A Parallel Programming Primer

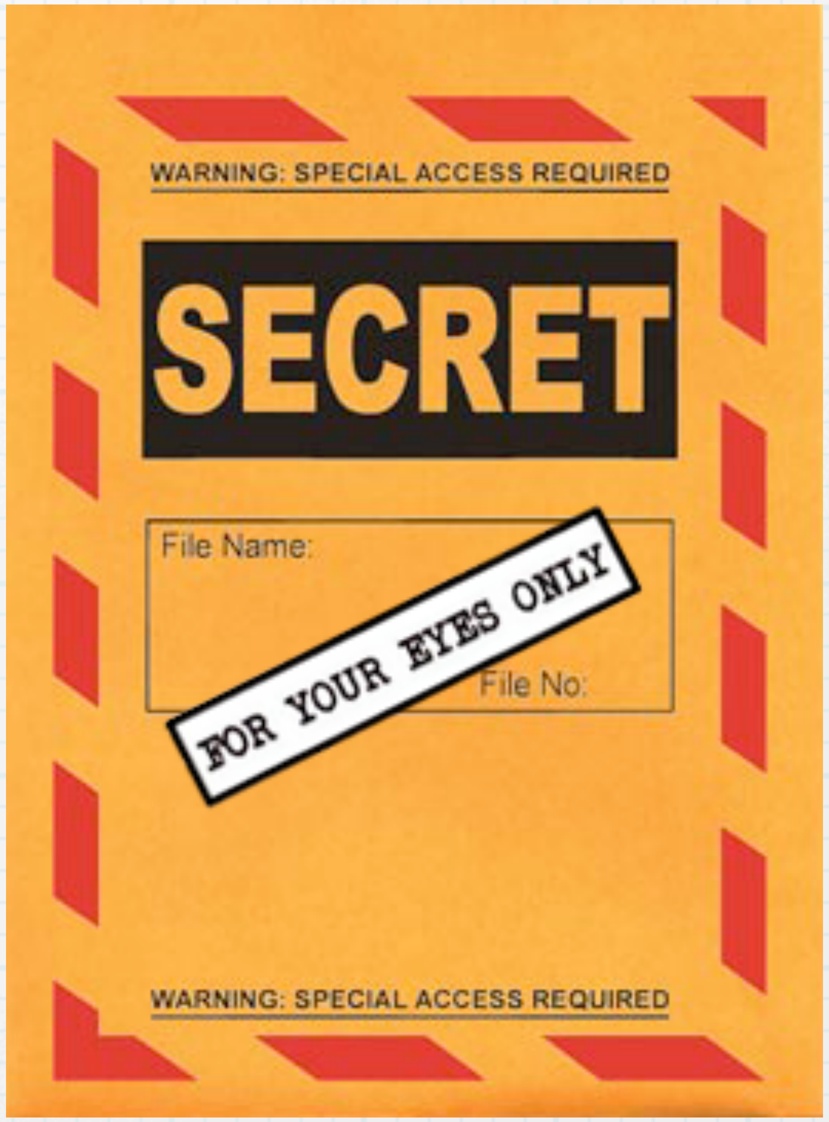
by
Alan G. Labouseur
alan@Labouseur.com

A Parallel Programming Primer

by
Alan G. Labouseur
alan@Labouseur.com

A Parallel Programming Primer

by
Alan G. Labouseur
alan@Labouseur.com


WARNING: SPECIAL ACCESS REQUIRED
SECRET
File Name:
FOR YOUR EYES ONLY
File No:
WARNING: SPECIAL ACCESS REQUIRED

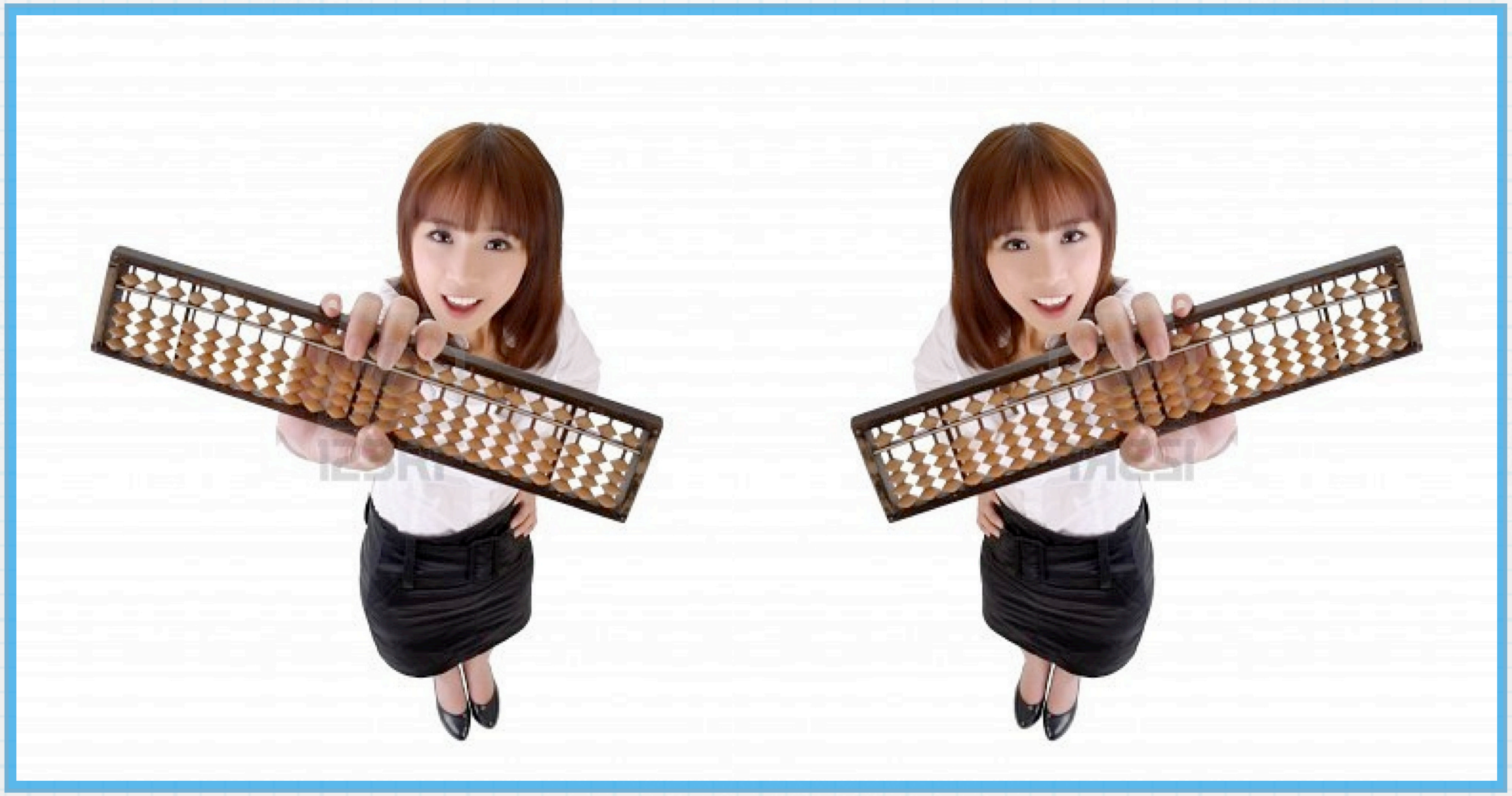
A Parallel Programming Primer

by
Alan G. Labouseur
alan@Labouseur.com

Mission Brief:

History

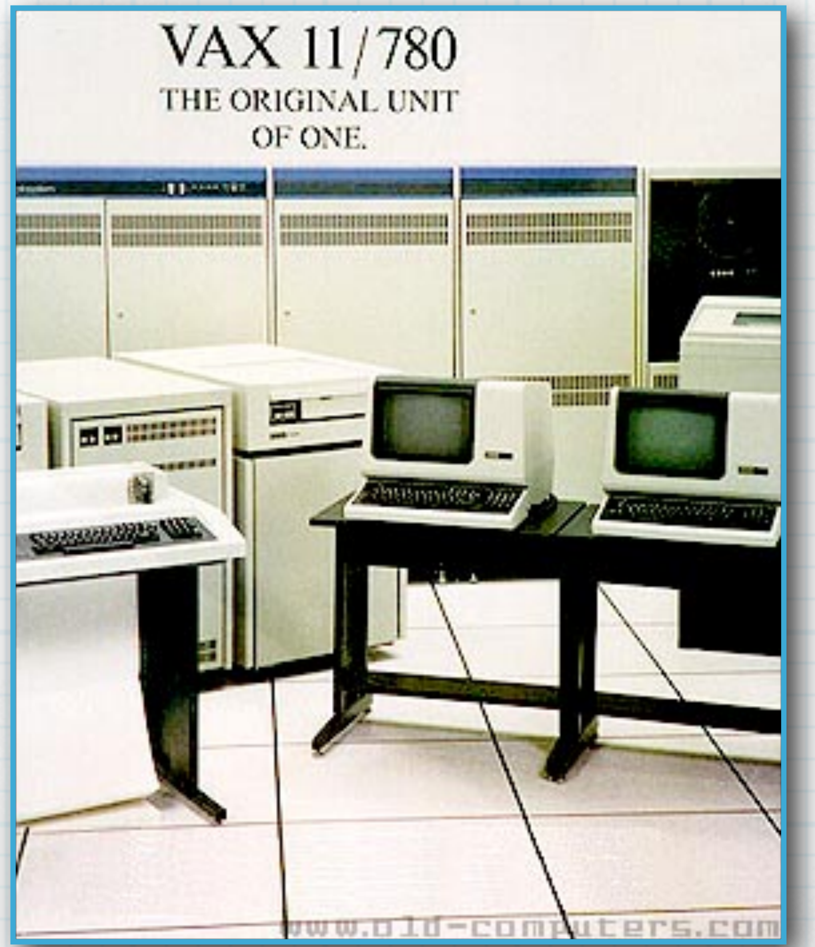
Early Parallel Programming



Data Parallel

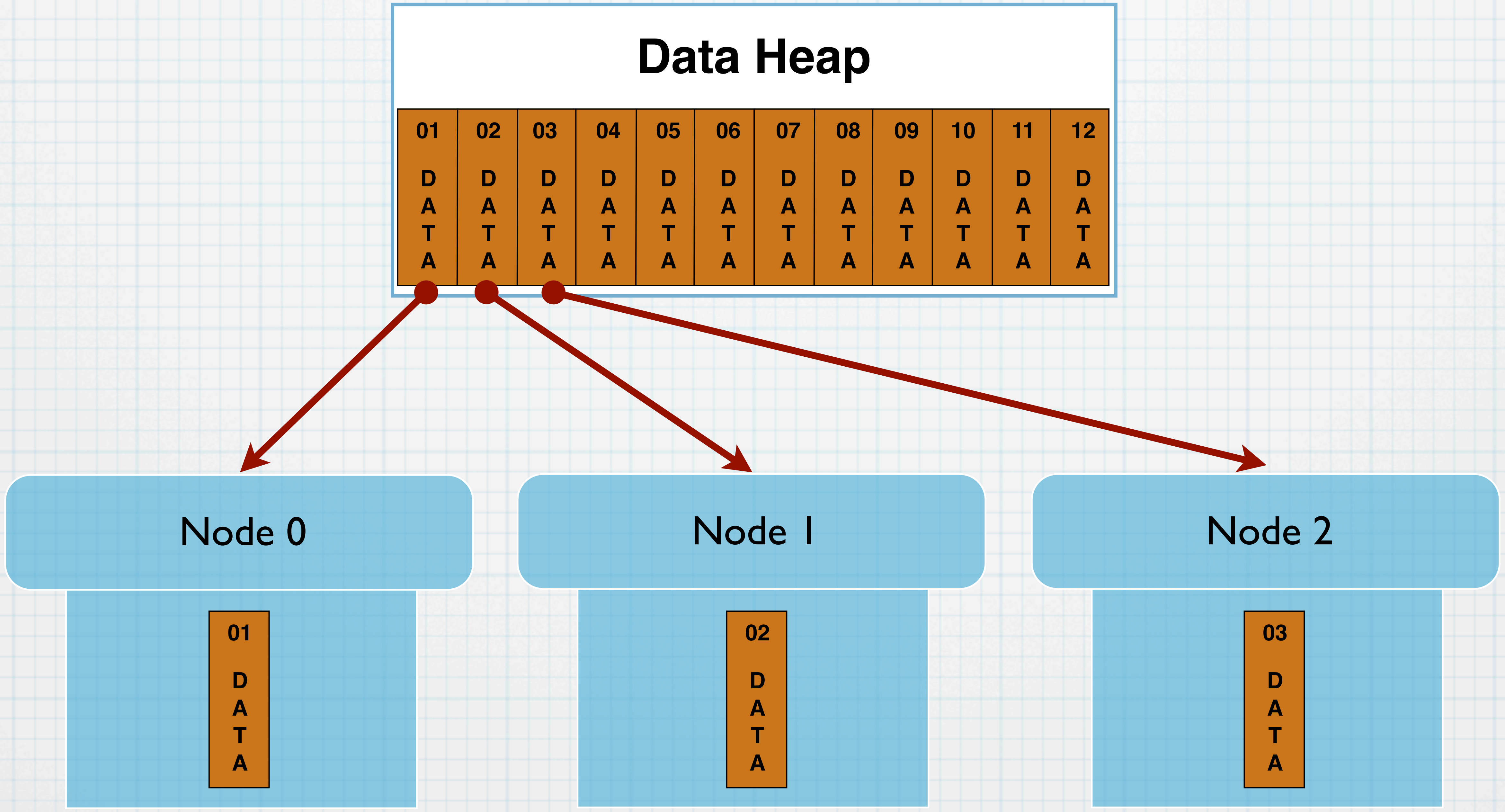
1984 - DeWitt's Gamma Database Machine

- Many processors, many disks
- Shared-nothing architecture
- Three *keys* to parallelism:
    1. Coordinated scheduling
    2. Parallel hash algorithms for relational operators
    3. Tables are horizontally partitioned/declustered
- Three declustering strategies



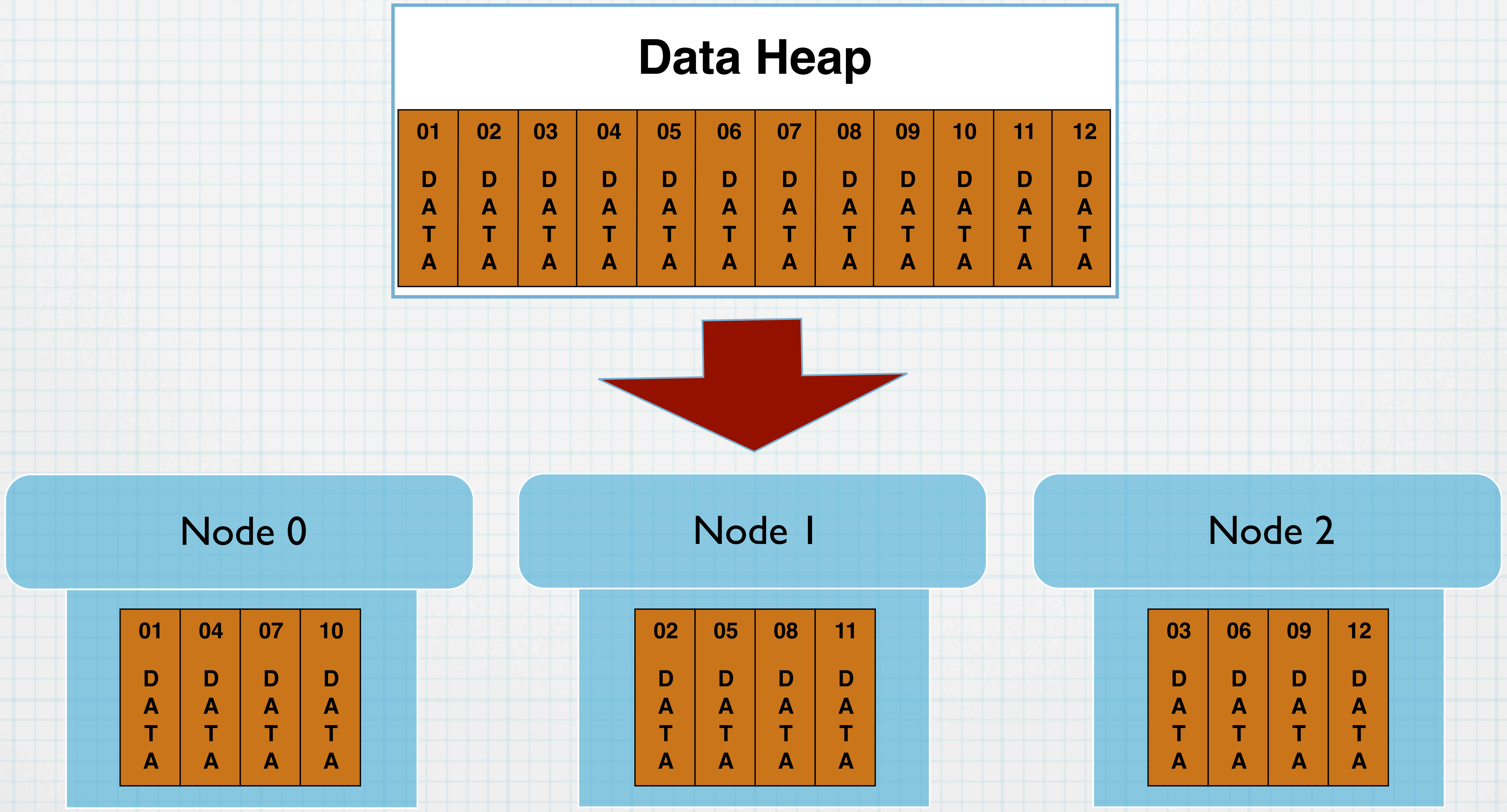
Data Parallel

Round-Robin Declustering



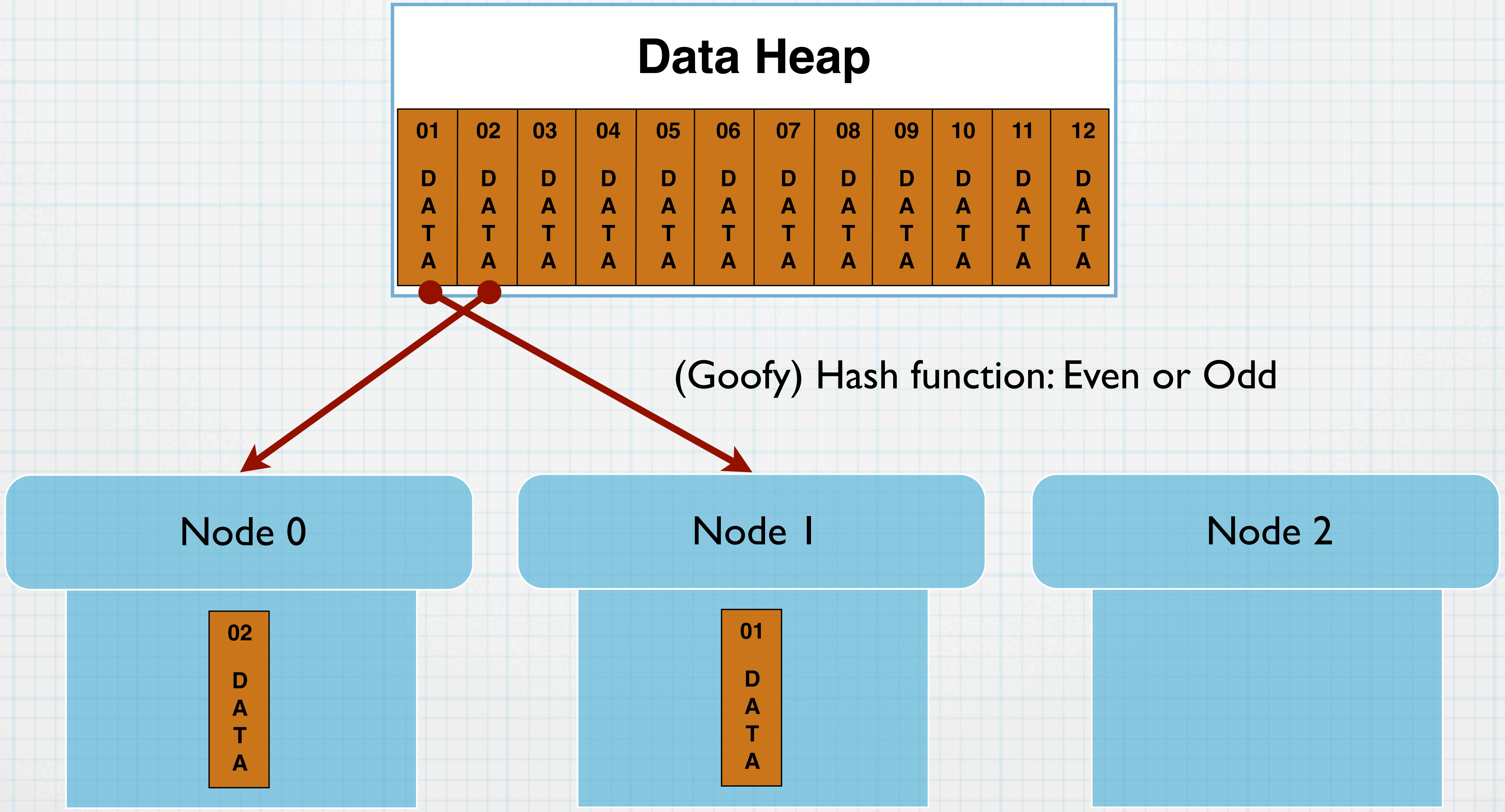
Data Parallel

Round-Robin Declustering



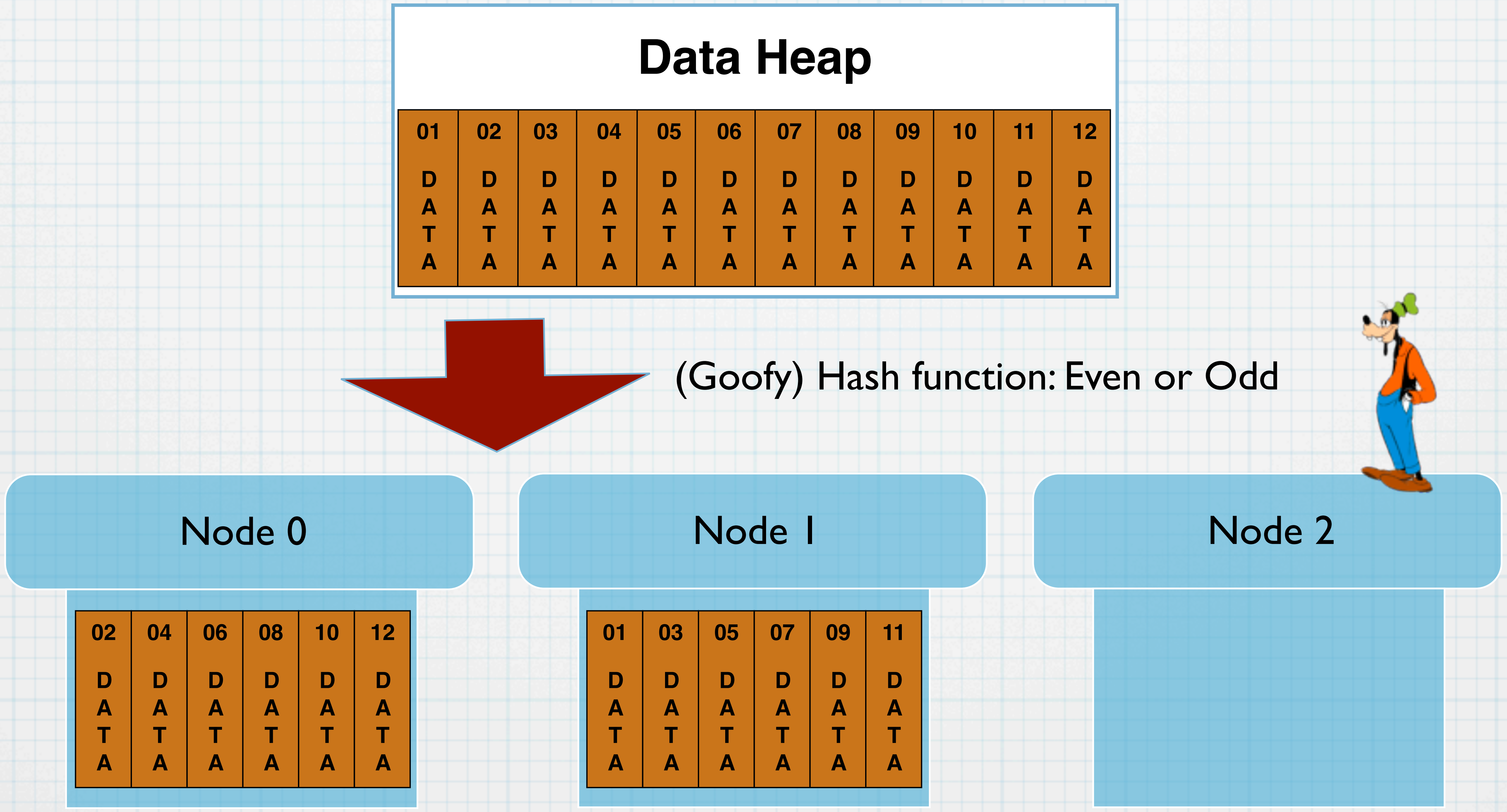
Data Parallel

Hashed Declustering



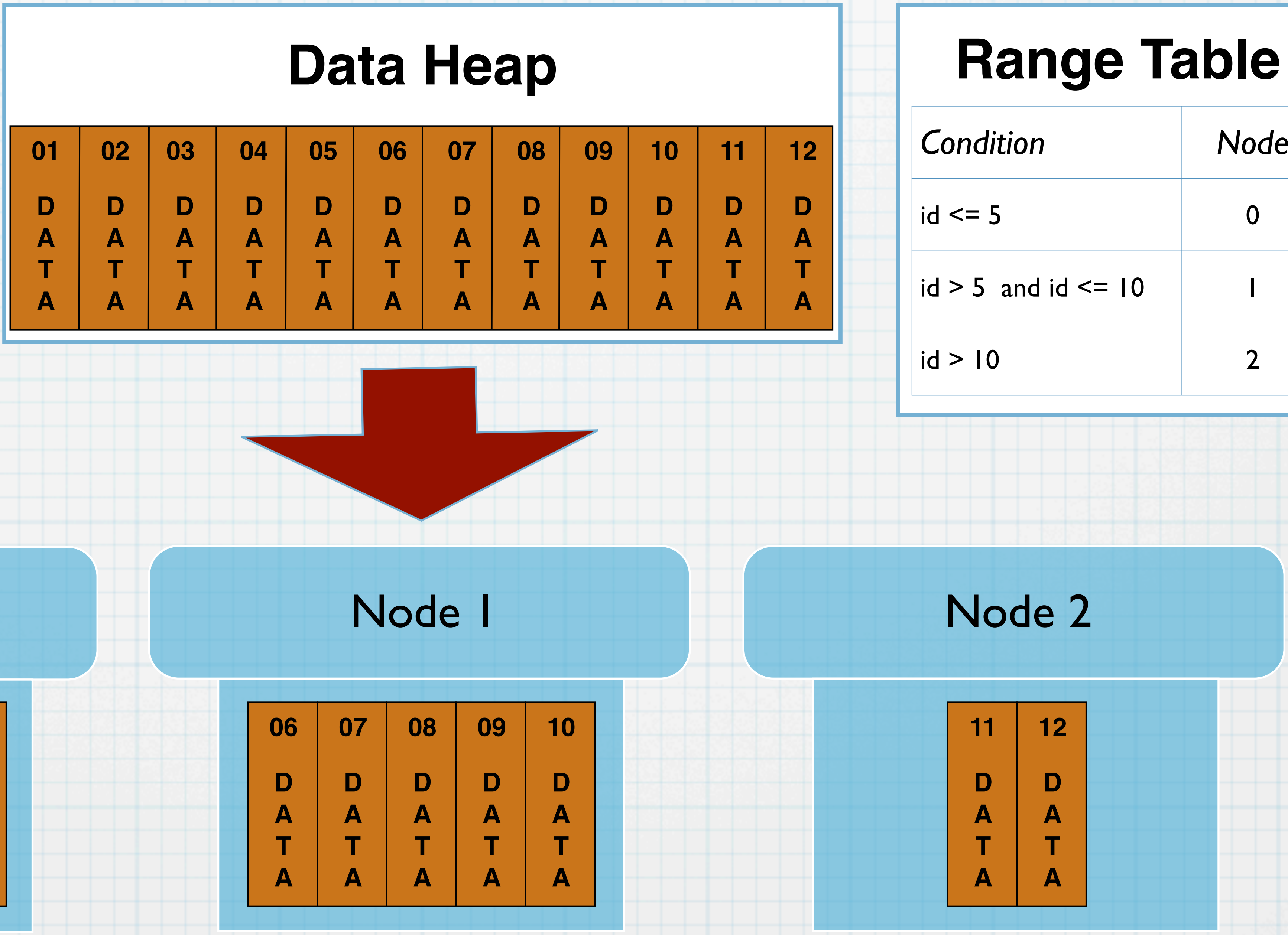
Data Parallel

Hashed Declustering



Data Parallel

Sharded Declustering



Range Table

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

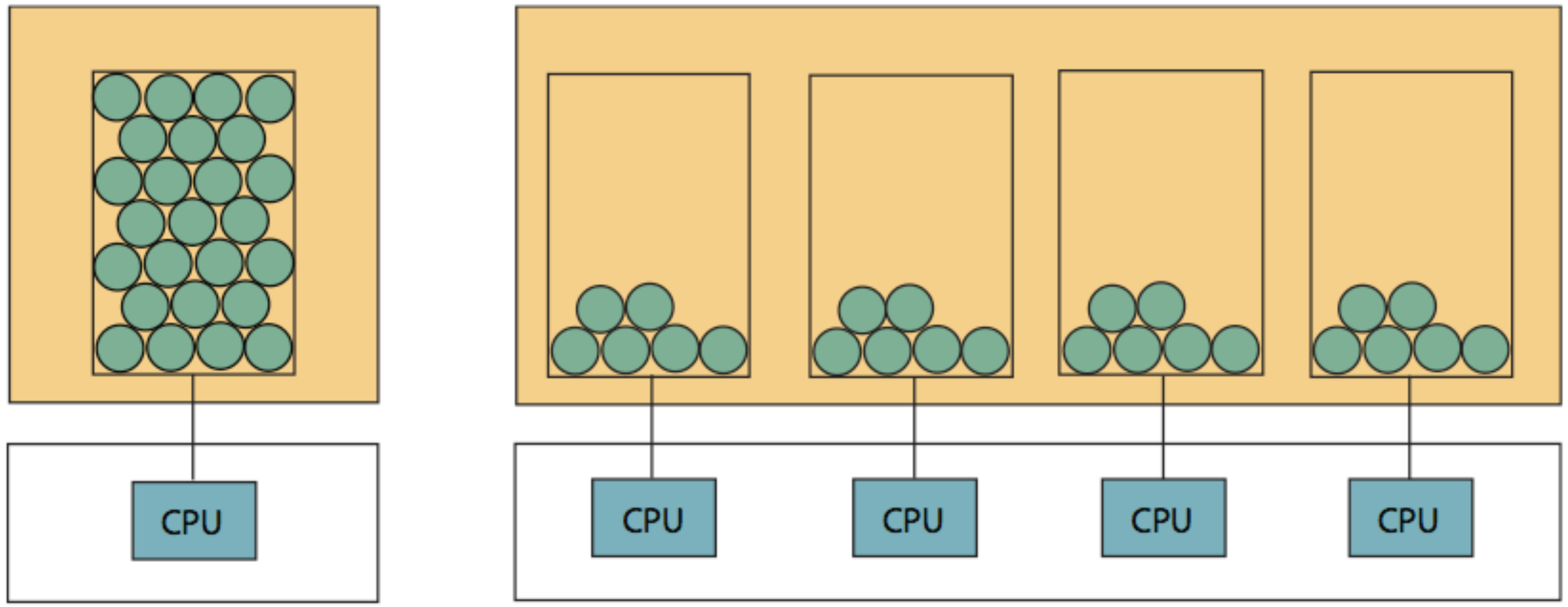
We’re done, right?

- NO! Distributing the data is only half the battle.

- Bad News: Moore’s Law is leveling out.
    - CPU cores aren’t getting much faster.
    - Thus limiting our capabilities on a single-CPU

- Good News: We’re getting more cores every day.
    - Intel i7 Ivy Bridge microprocessor
        - dual-core (2 real / 4 virtual)
        - quad-core (4 real / 8 virtual)

Task (Function) Parallel



Single-core Quad-core

All we need is programming language support.

A Family History

$$\begin{tikzcd}
\text{ALGOL} \arrow[d] & \textit{1960s} \\
\text{Pascal, C} \arrow[d] & \textit{1970s} \\
\text{C++} \arrow[d] & \textit{1980s} \\
\text{Java, Delphi} \arrow[d] & \textit{1990s} \\
\text{C\#} & \textit{2000s}
\end{tikzcd}$$

A Family History

$$\begin{tikzcd}
\text{ALGOL} \arrow[d] & \textit{1960s} \\
\text{Pascal, C} \arrow[d] & \textit{1970s} \\
\textbf{C++} \arrow[d] & \textit{1980s} \\
\textbf{Java}\text{, Delphi} \arrow[d] & \textit{1990s} \\
\textbf{C\#} & \textit{2000s}
\end{tikzcd}$$

Yesterday

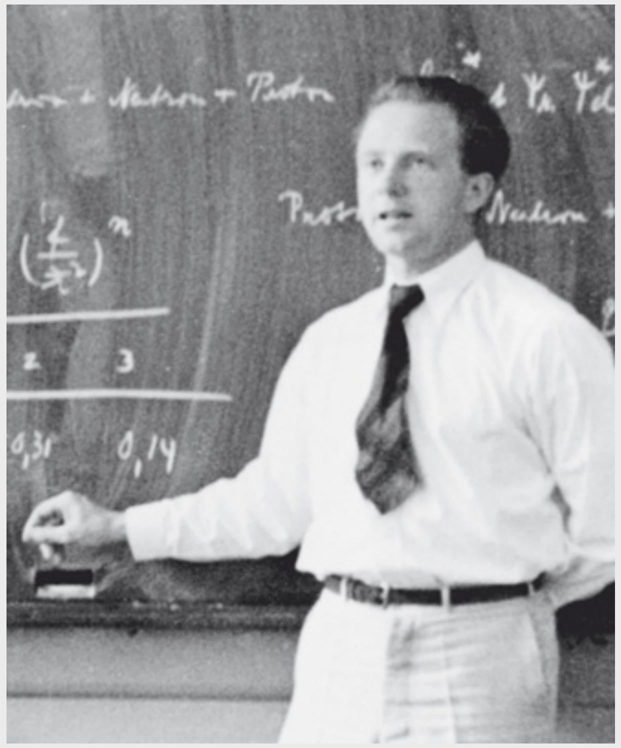
- C++ / Java / C#
  - Shared-state concurrency
  - Encapsulation is not complete
    - Thread scheduling does not obey encapsulation rules regardless of how you write your objects.
    - Most code is unsafe for scaling up or out.

Yesterday

- C++ / Java / C#
  - Concurrency is in the plumbing
    - Developers are responsible for determinism...
    - by adding locks, semaphores, monitors, which cause race conditions and deadlock.
  - Thread safety a programmer problem.

Yesterday

- C++ / Java / C#
  - Refactoring is complicated and error-prone.
    - Needs IDE to help out
  - Mutable State
    - Side effects
    - Heisenbugs
  - Difficult to parallelize.



Another Family History

$$\begin{tikzcd}
\text{ALGOL} \arrow[d] & \textit{1960s} & \text{LISP} \arrow[d] \\
\text{Pascal, C} \arrow[d] & \textit{1970s} & \text{ML} \arrow[d] \\
\text{C++} \arrow[d] & \textit{1980s} & \text{Erlang} \arrow[d] \\
\text{Java, Delphi} \arrow[d] & \textit{1990s} & \text{Haskell} \arrow[d] \\
\text{C\#} & \textit{2000s} & \text{F\#, Scala}
\end{tikzcd}$$

Another Family History

$$\begin{tikzcd}
\text{ALGOL} \arrow[d] & \textit{1960s} & \text{LISP} \arrow[d] \\
\text{Pascal, C} \arrow[d] & \textit{1970s} & \text{ML} \arrow[d] \\
\text{C++} \arrow[d] & \textit{1980s} & \textbf{Erlang} \arrow[d] \\
\text{Java, Delphi} \arrow[d] & \textit{1990s} & \text{Haskell} \arrow[d] \\
\text{C\#} & \textit{2000s} & \text{F\#, } \textbf{Scala}
\end{tikzcd}$$

Today

- C++ / Java / C#
  - Shared-state concurrency
  - Encapsulation is not complete
    - Thread scheduling does not obey encapsulation rules regardless of how you write your objects.
    - Most code is unsafe for scaling up or out.

- Erlang / Scala
  - Shared-nothing concurrency
  - Encapsulation of state and behavior is complete
    - **Actors** are completely isolated, only communicating with messages.
    - Most code is safe for scaling up and out.

Today

- C++ / Java / C#
  - Concurrency is in the plumbing
    - Developers are responsible for determinism...
    - by adding locks, semaphores, monitors, which cause race conditions and deadlock.
  - Thread safety a programmer problem.

- Erlang / Scala
  - Actors hoist the concurrency abstraction from the plumbing to the workflow.
    - Developers are responsible workflow...
    - by passing immutable messages to non-blocking asynchronous actors.
  - **Thread safety is a runtime feature.**

Today

- C++ / Java / C#
  - Refactoring is complicated and error-prone.
    - Needs IDE to help out
  - Mutable State
    - Side effects
    - Heisenbugs
  - Difficult to parallelize.

- Erlang / Scala
  - Type inference makes much refactoring instant and error-free.
    - No IDE help, it’s in the language
  - Immutable State
    - **No side effects**
    - Fewer bugs
  - Easier to parallelize (within the limits noted by Amdahl and Gustafson).

Today

- Erlang in the world
  - Amazon, Facebook, British Telecom, T-Mobile, Xerox, Jane Street, Google, Apple, Basho, Ericsson, Heroku, InfoQ, and many others, especially in Europe.
- Scala in the world
  - Twitter, LinkedIn, FourSquare, Siemens, Sony Pictures, Tumblr, UBS, Morgan Stanley, Capital IQ, Google, HP, eBay, zeebox, Heroku, and many more.

Tomorrow

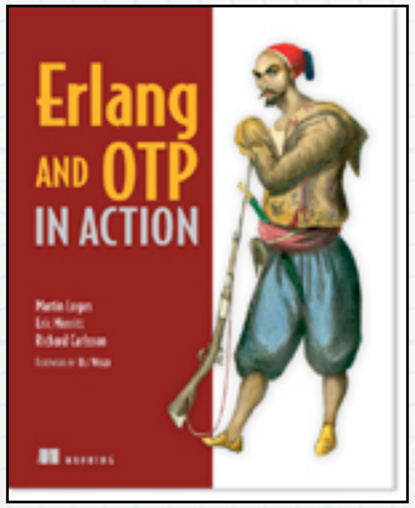
- Go out and learn Erlang.

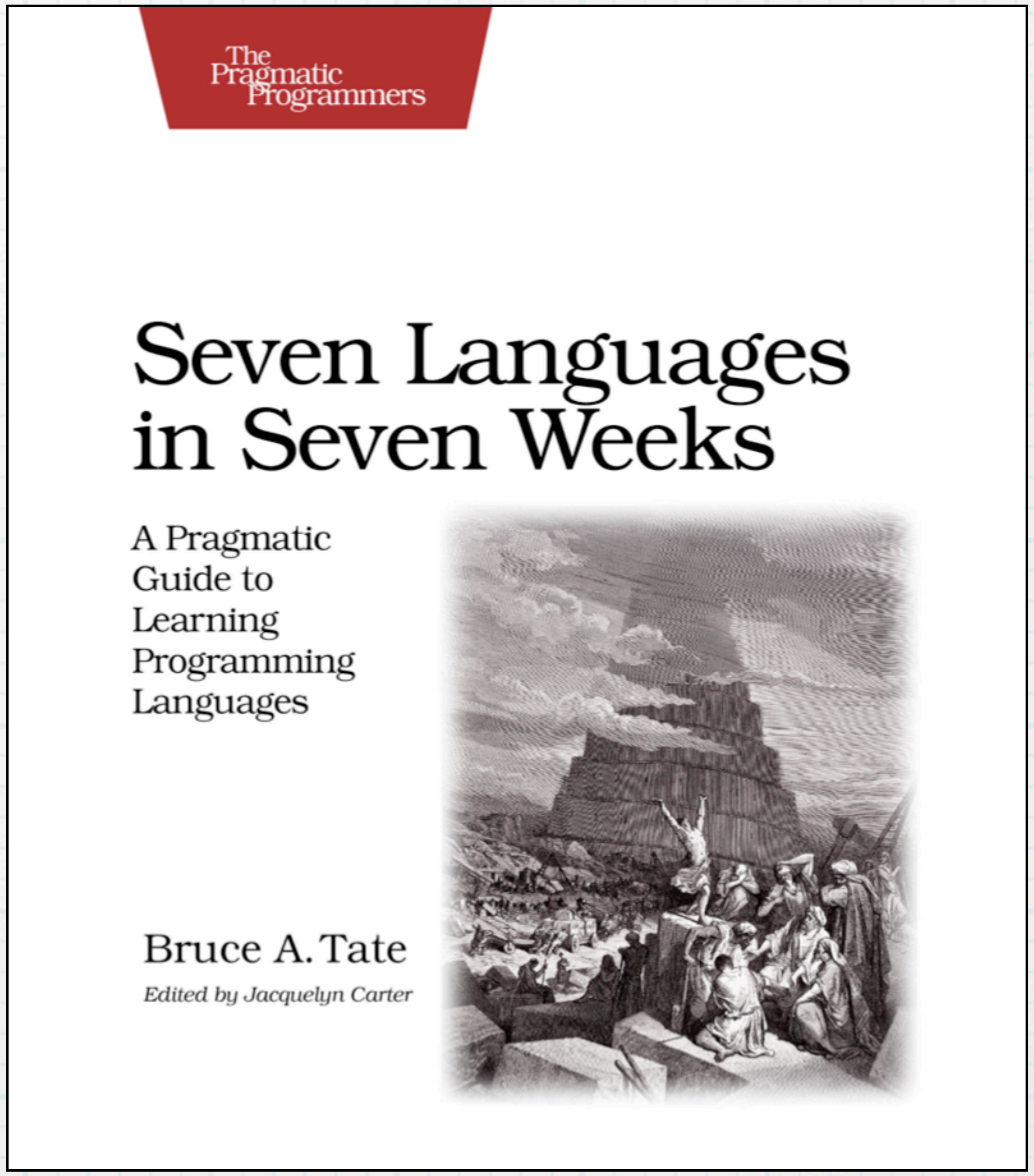
The Pragmatic Programmers

Programming Erlang

Software for a Concurrent World

Joe Armstrong


Erlang and OTP in Action


The Pragmatic Programmers

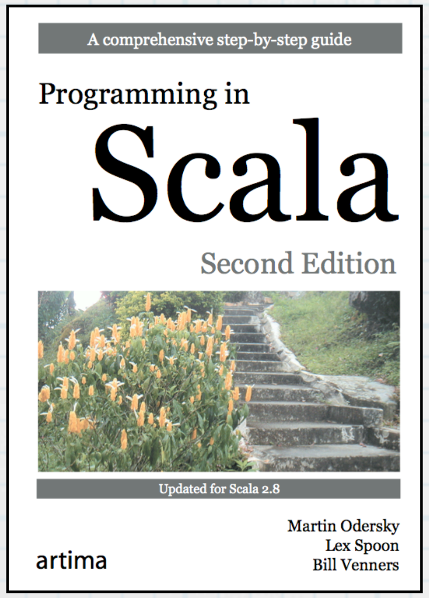
Seven Languages in Seven Weeks

A Pragmatic Guide to Learning Programming Languages

Bruce A. Tate

Edited by Jacquelyn Carter

- Go out and learn Scala.


A comprehensive step-by-step guide

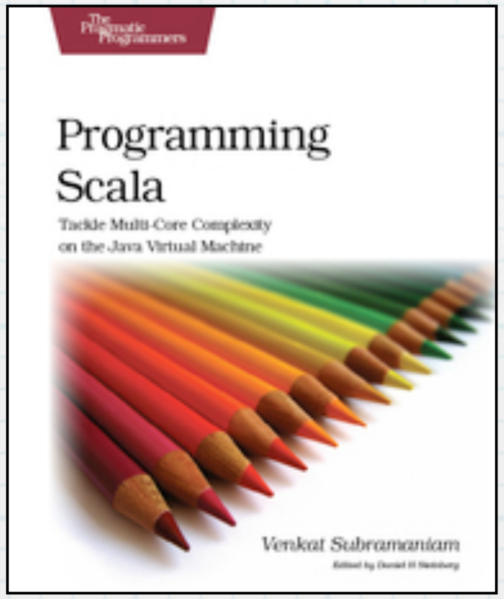
Programming in Scala

Second Edition

Updated for Scala 2.8

Martin Odersky
Lex Spoon
Bill Venners

artima


The Pragmatic Programmers

Programming Scala

Tackle Multi-Core Complexity on the Java Virtual Machine

Venkat Subramaniam