CS 265
Stratos Idreos
BIG DATA SYSTEMS
NoSQL | Neural Networks | SQL | Graph | Data Science
<table>
<thead>
<tr>
<th>Category</th>
<th>Primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node organization</td>
<td><strong>1 Key retention.</strong> <em>No:</em> node contains no real key data, e.g., intermediate nodes of b+trees and linked lists. <em>Yes:</em> contains complete key data, e.g., nodes of b-trees, and arrays. <strong>Function:</strong> contains only a subset of the key, i.e., as in tries.</td>
</tr>
<tr>
<td></td>
<td><strong>2 Value retention.</strong> <em>No:</em> node contains no real value data, e.g., intermediate nodes of b+trees, and linked lists. <em>Yes:</em> contains complete value data, e.g., nodes of b-trees, and arrays. <strong>Function:</strong> contains only a subset of the values.</td>
</tr>
<tr>
<td></td>
<td><strong>3 Key order.</strong> Determines the order of keys in a node or the order of fences if real keys are not retained.</td>
</tr>
<tr>
<td></td>
<td><strong>4 Key-value layout.</strong> Determines the physical layout of key-value pairs. <strong>Rules:</strong> requires key retention != no or value retention != no.</td>
</tr>
<tr>
<td></td>
<td><strong>5 Intra-node access.</strong> Determines how sub-blocks (one or more keys of this node) can be addressed and retrieved within a node, e.g., with direct links, a link only to the first or last block, etc.</td>
</tr>
<tr>
<td></td>
<td><strong>6 Utilization.</strong> Utilization constraints in regards to capacity. For example, &gt;= 50% denotes that utilization has to be greater than or equal to half the capacity.</td>
</tr>
<tr>
<td></td>
<td><strong>7 Bloom filters.</strong> A node's sub-block can be filtered using bloom filters. Bloom filters get as parameters the number of hash functions and number of bits.</td>
</tr>
<tr>
<td></td>
<td><strong>8 Zone map filters.</strong> A node's sub-block can be filtered using zone maps, e.g., they can filter based on mix/max keys in each sub-block.</td>
</tr>
<tr>
<td></td>
<td><strong>9 Filters memory layout.</strong> Filters are stored contiguously in a single area of the node or scattered across the sub-blocks. <strong>Rules:</strong> requires bloom filter != off or zone map filters != off.</td>
</tr>
<tr>
<td></td>
<td><strong>10 Fanout/Radix.</strong> Fanout of current node in terms of sub-blocks. This can either be unlimited (i.e., no restriction on the number of sub-blocks), fixed to a number, decided by a function or the node is terminal and thus has a fixed capacity.</td>
</tr>
<tr>
<td></td>
<td><strong>11 Key partitioning.</strong> Sets if there is a pre-defined key partitioning imposed. e.g., the sub-block where a key is located can be dictated by a radix or range partitioning function. Alternatively, keys can be temporally partitioned. If partitioning is set to none, then keys can be forward or backwards appended.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Domain</th>
<th>Hash Table</th>
<th>B+Tree/CSB+Tree/FAST</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>LPL</td>
<td>B+</td>
</tr>
<tr>
<td></td>
<td>H</td>
<td>LL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Primitives</th>
<th><strong>1 Key retention.</strong> <em>No:</em> node contains no real key data, e.g., intermediate nodes of b+trees and linked lists. <em>Yes:</em> contains complete key data, e.g., nodes of b-trees, and arrays. <strong>Function:</strong> contains only a subset of the key, i.e., as in tries.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>2 Value retention.</strong> <em>No:</em> node contains no real value data, e.g., intermediate nodes of b+trees, and linked lists. <em>Yes:</em> contains complete value data, e.g., nodes of b-trees, and arrays. <strong>Function:</strong> contains only a subset of the values.</td>
</tr>
<tr>
<td></td>
<td><strong>3 Key order.</strong> Determines the order of keys in a node or the order of fences if real keys are not retained.</td>
</tr>
<tr>
<td></td>
<td><strong>4 Key-value layout.</strong> Determines the physical layout of key-value pairs. <strong>Rules:</strong> requires key retention != no or value retention != no.</td>
</tr>
<tr>
<td></td>
<td><strong>5 Intra-node access.</strong> Determines how sub-blocks (one or more keys of this node) can be addressed and retrieved within a node, e.g., with direct links, a link only to the first or last block, etc.</td>
</tr>
<tr>
<td></td>
<td><strong>6 Utilization.</strong> Utilization constraints in regards to capacity. For example, &gt;= 50% denotes that utilization has to be greater than or equal to half the capacity.</td>
</tr>
<tr>
<td></td>
<td><strong>7 Bloom filters.</strong> A node's sub-block can be filtered using bloom filters. Bloom filters get as parameters the number of hash functions and number of bits.</td>
</tr>
<tr>
<td></td>
<td><strong>8 Zone map filters.</strong> A node's sub-block can be filtered using zone maps, e.g., they can filter based on mix/max keys in each sub-block.</td>
</tr>
<tr>
<td></td>
<td><strong>9 Filters memory layout.</strong> Filters are stored contiguously in a single area of the node or scattered across the sub-blocks. <strong>Rules:</strong> requires bloom filter != off or zone map filters != off.</td>
</tr>
<tr>
<td></td>
<td><strong>10 Fanout/Radix.</strong> Fanout of current node in terms of sub-blocks. This can either be unlimited (i.e., no restriction on the number of sub-blocks), fixed to a number, decided by a function or the node is terminal and thus has a fixed capacity.</td>
</tr>
<tr>
<td></td>
<td><strong>11 Key partitioning.</strong> Sets if there is a pre-defined key partitioning imposed. e.g., the sub-block where a key is located can be dictated by a radix or range partitioning function. Alternatively, keys can be temporally partitioned. If partitioning is set to none, then keys can be forward or backwards appended.</td>
</tr>
<tr>
<td>Categories</td>
<td>Children layout</td>
</tr>
<tr>
<td>------------</td>
<td>----------------</td>
</tr>
<tr>
<td><strong>Partitioning</strong></td>
<td></td>
</tr>
<tr>
<td>Rules:</td>
<td>Requires fanout/radix (!=) terminal.</td>
</tr>
<tr>
<td><strong>Immediate node links.</strong></td>
<td>Whether and how sub-blocks are connected.</td>
</tr>
<tr>
<td>Rules:</td>
<td>Requires key partitioning (!=) none.</td>
</tr>
<tr>
<td><strong>Skip node links.</strong></td>
<td>Each sub-block can be connected to another sub-block (not only the next or previous) with skip-links. They can be perfect, randomized or custom.</td>
</tr>
<tr>
<td>Rules:</td>
<td>Requires fanout/radix (!=) terminal.</td>
</tr>
<tr>
<td><strong>Area-links.</strong></td>
<td>Each sub-tree can be connected with another sub-tree at the leaf level through area links. Examples include the linked leaves of a B+Tree.</td>
</tr>
<tr>
<td>Rules:</td>
<td>Requires fanout/radix (!=) terminal.</td>
</tr>
<tr>
<td><strong>Sub-block physical location.</strong></td>
<td>This represents the physical location of the sub-blocks. Pointed: in heap. Inline: block physically contained in parent. Double-pointed: in heap but with pointers back to the parent.</td>
</tr>
<tr>
<td>Rules:</td>
<td>Requires fanout/radix (!=) terminal.</td>
</tr>
<tr>
<td><strong>Sub-block physical layout.</strong></td>
<td>This represents the physical layout of sub-blocks. Scatter: random placement in memory. BFS: laid out in a breadth-first layout. BFS layer list: hierarchical level nesting of BFS layouts.</td>
</tr>
<tr>
<td>Rules:</td>
<td>Requires fanout/radix (!=) terminal.</td>
</tr>
<tr>
<td><strong>Sub-block homogeneous.</strong></td>
<td>Set to true if all sub-blocks of the same type.</td>
</tr>
<tr>
<td>Rules:</td>
<td>Requires fanout/radix (!=) terminal.</td>
</tr>
<tr>
<td><strong>Sub-block consolidation.</strong></td>
<td>Single children are merged with their parents.</td>
</tr>
<tr>
<td>Rules:</td>
<td>Requires fanout/radix (!=) terminal.</td>
</tr>
<tr>
<td><strong>Sub-block instantiation.</strong></td>
<td>If it is set to eager, all sub-blocks are initialized, otherwise they are initialized only when data are available (lazy).</td>
</tr>
<tr>
<td>Rules:</td>
<td>Requires fanout/radix (!=) terminal.</td>
</tr>
<tr>
<td><strong>Sub-block links layout.</strong></td>
<td>If there exist links, are they all stored in a single array (consolidate) or spread at a per partition level (scatter).</td>
</tr>
<tr>
<td>Rules:</td>
<td>Requires immediate node links (!=) none or skip links (!=) none.</td>
</tr>
<tr>
<td><strong>Recursion allowed.</strong></td>
<td>If set to yes, sub-blocks will be subsequently inserted into a node of the same type until a maximum depth (expressed as a function) is reached. Then the terminal node type of this data structure will be used.</td>
</tr>
<tr>
<td>Rules:</td>
<td>Requires fanout/radix (!=) terminal.</td>
</tr>
</tbody>
</table>
Are keys retained? (yes, no, function)
Are values retained?
Utilization? (e.g., >50%)
Are keys retained? (yes, no, function)
Are values retained?
Utilization? (e.g., >50%)

Fanout (fixed/functional | unlimited | terminal |)
Key partitioning (none(fw-append | bw-append) | sorted | range() | radix() | function (func) | temporal(...))
Are keys retained? (yes, no, function)
Are values retained?
Utilization? (e.g., >50%)

Fanout (fixed-functional | unlimited | terminal |)
Key partitioning (none(fw-append | bw-append) | sorted | range() | radix() | function (func) | temporal(…))

Intra node access (direct | head_link | tail_link | link_function(func))
Are keys retained? (yes, no, function)
Are values retained?
Utilization? (e.g., >50%)

Fanout (fixed/functional | unlimited | terminal |)
Key partitioning (none(fw-append | bw-append) | sorted | range() | radix() | function (func) | temporal(…))

Intra node access (direct | head_link | tail_link | link_function(func))
Sub block links (next | previous | both | none)
Sub block skip links (perfect | randomized(prob: double) | function(func) | none)
Are keys retained? (yes, no, function)
Are values retained?
Utilization? (e.g., >50%)

Fanout (fixed/functional | unlimited | terminal |)
Key partitioning (none(fw-append | bw-append) | sorted | range() | radix() | function (func) | temporal(…))

Intra node access (direct | head_link | tail_link | link_function(func))

Sub block links (next | previous | both | none)
Sub block skip links (perfect | randomized(prob: double) | function(func) | none)

Zone Maps (min | max | both | exact | off)
Bloom filters (off | on(num_hashes: int, num_bits: int))
Filters layout (consolidate | scatter)
Links layout (consolidate | scatter)
Are keys retained? (yes, no, function)
Are values retained?
Utilization? (e.g., >50%)

Fanout (fixed/functional | unlimited | terminal |)
Key partitioning (none(fw-append | bw-append) | sorted | range() | radix() | function (func) | temporal(…))

Intra node access (direct | head_link | tail_link | link_function(func))
Sub block links (next | previous | both | none)
Sub block skip links (perfect | randomized(prob: double) | function(func) | none)

Zone Maps (min | max | both | exact | off)
Bloom filters (off | on(num_hashes: int, num_bits: int))

Filters layout (consolidate | scatter)
Links layout (consolidate | scatter)
Physical location (inline | pointed | double-pointed)
Physical layout (BFS | scatter)
UNORDERED ARRAY

no order (fw-append)

[256]

UNORDERED LIST OF ARRAYS

[256]
UNORDERED ARRAY

B+Tree

Recursion: log(n)

Sub-blocks homogeneous: true
Sub-block physical layout: scatter
Sub-block consolidation: false
Sub-block instantiation: lazy

Consolidated Filters (Fences only)
Balanced
Fixed fanout
Direct Addressing
Sub-block location: pointed

sorted

[k] [v]
POSSIBLE STRUCTURES

- Trie
- Array
- Skip-List
- Linked-List
- Hash-Table
- Sorted Array

POSSIBLE NODE DESIGNS
POSSIBLE NODE DESIGNS

POSSIBLE STRUCTURES

- Trie
- Array
- Skip-List
- Hash-Table
- Linked-List
- Sorted Array
- Unknown 1
- Unknown 2
- Unknown N
The Periodic Table of the Elements explains and predicts missing elements. Dimitri Mendelev structures elements based on atomic number, electron configuration, and recurring chemical properties.
“The taxonomy is used to shed light both on the nature of the design space and on the performance tradeoffs implied by many of the choices that exist in the design space.”
"The taxonomy is used to shed light both on the nature of the design space and on the performance tradeoffs implied by many of the choices that exist in the design space."

**TAXONOMY OF COMPLEX ALGORITHMS**

Transactionally cache consistency maintenance

---

Mike Franklin
# Periodic Table of Data Structures

<table>
<thead>
<tr>
<th>Classes of Designs</th>
<th>Classes of Primitives</th>
<th>B-trees &amp; Variants</th>
<th>Tries &amp; Variants</th>
<th>LSM-Trees &amp; Variants</th>
<th>Differential Files</th>
<th>Membership Tests</th>
<th>Zone Maps &amp; Variants</th>
<th>Bitmaps &amp; Variants</th>
<th>Hashing</th>
<th>Base Data &amp; Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning</td>
<td></td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td>DONE</td>
<td></td>
</tr>
<tr>
<td>Logarithmic Design</td>
<td></td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td>DONE</td>
<td></td>
</tr>
<tr>
<td>Fractional Cascading</td>
<td></td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Log-Structured</td>
<td></td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffering</td>
<td></td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Differential Updates</td>
<td></td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sparse Indexing</td>
<td></td>
<td>DONE</td>
<td>DONE</td>
<td></td>
<td>DONE</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adaptivity</td>
<td></td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

@IEEE.EngBul18

DASlab
@HarvardSEAS
### Periodic Table of Data Structures

<table>
<thead>
<tr>
<th>Classes of Designs</th>
<th>B-trees &amp; Variants</th>
<th>Tries &amp; Variants</th>
<th>LSM-Trees &amp; Variants</th>
<th>Differential Files</th>
<th>Membership Tests</th>
<th>Zone maps &amp; Variants</th>
<th>Bitmaps &amp; Variants</th>
<th>Hashing</th>
<th>Base Data &amp; Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning</td>
<td>□ DONE</td>
<td>□ DONE</td>
<td>□ DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>□ DONE</td>
<td>□ DONE</td>
</tr>
<tr>
<td>Logarithmic Design</td>
<td>□ DONE</td>
<td>□ DONE</td>
<td>□ DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>□ DONE</td>
<td>□ DONE</td>
</tr>
<tr>
<td>Fractional Cascading</td>
<td>□ DONE</td>
<td>□ DONE</td>
<td>□ DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>□ DONE</td>
<td>□ DONE</td>
</tr>
<tr>
<td>Log-Structured</td>
<td>□ DONE</td>
<td>□ DONE</td>
<td>□ DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>□ DONE</td>
<td>□ DONE</td>
</tr>
<tr>
<td>Buffering</td>
<td>□ DONE</td>
<td>□ DONE</td>
<td>□ DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>□ DONE</td>
<td>□ DONE</td>
</tr>
<tr>
<td>Differential Updates</td>
<td>□ DONE</td>
<td>□ DONE</td>
<td>□ DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>□ DONE</td>
<td>□ DONE</td>
</tr>
<tr>
<td>Sparse Indexing</td>
<td>□ DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>□ DONE</td>
<td>□ DONE</td>
</tr>
<tr>
<td>Adaptivity</td>
<td>□ DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>□ DONE</td>
<td>□ DONE</td>
</tr>
</tbody>
</table>
### Periodic Table of Data Structures

<table>
<thead>
<tr>
<th>Classes of Designs</th>
<th>B-trees &amp; Variants</th>
<th>Tries &amp; Variants</th>
<th>LSM-Trees &amp; Variants</th>
<th>Differential Files</th>
<th>Membership Tests</th>
<th>Zone Maps &amp; Variants</th>
<th>Bitmaps &amp; Variants</th>
<th>Hashing</th>
<th>Base Data &amp; Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Partitioning</strong></td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
</tr>
<tr>
<td><strong>Logarithmic</strong></td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
</tr>
<tr>
<td><strong>Differential Updates</strong></td>
<td>DONE</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Sparse Indexing</strong></td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Adaptivity</strong></td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PAPER MACHINE**
### periodic table of data structures

<table>
<thead>
<tr>
<th>classes of designs</th>
<th>B-trees &amp; Variants</th>
<th>Tries &amp; Variants</th>
<th>LSM-Trees &amp; Variants</th>
<th>Differential Files</th>
<th>Membership Tests</th>
<th>Zone maps &amp; Variants</th>
<th>Bitmaps &amp; Variants</th>
<th>Hashing</th>
<th>Base Data &amp; Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DONE</td>
<td>DONE</td>
</tr>
<tr>
<td>Logarithmic</td>
<td>DONE</td>
<td>DONE</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DONE</td>
<td>DONE</td>
</tr>
<tr>
<td>Differential</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Updates</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sparse Indexing</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adapativity</td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PAPER MACHINE**

- Updatable bitmap indexes
- @SIGMOD16
STARS IN THE SKY

POSSIBLE DATA STRUCTURES

$10^{24}$

$10^{32}$, 2-node

$10^{48}$, 3-node
DESIGN SPACE

**fundamental** building blocks

**properties** when combined
Memory Read
Update
COST?

fundamental building blocks
properties when combined
HOW TO JUDGE A DESIGN?

1. COMPLEXITY ANALYSIS
2. IMPLEMENTATION & TESTING
HOW TO JUDGE A DESIGN?

1. COMPLEXITY ANALYSIS
2. IMPLEMENTATION & TESTING
3. GENERALIZED MODELS
ACCESS PATH SELECTION in ANALYTICAL SYSTEMS
scan vs secondary index selection
ACCESS PATH SELECTION in ANALYTICAL SYSTEMS
scan vs secondary index selection

@SIGMOD 2017

algorithms/operators

data structure 1

data structure 2

data structure 3
ACCESS PATH SELECTION
scan vs secondary index selection
ACCESS PATH SELECTION
scan vs secondary index selection

Scan is best

Index is best

P. Sellinger et. al, 1979
ACCESS PATH SELECTION
scan vs secondary index selection

P. Sellinger et. all, 1979

Scan is best

Index is best

DO WE STILL NEED INDEXING? (AND IF YES HOW DO WE CHOOSE)
ACCESS PATH SELECTION in ANALYTICAL SYSTEMS
scan vs secondary index selection

P. Sellinger et. all, 1979

Scan is best

Index is best

selectivity
ACCESS PATH SELECTION in ANALYTICAL SYSTEMS
scan vs secondary index selection

P. Sellinger et. al, 1979

Scan is best

Index is best

selectivity

selectivity

# of concurrent queries

multi-core, SIMD, compression, columnar/hybrid, scan sharing, …
ACCESS PATH SELECTION in ANALYTICAL SYSTEMS
scan vs secondary index selection

Pat Selinger
P. Sellinger et. all, 1979

Index is best
Scan is best

selectivity

Dawn of time 2000 2010 2017 Future

Hardware Improvements
Column Stores
Main Memory
latency
bandwidth

# of concurrent queries
10%
1%
0%

selectivity threshold

scan vs secondary index selection @SIGMOD 2017

Tree Traversal + Leaf Traversal = Result Writing + Sorting = Base Scan + Predicate Eval. + Result Writing

\[
APS(q, S_{tot}) = \frac{q \cdot \left(1 + \left\lceil \log_b(N) \right\rceil \right) \cdot \left(\frac{BW_S \cdot C_M}{2} + \frac{b \cdot BW_S \cdot C_A}{2} + \frac{b \cdot BW_S \cdot f_p \cdot p}{2}\right)}{\max\left(ts, 2 \cdot f_p \cdot p \cdot q \cdot BW_S\right) + S_{tot} \cdot rw \cdot \frac{BW_S}{BW_R}}
\]

\[
S_{tot} \left(\frac{BW_S \cdot C_M}{b} + (aw + ow) \cdot \frac{BW_S}{BW_I} + rw \cdot \frac{BW_S}{BW_R}\right)
\]

\[
+ \frac{\max\left(ts, 2 \cdot f_p \cdot p \cdot q \cdot BW_S\right) + S_{tot} \cdot rw \cdot \frac{BW_S}{BW_R}}{\max\left(ts, 2 \cdot f_p \cdot p \cdot q \cdot BW_S\right) + S_{tot} \cdot rw \cdot \frac{BW_S}{BW_R}}
\]

\[
= \frac{S_{tot} \cdot \log_2 (S_{tot} \cdot N) \cdot BW_S \cdot C_A}{\max\left(ts, 2 \cdot f_p \cdot p \cdot q \cdot BW_S\right) + S_{tot} \cdot rw \cdot \frac{BW_S}{BW_R}}
\]

### Workload

<table>
<thead>
<tr>
<th>Workload</th>
<th>( q )</th>
<th>number of queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_i )</td>
<td>selectivity of query ( i )</td>
<td></td>
</tr>
<tr>
<td>( S_{tot} )</td>
<td>total selectivity of the workload</td>
<td></td>
</tr>
</tbody>
</table>

### Dataset

<table>
<thead>
<tr>
<th>Dataset</th>
<th>( N )</th>
<th>data size (tuples per column)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ts )</td>
<td>tuple size (bytes per tuple)</td>
<td></td>
</tr>
</tbody>
</table>

### Hardware

<table>
<thead>
<tr>
<th>Hardware</th>
<th>( C_A )</th>
<th>L1 cache access (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C_M )</td>
<td>LLC miss: memory access (sec)</td>
<td></td>
</tr>
<tr>
<td>( BW_S )</td>
<td>scanning bandwidth (GB/s)</td>
<td></td>
</tr>
<tr>
<td>( BW_{R} )</td>
<td>result writing bandwidth (GB/s)</td>
<td></td>
</tr>
<tr>
<td>( BW_I )</td>
<td>leaf traversal bandwidth (GB/s)</td>
<td></td>
</tr>
<tr>
<td>( p )</td>
<td>The inverse of CPU frequency</td>
<td></td>
</tr>
<tr>
<td>( f_p )</td>
<td>Factor accounting for pipelining</td>
<td></td>
</tr>
</tbody>
</table>

### Scan & Index

| Scan | \( rw \) | result width (bytes per output tuple) |
| Index | \( b \) | tree fanout |
| \( aw \) | attribute width (bytes of the indexed column) |
| \( ow \) | offset width (bytes of the index column offset) |
scan vs secondary index selection

HARD & SLOW

@SIGMOD 2017

Tree Traversal + Leaf Traversal = Result Writing + Sorting

Base Scan + Predicate Eval. + Result Writing

Workload: $q$, $s_i$, $S_{tot}$
- $q$: number of queries
- $s_i$: selectivity of query $i$
- $S_{tot}$: total selectivity of the workload

Dataset: $N$, $t_s$
- $N$: data size (tuples per column)
- $t_s$: tuple size (bytes per tuple)

Hardware: $C_A$, $C_M$, $BW_S$, $BW_R$
- $C_A$: L1 cache access (sec)
- $C_M$: LLC miss: memory access (sec)
- $BW_S$: scanning bandwidth (GB/s)
- $BW_R$: result writing bandwidth (GB/s)
- $BW_f$: leaf traversal bandwidth (GB/s)
- $p$: The inverse of CPU frequency
- $B$: Factor accounting for pipelining

Scan & Index: $rw$, $b$, $aw$, $ow$
- $rw$: result width (bytes per output tuple)
- $b$: tree fanout
- $aw$: attribute width (bytes of the indexed column)
- $ow$: offset width (bytes of the index column offset)

Equation 16:

$\text{APS}(q, S_{tot}) = \frac{q \cdot \left[ 1 + \frac{\log_b(N)}{N} \right] \cdot \left( BW_S \cdot C_M + \frac{b \cdot BW_f \cdot C_A}{2} + \frac{b \cdot BW_S \cdot f_p \cdot p}{2} \right)}{\max \left( ts, 2 \cdot f_p \cdot p \cdot q \cdot BW_S \right) + S_{tot} \cdot rw \cdot \frac{BW_S}{BW_R}}$

$S_{tot} \left( \frac{BW_S \cdot C_M}{b} + (aw + ow) \cdot \frac{BW_S}{BW_f} + rw \cdot \frac{BW_S}{BW_R} \right)$

$+ \frac{\max \left( ts, 2 \cdot f_p \cdot p \cdot q \cdot BW_S \right) + S_{tot} \cdot rw \cdot \frac{BW_S}{BW_R}}{\max \left( ts, 2 \cdot f_p \cdot p \cdot q \cdot BW_S \right) + S_{tot} \cdot \log_2 \left( S_{tot} \cdot N \right) \cdot BW_S \cdot C_A}$

$+ \frac{S_{tot} \cdot \log_2 \left( S_{tot} \cdot N \right) \cdot BW_S \cdot C_A}{\max \left( ts, 2 \cdot f_p \cdot p \cdot q \cdot BW_S \right) + S_{tot} \cdot rw \cdot \frac{BW_S}{BW_R}}$
POSSIBLE DATA LAYOUTS

write

read

memory
POSSIBLE DATA LAYOUTS

read

write

memory

operation
ALGORITHM & COST SYNTHESIS
synthesize access pattern

If ..., then ..., else

RULES

POSSIBLE DATA LAYOUTS

write
memory

operation
sorted keys
columnar layout
sorted keys
columnar layout

RULES

sorted search
DEPENDS ON HARDWARE ENGINEERING

sorted keys columnar layout

RULES

sorted search

binary search

interpolation search

using new SIMD instruction X

...
COMPONENTS OF KEY-VALUE ALGORITHMS

RULES

sorted keys
columnar layout

sorted search

binary search1
binary search2
interpolation search1
interpolation search2
using new SIMD instruction X

batched write
BF probe
scan

...
COMPONENTS OF KEY-VALUE ALGORITHMS

RULES

sorted keys
columnar layout

sorted search

batched write

BF probe

scan

LEARNING

binary search1
binary search2
interpolation search1
interpolation search2
using new SIMD instruction X

code, model

code, model

code, model
SYNTHESIS FROM LEARNED MODELS
coding, modeling, generalized models, and a touch of ML

1. MINIMAL CODE

e.g., binary search

```cpp
if (data[middle] < search_val) {
  low = middle + 1;
} else {
  high = middle;
}
middle = (low + high)/2;
```

1 11 17 37 51 66 80 94
SYNTHESIS FROM LEARNED MODELS
coding, modeling, generalized models, and a touch of ML

1. MINIMAL CODE

e.g., binary search

```c++
if (data[middle] < search_val) {
    low = middle + 1;
} else {
    high = middle;
}
middle = (low + high)/2;
```

2. BENCHMARK

![Graph showing time vs. data size]
SYNTHESIS FROM LEARNED MODELS
coding, modeling, generalized models, and a touch of ML

1. MINIMAL CODE
   e.g., binary search
   ```
   C++
   if(data[middle] < search_val) {
      low = middle + 1;
   } else {
      high = middle;
   }
   middle = (low + high)/2;
   ```

2. BENCHMARK
   ![Benchmark plot]
   - Data Size (KB)
   - Time (s)
   - Log-Linear Model
   - \( f(x) = ax + b \log x + c \)

3. FIT MODEL
   ![Fit model graph]
SYNTHESIS FROM LEARNED MODELS
coding, modeling, generalized models, and a touch of ML

1. MINIMAL CODE

```cpp
if (data[middle] < search_val) {
    low = middle + 1;
} else {
    high = middle;
}
middle = (low + high)/2;
```

2. BENCHMARK

FOLDING ALGORITHMIC, ENGINEERING, AND H/W, PROPERTIES INTO THE COEFFICIENTS

3. FIT MODEL

\[ f(x) = ax + b \log x + c \]


(we will see the NoSQL papers again in detail later on)