NetApp

# Neural Trees

## Using Neural Networks as an Alternative to Binary Comparison in Classical Search Trees

Douglas Santry

# Introduction – Finding Stuff

- Binary Comparison
  - The < operator has been directly supported by even the earliest digital computers.
  - Binary search was used from the beginning, but binary search as we know it is the result of Lehmer's paper in 1960.
  - $N \cdot log_2(N)$ search is provably optimal for binary comparison.
- B Tree
  - Described by Bayer in 1970.
  - Provably optimal with respect to the number of comparisons.
  - The parameter, B, trades off memory for number media accesses.
  - Countless permutations.
- $B^\varepsilon$ spectrum
  - Today we think in terms of the $B^\varepsilon$ spectrum: $\varepsilon$ trades off between updates and searches.
  - The spectrum is the direct result of the binary comparison operator: it totally determines the physical structure of the tree.



NetApp

# Alternatives to Binary Comparison Are Not New

- Learned Indices (Kraska et al, SIGMOD 2018)
  - Data are considered as a cumulative distribution: $index_{i+1} = N_i \cdot CDF_i(key)$
  - A tree with a neural network root, linear regression between root and leaves.
  - Binary comparison is used in the leaves.
- Interpolation Search (Van Sandt et al, SIGMOD 2019)
  - Originally proposed in 1957 by Petersen.
  - Search based on linear regression.
- Operates on in-memory contiguous arrays of sorted data.
  - Binary comparison determines physical layout.
  - Insertion requires `memcpy()` of everything in front.
- Not appropriate for secondary storage as $index_i$ requires indirection.
- Learn the data directly (distribution)
- Read-only

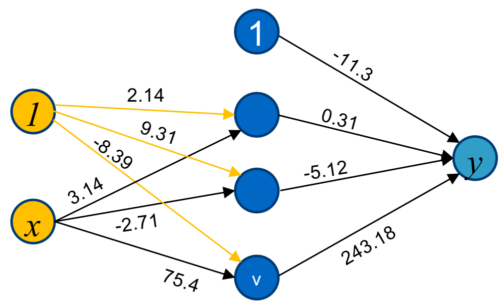


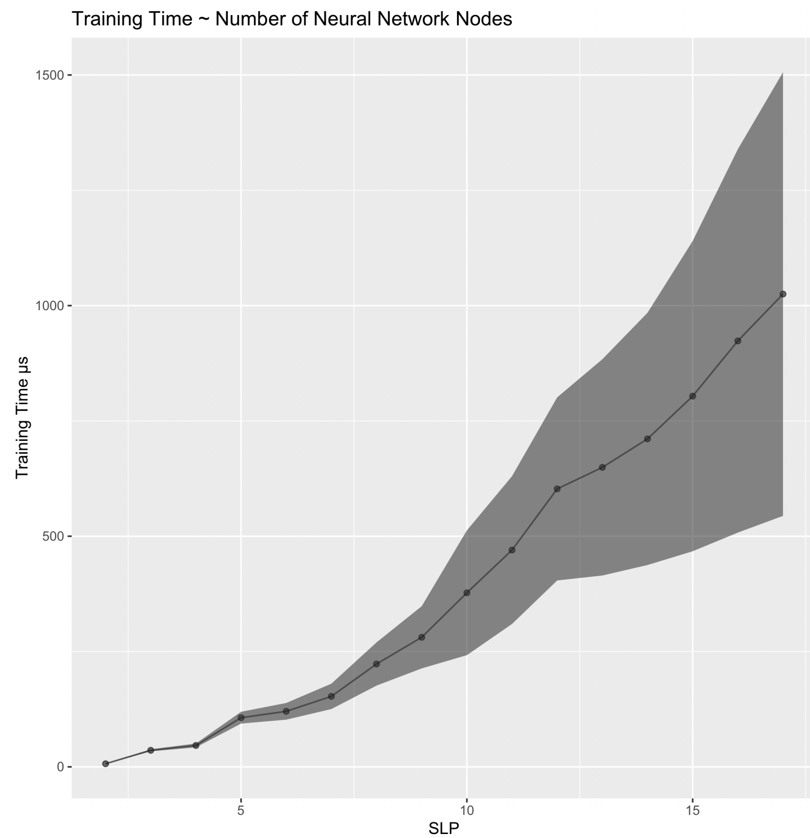
NetApp

# Key Technical Contributions

- Supports secondary storage.
  - Discards requirement for contiguous sorted array in memory.
  - Indexing secondary storage is not a CDF, it is a mapping.
- Neural networks inherently include the indirection required for secondary storage.
  - Linear interpolation requires a mapping from logical index to physical address.
- Employs many tiny neural networks that are *quick* to train.
  - Training is in the write path.
- Straddles classical search trees by learning *paths*, not the data directly.
  - There are fewer paths than data.
  - Paths are relatively static compared to data.
  - Neural networks are more like network routers.
- Addresses inference error
  - Inferencing mistakes are expensive in a secondary storage index: superfluous reads.

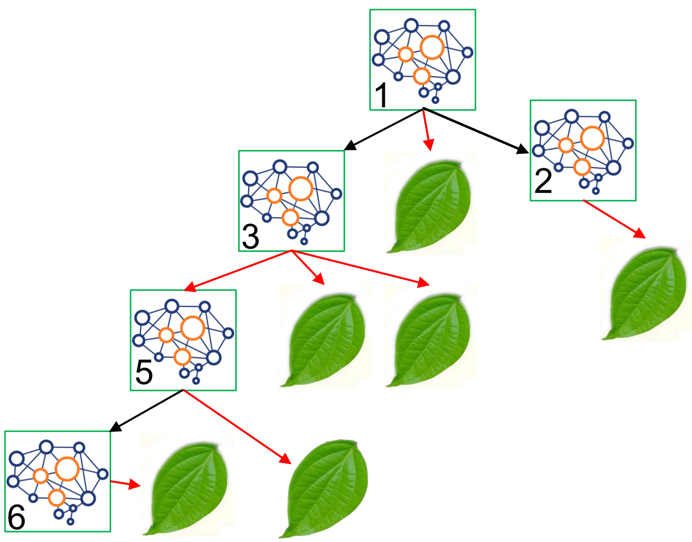
# Single Layer Perceptrons (SLP)



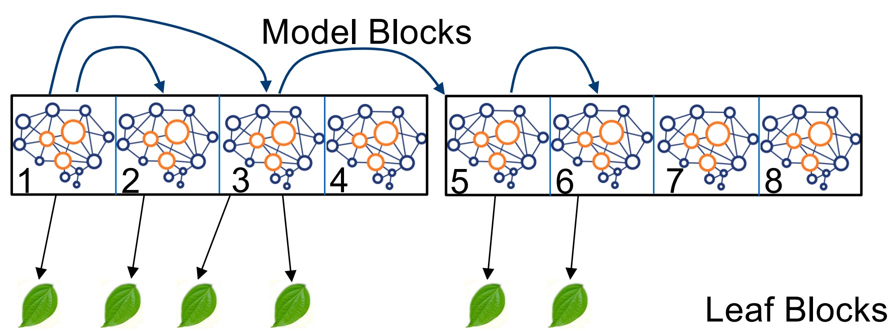



# Neural Tree Architecture

Logical Structure

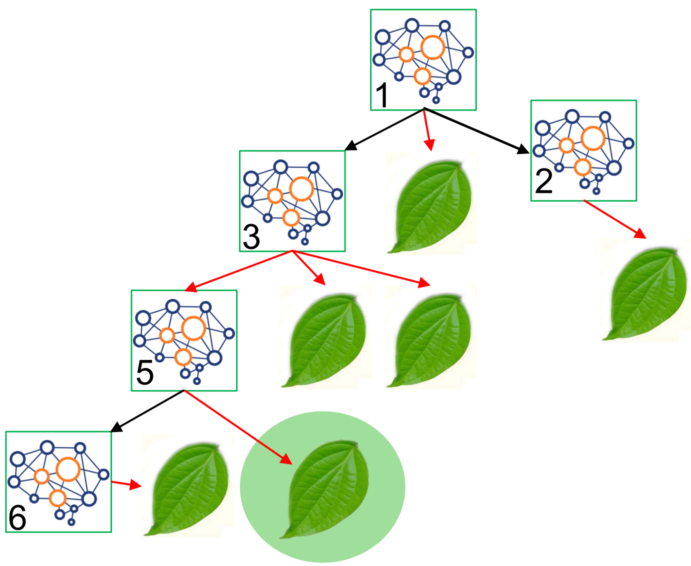



Physical Structure

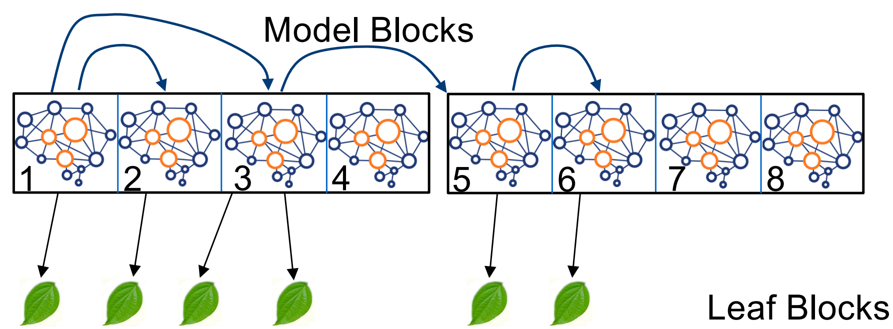


# Neural Tree Architecture

Logical Structure

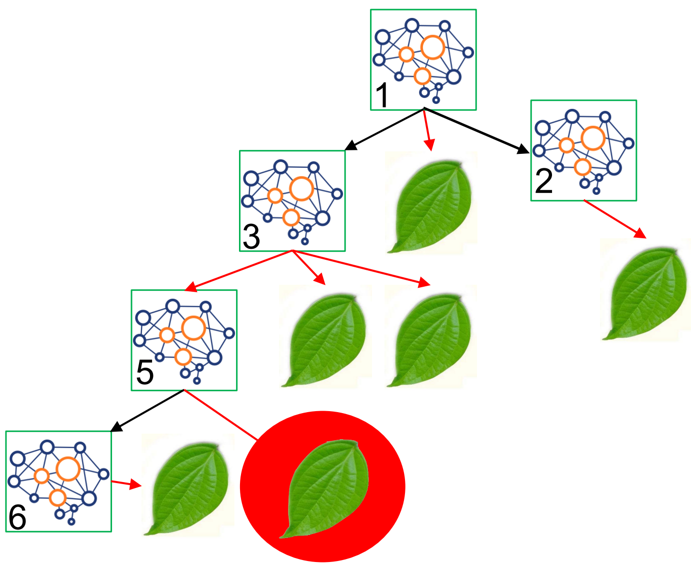



Physical Structure

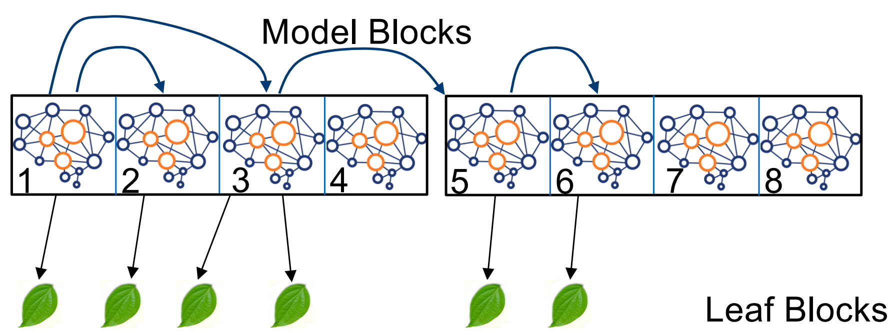


# Neural Tree Architecture

Logical Structure



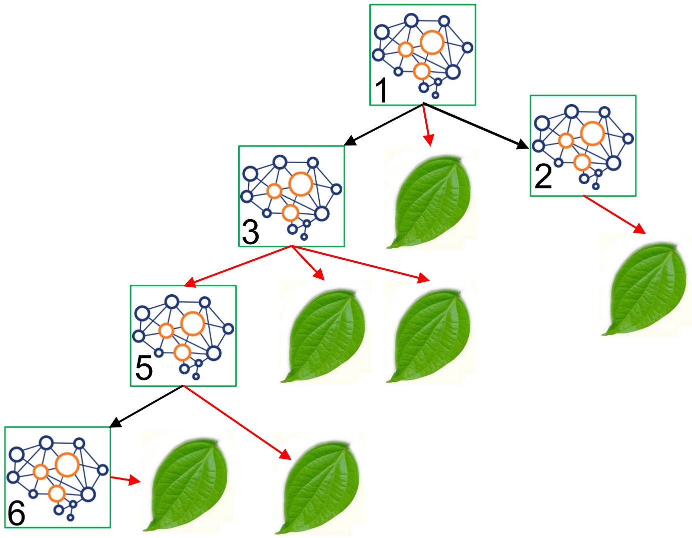

Physical Structure

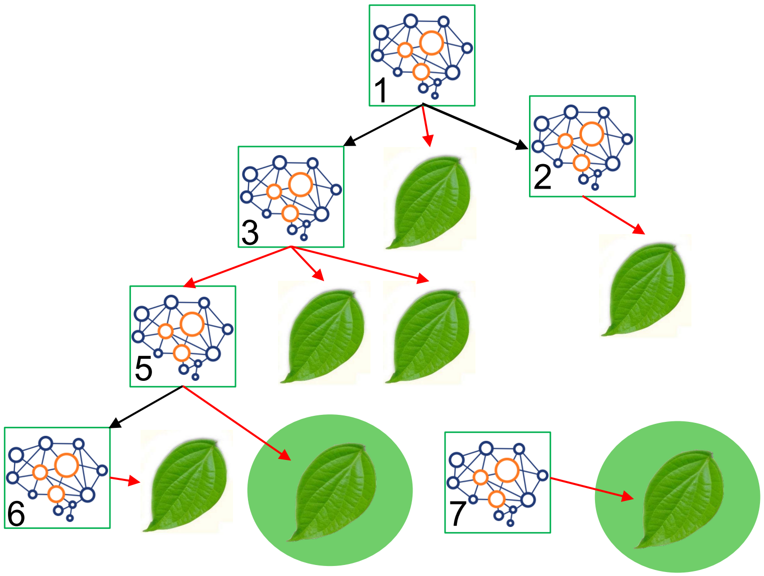





NetApp

# Overflow

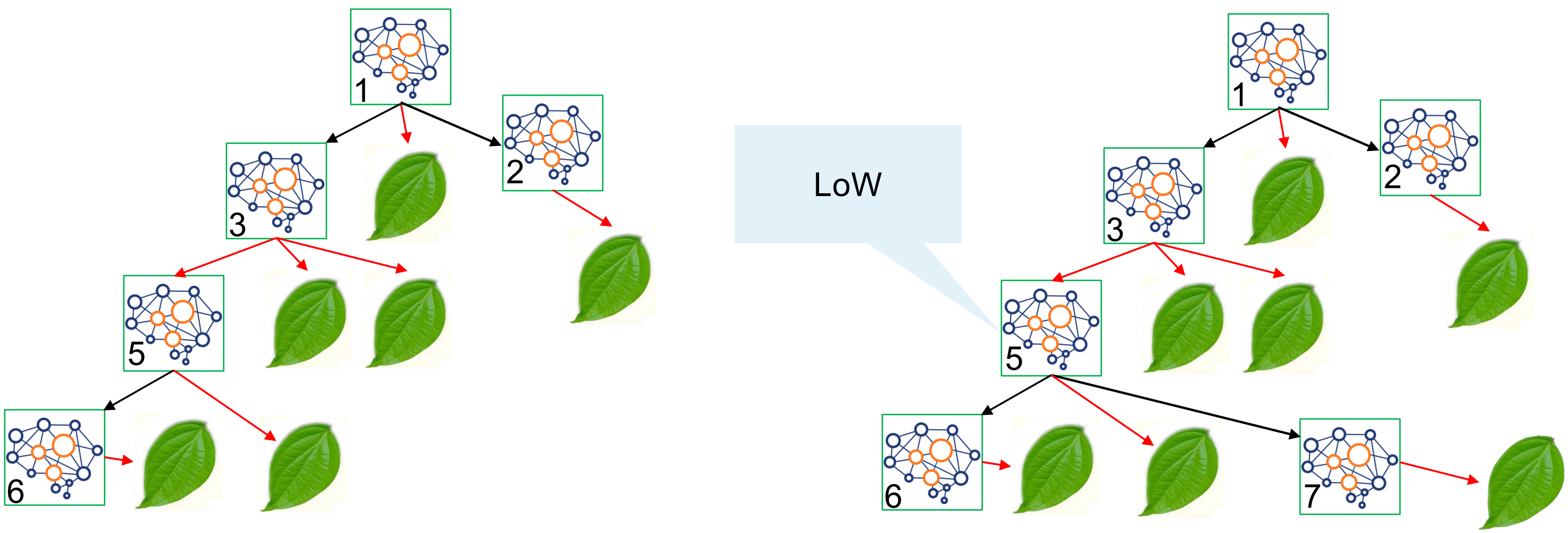





NetApp

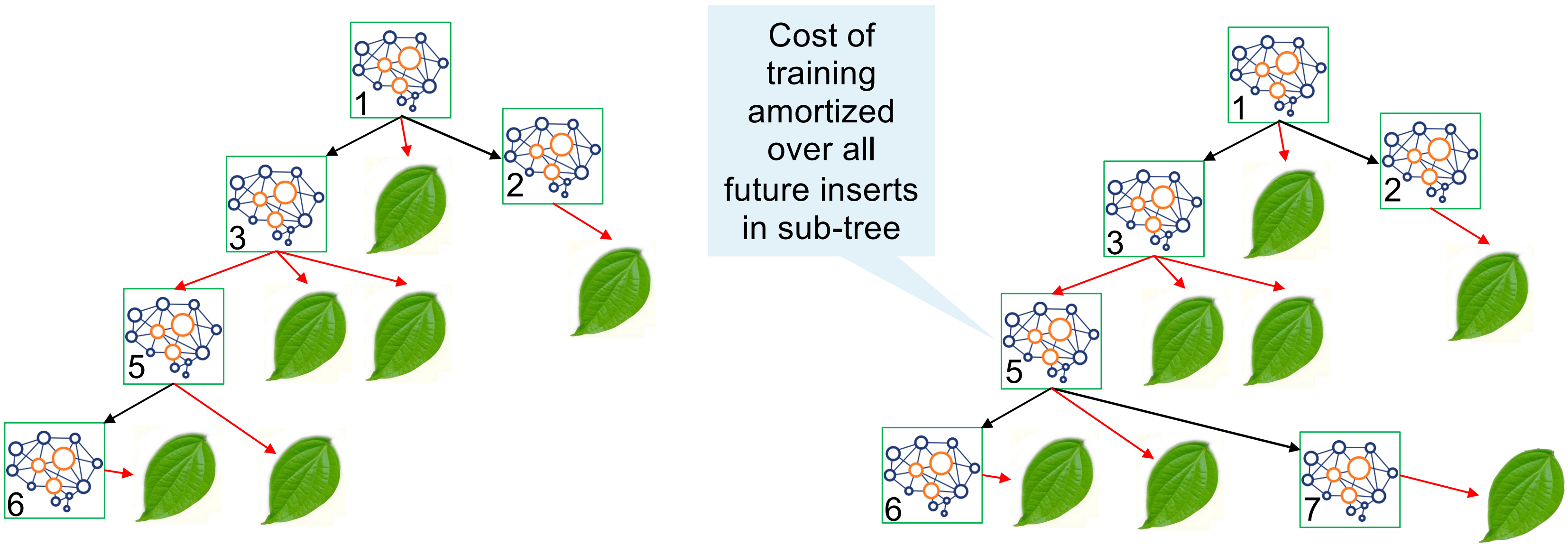
# Learning on Write (LoW)






NetApp

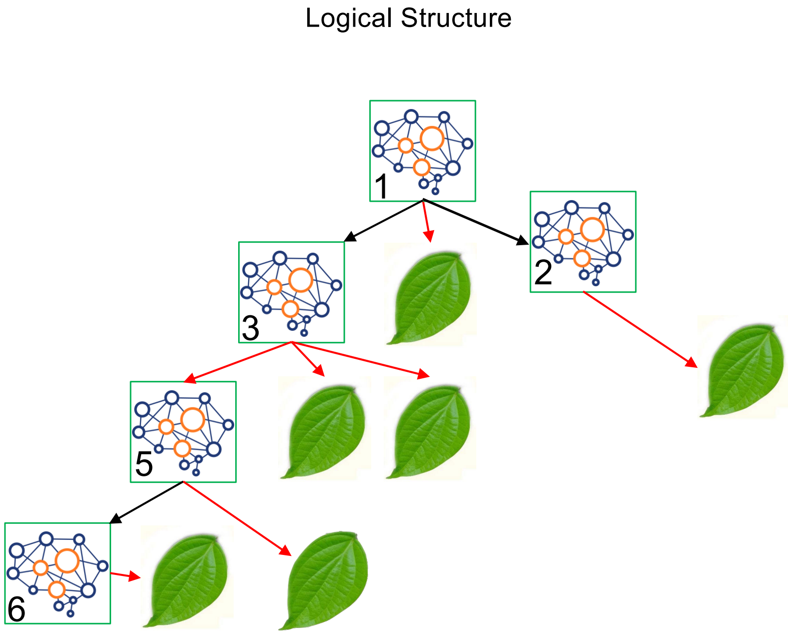
# Learning on Write (LoW)

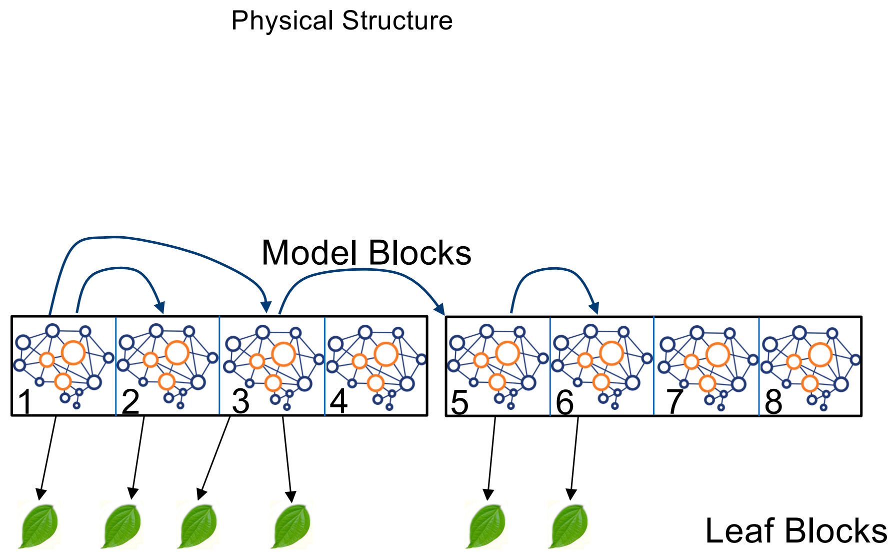





NetApp

# Neural Tree Media Access Tuning



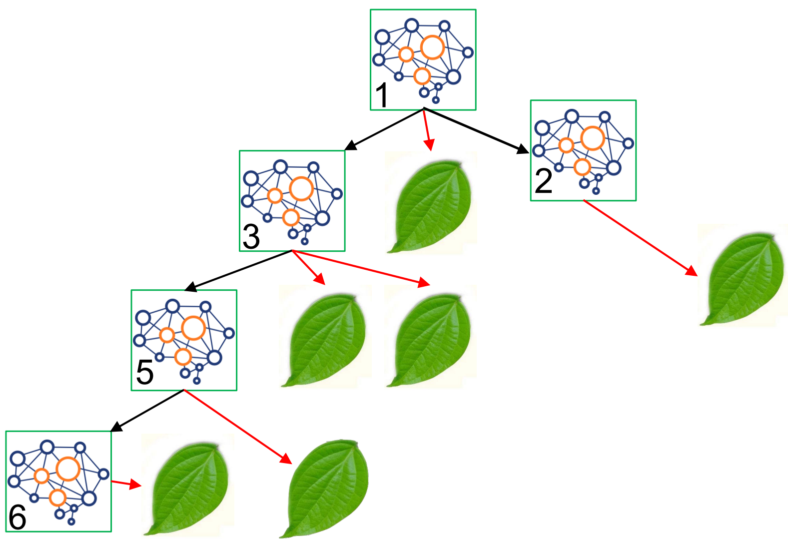





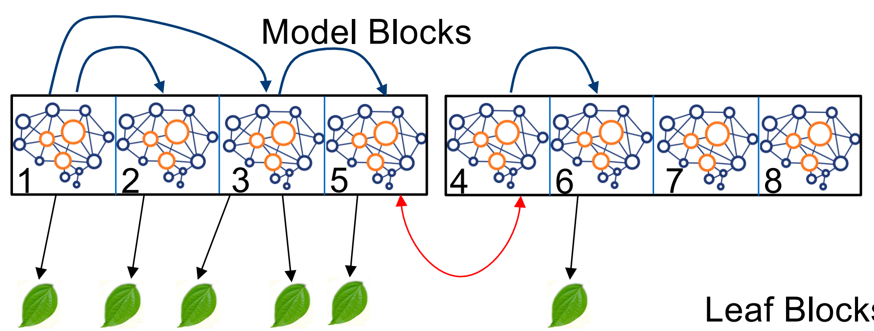

NetApp

# Neural Tree Media Access Tuning: Swap Models

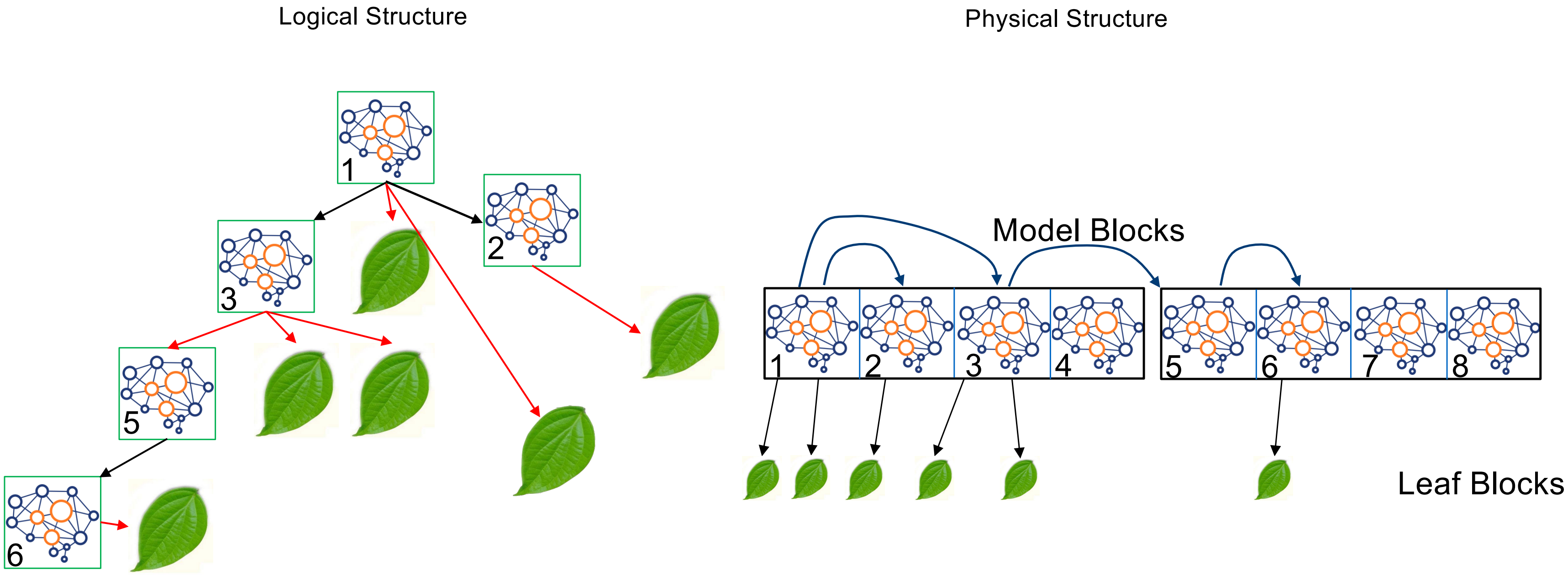
Logical Structure




Physical Structure



# Neural Tree Media Access Tuning: Short Circuit Models



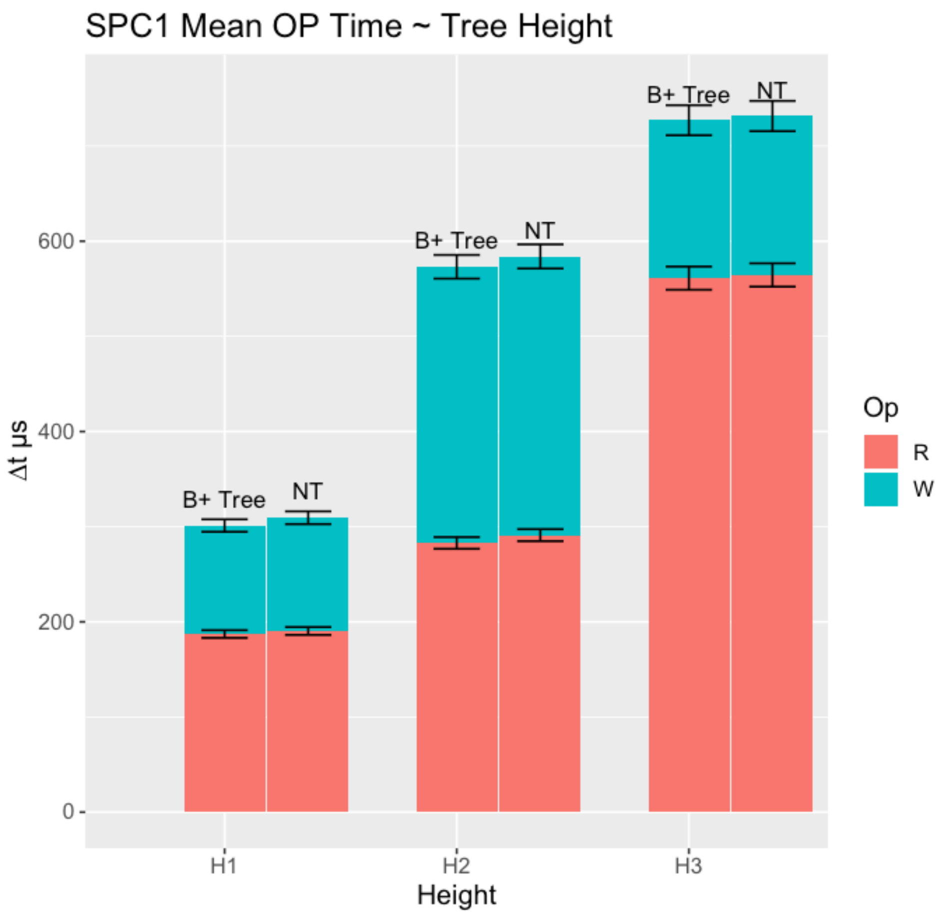


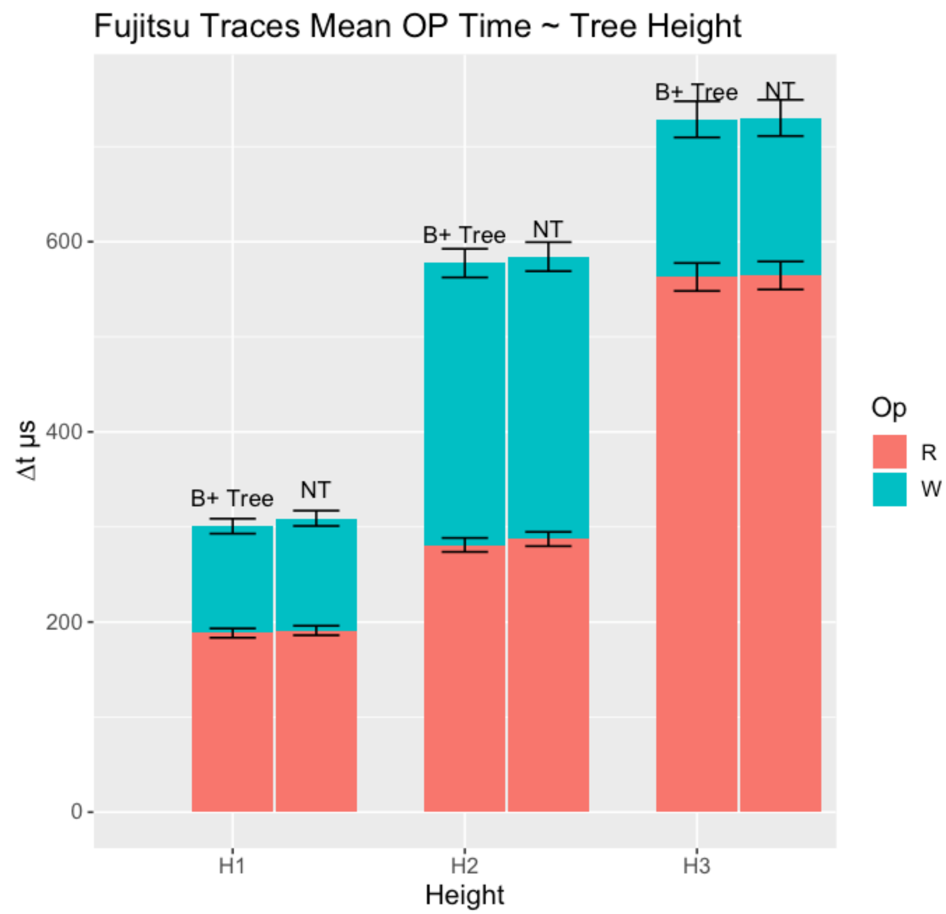

NetApp

# Neural Tree Models

- SLP neural networks so the number of weights is 3·N + 1
  - C `float`
  - Cacheline efficient
  - Information density higher per byte, 4k pages yield fan-out: 723 vs. 500
- Neural networks have ranges and domains of [-1, 1].
- The keys and values of an index are arbitrary: they can be *anything*.
  - The first job is to turn the key in to a number between [-1, 1]: $\alpha(key)$
  - The inference needs to be something useful (address of next model or leaf): $\beta(y)$
- Recursive: $f_i(\alpha(key); w_i) \to y,\ f_{i+1} = \beta(y)$
  - α and β are the "secret sauce" of a NT implementation.
- Training set constructed as: $\{ <x_1, \beta_1^{-1}(address_1)>, \ldots, <x_N, \beta_N^{-1}(address_N)> \}$
- α() maps a key to a known "good value", a value from the training set.
  - The guarantee: $\alpha\ (any\ key) \in \{ x_1, \ldots x_N \}$
  - Thus $y$ will also be in the training set, by design.
  - This means that error is totally controlled in the training process.

# Evaluation: Insertion as a Function of Height

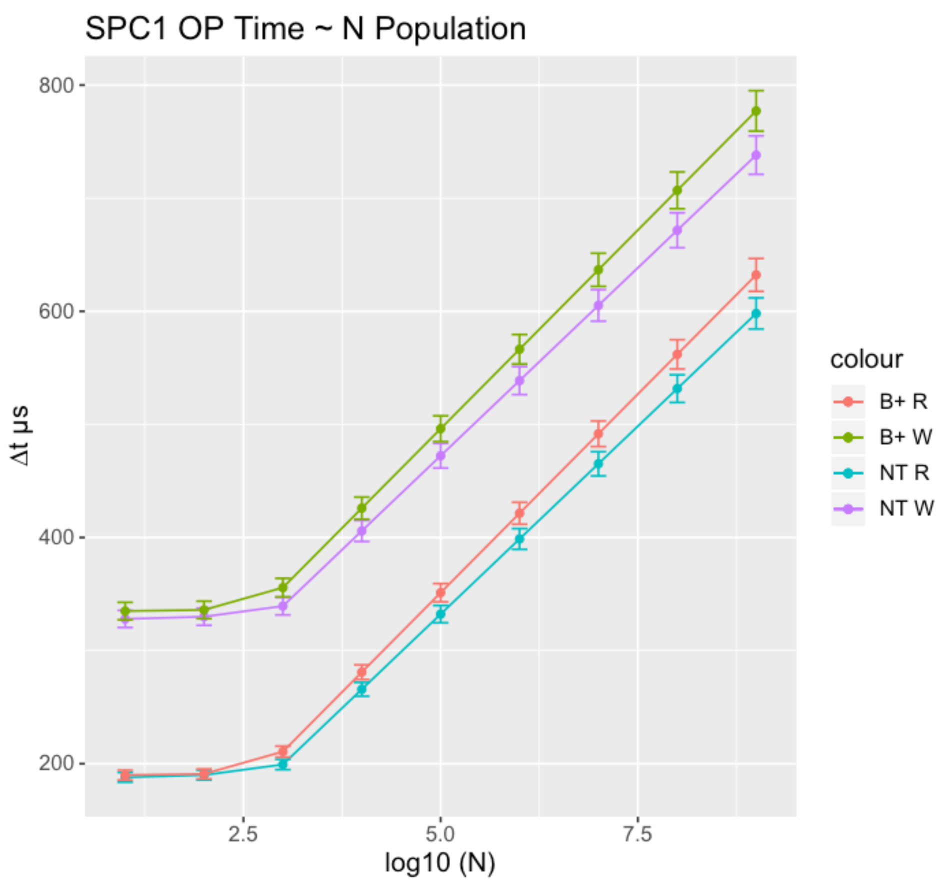


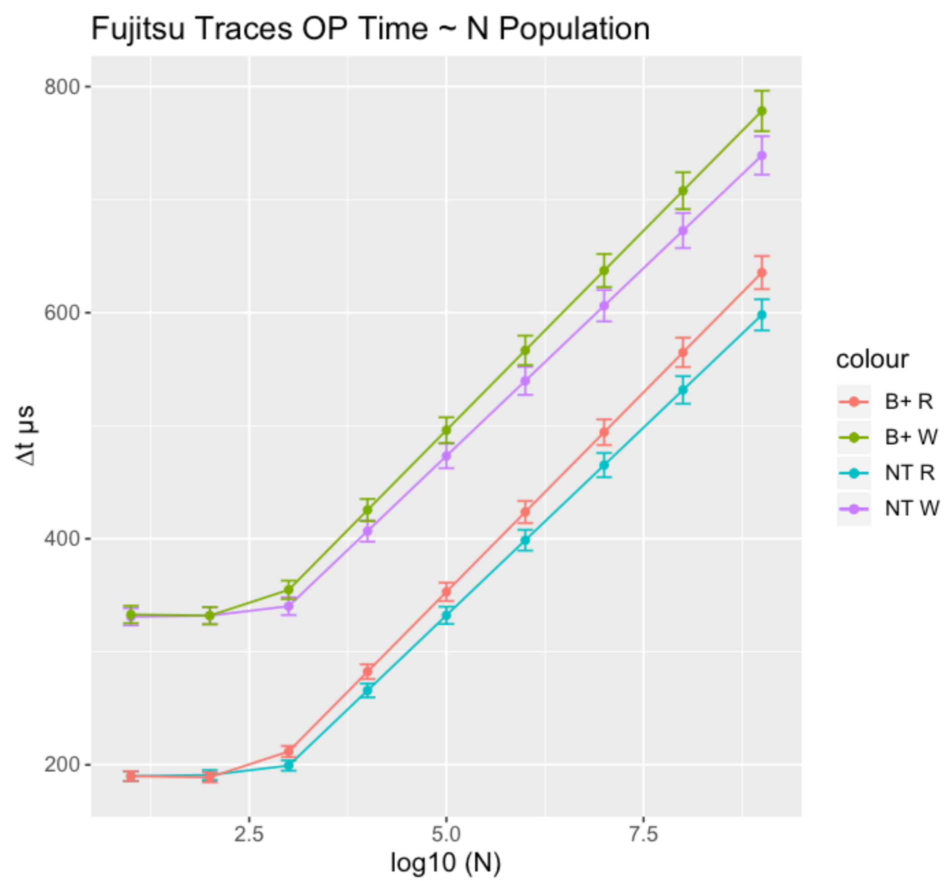






NetApp

# Evaluation: Effects of Population on Insertion Times






NetApp

## Future Work

- The ”secret sauce” of a Neural Tree implementation is the pair of functions, ($\alpha$, $\beta$). Many are possible. Neural Trees let us control the data layout. Are there richer ways?
  - Temporal: organize data according to creation or access time?
  - Ontological, what would annotated data look?
- Neural Trees can also mimic other data structures. For example, geometric applications often use R-Trees and Hilbert Trees. Can a Neural Tree can be used to mimic both?
  - Moreover, once a Neural Tree has been implemented, can new behaviors simply be programmed with a new $\alpha()$? This should be far easier (and cheaper) than implementing an entire new data structure.
- What would a file system based on Neural Trees look like?
  - Directory search could be so much richer than lexical matching.
- What do snapshots look like in a LoW world?
  - CoW on the physical structure would work, but can something be done with LoW in the logical representation?

Thank you.