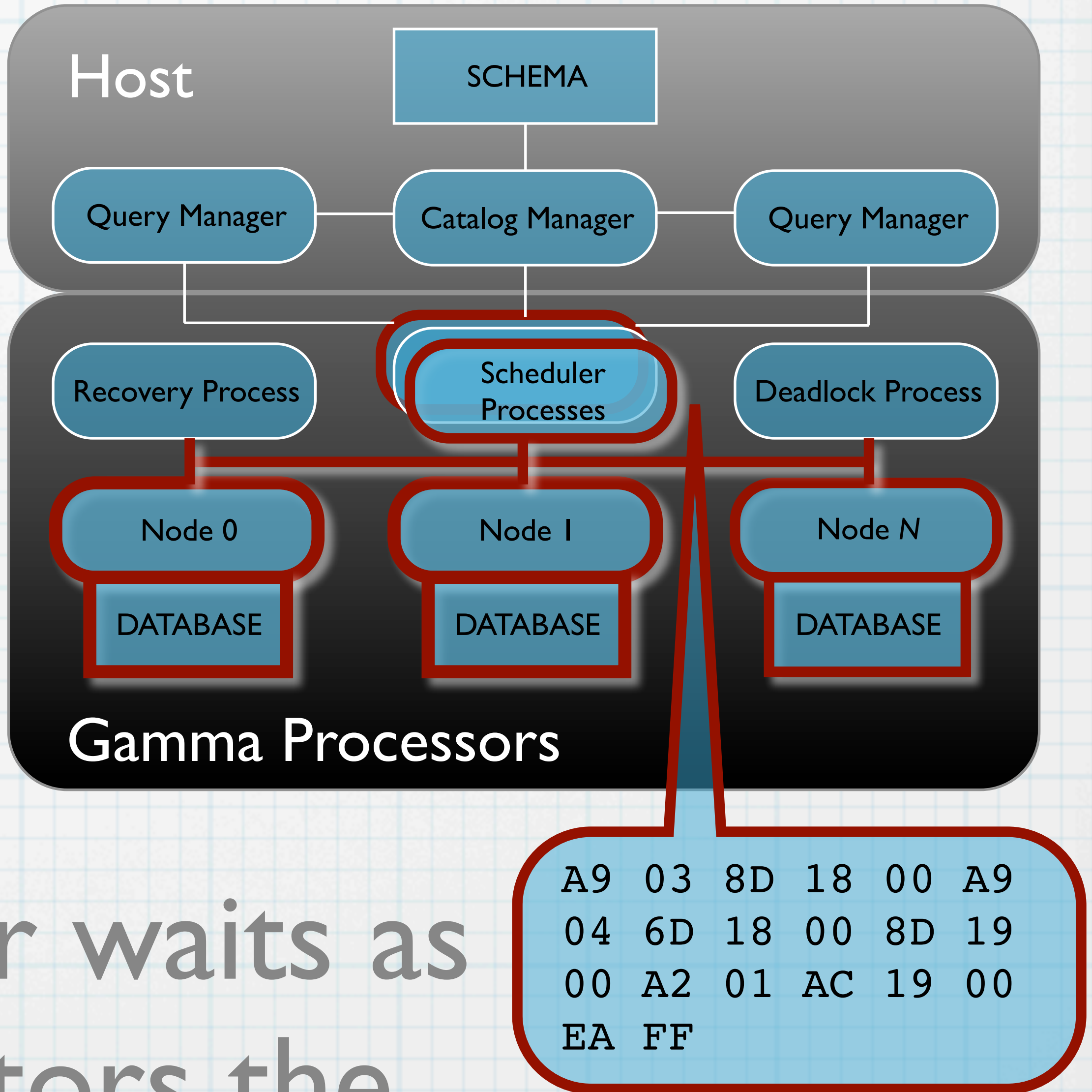
Software Architecture

- Query Execution Overview
  - Now the Query Manager connects to an idle scheduler,
  - and sends it the planned, compiled query.



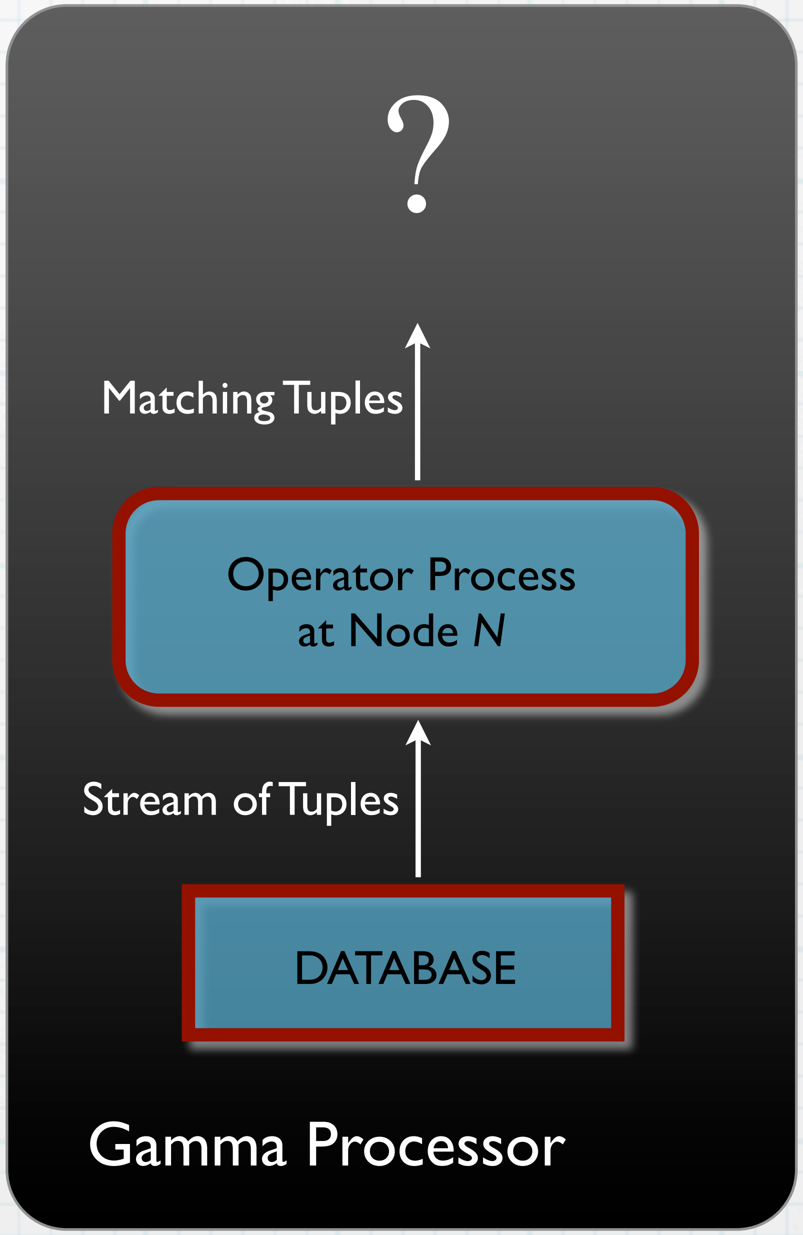
Software Architecture

- Query Execution Overview
  - The scheduler activates operator processes (select, join, etc.) at various nodes to execute the query.
  - The Query Manager waits as the scheduler monitors the progress.



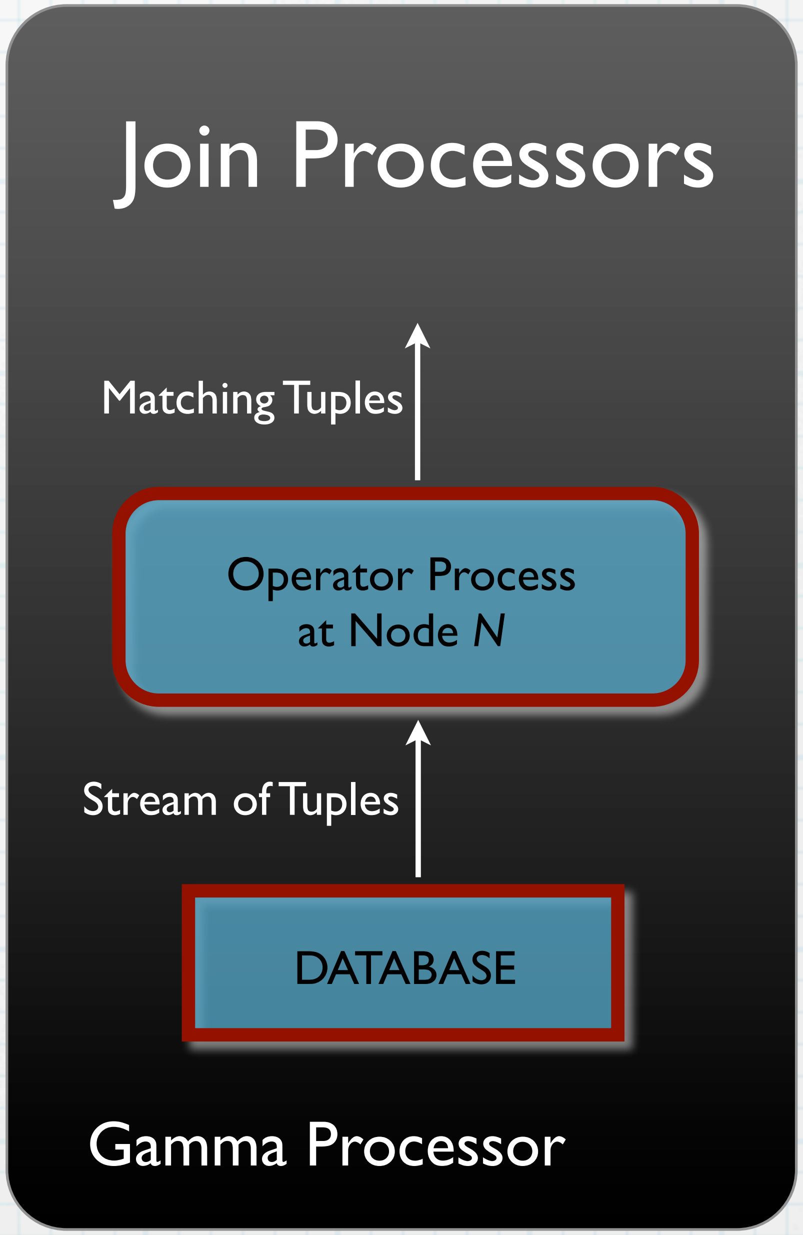
Software Architecture

- Query Execution Overview
  - Each participating operator process reads tuples from the database at its node,
  - performs its operation (index select, scan, etc.)
  - and sends the matching tuples . . . somewhere?



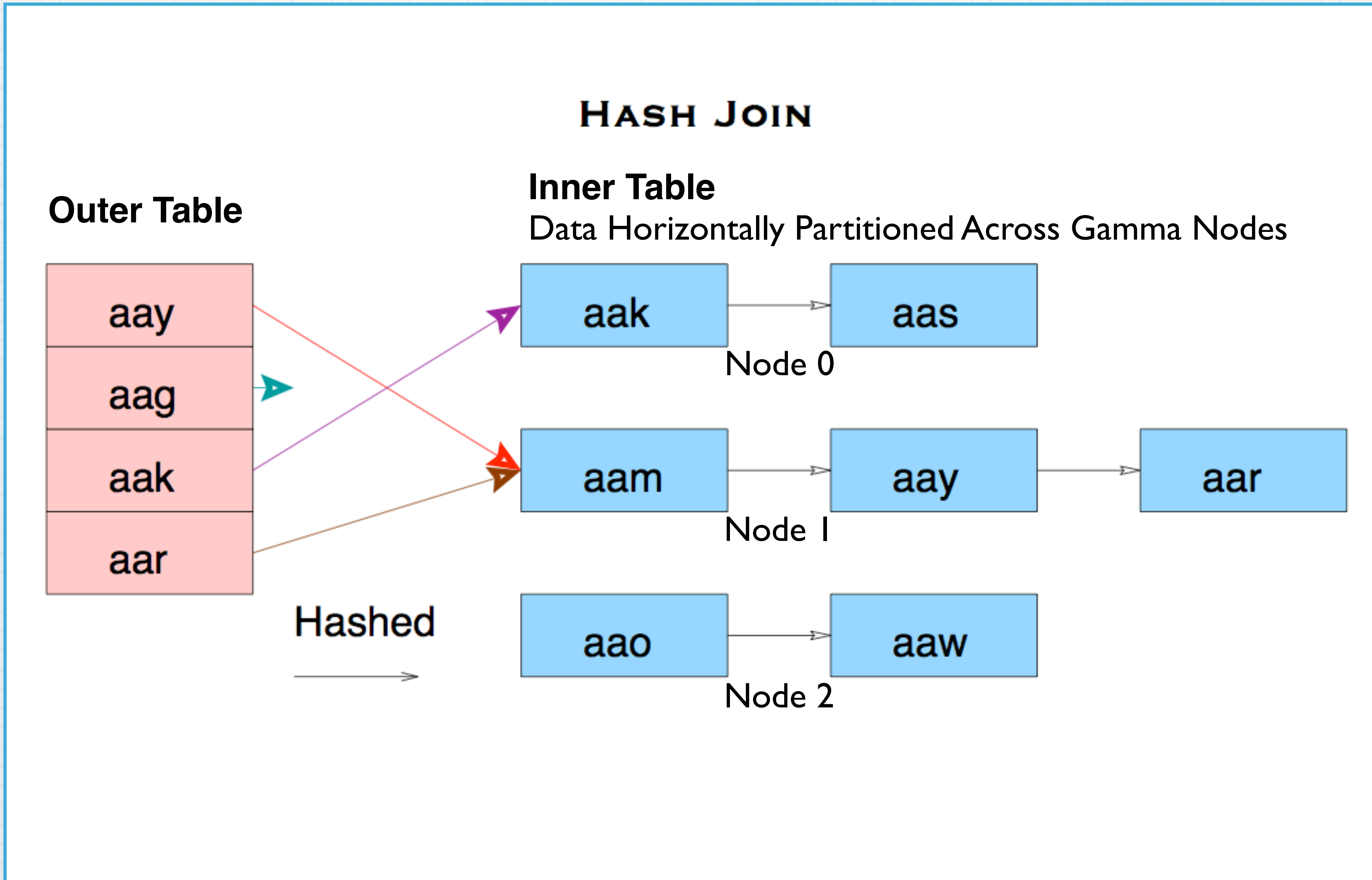
Software Architecture

- Query Execution Overview
  - If we're doing a join, then there are other processes available to help with the join.
  - But who gets what?
  - How do we parallelize the work of a join?
  - Remember the Hash Join?



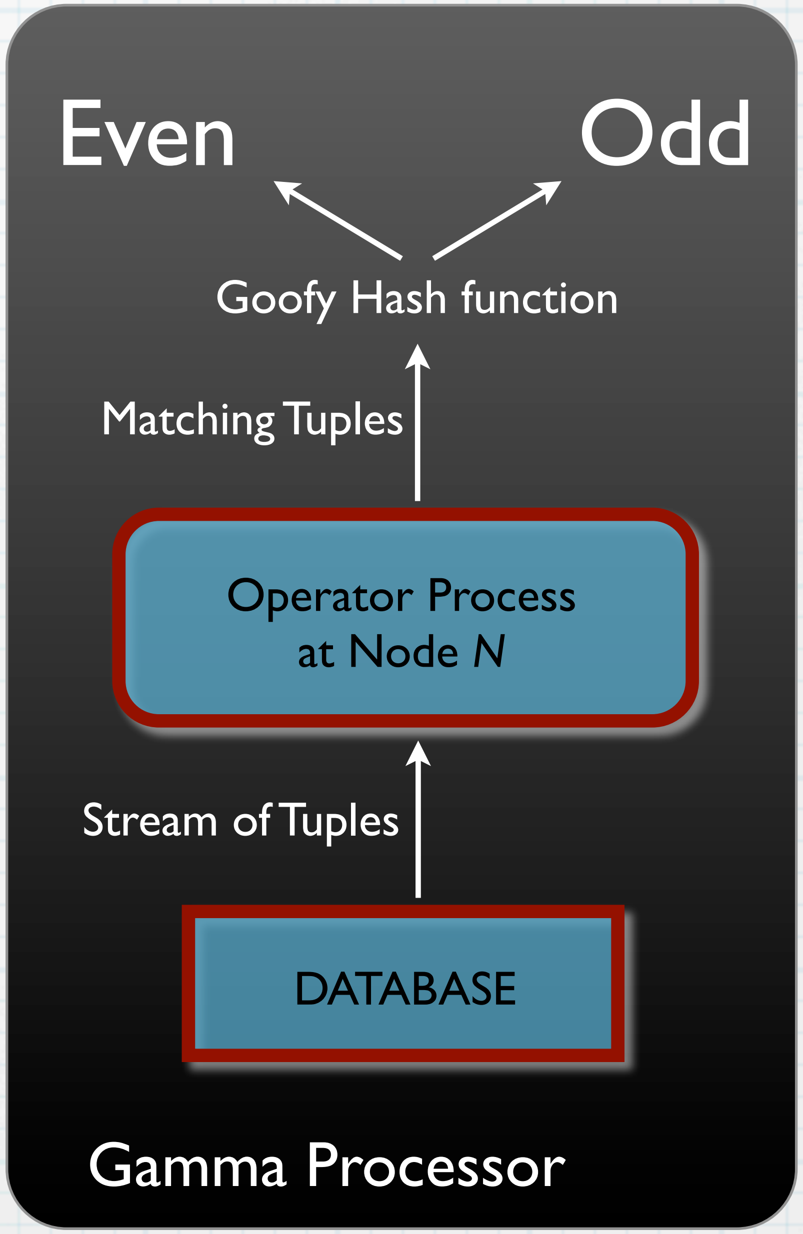
Software Architecture

- Gamma's Hash Join [4] modified



Software Architecture

- Query Execution Overview
  - The operator process performs a **hash** on the **join attribute** of each resulting tuple,
  - and sends it to the appropriate join node.
  - But where is that node?

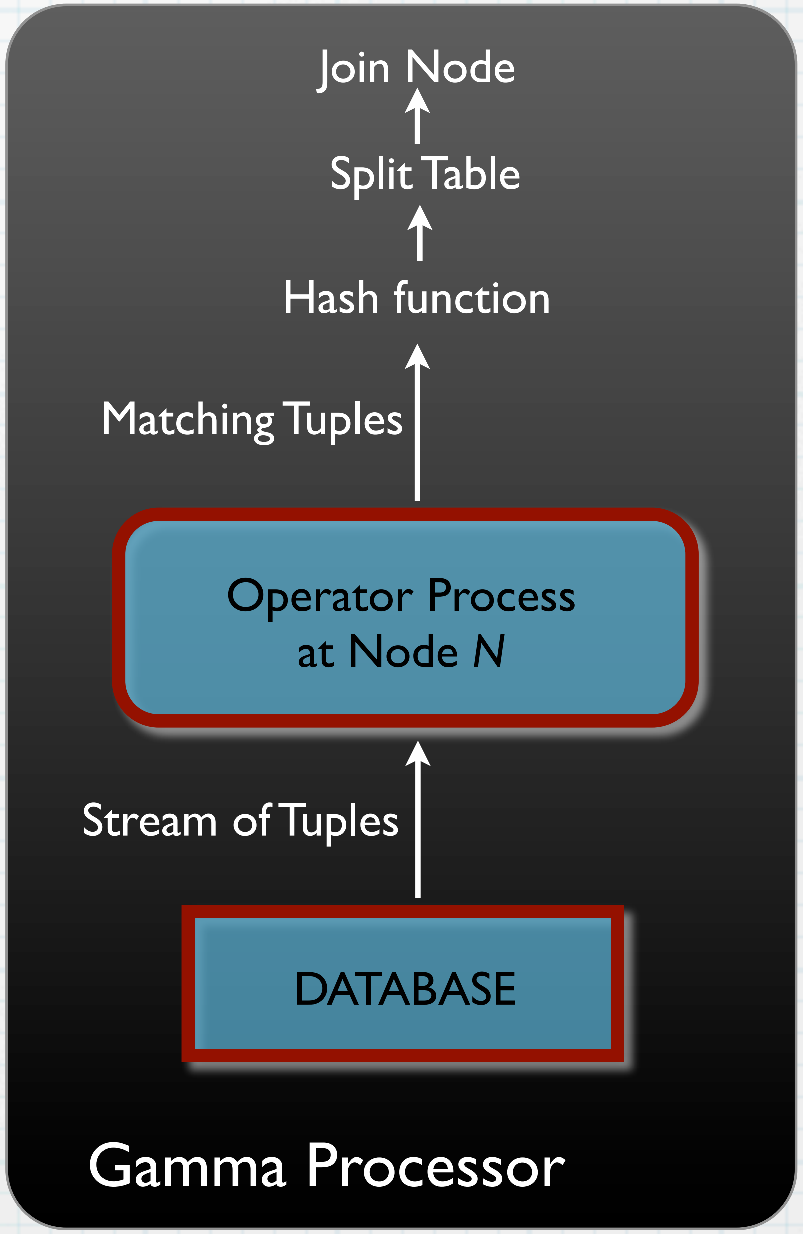


Software Architecture

- Query Execution Overview
  - Gamma builds **Split Tables** to demultiplex matching tuples to join operator processes.

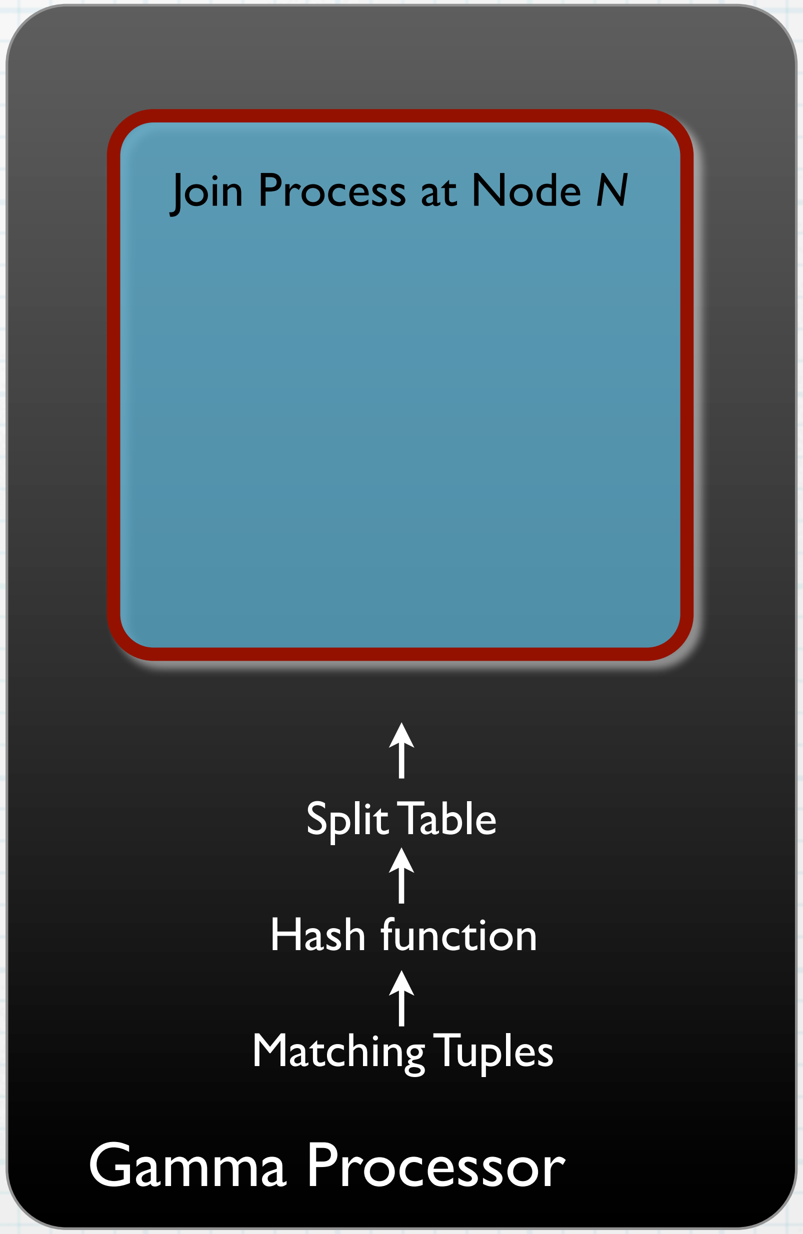
Split Table

Value	*Destination Process*
Even	0
Odd	1



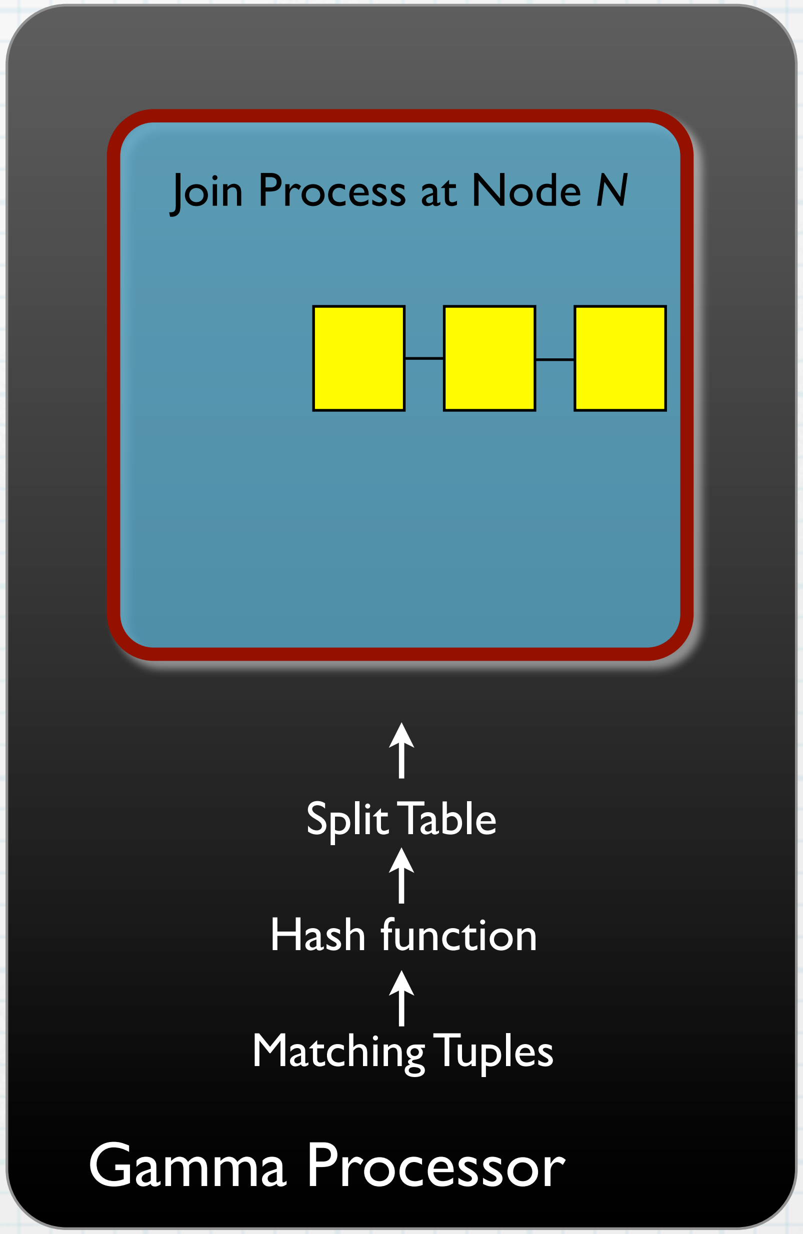
Software Architecture

- Query Execution Overview
  - Each join process operates in two phases (controlled by the scheduler)
    - Building Phase
    - Probing Phase



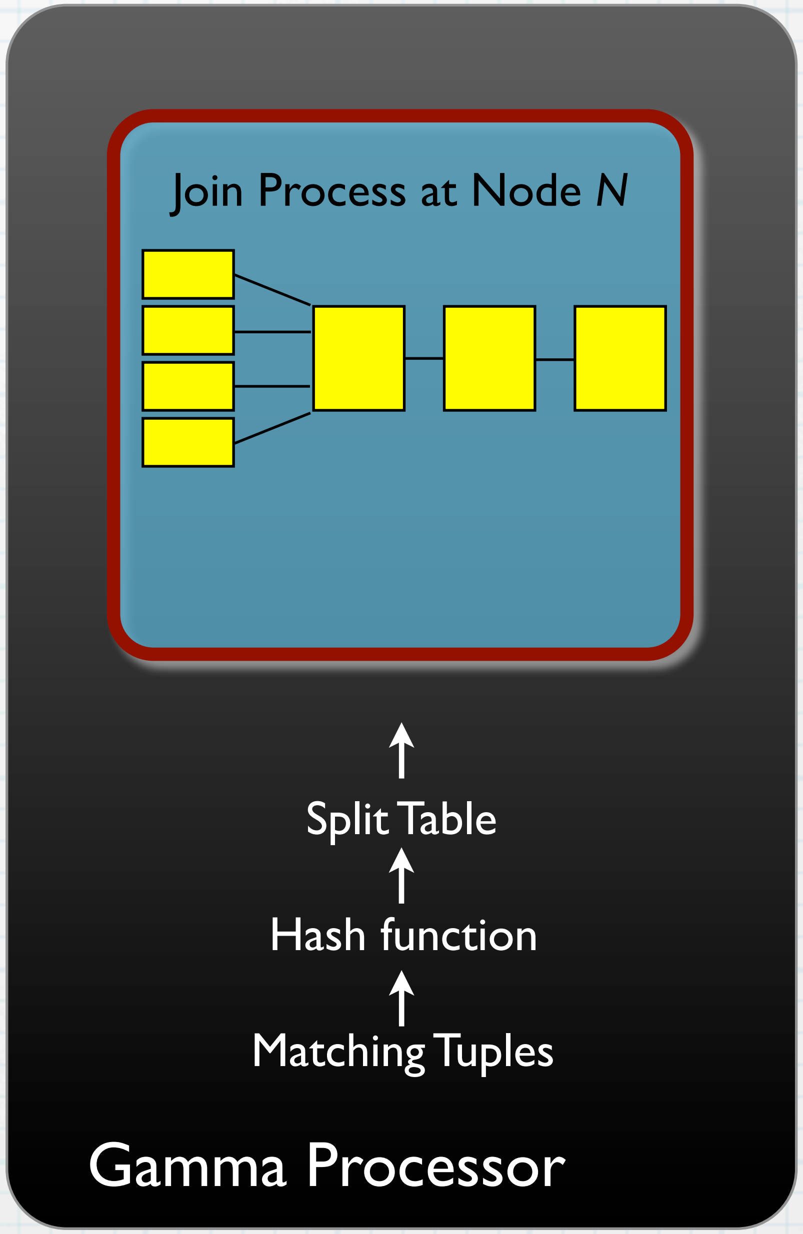
Software Architecture

- Query Execution Overview
  - Each join process operates in two phases:
    - Building Phase
      - An in-memory hash table is built for the join's inner table.



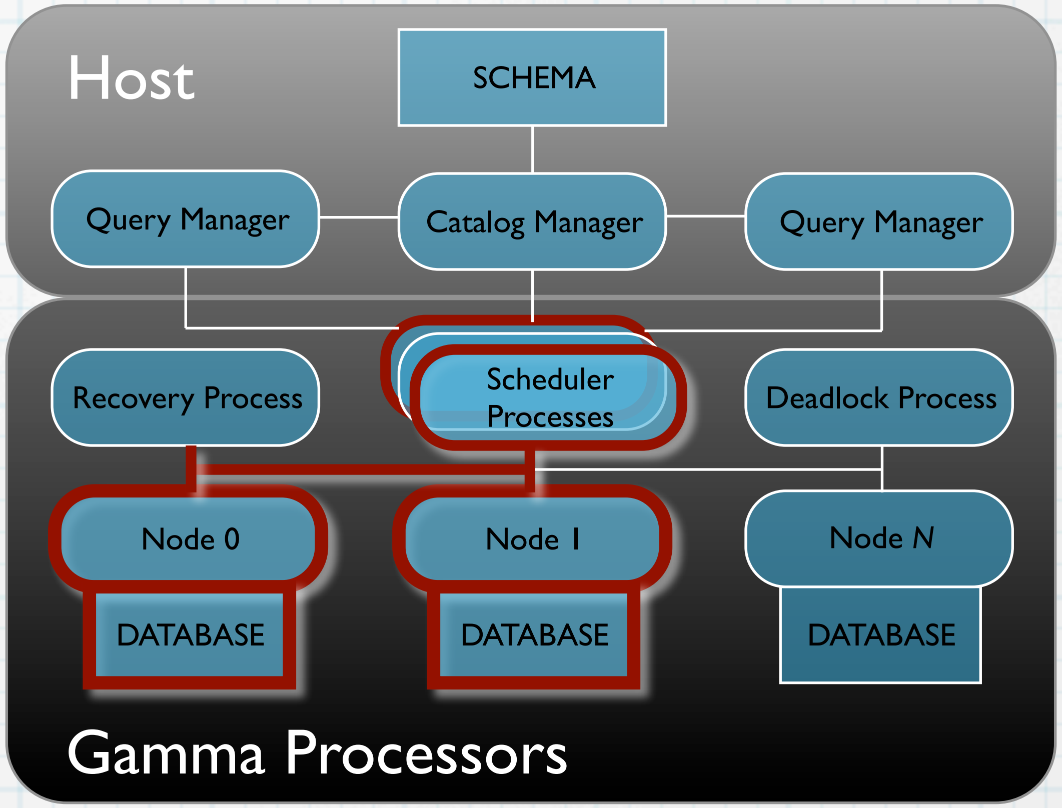
Software Architecture

- Query Execution Overview
  - Each join process operates in two phases:
    - Building Phase
    - **Probing Phase**
      - Tuples from the **outer table** are used to probe the hash table for matches.



Software Architecture

- Query Execution Overview
  - The scheduler, who has been monitoring and controlling all of this, collects the partial results as the various probing phases complete.

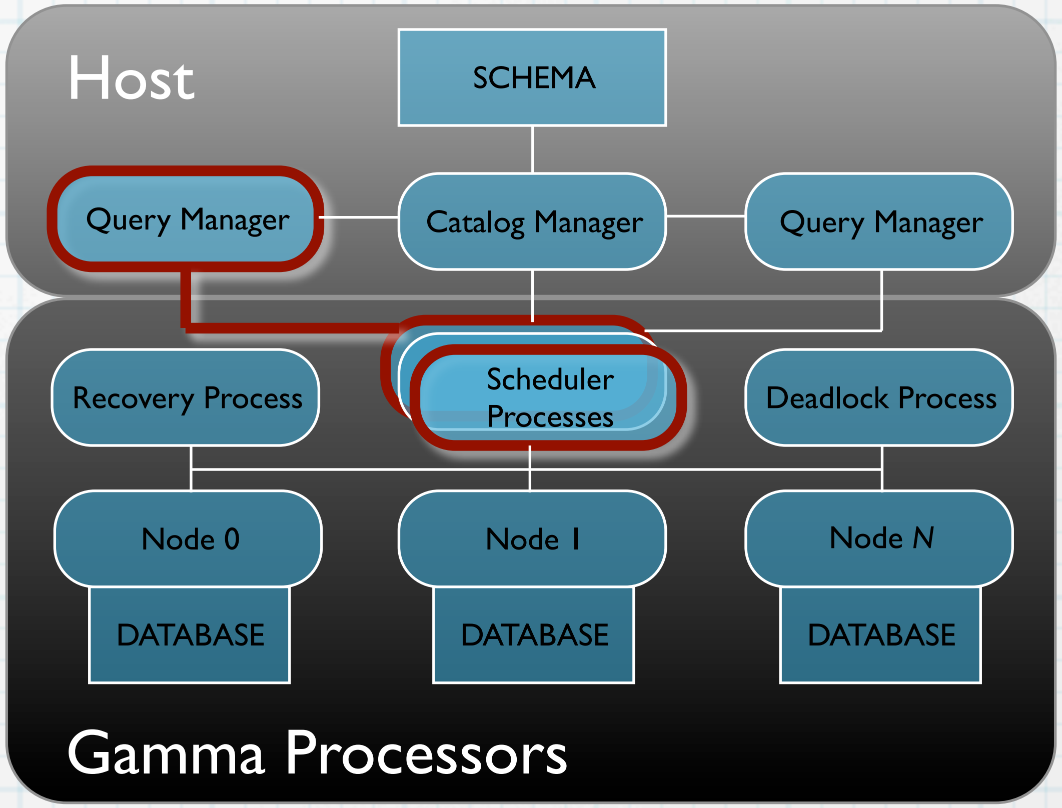


Split Table

Value	*Destination Process*
Even	0
Odd	1

Software Architecture

- Query Execution Overview
  - Finally, the Query Manager reads the combined results from the scheduler and returns them to the user.



  - `Warning: No Rows Selected.`

The Plan

- History
- Hardware Architecture
- Software Architecture
- Query Algorithms
- Transactions
- Performance
- Summary

(That was cool, wasn't it?)

Query Algorithms

- Selection - two cases
  - Selection on a partitioning attribute
    - Scheduler initiates selection operator on a subset of nodes.
  - Selection on a non-partitioning attribute or we used round-robin partitioning in the first place
    - Scheduler initiates the selection operation at all nodes.

Query Algorithms

- Aggregates - sum, min, max, etc.
  - Each participating node maps the aggregate operator to the elements of its portion of the data in parallel.
  - The individual node results are collected (by the scheduler) and combined (reduced) to the final answer.
  - Does this sound familiar?

Query Algorithms

- Aggregates - sum, min, max, etc.
  - Each participating node maps the aggregate operator to the elements of its portion of the data in parallel.
  - The individual node results are collected (by the scheduler) and combined (reduced) to the final answer.
  - Does this sound familiar? It should.

Query Algorithms

- Updates - insert, update, delete
  - Mostly done as typical RDBMS do it.
  - Exception: modifying the partitioning attribute.
    - Use the split tables or partition data in the system catalog held at the Catalog Manager to reroute the modified tuples to the proper node.

The Plan

- History
- Hardware Architecture
- Software Architecture (Still cool.)
- Query Algorithms
- Transactions
- Performance
- Summary

Transactions

- (Pessimistic) Concurrency Control - Locks
  - Basic Lock Types
    - S: shared / read
    - X: exclusive / write
  - Lock Terms
    - Short-term: until end of access
    - Long-term: until end of transaction

Transactions

- Concurrency Control - Locks
  - Lock Types + Lock Terms = Lock Modes

[6]

	Write locks on rows of a table are long-term	Read locks on rows of a table are long-term	Read and write locks on predicates are long-term
Read Uncommitted (dirty reads)	No (but it's read-only)	No Read locks at all	No predicate locks at all
Read Committed	Yes	No	Short-term Read predicate locks Long-term Write predicate locks
Repeatable Read	Yes	Yes	Short-term Read predicate locks Long-term Write predicate locks
Serializable	Yes	Yes	Long-term Read and Write predicate locks

Figure 10.9 Long-Term Locking Behavior of SQL-99 Isolation Levels[2]

  - Gamma's Lock Modes: S, X, IS, IX, SIX
    - The "I" is for "intent"

Transactions

- Concurrency Control - Locks
  - Lock Granularity
    - Database, Table, Page, Row, Field
    - Gamma supports “file” and page locking granularities.
      - This means there could be a lot of **lock contention** in the average to worst case, depending on the data and its indexes.

Transactions

- Concurrency Control - Locks
  - Two-phase locking
    - Growing phase: acquiring locks
    - Shrinking phase: releasing locks
  - This helps relieve some lock contention.
  - But **deadlock** is still a worry.

Transactions

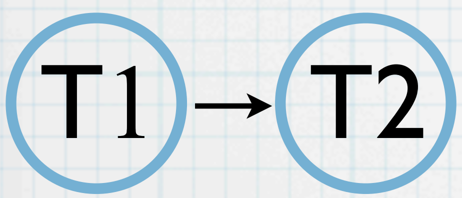
- Concurrency Control - Deadlock
  - Deadlock - mutual waiting, the dreaded deadly embrace
    - Transaction T1 needs resources A, and B, has a lock on A, waiting for B.
    - Transaction T2 needs resources A and B, has a lock on B, waiting for A.
  - What will we do? What **will** we do!?

Transactions

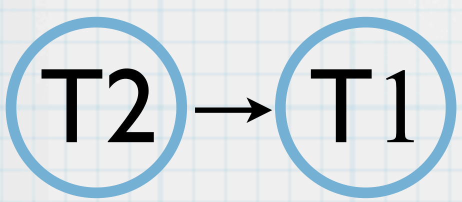
- Concurrency Control - Deadlock
  - Each Gamma Node has a Lock Manager that maintains a wait-for graph
    - One vertex (V) for each transaction
    - An edge from V_i to V_j means that V_i is blocked and waiting for a resource that V_j is holding (has locked).

Transactions

- Concurrency Control - Deadlock
  - Deadlock - mutual waiting, the dreaded deadly embrace



    - Transaction T1 needs resources A, and B, has a lock on A, **waiting for** B at T2.

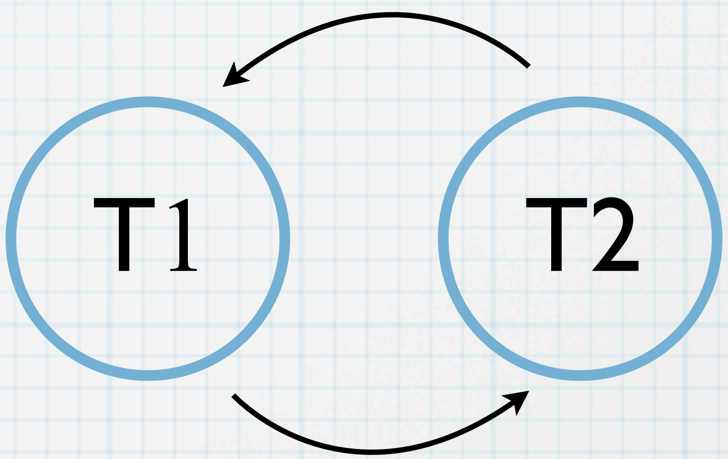


    - Transaction T2 needs resources A and B, has a lock on B, **waiting for** A at T1.

  - Combine the pieces into one wait-for graph to detect deadlock.

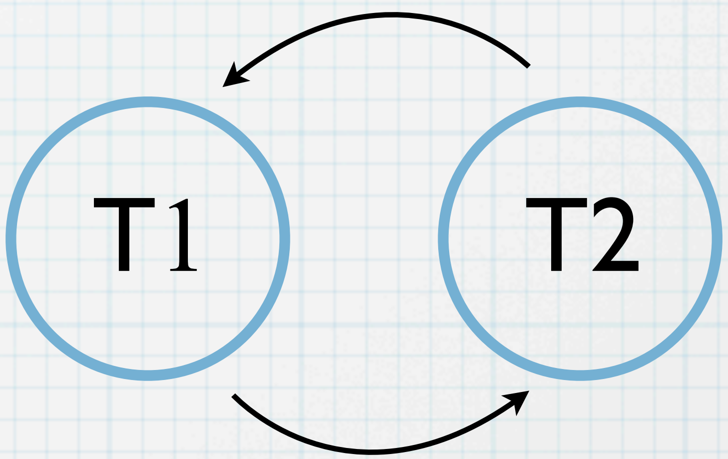
Transactions

- Concurrency Control - Deadlock
  - Combine the pieces into one wait-for graph to detect cycles and therefore deadlock.
  - Gamma does this across many nodes.
    - Lock Managers periodically exchange wait-for graphs with a central node who stitches them together for central deadlock detection.



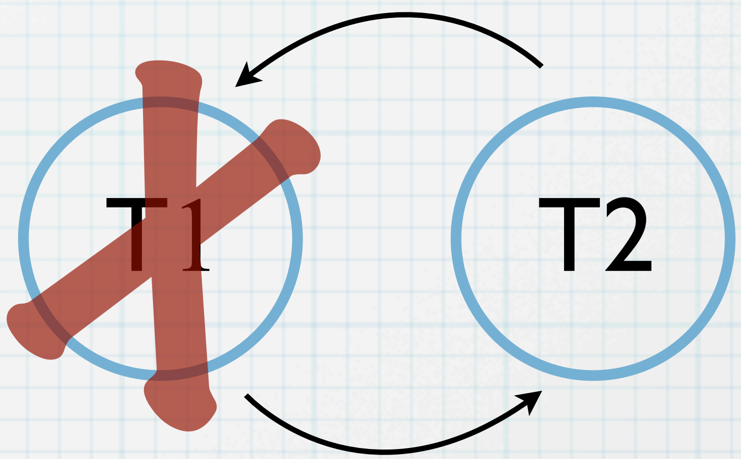
Transactions

- Concurrency Control - Deadlock
  - One we've detected deadlock, what do we do?



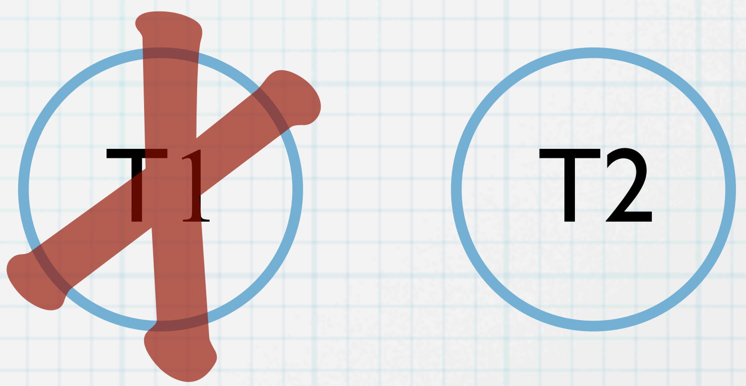
Transactions

- Concurrency Control - Deadlock
  - One we’ve detected deadlock, what do we do?
  - Kill (roll back) the transaction that’s holding the fewest locks.



Transactions

- Concurrency Control - Deadlock
  - One we’ve detected deadlock, what do we do?
  - Kill (roll back) the transaction that’s holding the fewest locks.



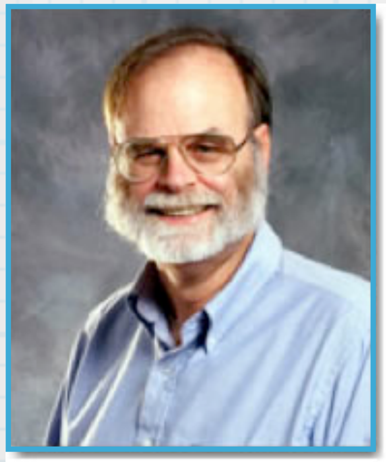
  - This unclogs the wait-for graph and lets the other transactions proceed.

Transactions

- Log Manager
    - When an operator process updates a record it generates a log record that contains . . .
        - LSL: Log Sequence Number
        - Before Image of the data
        - After Image of the data

Transactions

- Log Manager
  - Log records are sent to Log Manager processes at various nodes where they are collected, merged, and written to disk a page at a time.
  - This process seems pretty fragile to me and I'm not convinced it worked.
    - Jim Gray had this figured out and documented in his famous paper 1981 paper "The Recovery Manager of the System R Database Manager".



Transactions

- Recovery
  - Log records can be read by the Log Manager and transactions "undone" in reverse LSN order, using before images.
  - There's more to do (checkpoints, write-ahead durability, and more). They were still working on it at the time this paper was written.
    - DeWitt published at least five papers with Jim Gray, one in 2005, the others in the early 1990s.

Transactions

- Failure Management
  - Gamma keeps a **primary copy** and a **backup copy** of each table.
  - Reads are serviced from the primary copy.
  - Writes update both copies.
    - I hope the data is (exclusive) locked until the primary copy is updated.

The Plan

- History
- Hardware Architecture
- Software Architecture
- Query Algorithms
- Transactions
- Performance
- Summary

Performance

- The authors conducted many benchmark experiments. Let's look at two of the most interesting ones.

  1. Constant number of processors (30), vary the number of tuples - Measure performance relative to table size.

  2. Constant number of tuples (1M), vary the number of processors - Measure speed up / scale up

Performance

- 30 processors, variable tuples, 6 queries

SELECTION QUERIES. 30 PROCESSORS WITH DISKS (ALL EXECUTION TIMES IN SECONDS)

Query Description	Number of Tuples in Source Relation 100,000	1,000,000	10,000,000
1% nonindexed selection	0.45	8.16	81.15
10% nonindexed selection	0.82	10.82	135.61
1% selection using clustered index	0.35	0.82	5.12
10% selection using clustered index	0.77	5.02	61.86
1% selection using non-clustered index	0.60	8.77	113.37
single tuple select using clustered index	0.08	0.08	0.14

[1]

Performance

- 30 processors, variable tuples, 6 queries

SELECTION QUERIES. 30 PROCESSORS WITH DISKS (ALL EXECUTION TIMES IN SECONDS

Query Description	Number of Tuples in Source Relation 100,000	1,000,000	10,000,000
1% nonindexed selection	0.45	8.16	81.15
10% nonindexed selection	0.82	10.82	135.61
1% selection using clustered index	0.35	0.82	5.12
10% selection using clustered index	0.77	5.02	61.86
1% selection using non-clustered index	0.60	8.77	113.37
single tuple select using clustered index	0.08	0.08	0.14

[1]

Performance

- 30 processors, variable tuples, 6 queries
- Linear increases

SELECTION QUERIES. 30 PROCESSORS WITH DISKS (ALL EXECUTION TIMES IN SECONDS)

Query Description	Number of Tuples in Source Relation 100,000	1,000,000	10,000,000
1% nonindexed selection	0.45	8.16	81.15
10% nonindexed selection	0.82	10.82	135.61
1% selection using clustered index	0.35	0.82	5.12
10% selection using clustered index	0.77	5.02	61.86
1% selection using non-clustered index	0.60	8.77	113.37
single tuple select using clustered index	0.08	0.08	0.14

[1]

Performance

- 30 processors, variable tuples, 6 queries
- Constant performance here

SELECTION QUERIES. 30 PROCESSORS WITH DISKS (ALL EXECUTION TIMES IN SECONDS)

Query Description	Number of Tuples in Source Relation 100,000	1,000,000	10,000,000
1% nonindexed selection	0.45	8.16	81.15
10% nonindexed selection	0.82	10.82	135.61
1% selection using clustered index	0.35	0.82	5.12
10% selection using clustered index	0.77	5.02	61.86
1% selection using non-clustered index	0.60	8.77	113.37
single tuple select using clustered index	0.08	0.08	0.14

[1]

Performance

- 30 processors, variable tuples, 6 queries
- Not constant performance here. Why?

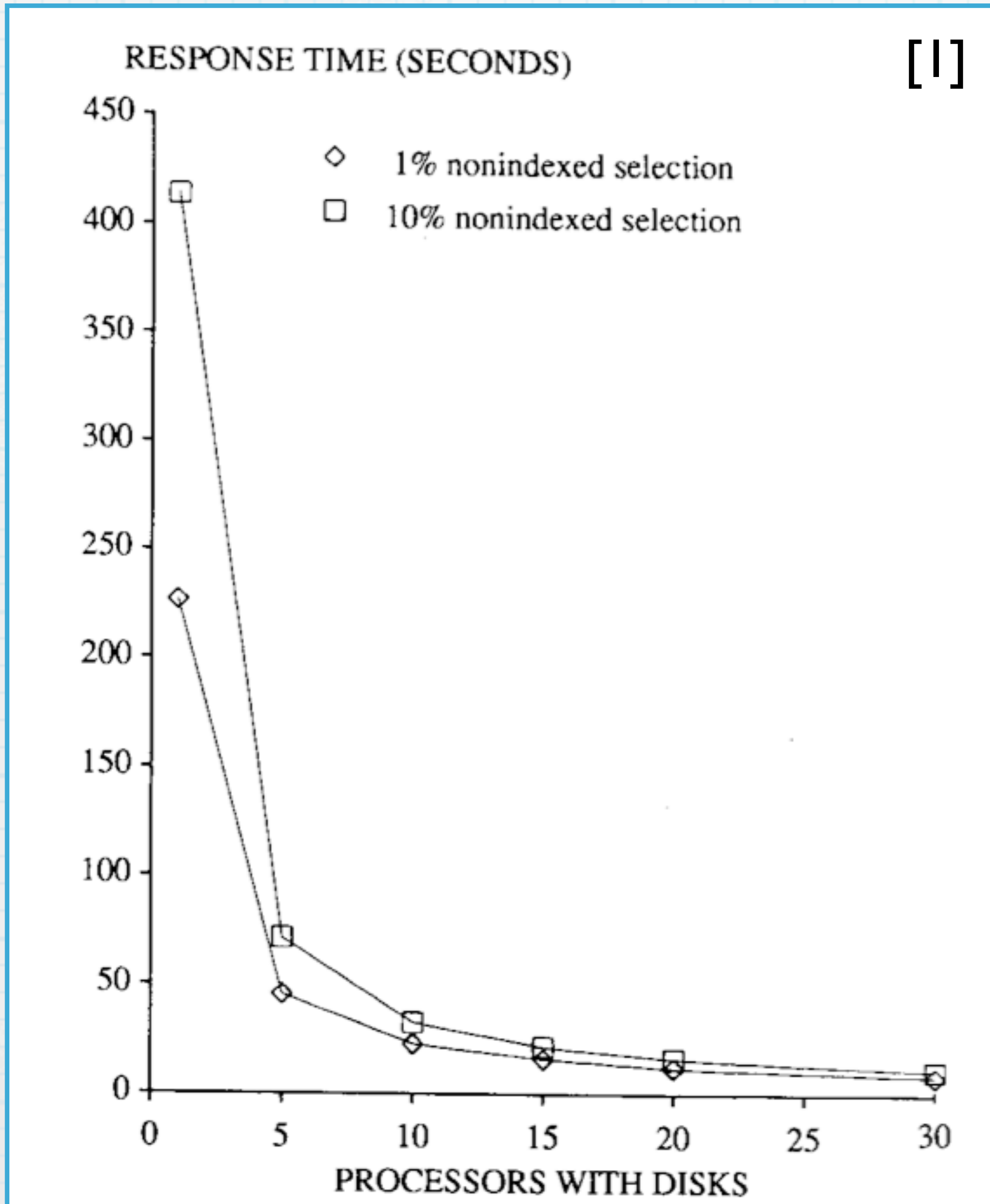
SELECTION QUERIES. 30 PROCESSORS WITH DISKS (ALL EXECUTION TIMES IN SECONDS)

Query Description	Number of Tuples in Source Relation 100,000	1,000,000	10,000,000
1% nonindexed selection	0.45	8.16	81.15
10% nonindexed selection	0.82	10.82	135.61
1% selection using clustered index	0.35	0.82	5.12
10% selection using clustered index	0.77	5.02	61.86
1% selection using non-clustered index	0.60	8.77	113.37
single tuple select using clustered index	0.08	0.08	0.14

[1]

Performance

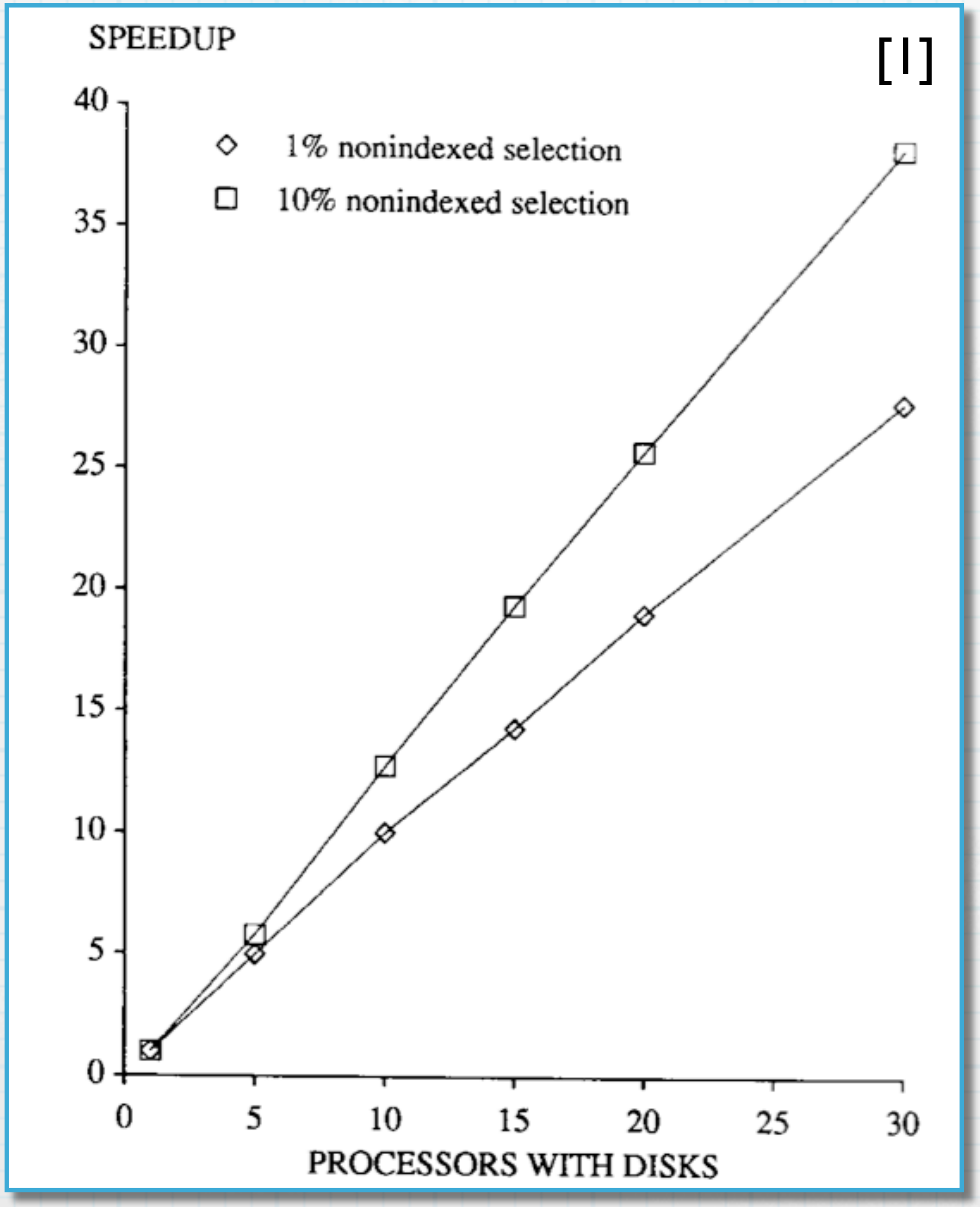
- 1M tuples, variable processors, 2 queries



- Query response time decreases as the number of nodes/processors increase.
- This is speed-up (or scale-up)

Performance

- 1M tuples, variable processors, 2 queries



- Same data expressed as speed-up.
- Why does the query with 10% selectivity speed up less?

The Plan

- History
- Hardware Architecture
- Software Architecture
- Query Algorithms
- Transactions
- Performance
- Summary

Summary

- David J. DeWitt's **Gamma was a big deal**.
- A few projects/areas citing DeWitt, et al. [5]

DB2 Parallel Edition	NUMA Clusters
IBM S/390 Parallel Sysplex	vehicular ad-hoc networks
Map-reduce	SAP
Sensor Networks	extensible web crawlers
Data Mining, OLAP, and BI	parallel query processing

Summary

- David J. DeWitt's Gamma was a big deal.
  - In 1995, David was named a Fellow of the ACM and received the ACM SIGMOD Innovations Award for his contributions to the database field. [2]

Summary

- David J. DeWitt's Gamma was a big deal.
  - In 2009, the ACM recognized the seminal contributions of the Gamma parallel database system project with the ACM Software Systems Award. [2]

Summary

- Gamma was a fast, parallel, relational database that scaled with the number of processors and the size of the data and influenced many systems we still use today.

Questions? Comments?

Thank you for your attention.

Summary

- References

(1) DeWitt, et.al, *The Gamma Database Machine Project*, IEEE Transactions on Knowledge and Data Engineering, Vol. 2 No. 1, March 1990. pp 44-62

(2) David DeWitt's home page, http://pages.cs.wisc.edu/~dewitt/includes/publications.html. Accessed February 3, 2012

(3) M. Stonebraker, *The Case for Shared Nothing*, *Database Eng*., vol. 9 no. 1, 1986.

(4) Momjian, Bruce, *PostgreSQL Internals Through Pictures*, Enterprise DB, January, 2004

(5) *The Gamma Database Machine Project* ACM Digital Library Bibliometrics http://dl.acm.org/citation.cfm?id=627276.627398&coll=DL&dl=GUIDE&CFID=65801349&CFTOKEN=74568999. Accessed February 11, 2012.

(6) *Database: Principles, Programming, and Performance*, 2nd edition, by Patrick and Elizabeth O'Neil