ALEX: An Updatable Adaptive Learned Index

Omkar Desai

Advisor: Prof. Bryan Kim

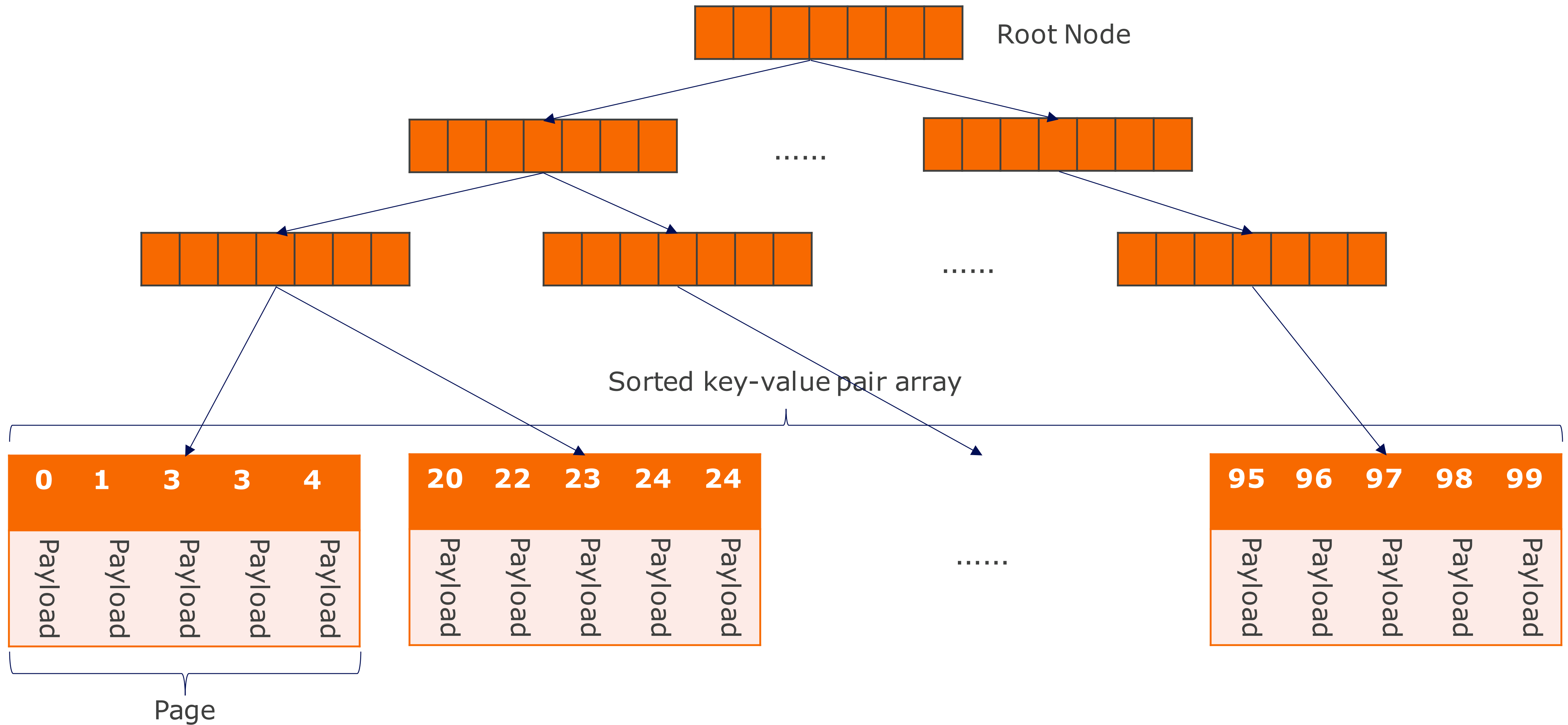
Fundamental building blocks of Data Systems

- B-Tree
- Hash Map
- Bloom Filters
- Queues
- Etc.

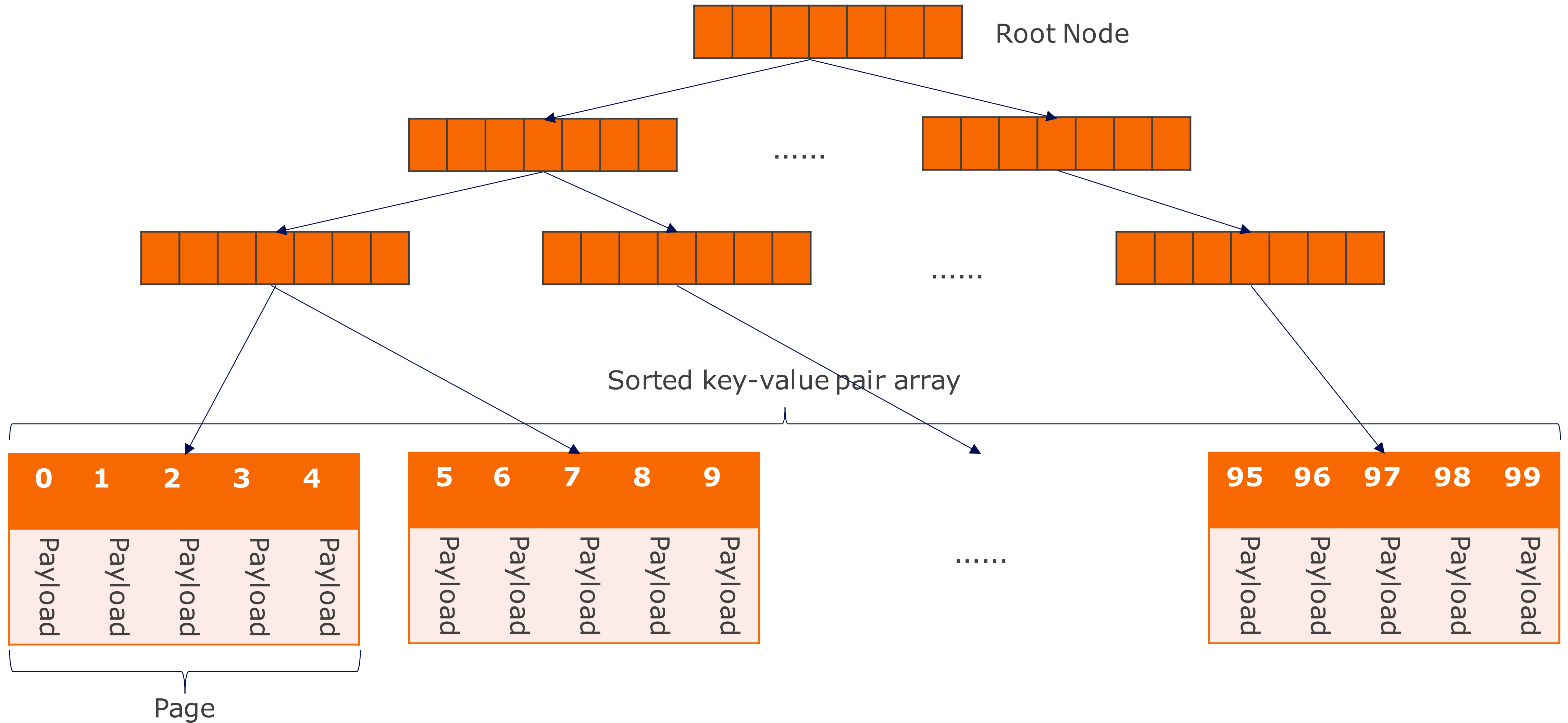
Fundamental building blocks of Data Systems

- B-Tree
- Hash Map
- Bloom Filters
- Queues
- Etc.

B-Tree example



B-Tree example: Integers from 0 to 100

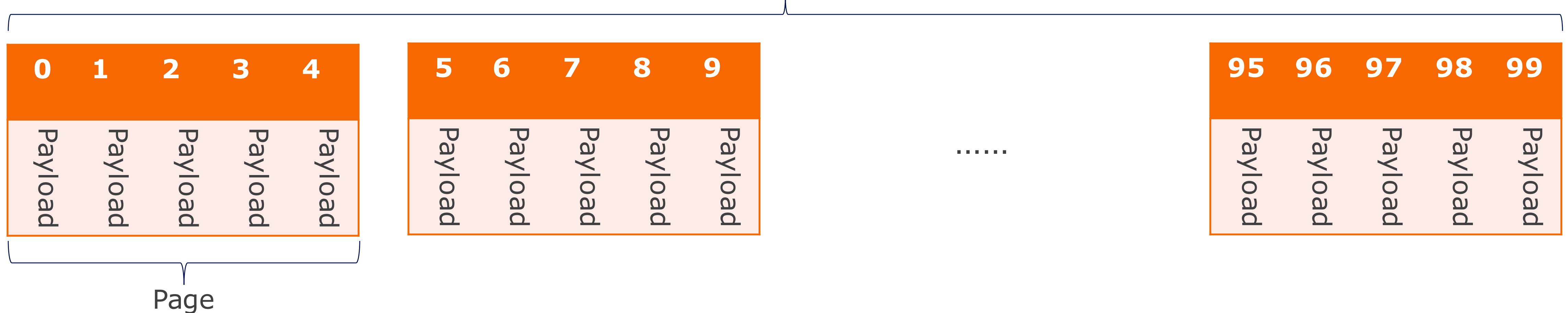


B-Tree example: Integers from 0 to 100

We could just put all the data into an array and fetch on the key

E.g.: data_array[lookup_key] O(1) lookup

O(1) memory

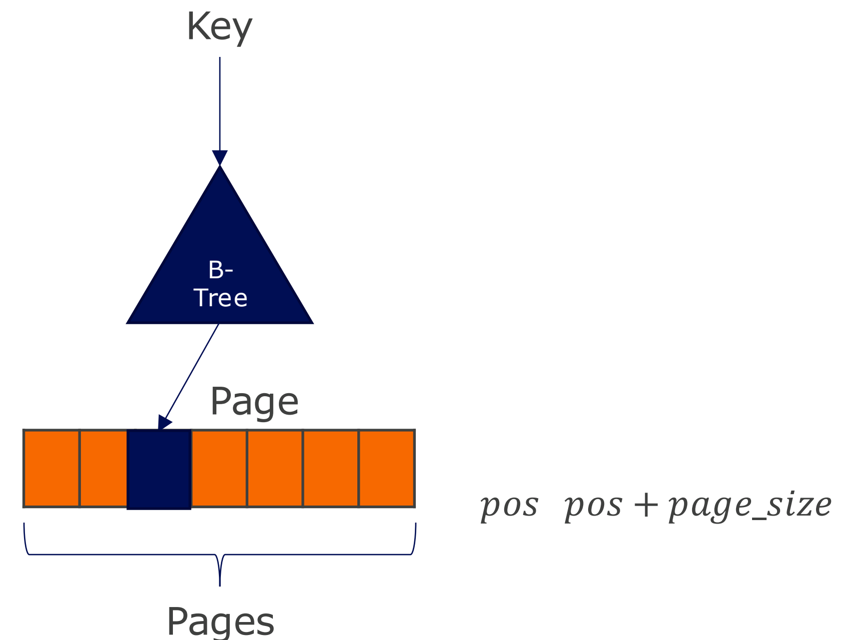


We want to know the distribution of data in a dataset

The B-Tree is actually building a model of the CDF of the data

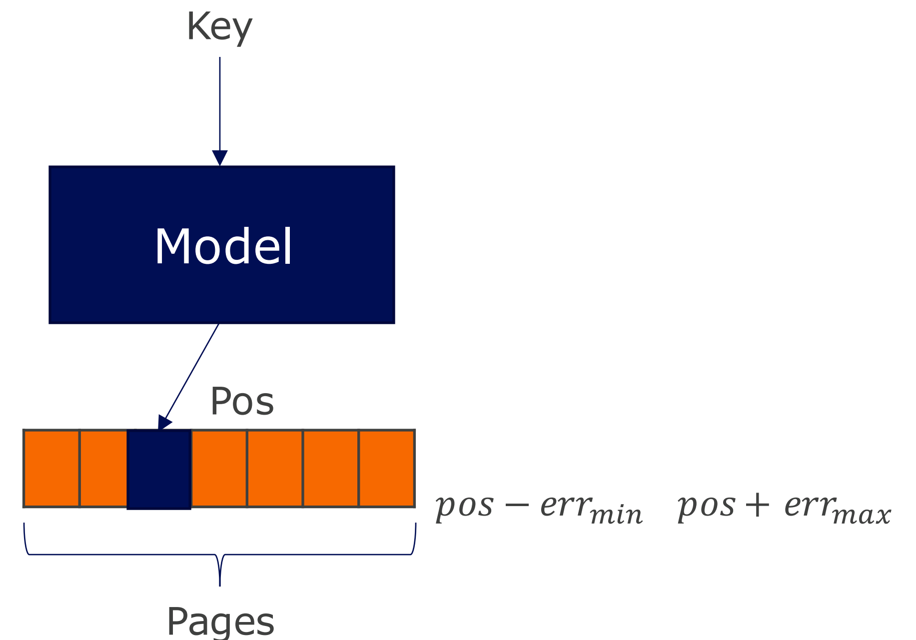
B-Tree

- A B tree maps a key to a page in the data
- Once the page is found, a search is performed to find the requested key
- The search bound is that page



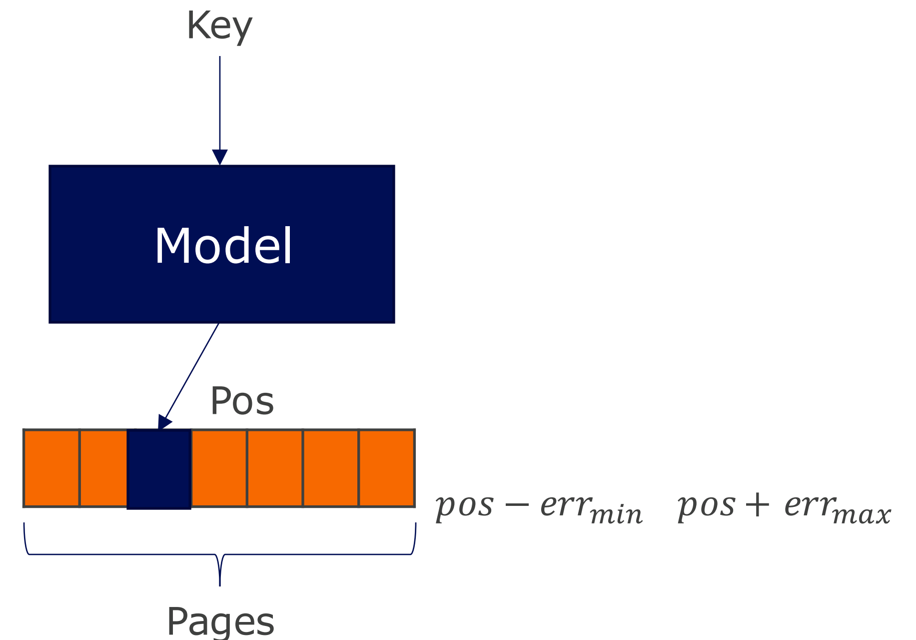
B-Tree as a model

- Replace B-Tree with model as: $f(key) \rightarrow pos$
- Position is predicted with an error bound
- The last step is to search within that error bound

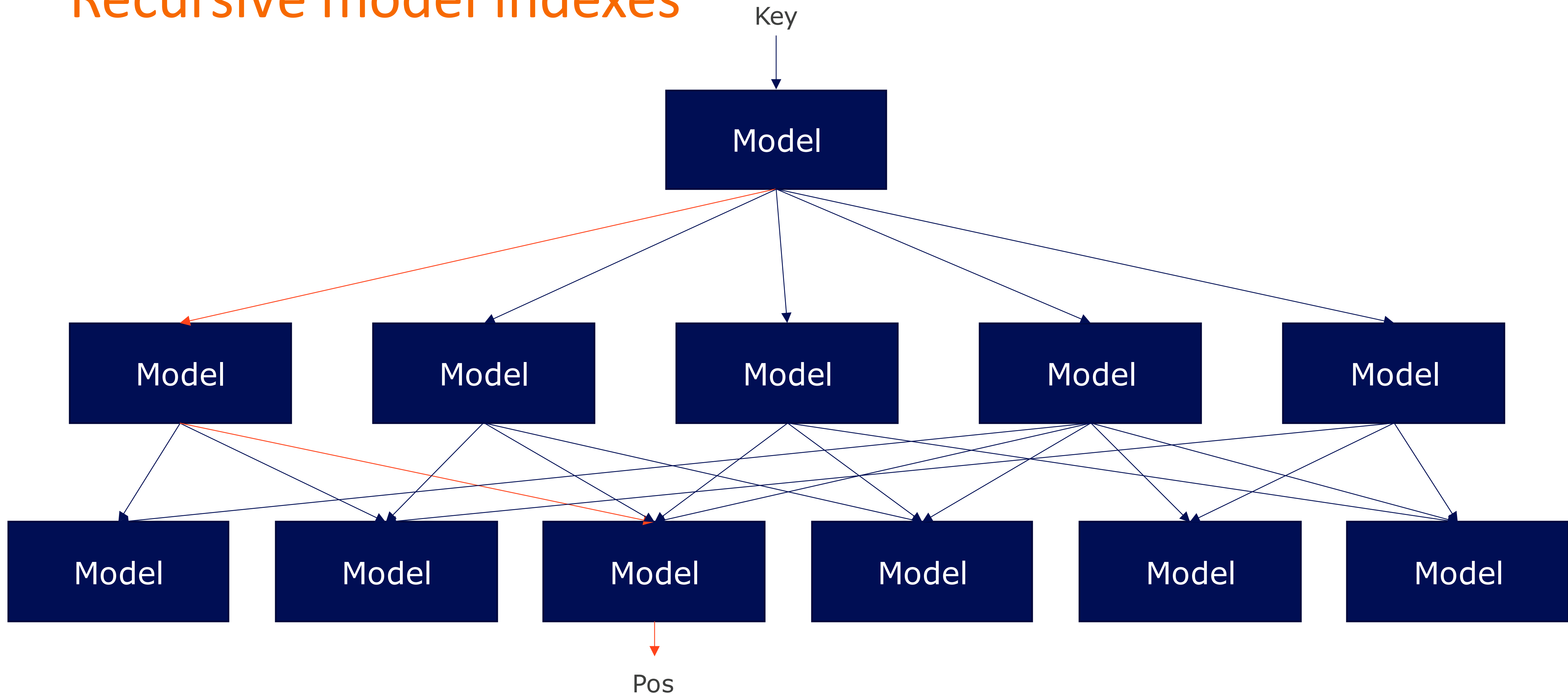


B-Tree as a model

- Replace B-Tree with model as: $f(key) \rightarrow pos$
- Pos is predicted with an error bound
- The last step is to search within that error bound
- **Essentially, we are trying to build a CDF of the data distribution.**

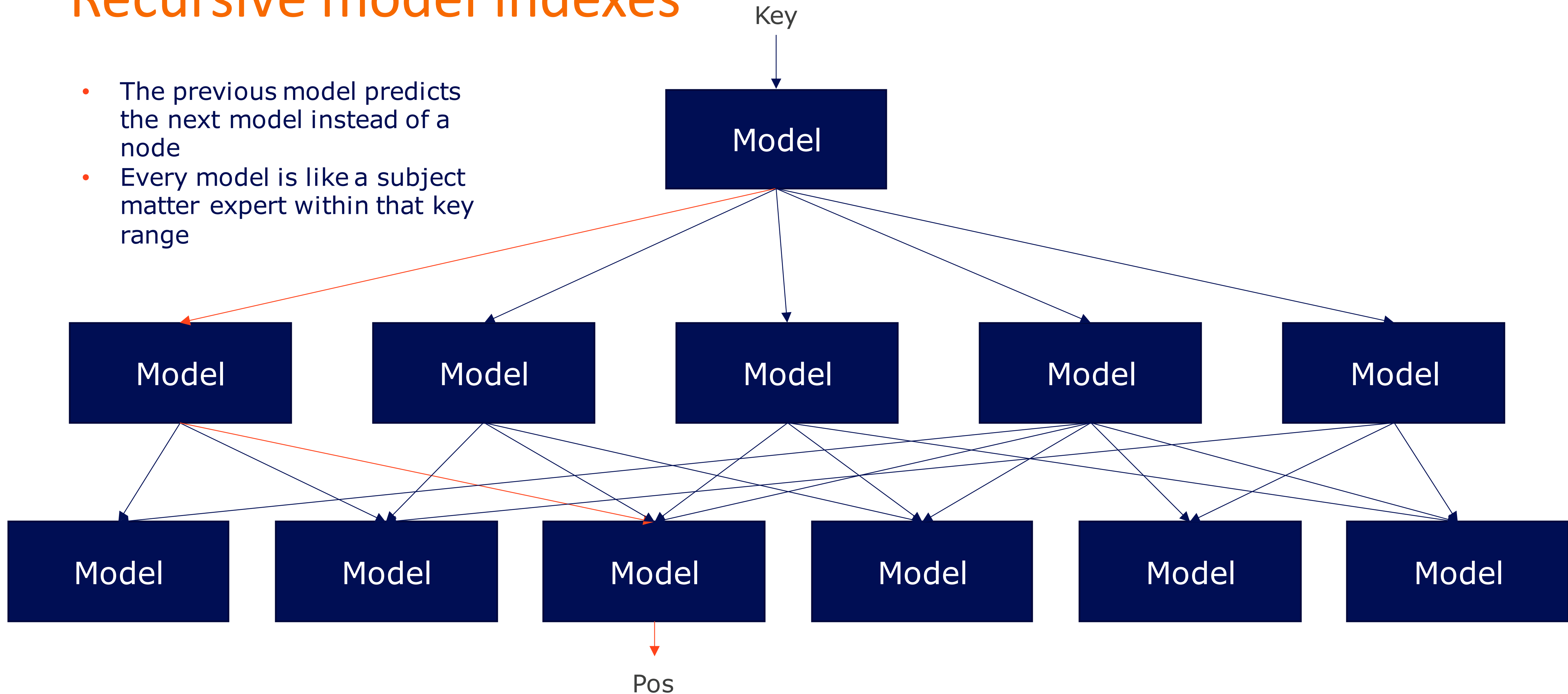


Recursive model indexes



Recursive model indexes

- The previous model predicts the next model instead of a node
- Every model is like a subject matter expert within that key range



ALEX

An Updatable Adaptive Learned Index

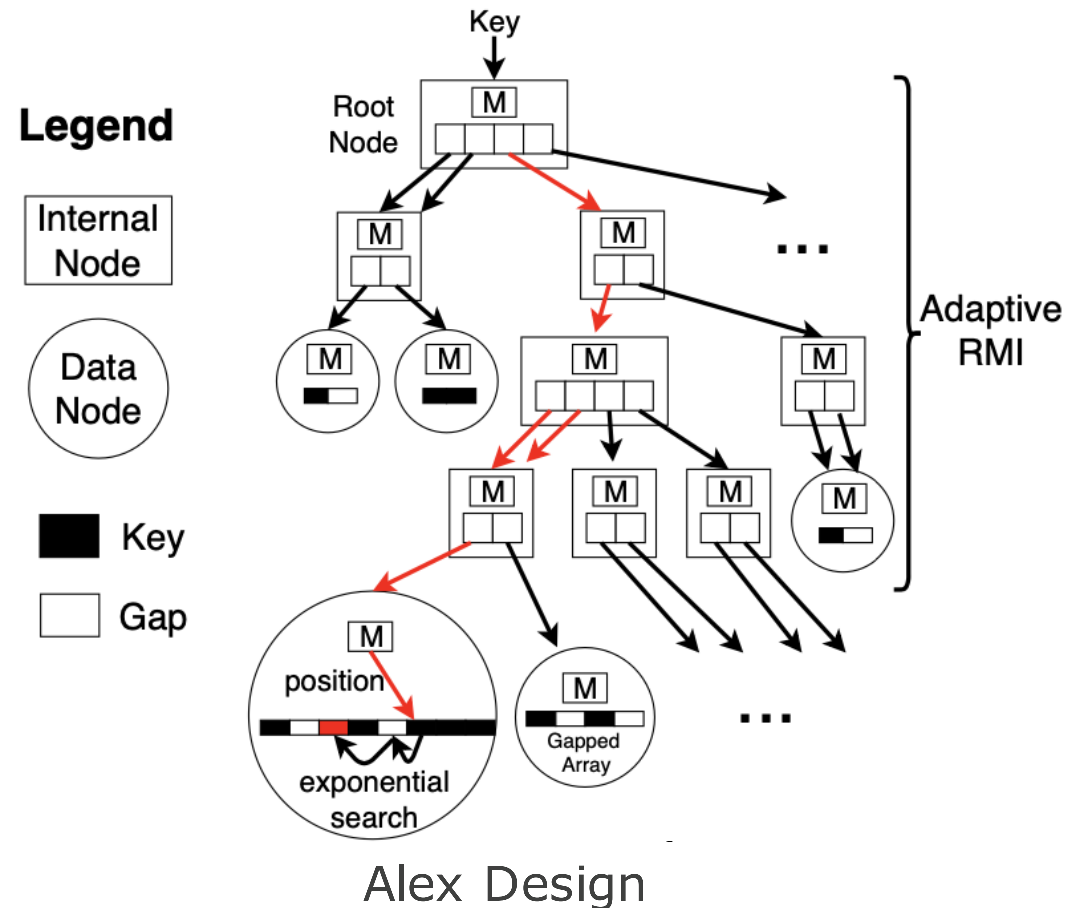
OLTP Operations

ALEX is a fully dynamic learned index data structure that aims to provide efficient support for:

1. point lookups
2. short range queries
3. Inserts
4. Updates
5. Deletes and
6. bulk loading

Important contributions

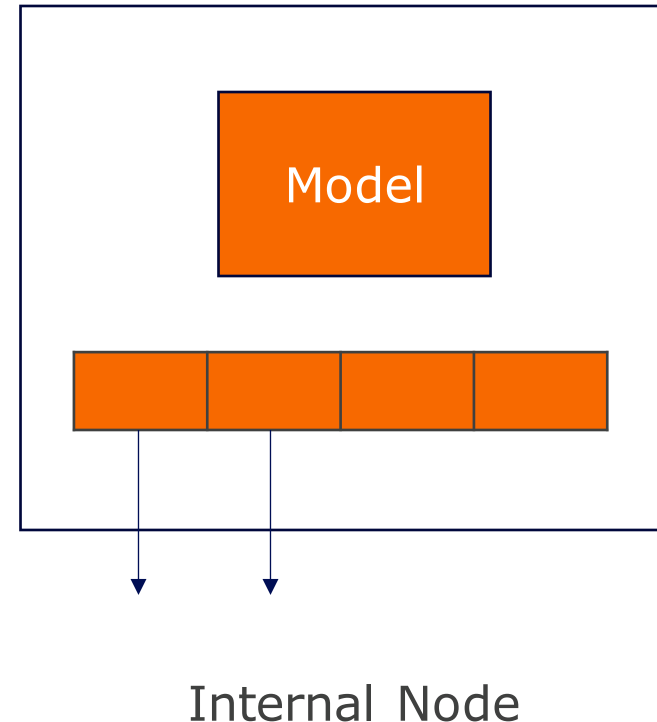
- Gapped Array (GA) layout of data nodes amortizes the cost of inserts
- RMI that can be updated dynamically and efficiently at runtime based on the workload and data distribution
- Exponential search within the data node from the predicted pos
- Model based insertion inserts into the position predicted by the model



Alex Design

Data Structures – Internal Node

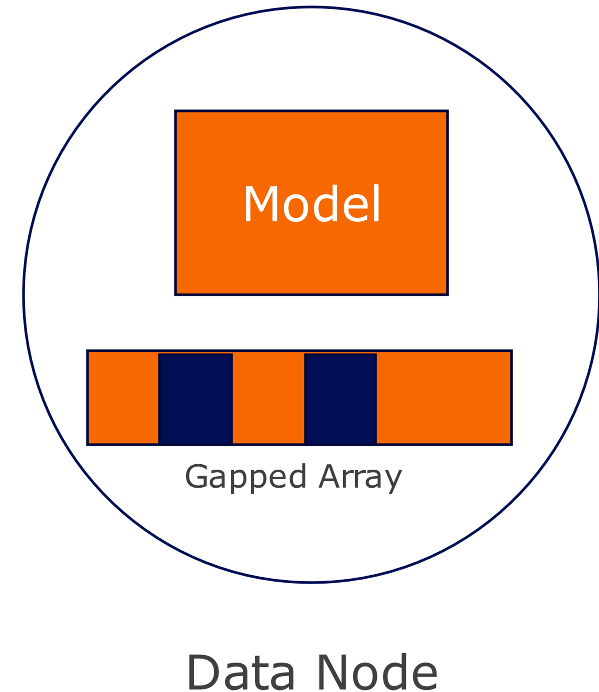
- Internal nodes store a linear regression model and an array containing pointers to children nodes
- Internal nodes compute the location in the pointers array, of the next child pointer to follow
- the role of the internal nodes in ALEX is to provide a flexible way to partition the key space so as to create a roughly linear distribution of data



Internal Node

Data Structures – Data Node

- The Data Node stores a linear regression model and 2 gapped arrays, for keys and payloads
- The gapped array absorbs new inserts and amortizes the cost of new inserts
- Every Data Node maintains a bitmap which tracks the weather each node in the GA is occupied or a gap



Algorithms – Insert with model-based inserts

- The model predicts the insert position of the new element using model-based inserts
- If prediction is incorrect (if it destroys sorted order) exponential search is used
- If the insert pos is a gap, insertion is done. Else a gap is created by shifting data before the insert (inefficient)

Algorithms – Insert with model-based inserts

- To insert into a full data node (70% threshold), ALEX uses 2 methods and selects between them based on simple cost models.
  - Expansions
  - Splits
- Expansion – Allocate a larger Gapped Array, scale or retrain the linear regression model, perform model-based inserts of all elements.
- Splits - Split nodes either sideways or downwards. Keys are split among the 2 new data nodes so that each node is responsible for half the key space.

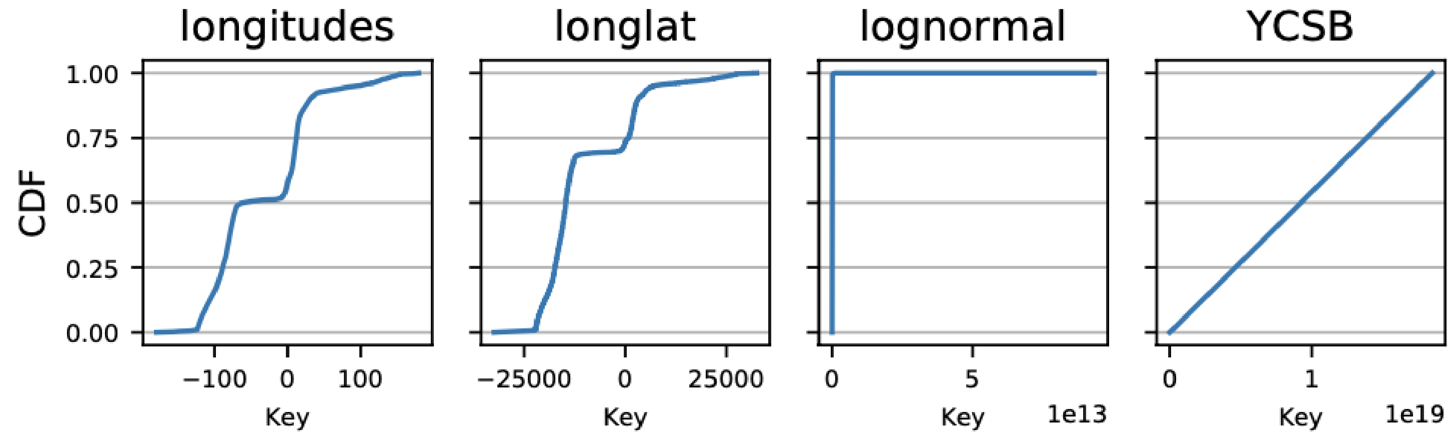
Algorithms – Bulk Inserts

- Bulk insertion is used to insert a large amount of data at initialization or to rebuild an index.
- At each node, decide weather the node should be internal or data.
- If internal, compute the fanout of the node, divide the keys equally and recurse on each of its child nodes.

Evaluation methodology

- Alex is tested with B+Tree, model B+Tree, Learned Index and Adaptive Radix Tree (ART)
- 4 Datasets:
  - Longitudes
  - Longlat (non linear)
  - Lognormal
  - YCSB (linear)
- 4 Workloads:
  - Read only
  - Read heavy (95%R, 5%W)
  - Write heavy (50%R, 50%W)
  - Short range queries (95%R, 5%W)
  - Write only

Evaluation methodology



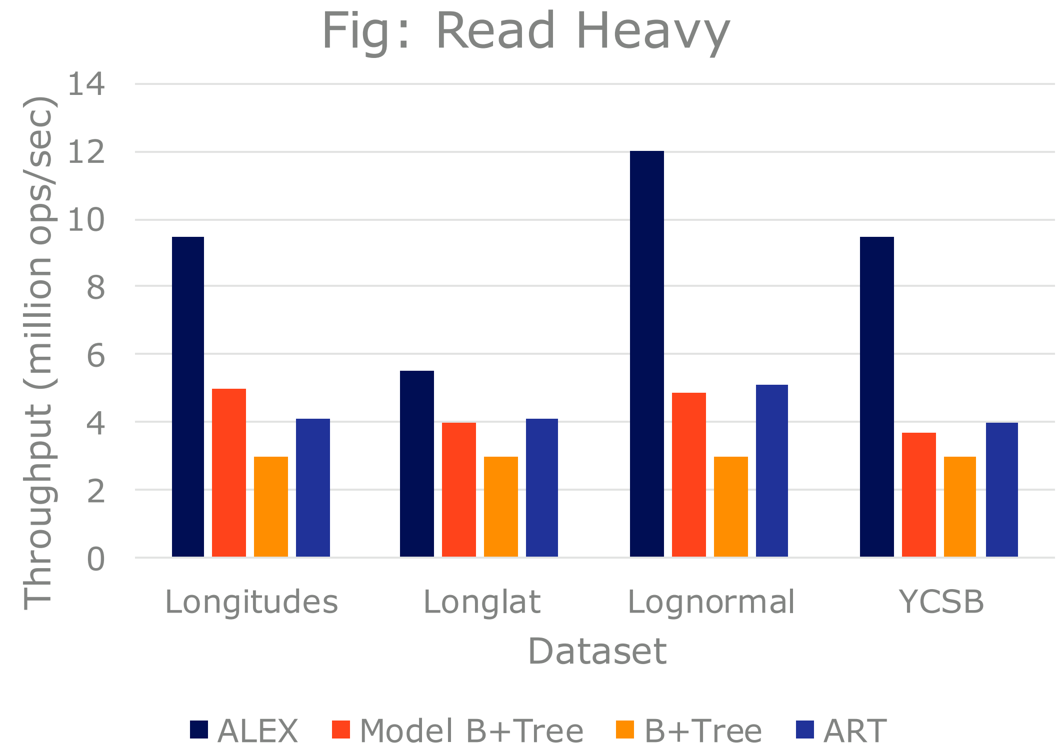
	Longitudes	Longlat	Lognormal	YCSB
Num Keys	1B	200M	190M	200M
Key type	Double	Double	64 bit int	64 bit int
Payload size	8 Bytes	8 Bytes	8 Bytes	80 Bytes
Total size	16 GB	3.2 GB	3.04 GB	17.6 GB

Performance Metrics – Snapshot

- On read only workloads, ALEX achieves up to 4.1×, 2.2× higher throughput and 800×, 15 smaller index size than the B+Tree and Learned index, respectively.
- On read write workloads, ALEX achieves up to 4.0, 2.7x higher throughput and 2000x, 475x smaller index size than the B+Tree and Model B+Tree, respectively.
- Alex takes 50% more time to perform bulk loads.

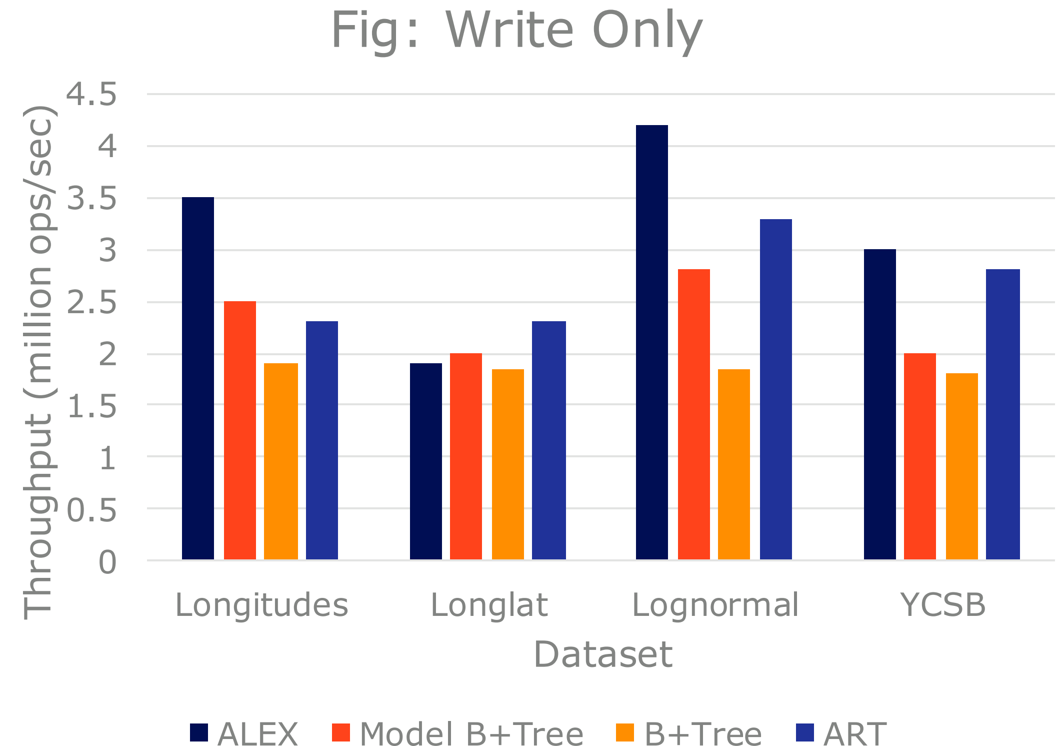
Performance Metrics – Read heavy workload

- Read heavy workloads perform very well on ALEX, doing better than B+Tree, Model B+Tree and ART.
- Space requirement is small when the dataset is highly linear.
- In more difficult to model datasets, index size of ALEX is comparable to Model B+Tree and B+Tree. This happens because the average and max depth increases to 1.5 and 4 respectively (compare to 1 and 1 for YCSB).



Performance Metrics – Write only workload

- Write heavy workloads perform comparable to B+Tree and Model B+Tree performance.
- In case of indexes that are hard to model e.g.: Longlat, Model B+Tree and B+Tree perform better than ALEX.
- ALEX is slow because of frequent splits and copies.
- Space requirement is still small



Performance Metrics – Lessons learned

- ALEX achieves a considerable speedup while maintaining a significantly smaller index size.
- 4x faster than B+Tree with a 2000x smaller index size.
- For inserts, learned index is orders of magnitude slower than ALEX and B+Tree.
- As the ratio of writes is increased, the performance of ALEX takes a hit. This is because of copying and splitting operations on the nodes.
- Bulk loading is slow on ALEX. This is because of frequent creation of *fanout trees.*
- Performance is impacted when in model-based inserts, the predicted insert position is not a gap.

Lessons Learned

- Data distribution can be converted into a CDF that can be learned using very simple linear regression models
- Models are magnitudes smaller than traditional B+Tree nodes
- Instead of learning using 1 complex neural network, Using many simple regression models that are *'subject matter experts'* in a particular key space is more efficient as the CDF may change over time
- Overfitting is a good thing here, generally
- The fanout tree is the cause of the slow bulk load performance and that could be an interesting area to look in

Thank you

Omkar Desai
PhD student
Advisor: Prof. Bryan Kim
odesai@syr.edu

References

- Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In Proceedings of the 2018 International Conference on Management of Data (Houston, TX, USA) (SIGMOD fi18). Association for Computing Machinery, New York, NY, USA, 489–504. https://doi.org/10.1145/3183713.3196909
- Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20). Association for Computing Machinery, New York, NY, USA, 969–984. DOI:https://doi.org/10.1145/3318464.3389711
- https://www.youtube.com/watch?v=NaqJO7rrXy0&t=897s&ab_channel=stanfordonline