CS 591: Data Systems Architectures

Prof. Manos Athanassoulis

mathan@bu.edu

http://manos.athanassoulis.net/classes/CS591
Storage Layouts
Rows vs Cols vs Hybrid

A B C D
A B C D
A B C D
A B C D
A B C D
A B C D
A B C D
A B C D
Storage Layouts
Rows vs Cols vs Hybrid

New Hardware
Flash Storage
Multi-core
CS591 progress bar

Storage Layouts
Rows vs Cols vs Hybrid

New Hardware
Flash Storage
Multi-core

Indexing
When to use?
UpBit

index

or

scan
Storage Layouts
- Rows vs Cols vs Hybrid

New Hardware
- Flash Storage
- Multi-core

Indexing
- When to use?
- UpBit

<table>
<thead>
<tr>
<th></th>
<th>A=10</th>
<th>UB</th>
<th>A=20</th>
<th>UB</th>
<th>A=30</th>
<th>UB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

UB
Storage Layouts
Rows vs Cols vs Hybrid

NoSQL Engines
LSM-Trees
Hash-based

New Hardware
Flash Storage
Multi-core

Indexing
When to use?
UpBit

buffer
Bloom filters
fence pointers

memory

storage

X

UpBit
NoSQL Engines
LSM-Trees
Hash-based

buffer
Bloom filters
fence pointers

memory

storage

X
When the asynchronous 1ush operation completes, the 1ush-status set becomes valid, and retry.

Blind updates simply append a new record to the tail of the log, and resets the o-

set value and hence would not be accessing those memory addresses. Note that we must ensure that the to-be-evicted page is safe, we know that all threads would have seen the updated closed-status array entry for the older page frame. When this epoch becomes safe – because threads refresh epochs at every counter increment results in appending the new counter to the tail of the log (reading the older value from storage if necessary), the thread invokes a region-loading of log records to secondary storage. This action is invoked only when the tail enters a new page frame, and the 1ush page resides in main memory at the current head offset.

If yes, the record is in memory and we can proceed as before. If not, we issue an asynchronous read request.

Deletes insert a tombstone record (again, using a header bit), and require log garbage collection.

Updates are first marked invalid and then garbage collected.

Deletions update the log, and log is read-only.

Increasing Logical Address

Read-Copy-Update

In-Place-Update

LA = 0

Stable

Read-Only

Mutable

LA = ∞

Disk In-Memory

LA = ∞

When the tail enters a new page frame, we need to manage the o-

set greater than the last served logical address, we

New record allocation always happens at the tail. We maintain

addresses. Note that we must ensure that the to-be-evicted page is safe, we know that all threads would have seen the updated closed-status array entry for the older page frame. When this epoch becomes safe – because threads refresh epochs at

increasing latch-free concurrent access to records.

Traditional databases use a latch to pin pages in the bu-

er Maintenance

-Loading of log records to secondary storage. The in-memory portion is composed of read-only and

storage in a latch-free manner, as threads perform unrestricted

of the page is set to

LA = ∞

mutable

-Only

-Mutable

-Update

-Read

-Transactional

LA = ∞

LA = 0

Stable

Read-Only

Mutable

LA = ∞

Disk In-Memory

LA = ∞

As the tail grows, an existing page frame may need to be evicted and

ush page before every access so that it is not evicted when in use. For

5.3 Operations with Append-Only Allocator

addresses. Note that we must ensure that the to-be-evicted page is safe, we know that all threads would have seen the updated closed-status array entry for the older page frame. When this epoch becomes safe – because threads refresh epochs at

increasing latch-free concurrent access to records.

Traditional databases use a latch to pin pages in the bu-

er Maintenance

-Loading of log records to secondary storage. The in-memory portion is composed of read-only and

storage in a latch-free manner, as threads perform unrestricted

of the page is set to

LA = ∞

mutable

-Only

-Mutable

-Update

-Read
Data skipping has become an essential mechanism for improving performance in modern analytics databases. The key reason why Skipping-Oriented Partitioning (SOP) is sensitive to feature conflict is that it incurs an unnecessary cost for columns with high conflict. This cost arises from the fact that SOP incorporates a single horizontal partitioning scheme that produces only monolithic horizontal partitioning schemes. That is, SOP views every tuple as an atomic unit. While this perspective is natural for row-major data layouts, it becomes an unnecessary constraint for columnar layouts, and in such systems, a row-based approach can be partitioned differently. Unfortunately, this columnar approach led to the conflict between the horizontal partitioning schemes. By doing so, we can mitigate feature conflict and boost the performance of data skipping.

Consider the table in Figure 1(a). Suppose SOP extracts two features from the workload: course and year. Since SOP generates a single horizontal partitioning scheme that satisfies the "atomic-tuple" constraint and allows different columns to have different horizontal partitioning schemes, it may be rendered ineffective. While SOP has been shown to outperform previous techniques, its effectiveness depends on workload and data characteristics. Modern analytics applications can involve wide tables and complex workloads with diverse filter predicates and column-access patterns. For example, suppose a query contains a filter predicate that evaluates against metadata (such as the statistics of each column) and store the statistics as metadata. Incoming queries can evaluate their filter predicates against such metadata to be read or accessed. For example, suppose a query contains a filter predicate that evaluates against metadata (such as the statistics of each column). The query can be optimized by skipping blocks that are unlikely to be accessed.

In this paper, we propose a fine-grained partitioning scheme that was shown to improve data skipping. We evaluate our approach on a variety of real-world workloads and find that it outperforms previous techniques by a factor of 2-3. Moreover, our approach is scalable and can be applied to large datasets.

Figure 1: (a) A table with four tuples and three columns. (b) A skipping-oriented partitioning scheme for the table in (a). (c) A traditional horizontal partitioning scheme for the table in (a).
The information which key ranges each partition holds is answered by a simple scan of the qualifying partitions. The same reorganization is repeated for the partition into other partition all keys that are greater than or equal to where one partitions contains all keys less than partition into which indicates. If a range query selecting partitioned adaptively with respect to the incoming query pred-

workloads and data distributions, and the trade-off between convergence speed, weak robustness against different query showed, adaptive indexing must deal with high variance, slow convergence speed, as well. For instance, as the investigation of these works introduces a surprising amount of unpleasant problems, methods is that adaptivity, while offering many nice properties, the reason for the necessity of such a large number of per including exists today. In our recent studies, we analyzed large amount of specialized adaptive indexes under various query access patterns and key distributions.

In this regard, in this paper we explore the well-researched class of database cracking as well. Various types of database cracking as well as sorting can be expressed via a function partition-in-

incrementally converges towards the one of a traditional index. Stochastic testing shows, adaptive indexing, it actually has to be extended with numer-

Indexing

New Hardware
Flash Storage
Multi-core

NoSQL Engines
LSM-Trees
Hash-based

Storage Layouts
Rows vs Cols vs Hybrid

Adaptive Indexing
Data Skipping
Adaptive Indexing

Fig. 1: Figure 1 visualizes the concept.

When to use?
UpBit

Indexing

Index Column
Index Column
Index Column
Index Column

Column

Indexing

Columns

NoSQL Engines
LSM-Trees
Hash-based

Storage Layouts
Rows vs Cols vs Hybrid

New Hardware
Flash Storage
Multi-core

Indexing

When to use?
UpBit

Indexing

Index Column
Index Column
Index Column
Index Column

Column

Indexing

Columns

Indexing

Figure 1 visualizes the concept.
CS591 progress bar

**Storage Layouts**
Rows vs Cols vs Hybrid

**New Hardware**
Flash Storage
Multi-core

**Indexing**
When to use?
UpBit

**NoSQL Engines**
LSM-Trees
Hash-based

**Indexing**
Data Skipping
Adaptive Indexing

**Scientific Data Management**
In-situ Query Processing

**Adaptive Partitioning**

- BF
- BF+BTree
- BF
- BTree
- BF
- BF
- BTree

**Cache**

**Positional Map**

**Raw Data File**
Today: Array Data Storage Manager

Up to now: **uni-dimensional** data (integers, real, string)

Array Data: **multi-dimensional** data

No unique order (cannot sort!)

How to store?

**Concepts**: multi-dimensional arrays, storage manager, tiles, thread-safe, dense vs. sparse arrays, global cell order, fragments, dense vs. sparse fragments, consolidation

why is this a challenge?
Do not forget: **reviews**

You can skip up to 3 reviews

18 classes: 5 long + 10 short + 3 skipped

*new rule*: you can do extra long reviews, 1 long counts as 3 short

Normally for full marks: 5 long + 10 short

or 6 long + 7 short

or 7 long + 4 short

or 8 long + 1 short
Do not forget: *project*

Do not leave your project work for last minute!

Until *Tuesday April 16*\(^{th}\) every group in OH to discuss progress

April 30 and May 2 project presentations:
problem + approach + results + open questions

Project presentations will also be peer-evaluated