Scans and Indexes in Main-Memory Optimized Data Systems

Michael S. Kester  Manos Athanassoulis  Stratos Idreos
Ask for what you want...

```
SELECT x
FROM table_A
WHERE
  y < 10;
```

...the system chooses how to retrieve it.
Select Operator

```
SELECT x
FROM table_A
WHERE y < 10;
```

```
select /* TPC_H Query 17 */
sum(l_extendedprice) / 7.0 as avg_yearly
from lineitem, part
where p_partkey = l_partkey
and p_brand = '[BRAND]' 
and p_container = '[CONTAINER]'
and l_quantity < ( 
select 0.2 * avg(l_quantity) 
from lineitem 
where l_partkey = p_partkey 
)
```
Access Path Selection

Full Base Data Scan

Secondary Index Scan
(Auxiliary copy of the data + structure)
Selectivity

A fixed threshold

Scan is best

Index is best

0%
Increasing pressure on indexes

Vertica / C-store
Vectorwise / MonetDB
SQLServer
IBM BLU
Oracle
SAP
...

Selectivity Crossover

1960  2010  2015?
Do we need >1 access path?

How do we choose an access path?
Do we need >1 access path?
Yes.
How do we choose an access path?
Like before, but different.
Variable selectivity crossover
Outline

System design changes in modern data systems

Modern access method modeling

Access Path Selection and FastColumns

Experimental Results

Next steps

Data Skipping & Adaptive Denormalization
SIMD

Multi-Core

Vectorized Processing
Shared Scans

S\(_1\) S\(_2\) S\(_3\)

r\(_1\) r\(_2\) r\(_3\)

S\(_1\)

r\(_1\) r\(_2\) r\(_3\)
Making Scans Faster (Multithreaded Processing)

1 Query

4 threads?

$r_1$
Partition the Relation

1 Query

4 threads?

\[ r_1 \]
4 Query

4 threads?

$r_1$
Partition the Queries

4 threads?

4 Query

C₁
C₂
C₃
C₄

r₁, r₂, r₃, r₄
Making Scans Faster (Multithreaded Processing)

Tuple Count = 60M
Threads = 64
Selectivity = ~0.1%
Making Scans Faster (SIMD)

```
simd_select(0x1878010,0x187a730, 8, 9, 5, 8)
```

```
SELECT x
FROM table_A
WHERE
  y BETWEEN 5 AND 8;
```
<table>
<thead>
<tr>
<th>rid</th>
<th>vec_four</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data + i</th>
<th>vec_left</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>vec_right</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>8</td>
</tr>
</tbody>
</table>
vec_left_result

cmpgt

1 1 1 0

vec_right_result

cmplt

1 0 1 1

and_si

1 0 1 1

1 0 1 0

vec_result
vec_result

\[
\begin{array}{cccc}
1 & 0 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
0xFFFFFFFF & 0x00000000 & 0xFFFFFFFF & 0x00000000 \\
\end{array}
\]

_mm_movemask_ps

m = 0x00001010

perm = 1,3,0,2
vec_result

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

permutevar (perm = 1,3,0,2)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

maskstore

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>

pos (before popcnt)

pos (after popcnt)

rid

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
</tr>
</tbody>
</table>

permutevar (perm = 1,3,0,2)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>12</td>
<td>9</td>
<td>11</td>
</tr>
</tbody>
</table>

maskstore

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
</tr>
</tbody>
</table>

rid

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>15</td>
<td>14</td>
<td>13</td>
</tr>
</tbody>
</table>
Making Scans Faster (SIMD)

Tuple Count = 60M
Threads = 64
Queries = 64

[Graph showing runtime in seconds vs. selectivity, comparing Sharescan (SISD) and Sharescan (SIMD)]
Do we need >1 access path?

How do we choose an access path?
Outline

System design changes in modern data systems

Modern access method modeling

Access Path Selection and FastColumns

Experimental Results

Next steps

Data Skipping & Adaptive Denormalization
Model: Scan Cost

- Base data scan
- Predicate evaluation
- Result set writing

$p_i$
Model: Scan Cost

Base data scan

Result set writing
Model: Shared Scan Cost

Base data scan

Result set writing
Model: Index Scan Cost
Model: Batched Index Scan Cost
APS = \frac{\text{index\_scan}}{\text{scan}}
APS = \frac{\text{index}\_\text{scan}}{\text{scan}}

APS = \frac{\text{Tree Trav.} + S \cdot \text{Leaf Trav.} + \sum s_i \cdot \text{Result Wr.} + \sum s_i \cdot \text{Sorting}}{\text{Scan Base Data} + \sum s_i \cdot \text{Result Wr.}}
$$APS = \frac{\text{index}\_scan}{\text{scan}}$$

$$APS = \frac{\text{Tree Trav.} + S \cdot \text{Leaf Trav.} + \sum s_i \cdot \text{Result Wr.} + \sum s_i \cdot \text{Sorting}}{\text{Scan Base Data} + (\sum s_i) \cdot \text{Result Wr.}}$$
When APS < 1, we choose the index, otherwise we use a scan.
Yes, we do need >1 access path
Concurrency moves the crossover
Do we need >1 access path?
Yes.
How do we choose an access path?
Like before, but different.
The model captures hardware
The optimizer needs to be fast, how much does this slow the system down?
How many more errors does this type of access path selection introduce?
## Crossover Predictor

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>HDD (ms)</td>
<td>250</td>
<td>10</td>
<td>8</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>HDD (MB/s)</td>
<td>1</td>
<td>40</td>
<td>100</td>
<td>500</td>
<td>500</td>
</tr>
<tr>
<td># tuples</td>
<td>$10^4$</td>
<td>$10^7$</td>
<td>$10^8$</td>
<td>$10^9$</td>
<td>$10^{10}$</td>
</tr>
<tr>
<td>Tuple Size</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>4</td>
</tr>
<tr>
<td>Branching</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>Crossover</td>
<td>12.8%</td>
<td>10.7%</td>
<td>5.5%</td>
<td>4.45%</td>
<td>0.1%</td>
</tr>
</tbody>
</table>

![Graph showing Crossover Selectivity over time](image)
Next Steps

Shared tree scans

Query scheduling

Parallel result writing

Access paths and hardware accelerators
Tree Scan Cost Components

Shared Tree Scan
Shared Tree Scan

Leaves of the tree

<table>
<thead>
<tr>
<th>...</th>
<th>10</th>
<th>13</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>29</td>
<td>33</td>
<td>...</td>
</tr>
<tr>
<td>13</td>
<td>12</td>
<td>90</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>...</th>
<th>3</th>
<th>65</th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td>62</td>
<td>63</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>66</td>
<td>67</td>
<td>...</td>
</tr>
</tbody>
</table>

Universal Result Set

<table>
<thead>
<tr>
<th>...</th>
<th>3</th>
<th>65</th>
<th>22</th>
<th>13</th>
<th>12</th>
<th>90</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>q₁</td>
<td>q₂</td>
<td>q₁</td>
<td>q₂</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Tree Scan Cost Components

Shared Tree Scan
Outline

System design changes in modern data systems
Modern access method modeling
Access Path Selection and FastColumns
Experimental Results
Next steps
Data Skipping & Adaptive Denormalization
Partial Universal Tables

Fast query performance without requiring join operators
Adaptive Data Skipping

Continuously reorganized light weight indexing
We need >1 Access Path.

Crossover selectivity is not a fixed point, it is influenced by concurrency.

Access Path Selection can be done quickly and efficiently.

Scan is best

Index is best

Concurrency

Thank you