CLOUD AND BIG DATA

PRESENTED BY MICHAL AMSTERDAM , POLINA MANEVICH

11.12.17

Spanner: Becoming a SQL System

David F. Bacon, Nathan Bales, Nicolas Bruno, Brian F. Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, Alexander Lloyd, Sergey Melnik, Rajesh Rao, David Shue, Christopher Taylor, Marcel van der Holst, and Dale Woodford

Data Page Layouts for Relational Databases on Deep Memory Hierarchies

Anatstasia Ailmaki, David J. DeWitt, Mark D. Hill

THE LOG-STRUCTURED MERGE-TREE (LSM-TREE)

Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil

AGENDA

- Spanner – Becoming a SQL System
  - Query Distribution
  - Query Range Extraction
  - Query Restarts
  - *Ressi* Data Layout – Combining PAX and LSM-tree into a new low level storage format
- Improving cache performance for DBMS
  - Common Data Page Layouts
  - What is the Parallel Attributes Across (PAX) model
  - The improvements the PAX model achieves on other layouts
- IMPROVING THE MAINTENANCE OF HISTORY TABLE INDEX EFFICIENCY AND COST
  - What is the Log-Structured Merge-Tree (LSM Tree)
  - The improvements the LSM Tree Achieves
  - Comparing the LSM Tree to other possible indexing options

SPANNER: BECOMING A SQL SYSTEM

2017

David F. Bacon, Nathan Bales, Nicolas Bruno,
Brian F. Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser,
Andrey Gubarev, Milind Joshi, Eugene Kogan, Alexander Lloyd,
Sergey Melnik, Rajesh Rao, David Shue, Christopher Taylor, Marcel
van der Holst, and Dale Woodford

SPANNER – BECOMING A SQL SYSTEM

BACON ET AL., *SIGMOD'17*

- Google's Spanner started out as a key-value store offering multi-row transactions, external consistency, and transparent failover across datacenters.
- Problem: Developers of many OLTP applications found it difficult to build these applications without a strong schema system, cross-row transactions, consistent replication and a powerful query language.
- Solution: Turn Spanner into a full featured SQL system, with query execution tightly integrated with the other architectural features of Spanner.
- Replicas of the data are served from datacenters around the world to provide low latency to scattered clients. Despite this wide replication, the system provides transactional consistency and strongly consistent replicas, as well as high availability.
- The Spanner query processor implements a dialect of SQL, and is built to serve a mix of transactional and analytical workloads, and to support both low-latency and long-running queries.

SPANNER INTRODUCTION

To sum up the change in two words, would probably be:

"more SQL!"

- In the past:

OLTP + ~~SQL~~ =

- Today:

Spanner SQL

while maintaining scalability

Why did Spanner evolve?

"A prime motivation for this evolution towards a more "database-like" system was driven by the experiences of Google developers trying to build on previous "key-value" storage systems. [E.g., Bigtable]."

Goals by the Spanner team:

"A scalable data management system must address manageability early on to remain viable. ACID transactions spanning arbitrary rows/keys is the next hardest challenge for scalable data management systems. Transactions are essential for mission-critical applications, in which any inconsistency, even temporary, is unacceptable. "

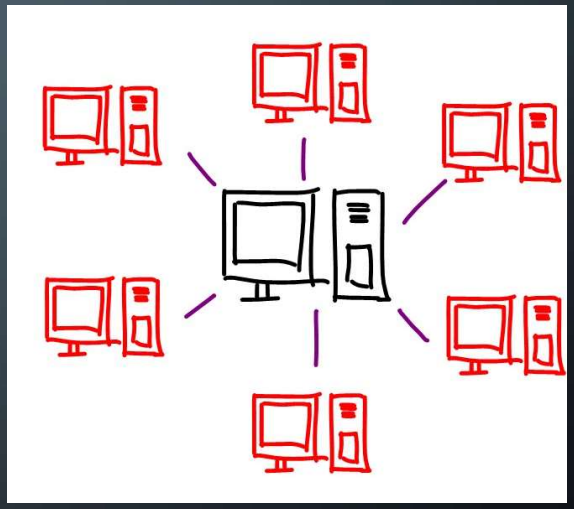
SPANNER – BACKGROUND

- Spanner is a horizontally row-range sharded, geo-replicated relational database system.
- Shards are distributed across multiple servers and replicated to multiple, geographically separated datacenters.
- Spanner's transactions use a replicated write-ahead redo log, and the Paxos consensus algorithm is used to get replicas to agree on the contents of each log entry.
- Concurrency control uses a combination of pessimistic locking and timestamps.
- Replicas retain several previous data versions to be able to serve stale reads.
- Read and write requests are addressed to a key or key range (that can be looked for through shard boundaries), not to a particular server.

SPANNER EVOLVING

We will focus on three main aspects in which **Spanner** evolved:

1) Supports *distributed* query execution efficiently.
2) Smart servers selection to minimize scanning and locking - using *range extraction*.
3) Expends so much effort to support *restartable* queries.



SPANNER – DISTRIBUTED QUERY EXECUTION

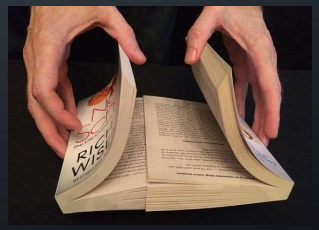
- The Spanner SQL query compiler represents distribution using explicit operators in the query algebra tree.
- ***Distributed Union*** operator is the fundamental building block, which ships a subquery to each relevant shard and concatenates the results.

$$Scan(T) \Rightarrow DistributedUnion[shard \subseteq T](Scan(shard))$$

- Partitionability:

$$F(Scan(T)) = OrderedUnionAll[shard \subseteq T]\left(F(Scan(shard))\right)$$

- Spanner supports ***Table interleaving*** - Co-locates rows from multiple tables sharing a primary key prefix.



Distributed Union operator?

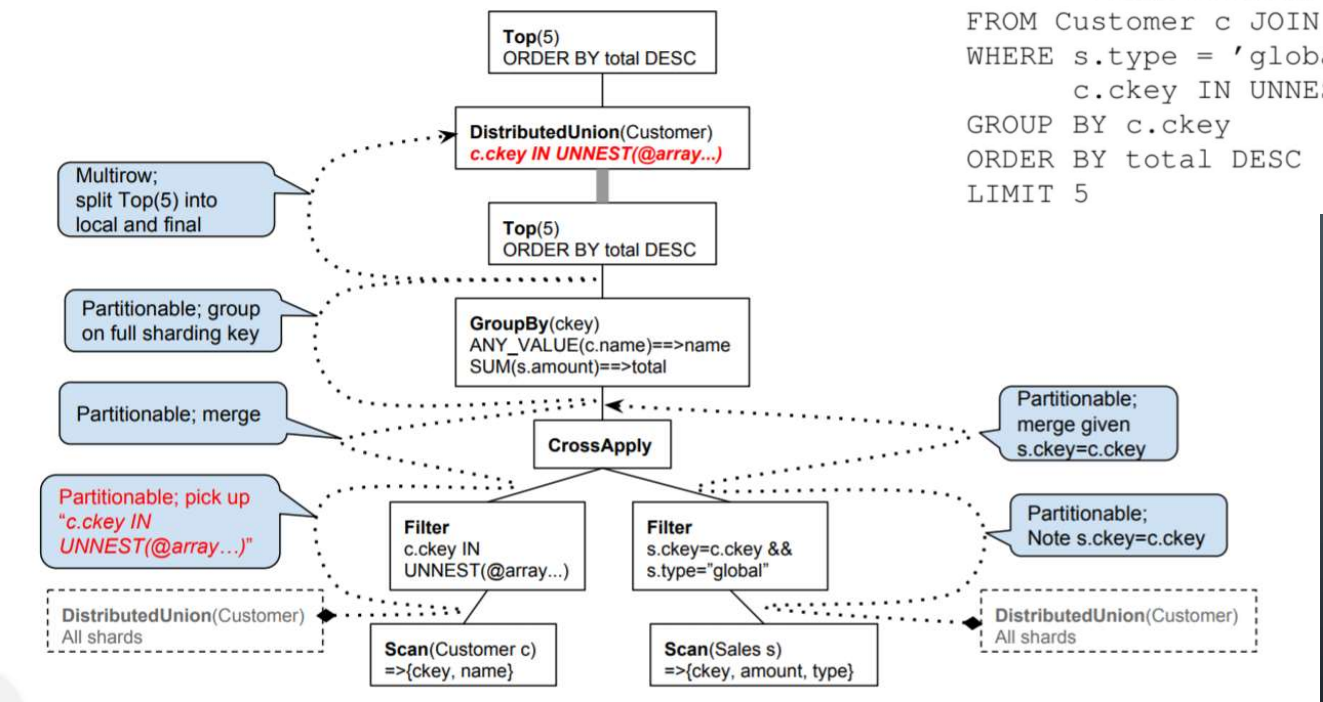
"Distributed Union is a fundamental operation in spanner, especially because the sharding of a table may change during query execution, and after a query restart."

Pulled up the tree

"In order for this operator tree transformations to be equivalent, a property we call partitionability must be satisfied for any relational operation F that we want to push down."

SPANNER – DISTRIBUTED QUERY EXECUTION

```
SELECT ANY_VALUE(c.name) name,
       SUM(s.amount) total
FROM Customer c JOIN Sales s ON c.ckey=s.ckey
WHERE s.type = 'global' AND
      c.ckey IN UNNEST(@customer_key_arr)
GROUP BY c.ckey
ORDER BY total DESC
LIMIT 5
```



- Customer table is sharded on *ckey*.
- *Sales table is interleaved in Customer and thus shares the sharding key.*

SPANNER – DISTRIBUTED QUERY EXECUTION

- When the sharding key range extracted from the filter expression covers the entire table, Distributed Union may change its subquery evaluation strategy.
- Range extraction, extracts a set of ranges guaranteed to fully cover all table rows on which a subquery may yield results ⇒ the heart of shard pruning.
- Subqueries are dispatched to every relevant shard in parallel.
- Joins between independently distributed tables – is a huge subject by itself.

SPANNER – DISTRIBUTED QUERY EXECUTION

- Spanner exposes two kinds of APIs for issuing queries and consuming results:
  - Single-consumer API – used for a single client process consuming results.
  - Parallel-consumer API – used for multiple processes consuming query results in parallel.
- The Single-consumer API :
  - Sends the query initially to a root server that orchestrates further distributed execution.
  - Spanner will attempt to route the query directly to the server that owns all or part of the data referenced by the query.
- The parallel-consumer API guarantees that the concatenation of results from all the partitions yields the same unordered set of rows as for the query submitted through the single-consumer API.

Parallel-Consumer API:
"The first is to divide the work between the desired number of clients. The API receives a SQL query and the desired degree of parallelism, and returns a set of opaque query partition descriptors. In the second stage, the query is executed on the individual partitions, normally using requests initiated in parallel from separate machines.

SPANNER – **RANGE EXTRACTION**

Range extraction is the process of analyzing a query to determine what portions of tables it references.

There are three flavors of range extraction:

1) *Distributed range extraction:* Figures out which table shards are referenced by a query.
2) *Seek range extraction:* Determines what fragments of a relevant shard to read from the underlying storage stack.
3) *Lock range extraction:* Determines what fragments of a table are to be locked (pessimistic transactions) or checked for potential pending modifications (snapshot transactions).

Main techniques

"Our implementation of range extraction in Spanner relies on two main techniques:

At compile time, we normalize and rewrite a filtered scan expression into a tree of correlated self-joins that extract the ranges for successive key columns.

At runtime, we use a special data structure called a filter tree for both computing the ranges via bottom-up interval arithmetic and for efficient evaluation of post-filtering conditions."

SPANNER – **RANGE EXTRACTION**

```
SELECT d.*
FROM Documents d
WHERE d.ProjectId = @param1
  AND STARTS_WITH(d.DocumentPath, '/proposals')
  AND d.Version = @param2
```

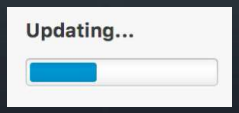
- Let us assume that the values of the parameters are : $@param1 = P1$, $@param2 = 2017 - 01 - 01.$
- Lets say that ten documents in that project have the prefix : "$/proposals/doc1$" etc.

- Range extraction should give us the following information:
  - Distribution Ranges: Simply P1, we route the query to the shard corresponding to P1.
  - Seek Ranges: will be ($P1$,/proposals/doc1 , 2017-01-01) etc.
  - Spanner supports column level locking – for the key columns, lock range extraction yields a single lock range (P1,/proposals). For non-key columns the minimal lock ranges coincide with the seek ranges.
- Range computation is in general a conservative approximation as isolating key columns in predicates can be arbitrarily complex and there are diminishing returns. In the worst case, for distribution range extraction, a query may be sent to a shard that ultimately returns no rows.

"… the $trade - offs$ of seeks vs scans and granularity of locking involve optimization decisions that can be very tricky and are beyond the scope of this paper"

SPANNER – QUERY RESTARTS

- To support query restarts Spanner extended its RPC mechanism with an additional parameter, a *restart token*. Restart tokens accompany all query results, sent in batches, one restart token per batch.
- This opaque restart token blob, when added as a special parameter to the original request prevents the rows already returned to the client to be returned again.
- Furthermore, the non-determinism which makes for high performance query execution opportunities complicates restart as results may be returned in a non-repeatable order.
- Some of the benefits delivered by the query restart mechanism include:
  - Hiding transient failures.
  - Simpler programming model: no retry loops.
  - Streaming pagination through query results.
  - Improved tail latency for online requests.
  - Forward progress for long-running queries.
  - Recurrent rolling upgrades.
  - Simpler Spanner internal error handling



"The ability to gradually upgrade all machines to a new version within a week or so while running a few versions concurrently has been a cornerstone of Spanner's development agility.

SPANNER – OTHER INTERESTING BITS AND PIECES

- Google moved to a standard internal SQL dialect ("Standard SQL") shared by all of their systems (e.g., Spanner, F1, Dremel, BigQuery), to make this work took quite a bit of effort.
- Spanner now has a new low-level storage format called ***Ressi***, designed from the ground-up for handling SQL queries over large-scale distributed databases with a mix of OLTP and OLAP workloads.
  - Stores a database as an ***LSM tree*** whose layers are periodically compacted.
  - Organizes data into blocks in row-major order, but lays out the data within a block in column-major order - essentially, the ***PAX layout***.

DATA PAGE LAYOUTS FOR RELATIONAL DATABASES ON DEEP MEMORY HIERARCHIES

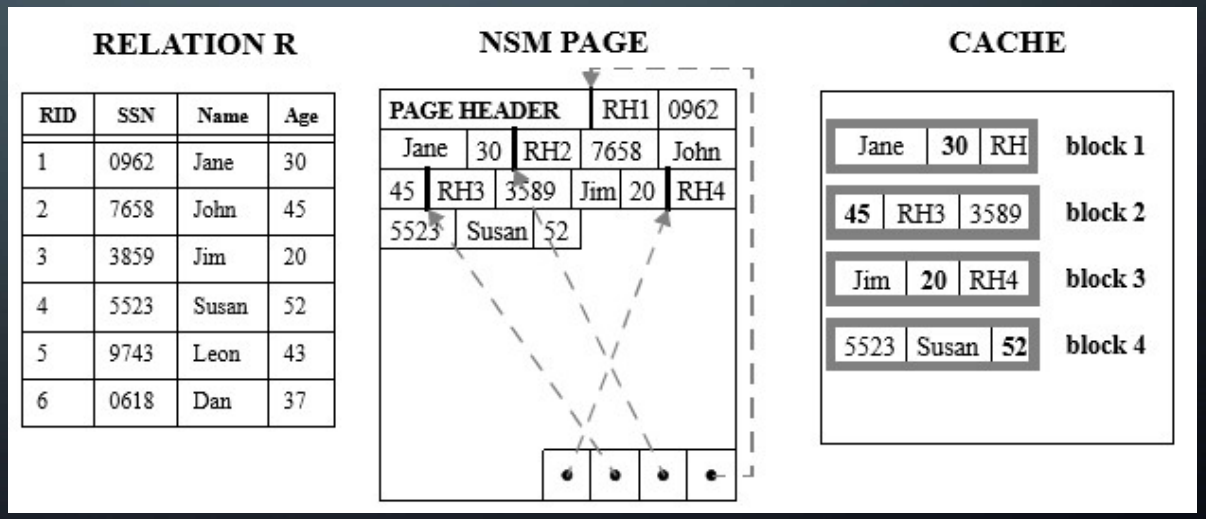
2002

Anatstasia Ailmaki, David J. DeWitt, Mark D. Hill

THE N-ARY STORAGE MODEL (NSM)

- NSM stores records contiguously starting from the beginning of each disk page.
- Each new record is typically inserted into the first available free space starting from the beginning of the page.
- Records may have variable length, therefore it uses offset table at the end of the page to locate the beginning of each record.

```
Select name
From R
Where age <40;
```

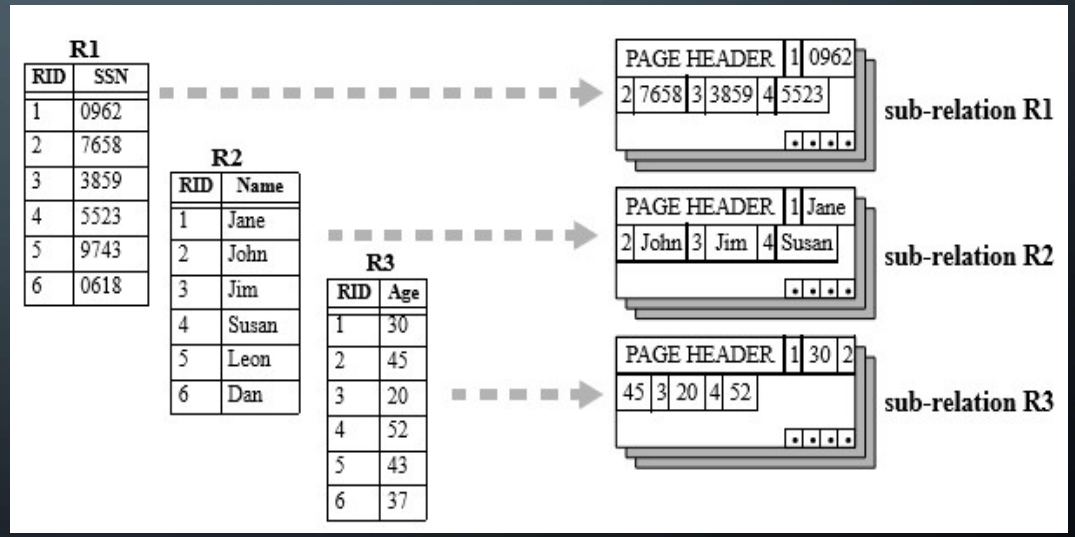


THE DECOMPOSITION STORAGE MODEL (DSM)

- Vertical partitioning is the process of striping a relation into sub-relations, each containing the values of a subset of the initial relation's attributes.
- Sub-relations are stored as regular relations in slotted pages, enabling each attribute to be scanned independently.

RELATION R

RID	SSN	Name	Age
1	0962	Jane	30
2	7658	John	45
3	3859	Jim	20
4	5523	Susan	52
5	9743	Leon	43
6	0618	Dan	37



NSM AND DSM - DISADVANTAGES

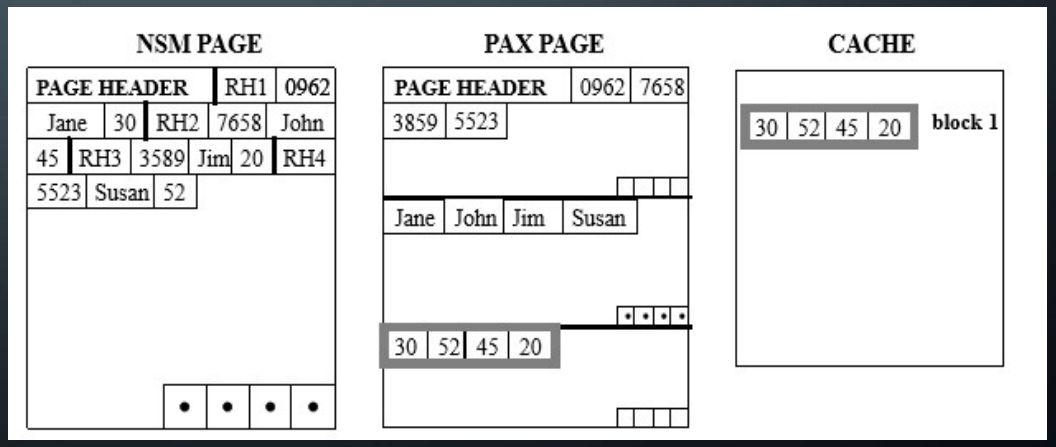
- Most query operations access only a small fraction of each record.
- For example given a relation saved in the NSM model a sequential scan of a certain attribute will miss the cache once for each record! Therefore access main memory once for every value.
- NSM Loading the data:
  - Wastes bandwidth.
  - Pollutes the cache with useless data.
  - Possibly forces replacement of information that may be needed in the future (which means even more delays).
- DSM incurs significant space overhead (record id is saved for each sub-relation separately).

SO WHAT IS PAX ?

- PAX – Partitioning Attributes Across.
- It is a new layout for data records that combines the best of two worlds.
- PAX :
  - Maximizes inter-record spatial locality within each column and page – eliminating unnecessary requests to main memory without space penalty.
  - Incurs a minimal record reconstruction cost.
  - Is orthogonal to other design decisions because it only affects the layout of a data stored on a single page.

RELATION R

RID	SSN	Name	Age
1	0962	Jane	30
2	7658	John	45
3	3859	Jim	20
4	5523	Susan	52
5	9743	Leon	43
6	0618	Dan	37



SO WHAT IS PAX? (CONT.)

- Within each page it groups all values of a particular attribute together on a “mini-page”.
- During a sequential scan – the cache is fully utilized- on each miss a number of the same attribute’s values are loaded into the cache together.
- Reconstruction of a record is of minimal cost (Only needed join between mini-pages since we don’t need to look beyond the page).

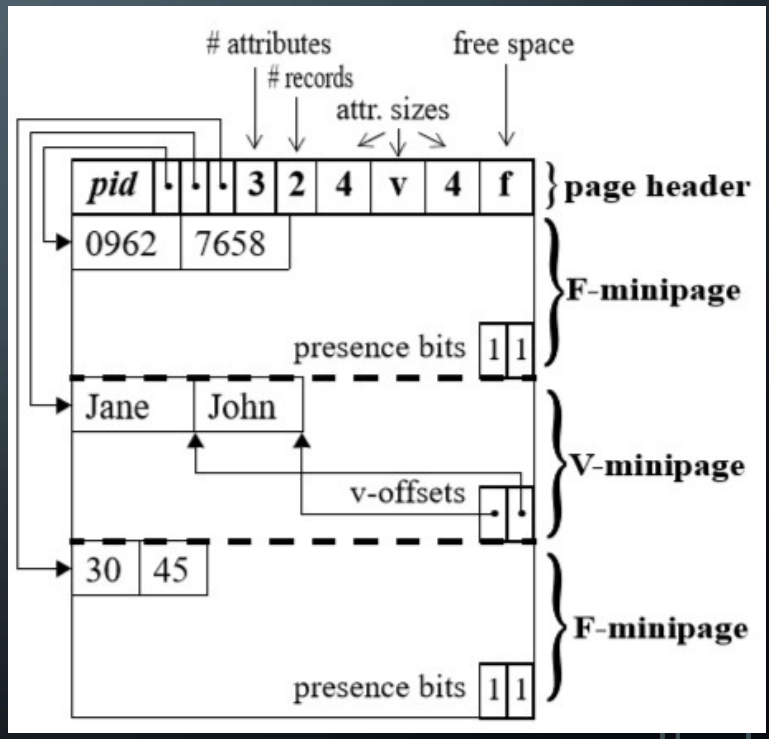
PAX PAGE STRUCTURE

- Each PAX page is partitioned into n mini-pages.
- The page header contains offsets to the beginning of the mini-pages.
- The structure of each mini-page is determined as follows:
  - Fixed-length attribute values: are stored in F-minipage. At the end of each F-minipage there is a bit vector with one entry per record that denotes null values for nullable attributes.
  - Variable-length attribute values: are stored in V-minipage. V-minipage are slotted, with pointers to the end of each value. Null values are denoted by Null Pointers.
- Each newly allocated page contains a page header and a number of minipages equal to the degree of the relation.

A relation with degree n (number of attributes).

RELATION R

RID	SSN	Name	Age
1	0962	Jane	30
2	7658	John	45
3	3859	Jim	20
4	5523	Susan	52
5	9743	Leon	43
6	0618	Dan	37

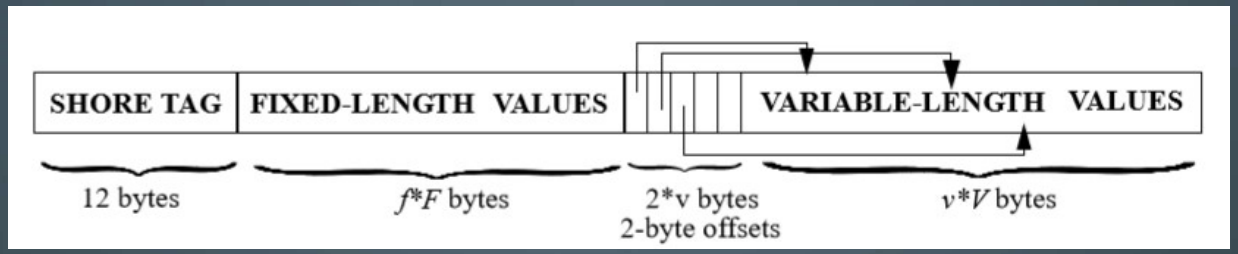


PAX – THE BEST OF BOTH WORLDS

Characteristic	NSM	DSM	PAX
Inter-record spatial locality	—	✓	✓
Low record reconstruction cost	✓	—	✓

- The data placement scheme determines two factors that affect performance:
  - Inter-record spatial locality – minimizes data cache-related delays since cache is fully utilized.
  - Record reconstruction cost minimizes the delays associated with retrieving multiple fields of the same record, since all values of each record are kept in the same page.
- Another advantage – implementation of PAX on existing DBMS requires only change to the page level data manipulation code.

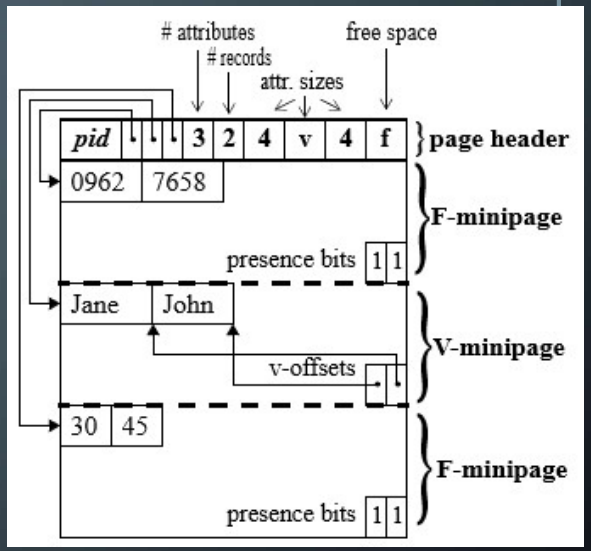
TESTING ENVIRONMENT IMPLEMENTATION



An example NSM Record structure in Shore

- The finest granularity for accessing information shore provides is the record.
- NSM- was implemented by adding attribute-level functionality on top of the existing Shore file manager.
- DSM- was implemented on top of Shore by decomposing the initial relation into n Shore files that are stored in slotted pages using NSM.
- PAX- was implemented as an alternative data page organization in Shore.

PAX DATA MANIPULATION ALGORITHMS IMPLEMENTATION



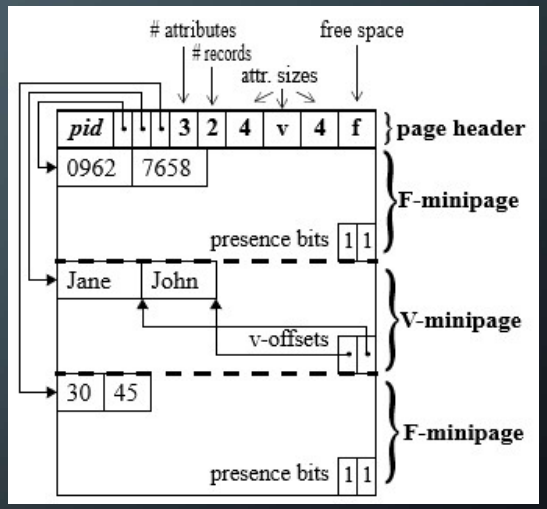
Bulk-Loading and Insertions:

- The algorithm for bulk-load records from a data file will be:
  - Allocate each minipage on the page based on attribute value size.
  - Insert records by copying each value into appropriate minipage. When variable-length values are present – minipage boundaries may need to be adjusted:
    - If a record fits in the page but its individual attribute values do not – recalculate minipage sizes based on the average value sizes in the page so far.
  - When the page is full – allocate new page with the initial minipage sizes equal to the ones in the previously populated page.

PAX DATA MANIPULATION ALGORITHMS IMPLEMENTATION

Updates:

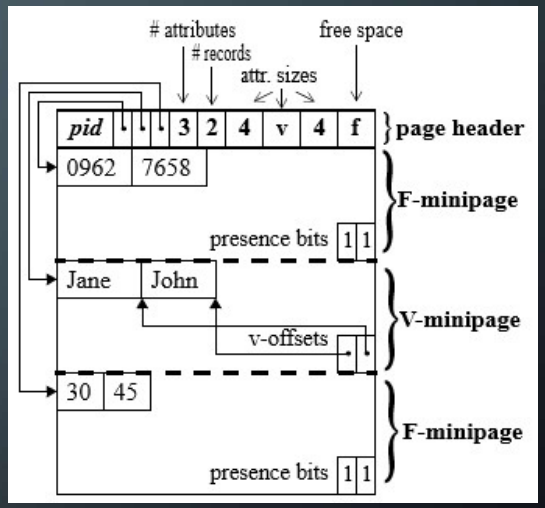
- NSM updates attribute values within the record.
  - For updates on variable-length attributes: page reorganization may be needed and the slot table must be updated.
  - If the updated record grows beyond the free space available in the page – it will be moved to another page.
- PAX updates value by computing the offset of the attribute in the corresponding minipage.
  - Variable length attribute updates require ***only*** V-minipage-level reorganizations.
  - If the new value is longer than the space available in the V-minipage – it borrows space from the neighboring minipage.
  - If the neighboring minipage do not have sufficient space – the record is moved to a different page.



PAX DATA MANIPULATION ALGORITHMS IMPLEMENTATION

Deletions:

- NSM uses the slot array to mark deleted records and free space can be filled upon future insertions.
- PAX keeps track of deleted records using a bitmap at the beginning of the page.
- Upon record deletion PAX:
  - Reorganizes minipage contents to fill the gaps to minimize fragmentation that could affect PAXs optimal cache utilization.
  - Attribute value offsets are calculated by converting the record id to the record index within the page.



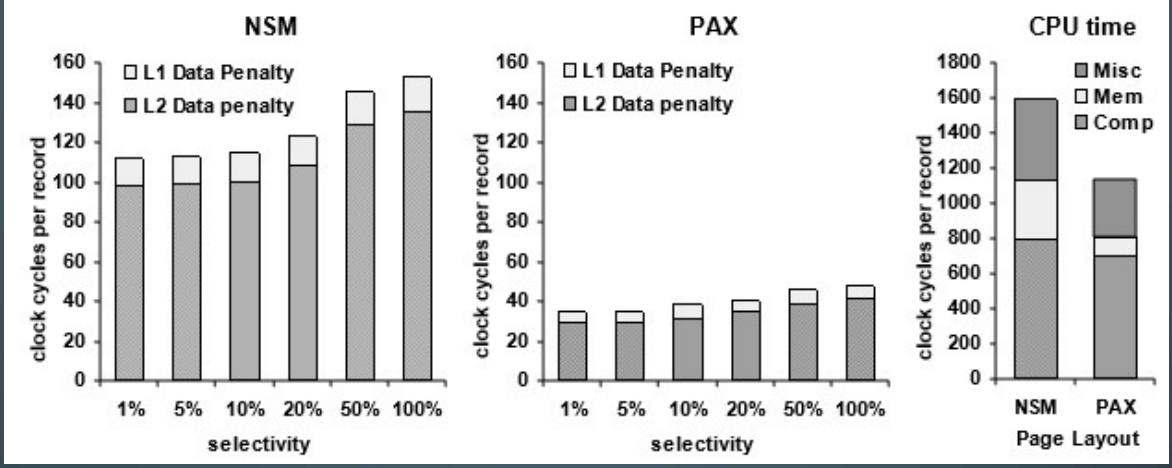
EVALUATION METHODOLOGY

To evaluate PAX performance and compare it to NSM and DSM the following steps were done:

1. Use plain range selection queries on a memory-resident relation that consists of fixed-length attributes.
2. Use both range and complex decision-support queries to obtain results on more complicated queries.
3. Evaluate PAX across three fundamentally different processor and memory hierarchy designs using DSS workload.

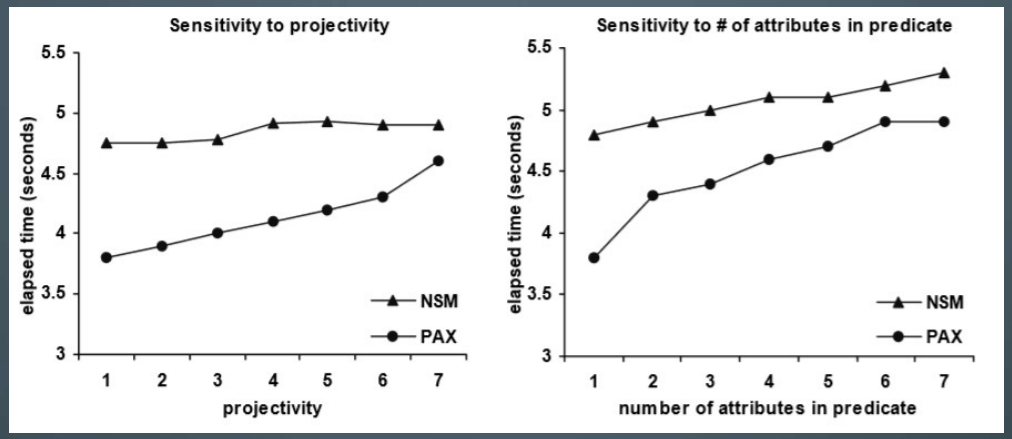
The Third stage completes the experiment cycle by ensuring that the performance results do not depend on a specific processor and memory architecture.

IMPACT ON CACHE BEHAVIOR AND EXECUTION TIME



- The figure on the right illustrates that PAX reduces the delays related to the data cache, and therefore runs queries faster.
  - The graphs on the left and center show the processor stall time per record due to data misses at both cache levels for PAX and NSM.
  - Due to the spatial locality – PAX reduces the data –related penalty at both cache levels.
  - The graph on the right shows the overall processor stall time is 75% less when using PAX, because it doesn't need to wait as long for data to arrive from main memory.

SENSITIVITY ANALYSIS



- As the number of attributes in the query increases, the elapsed execution times of NSM and PAX converge.
- In the left graph the projectivity of the query is varied – PAX is faster even when the result relation includes all attributes.
- In the right graph the number of attributes in the selection predicate is varied – PAX is again faster for locality reasons.
- In these experiments DSM's performance is about a factor of 9 slower then NSM and PAX.

THE LOG-STRUCTURED MERGE-TREE (LSM-TREE)

1996

Patrick E. O'Neil, Edward Cheng, Dieter Gawlick,
and Elizabeth J. O'Neil

THE LSM-TREE

The LSM-Tree is a disk-based data structure designed to provide low-cost indexing for a file experiencing a high rate of record inserts (and deletes) over an extended period.

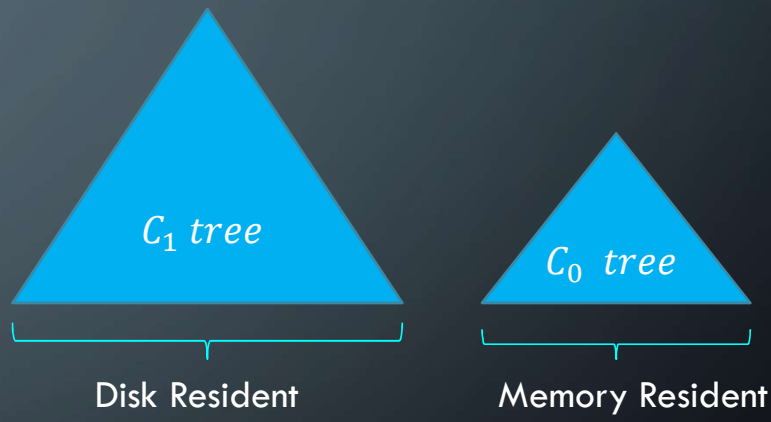
WHAT IS A LSM TREE?

- The LSM Tree access method enables us to perform frequent index inserts to an index with much less disk arm use then with other data structures.
- It also supports operations of indexing such as: deletes, updates, long latency finds with the same deferred efficiency.
- We are trading off disk accesses for memory buffers while the tradeoff gives economic gain.
- In cases where retrieval is much less frequent than insert reducing the cost of index inserts is of paramount importance, at the same time, find access is frequent enough that an index of some kind must be maintained.

The five minute rule:
We can reduce system costs by purchasing memory buffer space to keep pages in memory, thus avoiding disk I/O, when page reference frequency exceeds about once every 60 seconds.
The time period of 60 seconds is approximate.

THE TWO COMPONENT LSM TREE

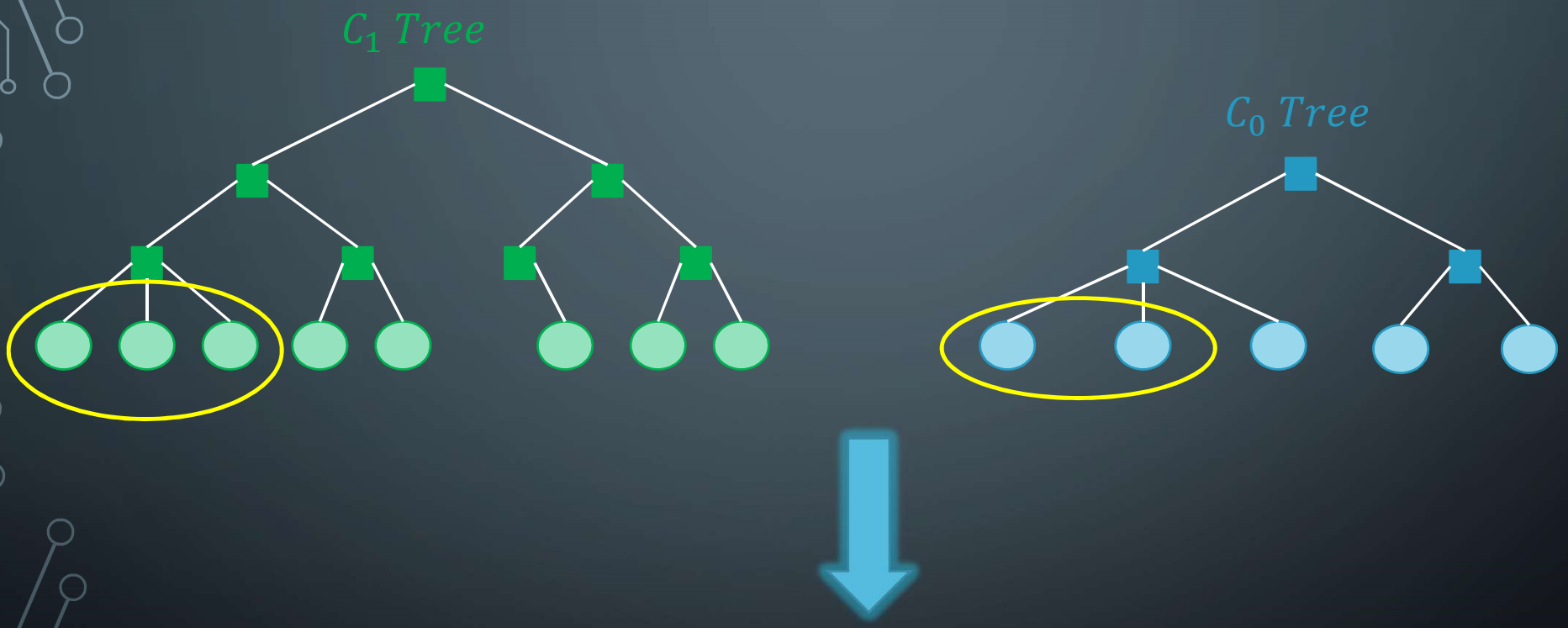
- The C_0 tree is not expected to have a B-tree-like Structure.
- The C_1 tree is a B-tree – providing efficient exact-match access along a path of single page index nodes down to the leaf level.
- There are optimal threshold sizes for the components that can be defined using formula for the I/O cost for inserts into an LSM-tree.



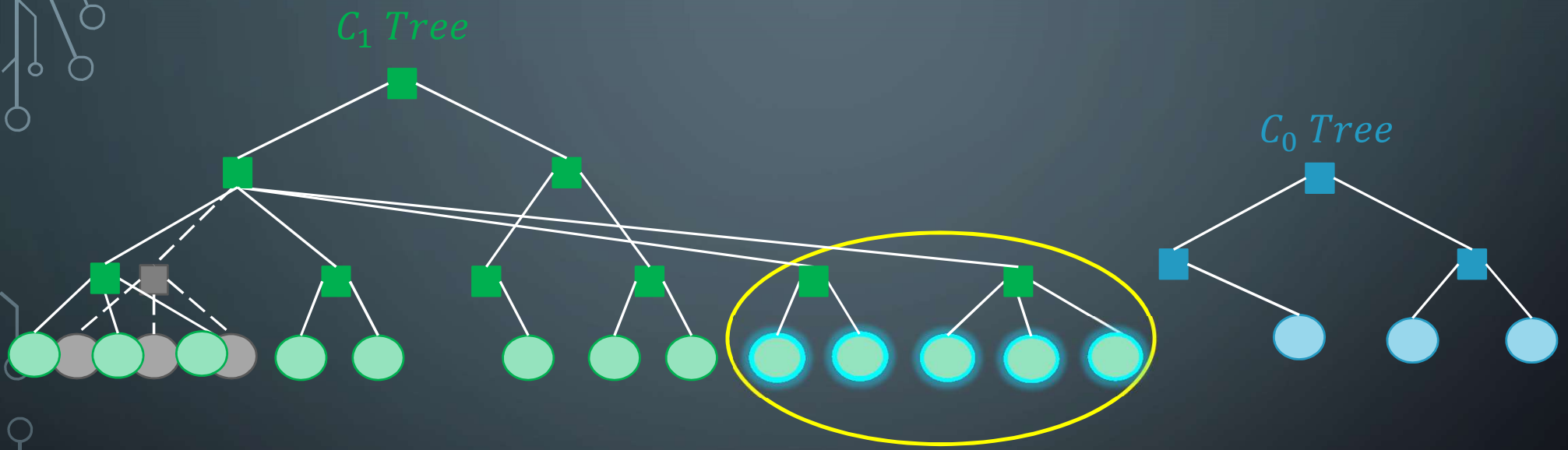
HOW A TWO COMPONENT LSM TREE GROWS

- Why put Data on different media ?
  - $1) \Rightarrow \ Speed \ \& \ Access \ pattern$
  - $2) \Rightarrow \ Price$
  - $3) \Rightarrow \ Durabillity$
- Why we need to merge ?
  - $\Rightarrow \ Because \ we \ put \ data \ on \ different \ media$
- How to merge ?
  - $\Rightarrow \ Lets \ look \ at \ some \ examples$

HOW A TWO COMPONENT LSM TREE GROWS



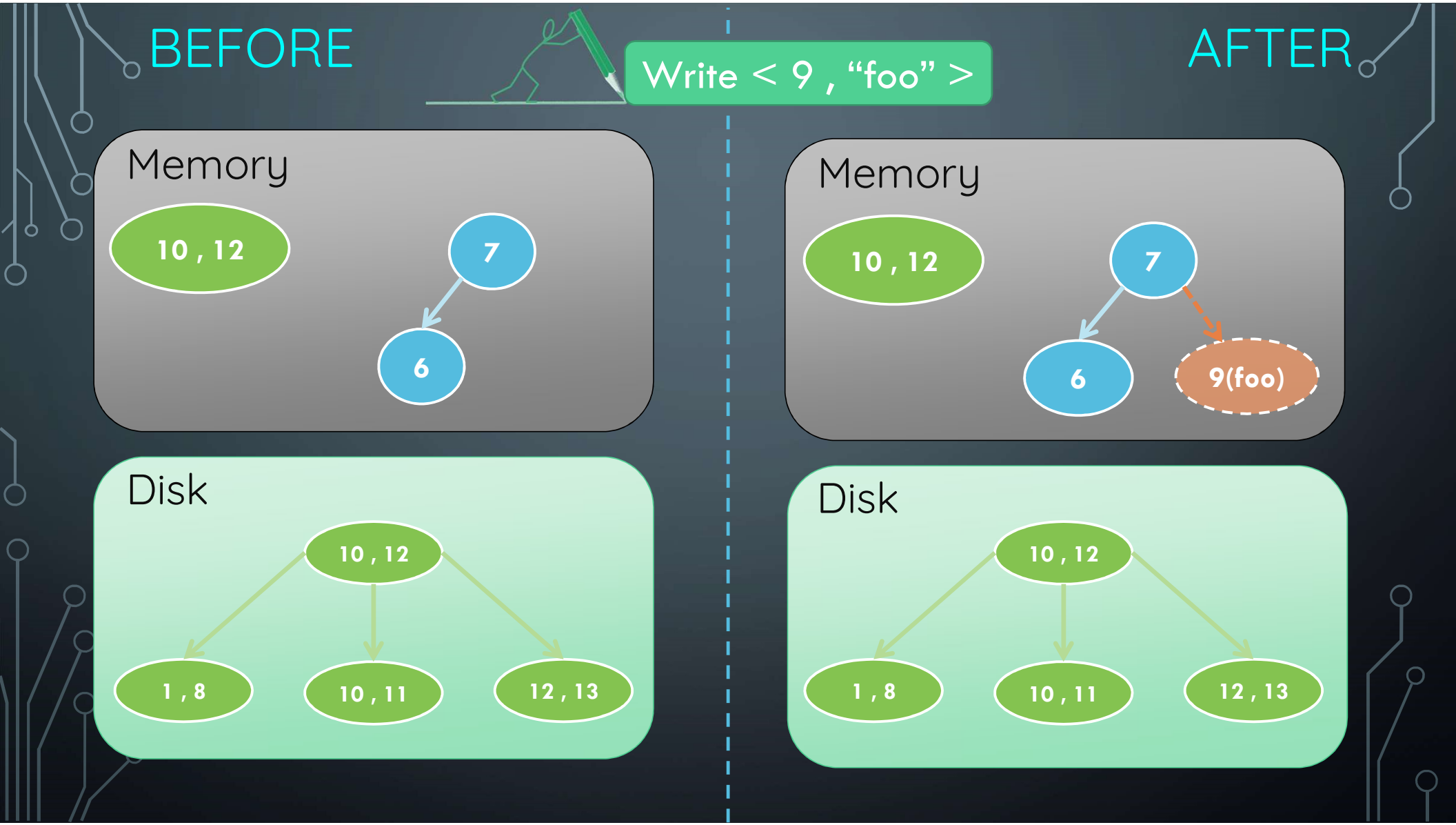
HOW A TWO COMPONENT LSM TREE GROWS

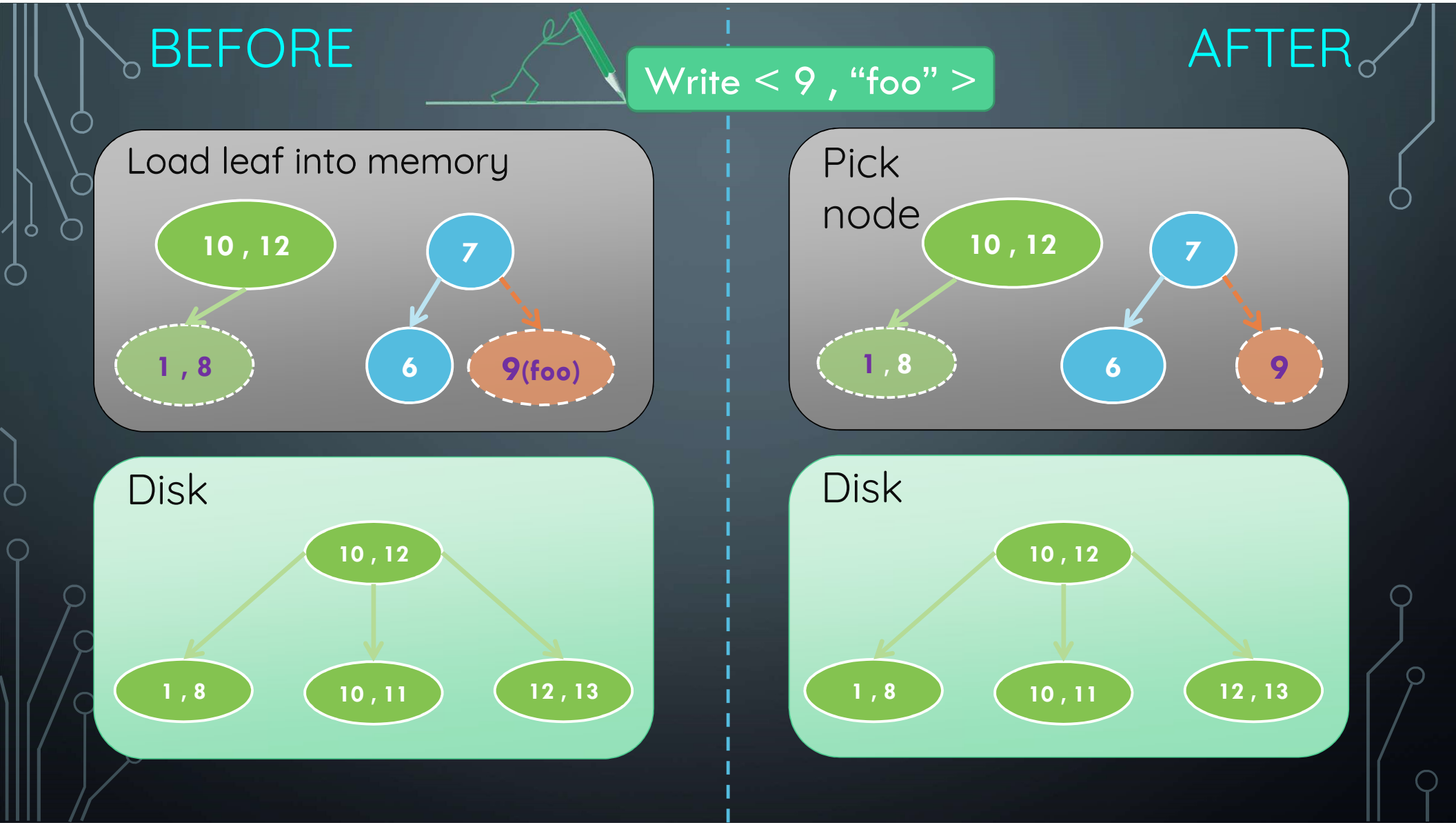


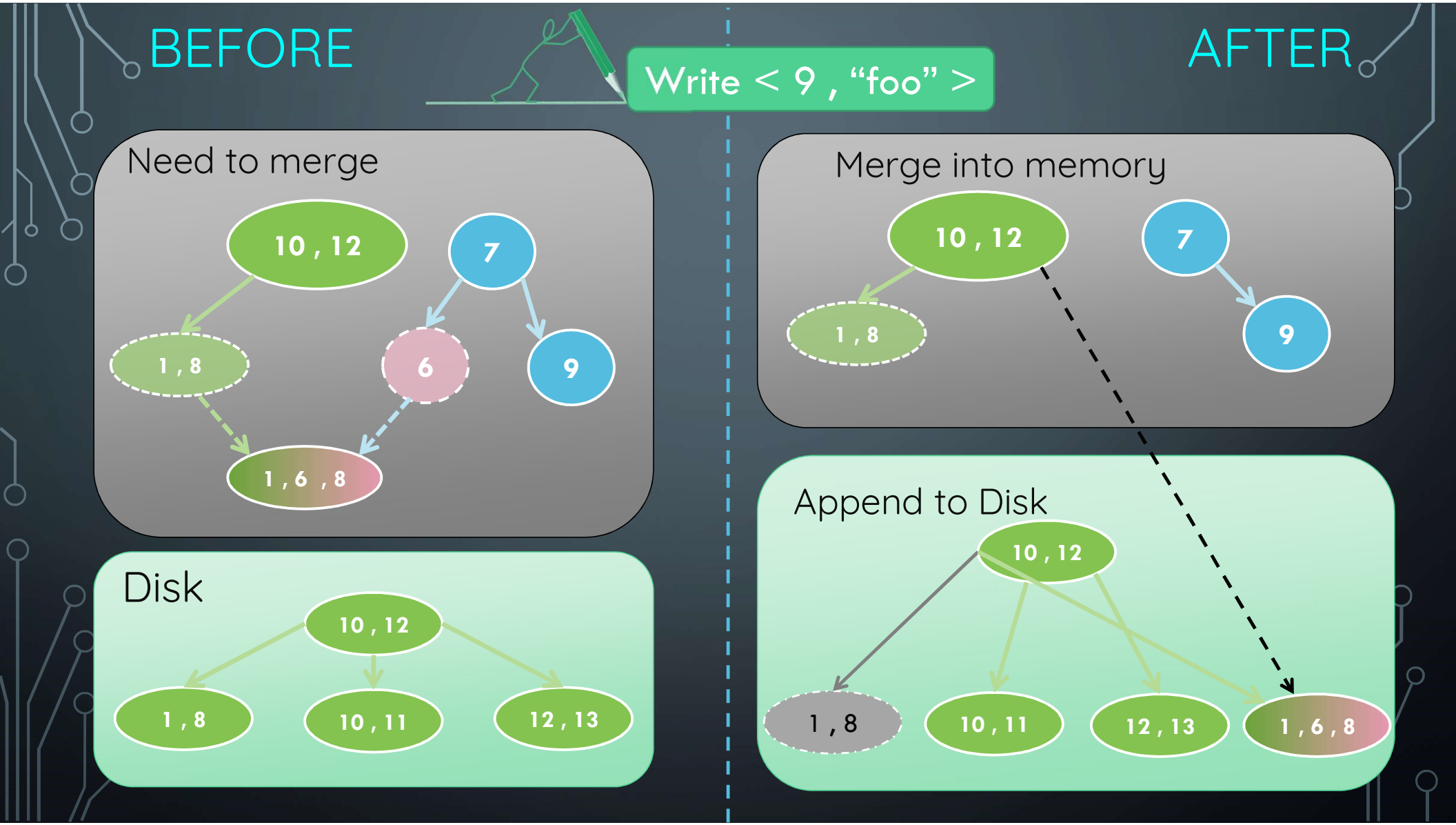
You don't write to the next level until you have to, and you write in the fastest way.

HOW A TWO COMPONENT LSM TREE GROWS

- When the growing C_0 tree first reaches its threshold size a leftmost sequence of entries is deleted from the C_0 tree and reorganized into a C_1 tree leaf node packed 100% full.
- Successive leaf nodes are placed left-to-right in the initial pages of a buffer resident multi-page block until the block is full.
- Then this block is written out to disk to become the first part of the C_1 tree disk-resident leaf level.
- Successive multi page blocks of the C_1 tree leaf level in ever increasing key-sequence order are written out to disk to keep the C_0 tree threshold size from exceeding its threshold.
- After the rightmost leaf entry of the C_0 tree is written out to the C_1 tree for the first time the process starts over on the left end of the two trees.
- In the LSM-Tree, blocks are totally freed up on the trailing edge of the rolling merge, so no extra I/O is involved.


BEFORE
Write < 9 , “foo” >
AFTER
Memory
10 , 12
7
6
Disk
10 , 12
1 , 8
10 , 11
12 , 13
Memory
10 , 12
7
6
9(foo)
Disk
10 , 12
1 , 8
10 , 11
12 , 13


BEFORE
Write < 9 , “foo” >
AFTER
Load leaf into memory
10 , 12
7
1 , 8
6
9(foo)
Pick node
10 , 12
7
1 , 8
6
9
Disk
10 , 12
1 , 8
10 , 11
12 , 13
Disk
10 , 12
1 , 8
10 , 11
12 , 13


BEFORE
Write < 9 , “foo” >
AFTER
Need to merge
10 , 12
7
1 , 8
6
9
1 , 6 , 8
Disk
10 , 12
1 , 8
10 , 11
12 , 13
Merge into memory
10 , 12
7
1 , 8
9
Append to Disk
10 , 12
1 , 8
10 , 11
12 , 13
1 , 6 , 8

OPERATIONS IN THE LSM TREE INDEX - FINDS

- When an exact-match find or range find is performed through the LSM-tree index (for a case of two components) :
  1) The C_0 tree is searched.
  2) The C_1 tree is searched.
- As a rule, in order to guarantee that all entries in the LSM-tree have been examined, it is necessary for an exact-match find or range find to access ***each*** component C_i through its index structure.
- In cases where the most frequent find references are to recently inserted values, many finds can be completed in the C_0 tree.

OPERATIONS IN THE LSM TREE INDEX – DELETES AND UPDATES

Deletes:

- When an indexed row is deleted, if a key-value entry is not found in the appropriate position in the C_0 tree, a delete node entry can be placed in that position.
- The actual delete can be done at a later time during the rolling merge process, when the actual index entry is encountered.
- Find requests must be filtered through delete node entries so as to avoid returning references to deleted records.

Updates:

- Updates of records that cause changes to indexed values are unusual in any kind of applications, but such updates can be handled by LSM-trees in a deferred manner if we view updates as delete followed by an insert.

CONCURRENCY IN THE LSM TREE

There are three distinct types of physical conflict:

1) A find operation should not access a node of a disk-based component at the same time that a different process performing a rolling merge is modifying the contents of the node.
2) A find or insert into the C_0 component should not access the same part of the tree that a different process is simultaneously altering to perform a rolling merge out to C_1.
3) The cursor for the rolling merge from C_{i-1} out to C_i will sometimes need to move past the cursor for the rolling merge from C_i out to C_{i+1}, since the rate of migration out from the component C_{i-1} is always at least as great as the rate of migration out from C_i and this implies a faster rate of circulation of the cursor attached to the smaller component C_{i-1}. Whatever concurrency method is adopted must permit this passage to take place without one process (migration out to C_i) being blocked behind the other at the point of intersection (migration out from C_i).

COMPARISON BETWEEN LSM AND OTHER CONTINUUM STRUCTURES

- Assumption: Newly inserted entries generally are placed in arbitrary positions among the index entries that are already present.
- Continuum Structure: A structure where an immediate placement of a newly inserted index entry is to its ultimate collation order.
- Inserts of new entries in Continuum Structures require $2\ I/Os$ on average - since the size of the index in which they must be placed cannot economically be buffered in memory.
- The LSM-Tree beats the Continuum Structure in terms of I/O performance, reducing the disk arm load as much as two orders of magnitude in certain situations.

B-TREE VS. LSM-TREE COSTS COMPARISON

- We will look at insert costs only.
- Insertion rate of 16,000 Bytes per Second ⇒ 1000 entries (16 Bytes each) per second.
- Resulting in an index of 576 million entries for 20 days of data (9.2 GB of data).

B-tree: the disk I/O will be the limiting factor.

- We are required to use enough disk space to provide 2000 random I/Os per second to update random pages at the leaf level.
- The cost of disk arm to provide one page/second I/O rate – 25$ ⇒ total cost is $50,000.
- The cost for memory – $6400
- ***Total cost – $56,400***

LSM-Tree:

- To store all records on disk – $9200
- The cost for 20MB C_0 component and 2MB of memory for merging blocks– $2200
- ***Total cost – $11,400***

ANY QUESTIONS ?