CS165 Midterm Preparation
Today: Review quizzes

The scan operator and the costs of materialization

Updating data in modern stores

Index design and access

External Sorting

The Halloween problem
Selects and Early/Late Materialization

Cs165 midterm prep
select operator
select min(C) from R where A<10 & B<20

• write query plan and logic of each operator
select min(C) from R where A<10 & B<20

Things to consider
- Which operator to begin? select vs min
- Which filter to apply first?
select min(C) from R where A<10 & B<20

- Sort C
select min(C) from R where A<10 & B<20

- Sort C
- At each min(C), check if A<10 & B < 20
select min(C) from R where A<10 & B<20

- Sort C
- At each min(C), check if A<10 & B < 20
- Shortcomings?
select min(C) from R where A<10 & B<20

- Sort C
- At each min(C), check if A<10 & B < 20
- Shortcomings?
  ◦ Might have to touch all of C, A, and B without getting useful information
select min(C) from R where A<10 & B<20

A<10

- positions_A = A<10
select min(C) from R where A<10 & B<20

- positions_A = A<10
- positions_B = B < 20
select min(C) from R where A<10 & B<20

A<10
- positions_A = A<10
- positions_B = B < 20
- positions_C = positions_A & positions_B
select min(C) from R where A<10 & B<20

- positions_A = A<10
- positions_B = B < 20
- positions_C = positions_A & positions_B
- find min from these available C
select min(C) from R where A<10 & B<20

- positions_A = A<10
- positions_B = B < 20
- positions_C = positions_A & positions_B
  - find min from these available C
  - Shortcomings?
select min(C) from R where A<10 & B<20

A<10
- positions_A = A<10
- positions_B = B < 20
- positions_C = positions_A & positions_B
- find min from these available C
- Shortcomings?
  ◦ Looking through all of A and B
select min(C) from R where A<10 & B<20

A<10
- positions_A = A<10
- positions_B = B < 20
- positions_C = positions_A & positions_B
- find min from these available C
- Shortcomings?
  ◦ Looking through all of A and B
select min(C) from R where A<10 & B<20

- positions_A = A<10
select min(C) from R where A<10 & B<20

A<10
- positions_A = select(A<10)
- values_B = fetch(B, positions_A)
select min(C) from R where A<10 & B<20

A<10
- positions_A = select(A<10)
- values_B = fetch(B, positions_A)
- positions_B = select(values_B<20)
select min(C) from R where A<10 & B<20

A<10
- positions_A = select(A<10)
- values_B = fetch(B, positions_A)
- positions_B = select(values_B<20)
- values_C = fetch(C, positions_B)
- min(values_C)
select min(C) from R where A<10 & B<20

Takeaways
- Applies filters to reduce size of working set
- Pass intermediate results to the next operator
- Apply aggregates/materialization last
Early/Late Materialization
Comparison

Early Materialization
• - row stores
• - materialize data early on

Late materialization
• - column stores
• - materialize data later on
select max(B), max(C), max(D), max(E) where A > v1

Early Materialization

- Materialize all A, B, C, D, E, and when A > v1, keep around B, C, D, E.
- At the end, find max(B), max(C), max(D), max(E)
select max(B), max(C), max(D), max(E) where A > v1

**LATE Materialization**

- p1 = select(A > v1)
- fetch(B,C,D,E)
- min(B), min(C), min(D), min(E)
Comparison

**Early Materialization**
- Performed as soon as the tuples are needed by query plan.
- Prevents multiple disk accesses by materializing data early.

**Late materialization**
- Performed as late as possible, sometimes at the query output.
- In selects, we mostly deal with positions when filtering and only materialize values at the end of the query.
Hybrid solution

- Sometimes a hybrid solution is better
- Balances the CPU cost of early materialization and the extra disk accesses required of late materialization
Updates on Modern Data Systems

CS165 midterm preparation
update row7=(A=a, B=b, C=c, D=d)

which is better to update and why?
how much does it cost to update a single row?
(think about pages, data movement)
how to update in column-stores?
(query plan + algorithms)
Update a row-store vs a col-store

• When updating an entire row, naturally, a row-store is more efficient
Update a row-store vs a col-store

- When updating an entire row, naturally, a row-store is more efficient

Why?
Update a row-store vs a col-store

A  B  C  D

A  B  C  D

A  B  C  D

A  B  C  D
Update a row-store vs a col-store

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

How are rows and columns organized in pages?
Update a row-store vs a col-store
Update a row-store vs a col-store

How many pages to be updated?
Update a row-store vs a col-store
Update a row-store vs a col-store

1 for row-stores and 4 (as many as the columns) for column-stores
How to update in column stores?

• We still need to support updates for column-stores!

• We can use a hybrid approach!
Updates on column-stores

A
B
C
D

base data

pending updates

Queries

select D from TABLE
becomes:
select D from TABLE[base data]
union
select D from TABLE[Pending updates]

Updates
Updates on column-stores

-- Merge base data with buffered updates on query time.
Updates on column-stores

-- Merge base data with buffered updates on query time.
-- Periodically migrate updates to base data.
Data structures for pending updates

• How to store pending updates?
  Differential Files

RowID, ColID, Val
RowID, ColID, Val
RowID, ColID, Val

or

RowID, ValA, ValB, ValC
RowID, ValA, ValB, ValC
RowID, ValA, ValB, ValC

and

RowID, Delete
RowID, Delete
RowID, Delete

or

Col A
Col B
Col C

RowID, Val
RowID, Val
RowID, Val

RowID, Val
RowID, Val
RowID, Val

RowID, Val
RowID, Val
RowID, Val

and

RowID, Delete
RowID, Delete
RowID, Delete
Data structures for pending updates

• How to store pending updates?

  Differential Files

  Organized in a Tree on RowID

<table>
<thead>
<tr>
<th>RowID, ColID, Val</th>
<th>RowID, ValA, ValB, ValC</th>
</tr>
</thead>
<tbody>
<tr>
<td>RowID, ColID, Val</td>
<td>RowID, ValA, ValB, ValC</td>
</tr>
<tr>
<td>RowID, ColID, Val</td>
<td>RowID, ValA, ValB, ValC</td>
</tr>
<tr>
<td>and</td>
<td>and</td>
</tr>
<tr>
<td>RowID, Delete</td>
<td>RowID, Delete</td>
</tr>
</tbody>
</table>

Col A          Col B          Col C
<table>
<thead>
<tr>
<th>RowID, Val</th>
<th>RowID, Val</th>
<th>RowID, Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>RowID, Val</td>
<td>RowID, Val</td>
<td>RowID, Val</td>
</tr>
<tr>
<td>RowID, Val</td>
<td>RowID, Val</td>
<td>RowID, Val</td>
</tr>
<tr>
<td>and</td>
<td>and</td>
<td>and</td>
</tr>
<tr>
<td>RowID, Delete</td>
<td>RowID, Delete</td>
<td>RowID, Delete</td>
</tr>
</tbody>
</table>
Query processing with pending updates

select D from TABLE

merge:
a scan (data naturally ordered by RowID)
with
an index scan (updates ordered by RowID due to index traversal)

Tree on RowID

A | B | C | D
---|---|---|---
RowID, ColID, Val
RowID, ColID, Val
RowID, ColID, Val
RowID, Delete
Index Design and Access
Designing an Index Using a B+Tree

Compare the access costs of using a sorted array and a B+tree. Include a discussion of major design decisions.

- 1 Billion (4 byte) integers
- 4 byte pointers
- 64kb minimum access granularity
What is a B+Tree?

- Linked Data Structure
- All data is held in the leaves
- Configurable Fanout (generally $>>2$
- $\log(n)$ lookups
- Guaranteed tree balance
B+tree design - Index Node Layout

Node Size = n * sizeof(int) + (n + 1) * sizeof(node*) + sizeof(len)
B+tree design - Calculate a good node size

Goals:

- Maximize Fanout (thereby minimizing index node tree height)
- Match the access granularity to the node size (minimize wasted bytes)
- Minimize Search Time (within a node)

\[(\text{length} \times \text{values} + \text{length} \times (\text{pointers} + 1) + \text{int}) \times 4 \text{ bytes} \leq 65536 \text{ bytes}\]
\[2(n + 1) \leq 16384 /* \text{Since each item is int} */\]
\[n \leq 8191\]

Height of the (index nodes of the) tree: \(\text{ceil}(\log_{8191} 1000000000) = 3\)

Use binary search within a node. (What if access granularity was (much) smaller?)
Bytes read on a point get

Sorted array and binary search:

\[ \text{ceil}(1000000000/65536) = 15259 \] -- Pages needed to hold the data

\[ \text{ceil}(\log(15259)) = 14 \times 65536 \] (Minimum Granularity) = 917504 bytes
Bytes read on a point get

Using the B+tree:

\[(3 \text{ (index)} + 1 \text{ (leaf)}) \times 65536 = 262144 \text{ bytes (~71.4\% less data)}\]
External Sorting
Given
Data does not fit in L1 memory
CPU can read/write to L1 only

Find
An algorithm to sort the array
The cost of sorting $C_s$
The cost of accessing the sorted array $C_a$

Memory Hierarchy
Sorting Algorithms
Binary Search
Basic Mathematics :)

CPU
Level n-1
Level n
3 pages

$L_{n-1}$

$N$ pages
CPU

3 pages

$L_{n-1}$

$L_n$

$N$ pages
CPU

3 pages

$L_{n-1}$

$N$ pages

$L_n$
We read and wrote every page once, cost is $2N$.
Now, we need to combine the 3-wise sorted pages.
CPU

3-wise sorted pages

3 pages

$L_{n-1}$

$L_n$

$N$ pages

3-wise sorted pages
3-wise sorted pages

CPU

Merge

3 pages

$L_{n-1}$

$L_n$

N pages

3-wise sorted pages
At the end of the first merge phase, we have combined the 3-wise sorted pages into 6-wise sorted pages.
Sorting: 1 step
Sorting: 1 step          Merging: $\log_2(N/M)$ steps
Sorting: 1 step  Merging: $\log_2(N/M)$ steps  1 step = $2N$ page access
Sorting: 1 step  \( \text{Merging: } \log_2(N/M) \text{ steps} \)

1 step = 2N page access

\[ 2N\left(\log_2(N/M) + 1\right) \]
Generalizing to $M$ pages
Generalizing to $M$ pages

**Observation 1:** The number of pages merged in one round depends on $M$, the size of $L1$. We can merge $M-1$ pages at the same time.

**Observation 2:** As a result, the base of the logarithm changes. In other words, we need fewer merging rounds:

$$2N(\log_{M-1}(N/M)+1)$$

$$2N(\log_{M-1}(\lceil N/M \rceil)+1)$$
How much data can we sort in $P$ passes?

How much memory do we need?

$$P \geq (\log_{M-1}(N/M)+1)$$

<table>
<thead>
<tr>
<th>$N$</th>
<th>$M=3$</th>
<th>$M=5$</th>
<th>$M=9$</th>
<th>$M=17$</th>
<th>$M=128$</th>
<th>$M=256$</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1,000</td>
<td>10</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>10,000</td>
<td>13</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>100,000</td>
<td>17</td>
<td>9</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1,000,000</td>
<td>20</td>
<td>10</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>10,000,000</td>
<td>23</td>
<td>12</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>100,000,000</td>
<td>26</td>
<td>14</td>
<td>9</td>
<td>7</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>30</td>
<td>15</td>
<td>10</td>
<td>8</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>
Further Questions

*Once sorted, what is the cost of access?*

*Binary search vs. Scan?*

\[ C_a = \log_2(N) \]

*When do we sort? Which column to sort?*
The Halloween Problem

CS165 - Midterm Review
Question

- Employee(Id, name, address, office, salary, year hired, ...)

- We have a B-tree index on table Employee which uses salary as the key and also contains attributes “name” and “year hired”.

- We want to give a 5% raise to all employees that work for more than 10 years in the company and have a salary lower than 100K.

Tasks:

1) Write the SQL query

2) How to update the B-tree?

3) What is the query plan?
UPDATE Employee
SET salary = salary * 1.05
WHERE (year(curdate()) - year_hired) > 10
AND salary < 100000;
How to update the B-Tree?

Naïve approach:

• Scan index for all salaries < 100K
• Update during scan

Problem:

• B-tree updates change physical position in index

→ Keys might be visited multiple times!
Example

• Index values:

(40K, 41K, 100K)
Example

• Index values:

(40K, 41K, 100K)
Example

- Index values:
  \((40\text{K}, 41\text{K}, 100\text{K})\)
  \(\Rightarrow\) \((41\text{K}, 42\text{K}, 100\text{K})\)
Example

• Index values:

  \((40K, 41K, 100K)\)

  \(\rightarrow\)  \((41K, 42K, 100K)\)
Example

• Index values:

  \((40K, 41K, 100K)\)

  \(\rightarrow \ (41K, 42K, 100K)\)

  \(\rightarrow \ (42K, 43K, 100K)\)
Example

• Index values:

  (40K, 41K, 100K)

  $\rightarrow$ (41K, 42K, 100K)

  $\rightarrow$ (42K (!), 43K, 100K)
Example

- Index values:

  (40K, 41K, 100K)

  -> (41K, 42K, 100K)

  -> (42K, 43K, 100K)

  ...

  -> (100K, 101K, 104K)

Everybody will get as many raises as they need until they earn >= 100K!
* The **Halloween Problem** was discovered on Halloween (Which Halloween remains a mystery. Probably in 1975)
Halloween Protection - Part 1

**Idea:** Isolate rows chosen by the filter from the effect of the update (Separate read and write cursors)

→ Get qualifying IDs first, then update in one go
Halloween Protection - Part 2

**Idea:** Remember already updated tuples and only update unmodified tuples

→ Track updated tuples

(e.g. using a bit vector, hash table or timestamps)