Notes :Indexing for Interactive Exploration of Big Data Series Paper

What is the problem :

Large data series – produced by numerous applications with the **need** to query them as soon as the data is available

The currently indexing mechanism : which involves building the FULL tree upfront – at indexing time

Leads to a significantly large data-to-query time.

Adaptive Data Indexing – is a technique that reduced the data to query time

Main Idea

* interactively and adaptively build parts of the index – driven by workload/query.
* Initialization cost is kept at very low levels - creating only a partial tree structure deep enough to not penalize
* the first queries with a lot of splits.
* - Algorithms focus on how to incrementally sort columns in main memory colm stores.
* .this is more focused on tree index, ( as it focuses on queries involving data series - questio below)

3 new approaches presented

* ADS - minimizes the data to query gap by delaying actions until they are absolutely necessary.
* (ADS+) Filling it on demand - As more queries are performaned on the columns - column reaches closer to sorted state
* (PADS+) adapting leaf sizes on-the-fly and with varying leaf sizes across the index -

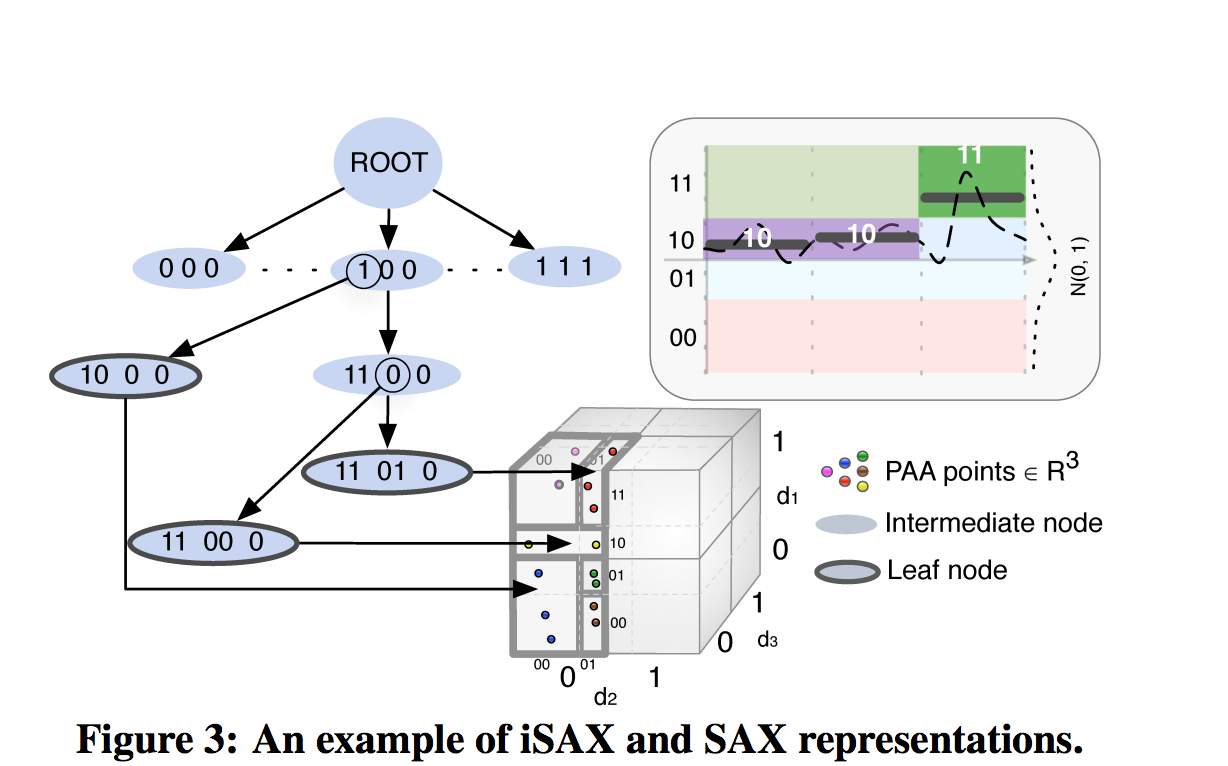
TO Index to Not TO Index:

* only beneficial when - single long data series and queries for small subsequences
* not useful for mixed database of several data series
* indexing required to efficiently support data exploration

Background Information:

Data Series Representations and the iSAX Index

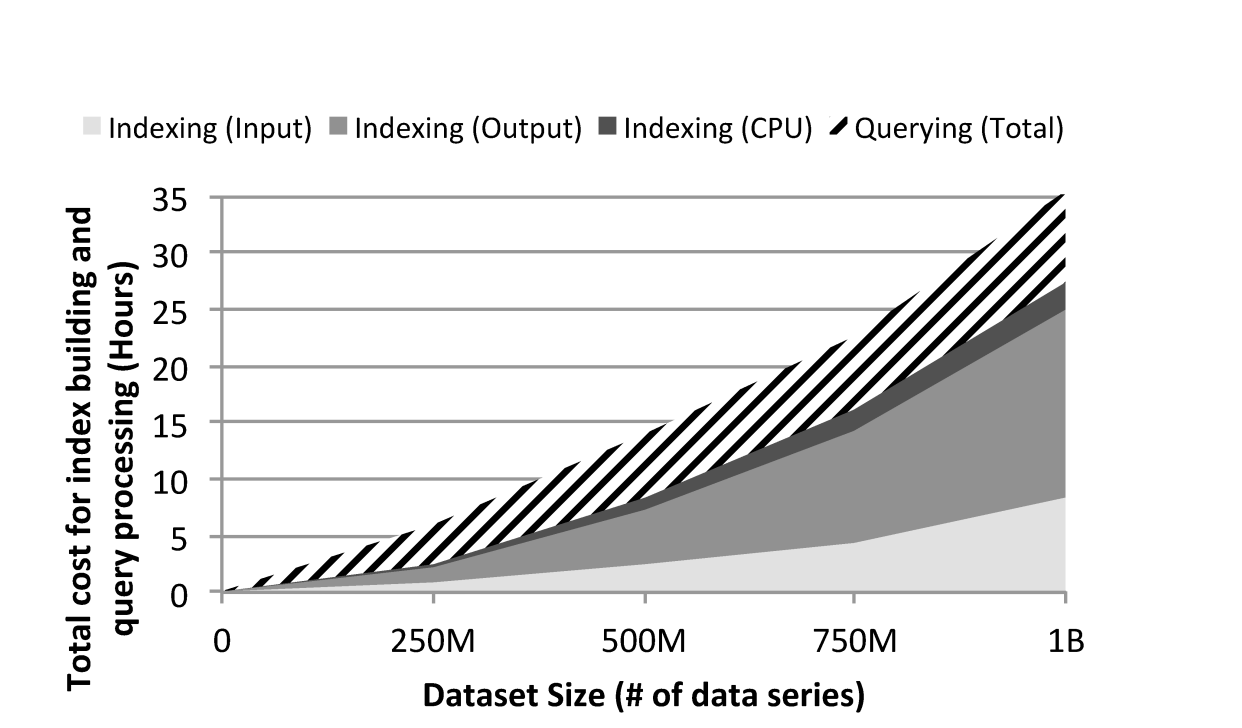
1. Segmented means or Piecewise Aggregate Approximation (PAA)
2. Bits have ordering - need as many bits as
   1. Dimensionally reduces the time domain by dividing the series into segments and calculating average value for each segment
   2. The SAX ( Symbolic Aggregate approximation) technique, the value space is partitioned into segments that follow the normal distribution.
   3. Each value can then be represented by a character/smaller number of bits – that corresponds to the segment that it falls in.
   4. This leads to a representation with a very small memory footprint – imp requirement for very large data series.



* 1. In the figure above the data series is divided into 4 equal segments – 00, 01,10, 11.
  2. SAX word for this segmentation of size 3 will be 10,10,11

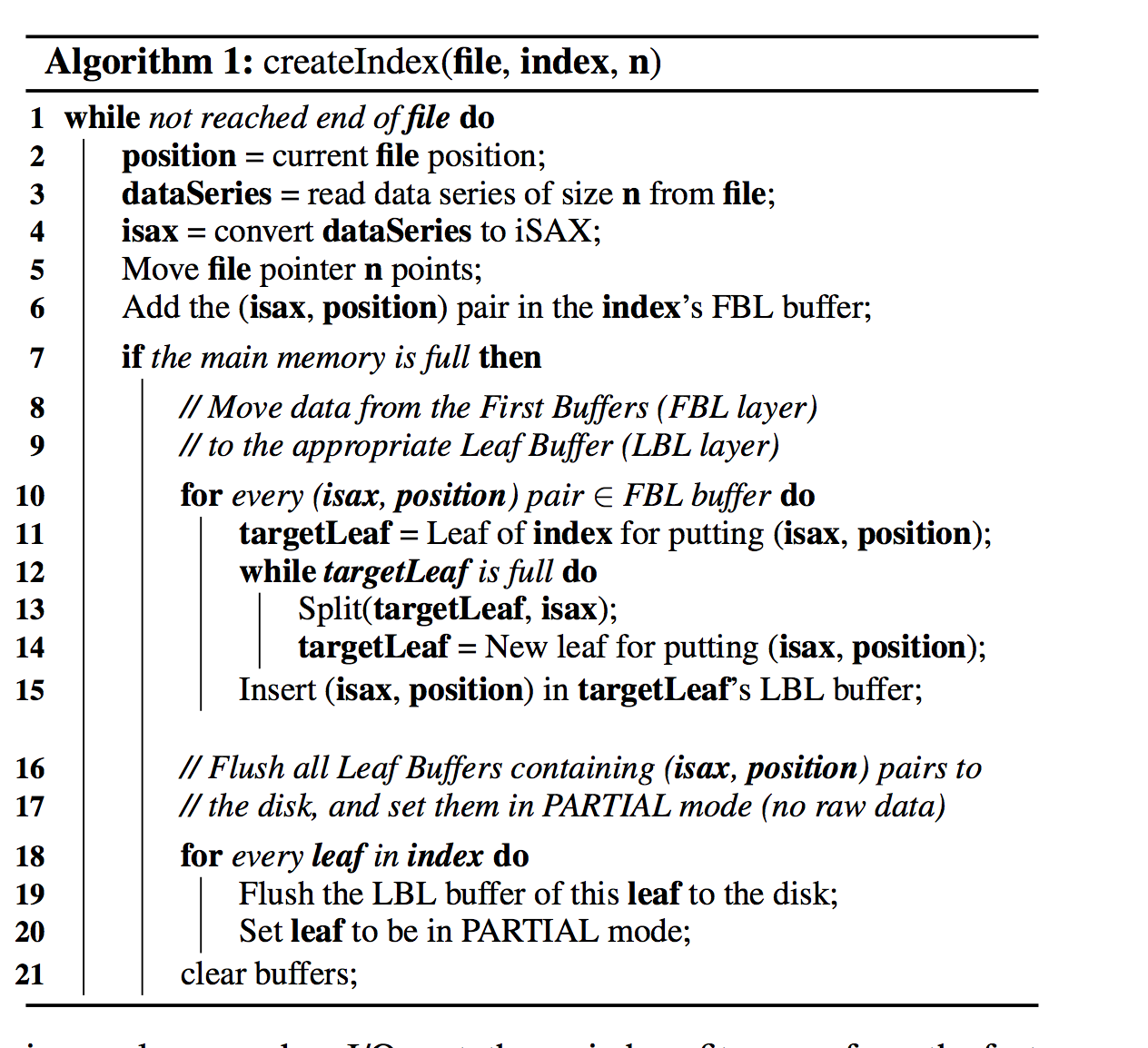
# ADS : Adaptively loading data series in the index:

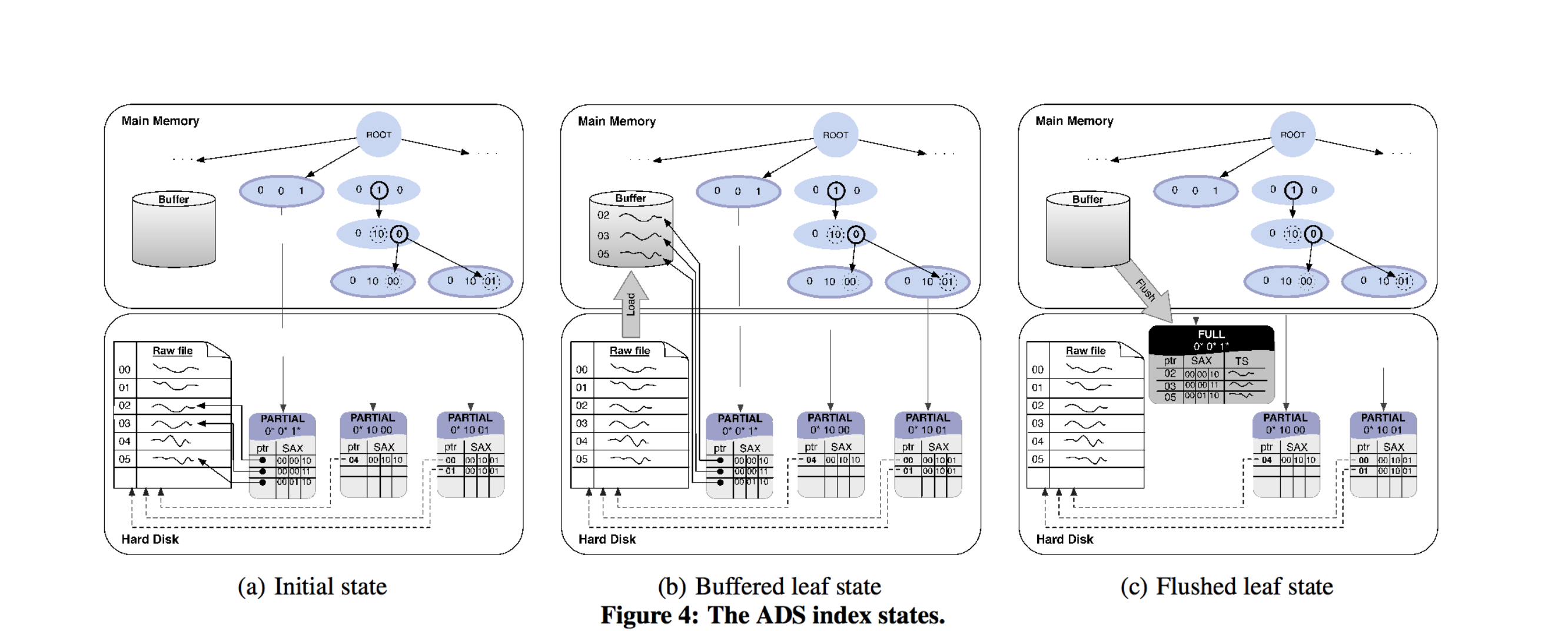
1. Construction of the leaf nodes of the index to the query time
2. Indexing phase only creates iSAX representation of each data series
3. The actual data remains in raw file
4. Loaded in the tree on a query request
5. From the figure we can see that iSAX is I/O bound – in reading raw data and then serializing the tree .



# ADS Index Creation:

1. scans the file
2. record the position
3. read t size n of data series
4. convert it to iSax representation,
5. add the iSAX and the position info to the buffers
6. when the buffer is full – move the data to the appropriate leaf buffer - split as when needed
7. Sequentially flush each leaf buffers in the index to the disk
8. Leaf nodes kept in the disk





performance improvement :

1. Does only one scan of the data series
2. Does not move raw series through the tree
3. Leaf is light weight – data series remain in the raw file. – helps with I/O and memory

Question : For example, a data series of 256 points with a float precision of 4 bytes, can be efficiently summarized with 16 characters of 1 byte each.

## Buffering :

1. Buffers at 2 levels :

a. FBL – First Buffer level - set of buffers corresponding to the children of the root

b. LBL – Leaf Buffer Level – second layer corresponding to the leaf of the index – on the disk

## Querying and Refining ADS

1. Refines the index during query processing time
2. Querying process consists of
   1. Converting the query to iSAX representation
   2. Search the index nodes for a matching iSAX
   3. If the search ends in a leaf node marked –“Partial”
      1. Fetch the partial leaf from the disk into memory
      2. Reads all positions in this leaf
      3. Sort these positions to ensure sequential access of raw file( which is good for performance – reduces cache misses)
      4. Fetch the raw data series
      5. Assigned to this leaf node and kept in the LBL ( Leaf buffer level )
      6. The leaf node contains pointers to this buffered data
      7. LBL will get flushed to the disk when the main memory is full.
      8. When leaf node is flushed it is marked as “FULL’
      9. Any future queries will need to fetch the leaf node from the disk ( if flushed ) or read it from LBL.

Question : Real distance from the query ? what does it mean – BSF ( Best So far)

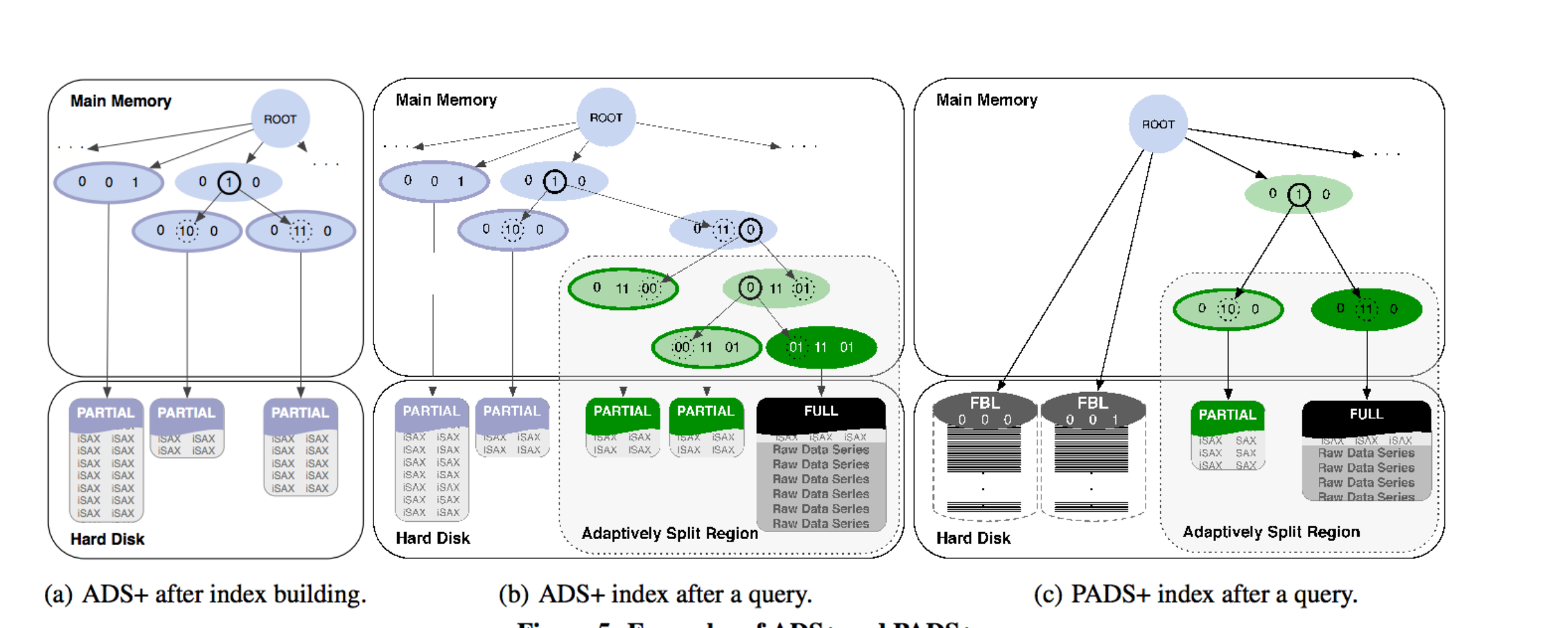
# ADS+ Index : Adaptive Leaf Size

1. Motivating the idea of ADS+ Index : Split cost of the ADS index
   1. Splits are expensive – I/O involved in the data movements to/from disk
   2. Split – directly effected by leaf size : tree with big leaf size – leads to faster index, slower query

Main Idea :

1. Two different leaf sizes
2. Large leaf size – optimize initialization cost . This will reduce split cost.
3. Small leaf size – optimize query cost . (Rely on the query to appropriately split the leaf , reduce leaf size in the hot areas
4. Future queries benefit from every split operation – as relevant data can be found by traversing the tree instead of scaning large leaves.
   1. Initial tree built with a constant ( large ) leaf size
   2. Querying – ADS+ refine the index leaf on-the-fly by recursively splitting it till leaf size <= query leaf size
      1. While leaves are created due to split – not all of them are needed for query
      2. These additional leafs are left as it – with a size < initial lef size and > query leaf size.
      3. When workload arrive that need these additional leaf- they will be split to reach the query leaf size.
5. Delays leaf materilization even further

# PADS+ ( Partial ADS+) : Variation of ADS+ for skewed workloads.



1. Improvises on the initialization phase – making it even more transparent.
2. Tailored for case where users have a few targeted queries or for skewed workloads.
3. Skewed workload – query only needs a part of the data or focuses on subsets

Main Idea

Build parts of index tree and for small subsets of data – depending on query

Initialization Phase:

1. Scans the file
2. Only creates a root node and the iSAX representation
3. Only maintains buffers in FBL.
4. Ready for handling query

Querying phase:

1. Similar to ADS+
2. Continuously and incrementally refined
3. Query directs the nodes that are split till they are small enough to contain data.
4. Query converted to iSAX representation – and finds the FBL for this representation
5. It pulls this FBL from the disk – and adaptively splits it – till its size <= query leaf size.
6. To optimize reading from disk for each query
   1. Initialize leaf size is set to infinite – thus split can be performed on large leaf BUFFERS
   2. In practice – 16 PAA segments ~ 2 ^16 FBL buffers
   3. For a 1B data series – each 2^16 buffers can have = 1B/2^15 = 15258.78 = 15K – iSAX
   4. If we use 1 byte for each character, 16 segments (0 to 15) --- 16 bytes for each data series
   5. Results in a FBL = 235 KB – trivial for splits.

Update and Deletes :

* Inserts are the most imp operation from data exploration perspective
* New data created, but past data not discarded
* Insert – achieved by appending the new data series to the end of the raw file, its positions and SAX representation is pushed through the tree
* FULL – materialized leaf
  + Flip a bit in the leaf- so the query knows there is new data and can adaptively index that
* Partial Leaf
  + No change
* Delete ( series)
  + Just mark it as deleted ( by setting a bit)
  + Same for FULL / Partial leaf

Evaluation:

