Hardware Conscious Algorithm Design

Niv Dayan, CS165
L1 cache
~ 10 KB, ~ < 1 ns

L2 cache
~ 100 KB, ~ 3 ns

L3 cache
~ 1 MB, ~ 10 ns

Main memory
~ 10 GB, ~ 100 ns

Secondary Storage
~ 100 GB, ~ 1 ms
L1 cache

L2 cache

L3 cache

Main memory

Secondary Storage

~ 10 KB, ~ < 1 ns

~ 100 KB, ~ 3 ns

~ 1 MB, ~ 10 ns

~ 10 GB, ~ 100 ns

~ 100 GB ~ 1 ms

Volatile

Persistent
The processor is fast!  

- L1 cache: ~10 KB, ~< 1 ns
- L2 cache: ~100 KB, ~3 ns
- L3 cache: ~1 MB, ~10 ns
- Main memory: ~10 GB, ~100 ns
- Secondary Storage: ~100 GB, ~1 ms

Possible bottleneck
Design Principles

Principle 1: exploit temporal locality
L1 cache

L2 cache

L3 cache

Main memory

Secondary Storage

Transfer unit

- 64 B
- 4KB

- ~ 10 KB, ~ < 1 ns
- ~ 100 KB, ~ 3 ns
- ~ 1 MB, ~ 10 ns
- ~ 10 GB, ~ 100 ns
- ~ 100 GB, ~ 1 ms

~ 100 GB
~ 1 ms

~ 100 ns
~ 10 ns
~ 3 ns
~ < 1 ns
~ ~ 10 KB
~ 100 KB
~ 1 MB
~ 10 GB
~ 100 GB
Design Principles

Principle 1: exploit temporal locality
Principle 2: exploit spatial locality
Examples

Suppose a unit of transfer is 12 bytes, or 3 integers.
Examples

Pathological application access pattern: 1, 4, 7, 2, 5, 8, 3, 6, 9

Cache

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

Main memory

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

<table>
<thead>
<tr>
<th>Cache hits</th>
<th>Cache misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Examples

Pathological application access pattern: 1, 4, 7, 2, 5, 8, 3, 6, 9

Cache

1 2 3

Main memory

1 2 3 4 5 6 7 8 9

<table>
<thead>
<tr>
<th>Cache hits</th>
<th>Cache misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Examples

Pathological application access pattern: 1, 4, 7, 2, 5, 8, 3, 6, 9

Cache

<table>
<thead>
<tr>
<th>Cache hits</th>
<th>Cache misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>
Examples

Pathological application access pattern: 1, 4, 7, 2, 5, 8, 3, 6, 9

<table>
<thead>
<tr>
<th>Cache hits</th>
<th>Cache misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>
Examples

Pathological application access pattern: 1, 4, 7, 2, 5, 8, 3, 6, 9

Cache

Main memory

<table>
<thead>
<tr>
<th>Cache hits</th>
<th>Cache misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>
Examples

Pathological application access pattern: 1, 4, 7, 2, 5, 8, 3, 6, 9

Cache

```
  4 5 6
```

Main memory

```
  1 2 3 4 5 6 7 8 9
```

<table>
<thead>
<tr>
<th>Cache hits</th>
<th>Cache misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>
Examples

Pathological application access pattern: 1, 4, 7, 2, 5, 8

<table>
<thead>
<tr>
<th>Cache hits</th>
<th>Cache misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>
Examples

Pathological application access pattern: 1, 4, 7, 2, 5, 8

<table>
<thead>
<tr>
<th>Cache</th>
<th>Main memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 8 9</td>
<td>1 2 3 4 5 6 7 8 9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache hits</th>
<th>Cache misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

100% cache misses!
Examples

Spatial locality access pattern: 1, 2, 3, 4, 5, 6

Cache

<table>
<thead>
<tr>
<th>Cache hits</th>
<th>Cache misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Main memory

1 2 3 4 5 6 7 8 9
Examples

Spatial locality access pattern: 1, 2, 3, 4, 5, 6

<table>
<thead>
<tr>
<th>Cache hits</th>
<th>Cache misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Main memory:

1 2 3 4 5 6 7 8 9

Cache:

1 2 3
Examples

Spatial locality access pattern: 1, 2, 3, 4, 5, 6

Cache

Main memory

<table>
<thead>
<tr>
<th>Cache hits</th>
<th>Cache misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>
Examples

Spatial locality access pattern: 1, 2, 3, 4, 5, 6

<table>
<thead>
<tr>
<th>Cache</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main memory</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache hits</th>
<th>Cache misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

33% cache misses
Examples

Temporal locality access pattern: 1, 3, 2, 2, 1, 1

Cache

Main memory

<table>
<thead>
<tr>
<th>Cache hits</th>
<th>Cache misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

1 2 3 4 5 6 7 8 9
Examples

Temporal locality access pattern: 1, 3, 2, 2, 1, 1

Cache hits | Cache misses
---|---
5 | 1
Examples

Temporal locality access pattern: 1, 3, 2, 2, 1, 1

<table>
<thead>
<tr>
<th>Cache</th>
<th>Main memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3</td>
<td>1 2 3 4 5 6 7 8 9</td>
</tr>
</tbody>
</table>

Cache hits | Cache misses
---|---
5 | 1

16% cache misses
Take-home Messages

• We can measure the cache-friendliness of an access pattern.
Take-home Messages

• We can measure the cache-friendliness of an access pattern.

• When we design an algorithm or data structure, we have control over how elements are ordered, and how we access them.
Take-home Messages

• We can measure the cache-friendliness of an access pattern.

• When we design an algorithm or data structure, we have control over how elements are ordered, and how we access them.

• We want to design algorithms and data structures that exhibit as much temporal and spatial locality as possible, so cache misses are minimized.
Case study 1: row-store vs. column store
## Case study 1: row-store vs. column store

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Title</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John Smith</td>
<td>DBA</td>
<td>110 K</td>
</tr>
<tr>
<td>2</td>
<td>Lisa Johnson</td>
<td>Programmer</td>
<td>100 K</td>
</tr>
<tr>
<td>3</td>
<td>Bob Williams</td>
<td>Architect</td>
<td>120 K</td>
</tr>
<tr>
<td>4</td>
<td>Mary Jones</td>
<td>Manager</td>
<td>130 K</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Title</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John Smith</td>
<td>DBA</td>
<td>110 K</td>
</tr>
<tr>
<td>2</td>
<td>Lisa Johnson</td>
<td>Programmer</td>
<td>100 K</td>
</tr>
<tr>
<td>3</td>
<td>Bob Williams</td>
<td>Architect</td>
<td>120 K</td>
</tr>
<tr>
<td>4</td>
<td>Mary Jones</td>
<td>Manager</td>
<td>130 K</td>
</tr>
</tbody>
</table>
Case study 1: row-store vs. column store

### Row-store

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Title</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John Smith</td>
<td>DBA</td>
<td>110 K</td>
</tr>
<tr>
<td>2</td>
<td>Lisa Johnson</td>
<td>Programmer</td>
<td>100 K</td>
</tr>
<tr>
<td>3</td>
<td>Bob Williams</td>
<td>Architect</td>
<td>120 K</td>
</tr>
<tr>
<td>4</td>
<td>Mary Jones</td>
<td>Manager</td>
<td>130 K</td>
</tr>
</tbody>
</table>

### Column-store

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Title</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John Smith</td>
<td>DBA</td>
<td>110 K</td>
</tr>
<tr>
<td>2</td>
<td>Lisa Johnson</td>
<td>Programmer</td>
<td>100 K</td>
</tr>
<tr>
<td>3</td>
<td>Bob Williams</td>
<td>Architect</td>
<td>120 K</td>
</tr>
<tr>
<td>4</td>
<td>Mary Jones</td>
<td>Manager</td>
<td>130 K</td>
</tr>
</tbody>
</table>

Scenario 1: there are mostly updates change the title and salary fields of one employee at a time.
Case study 1: row-store vs. column store

### Scenario 1:
There are mostly updates change the title and salary fields of one employee at a time.

**Row-store**

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Title</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John Smith</td>
<td>DBA</td>
<td>110 K</td>
</tr>
<tr>
<td>2</td>
<td>Lisa Johnson</td>
<td>Programmer</td>
<td>100 K</td>
</tr>
<tr>
<td>3</td>
<td>Bob Williams</td>
<td>Architect</td>
<td>120 K</td>
</tr>
<tr>
<td>4</td>
<td>Mary Jones</td>
<td>Manager</td>
<td>130 K</td>
</tr>
</tbody>
</table>

**Column-store**

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Title</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John Smith</td>
<td>DBA</td>
<td>110 K</td>
</tr>
<tr>
<td>2</td>
<td>Lisa Johnson</td>
<td>Programmer</td>
<td>100 K</td>
</tr>
<tr>
<td>3</td>
<td>Bob Williams</td>
<td>Architect</td>
<td>120 K</td>
</tr>
<tr>
<td>4</td>
<td>Mary Jones</td>
<td>Manager</td>
<td>130 K</td>
</tr>
</tbody>
</table>

**Row-store wins**
Case study 1: row-store vs. column store

### Row-store

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Title</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John Smith</td>
<td>DBA</td>
<td>110 K</td>
</tr>
<tr>
<td>2</td>
<td>Lisa Johnson</td>
<td>Programmer</td>
<td>100 K</td>
</tr>
<tr>
<td>3</td>
<td>Bob Williams</td>
<td>Architect</td>
<td>120 K</td>
</tr>
<tr>
<td>4</td>
<td>Mary Jones</td>
<td>Manager</td>
<td>130 K</td>
</tr>
</tbody>
</table>

### Column-store

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Title</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John Smith</td>
<td>DBA</td>
<td>110 K</td>
</tr>
<tr>
<td>2</td>
<td>Lisa Johnson</td>
<td>Programmer</td>
<td>100 K</td>
</tr>
<tr>
<td>3</td>
<td>Bob Williams</td>
<td>Architect</td>
<td>120 K</td>
</tr>
<tr>
<td>4</td>
<td>Mary Jones</td>
<td>Manager</td>
<td>130 K</td>
</tr>
</tbody>
</table>

Scenario 1: there are mostly updates change the title and salary fields of one employee at a time. **Row-store wins**

Scenario 2: there are mostly queries to find the average salary.
### Case study 1: row-store vs. column store

#### Scenario 1:
There are mostly updates that change the title and salary fields of one employee at a time. **Row-store wins**

#### Scenario 2:
There are mostly queries to find the average salary. **Column store wins**

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Title</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John Smith</td>
<td>DBA</td>
<td>110 K</td>
</tr>
<tr>
<td>2</td>
<td>Lisa Johnson</td>
<td>Programmer</td>
<td>100 K</td>
</tr>
<tr>
<td>3</td>
<td>Bob Williams</td>
<td>Architect</td>
<td>120 K</td>
</tr>
<tr>
<td>4</td>
<td>Mary Jones</td>
<td>Manager</td>
<td>130 K</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Title</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John Smith</td>
<td>DBA</td>
<td>110 K</td>
</tr>
<tr>
<td>2</td>
<td>Lisa Johnson</td>
<td>Programmer</td>
<td>100 K</td>
</tr>
<tr>
<td>3</td>
<td>Bob Williams</td>
<td>Architect</td>
<td>120 K</td>
</tr>
<tr>
<td>4</td>
<td>Mary Jones</td>
<td>Manager</td>
<td>130 K</td>
</tr>
</tbody>
</table>
Case study 1: row-store vs. column store

• Take-home message: the storage format that achieves the best temporal and/or spatial locality depends on the workload.

• When we design a data structure or algorithm, it is extremely useful to know some properties about the workload so that we can exploit different types of localities to improve performance.
Case study 2: array traversal

• Suppose we have a two dimensional C array of integers.
• We want to assign all integers to 0.
Case study 2: array traversal

• Suppose we have a two dimensional C array of integers.
• We want to assign all integers to 0.
• Two ways:

```c
for (int i = 0; i < dimension1_max; i++) {
    for (int j = 0; j < dimension2_max; j++) {
        array[i][j] = 0;
    }
}

for (int j = 0; j < dimension2_max; j++) {
    for (int i = 0; i < dimension1_max; i++) {
        array[i][j] = 0;
    }
}
```
Case study 2: array traversal

• Suppose we have a two dimensional C array of integers.
• We want to assign all integers to 0.
• Two ways:

```c
for (int i = 0; i < dimension1_max; i++) {
    for (int j = 0; j < dimension2_max; j++) {
        array[i][j] = 0;
    }
}
```

```c
for (int j = 0; j < dimension2_max; j++) {
    for (int i = 0; i < dimension1_max; i++) {
        array[i][j] = 0;
    }
}
```
Case study 2: array traversal

• Suppose we have a two dimensional C array of integers.
• We want to assign all integers to 0.
• Two ways:

```c
for (int i = 0; i < dimension1_max; i++) {
    for (int j = 0; j < dimension2_max; j++) {
        array[i][j] = 0;
    }
}
```

```c
for (int j = 0; j < dimension2_max; j++) {
    for (int i = 0; i < dimension1_max; i++) {
        array[i][j] = 0;
    }
}
```
Case study 2: array traversal

• Suppose we have a two dimensional C array of integers.
• We want to assign all integers to 0.
• Two ways:

```c
for (int i = 0; i < dimension1_max; i++) {
    for (int j = 0; j < dimension2_max; j++) {
        array[i][j] = 0;
    }
}
```

```c
for (int j = 0; j < dimension2_max; j++) {
    for (int i = 0; i < dimension1_max; i++) {
        array[i][j] = 0;
    }
}
```

Spatial locality

No locality
Case study 3: maximize temporal locality

- Suppose we have a table of data in disk
- Stored in traditional row-store format
- Spans many disk blocks
Case study 3: maximize temporal locality

- Suppose we have a table of data in disk
- Stored in traditional row-store format
- Spans many disk blocks
- We have some main memory
- Goal: keep in main memory frequently accessed blocks
Case study 3: maximize temporal locality

- Suppose we have a table of data in disk
- Stored in traditional row-store format
- Spans many disk blocks
- We have some main memory
- Goal: keep in main memory frequently accessed blocks
Case study 3: maximize temporal locality

- Central question: when and how to find a block to evict?
  - First in first out (FIFO)?
  - Least recently used (LRU)?
Case study 3: maximize temporal locality

- Central question: when and how to find a block to evict?
  - First in first out (FIFO)?
  - Least recently used (LRU)?

- What are the most efficient data structures needed to implement LRU?
  - How to find a block quickly?
  - How to find the least recently used block quickly?
Case study 3: maximize temporal locality

• LRU naïve solution: hash table
  • Finding block takes $O(1)$
  • Finding least recently used block takes $O(N)$
Case study 3: maximize temporal locality

• LRU naïve solution: hash table
  • Finding block takes $O(1)$
  • Finding least recently used block takes $O(N)$

• LRU naïve solution 2: linked list
  • Finding least recently used block takes $O(1)$
  • Finding block takes $O(N)$
Case study 3: maximize temporal locality

• LRU naïve solution: hash table
  • Finding block takes $O(1)$
  • Finding least recently used block takes $O(N)$

• LRU naïve solution 2: linked list
  • Finding least recently used block takes $O(1)$
  • Finding block takes $O(N)$
Case study 3: maximize temporal locality

• LRU naïve solution: hash table
  • Finding block takes O(1)
  • Finding least recently used block takes O(N)

• LRU naïve solution 2: linked list
  • Finding least recently used block takes O(1)
  • Finding block takes O(N)
Case study 3: maximize temporal locality

- Combine solutions:
Case study 3: maximize temporal locality

- Combine solutions:
  - Example 1: application accesses block Z
Case study 3: maximize temporal locality

- Combine solutions:
  - Example 1: application accesses block Z
Case study 3: maximize temporal locality

• Combine solutions:
  • Example 1: application accesses block Z
  • Example 2: application removes LRU block P
Case study 3: maximize temporal locality

• Combine solutions:
  • Example 1: application accesses block Z
  • Example 2: application removes LRU block P
Case study 3: maximize temporal locality

• Combine solutions:
  • Example 1: application accesses block Z
  • Example 2: application removes LRU block P

• Both operations take O(1).
Case study 3: maximize temporal locality

• Does this generalize?
• Should the caches also implement a buffer pool?
Case study 3: maximize temporal locality

• Does this generalize?
• Should the caches also implement a buffer pool?
• The caches do implement a limited form of an LRU eviction policy in hardware.
Case study 3: maximize temporal locality

- Take-home messages
  - We often don’t know future application access pattern
  - We still want to take advantage of temporal locality if any exists
  - Buffer pool allows doing that in a database system.
  - Define eviction policy
  - Implement policy efficiently using appropriate data structures

---

A	C	X

Buffer pool

A	B	C	D	X

Main memory

Disk
Case study 4: search in sorted array

- Basic solution: binary search
- E.g. search for key 47.
Case study 4: search in sorted array

• Basic solution: binary search
• E.g. search for key 47.
Case study 4: search in sorted array

• Basic solution: binary search
• E.g. search for key 47.
Case study 4: search in sorted array

• Basic solution: binary search
• E.g. search for key 47.
Case study 4: search in sorted array

• Basic solution: binary search
• E.g. search for key 47.
Case study 4: search in sorted array

- Basic solution: binary search
- E.g. search for key 47.
Case study 4: search in sorted array

• Basic solution: binary search
• E.g. search for key 47.
Case study 4: search in sorted array

• Basic solution: binary search
• E.g. search for key 47.

Found!

4  10  17  27  35  38  47  60  64  74  82  84  88  89  64
Case study 4: search in sorted array

• Basic solution: binary search
• E.g. search for key 47.
• We divide the problem in 2 every step.
• Thus, there are $O(\log_2(N))$ steps, where $N$ is the size of the array.
Case study 4: search in sorted array

• Basic solution: binary search
• E.g. search for key 47.
• We divide the problem in 2 every step.
• Thus, there are $O(\log_2(N))$ steps, where $N$ is the size of the array.
• Still best solution given memory hierarchy?

Found!
Case study 4: search in sorted array

- Consider an array of integers that is too big to fit in main memory.
- We store it in secondary storage.
- How long does it take to find and fetch an element to the processor?
Case study 4: search in sorted array

- Consider an array of integers that is too big to fit in main memory.
- We store it in secondary storage.
- How long does it take to find and fetch an element to the processor?

Bottleneck: moving data from disk to main memory
Case study 4: search in sorted array

• Suppose there are B integers in each block.
• Number of disk reads per binary search is: $O(\log_2(N) - \log_2(B))$
Case study 4: search in sorted array

- Suppose there are B integers in each block.
- Number of disk reads per binary search is: \( O(\log_2(N) - \log_2(B)) \)

- E.g. Let \( N \) be \( 2^{30} \), and let \( B \) be \( 2^{10} \), and each IO is 10 ms.
- Approx 20 IOs, so 200 milliseconds.

Disk blocks containing sorted array
Case study 4: search in sorted array

- Suppose there are $B$ integers in each block.
- Number of disk reads per binary search is: $O(\log_2(N) - \log_2(B))$

- E.g. Let $N$ be $2^{30}$, and let $B$ be $2^{10}$, and each IO is 10 ms.
- Approx 20 IOs, so 200 milliseconds.
- Can we do better?
Case study 4: search in sorted array

• Create tree on top.
• Each parent block knows the key range of its children blocks.
Case study 4: search in sorted array

• Create tree on top.
• Each parent block knows the key range of its children blocks.
• Suppose all of these blocks are saved in secondary storage.
• How many IOs?

Disk blocks containing sorted array
Case study 4: search in sorted array

- Create tree on top.
- Each parent block knows the key range of its children blocks.
- Suppose all of these blocks are saved in secondary storage.
- How many IOs?

Depth: $O(\log_B(N))$
Case study 4: search in sorted array

- E.g. Let N be $2^{30}$, and let B be $2^{10}$, and each IO is 10 ms.
- Approx 3 IOs, so 30 milliseconds. 85% reduction in cost.

Depth: $O(\log_B(N))$
Case study 4: search in sorted array

- Read $O(\log_B(N))$ nodes. In each node, do $O(\log_2(B))$ comparisons.
- Total comparisons: $O(\log_2(B) \times \log_B(N)) = O(\log_2(N))$
- We just changed their distribution across disk reads
Case study 4: search in sorted array

• As unit of transfer size decreases?
  • $O(\log_B(N))$ reads
  • As unit of transfer size decreases, $B$ decreases
  • Number of reads approaches $O(\log_2(N))$
  • The extra metadata becomes overhead
  • Solution becomes less competitive
Case study 4: search in sorted array

- Limitation of cost models
  - Our model is $O(\log_B(N))$
  - Works as long as latency difference is massive
  - Works for disk
  - Otherwise, it underestimates true cost

```
L1 cache
~ < 1 ns
```
```
L2 cache
~ 3 ns
```
```
L3 cache
~ 10 ns
```
```
Main memory
~ 100 ns
```
```
Secondary Storage
~ 1 ms
```
Case study 4: search in sorted array

• Take-home messages:
  • Always identify and model the bottleneck using baseline algorithm
  • Then identify techniques to alleviate the bottleneck.

• Different solutions may not work equally well for all levels of the hierarchy
• Modelling is power but has limitations

• Basic mechanism behind a B-tree.
Case study 4: search in sorted array

• Extension: Suppose we want to optimize for multiple levels.
• I.e., as few reads as possible across all levels, not just from disk
Case study 4: search in sorted array

• Extension: Suppose we want to optimize for multiple levels.
• I.e., as few reads as possible across all levels, not just from disk

Conclusion

• Exploiting temporal and spatial locality
• Quantifying cache friendliness of algorithm
• Exploit spatial locality with row-stores or column stores
• Exploit spatial locality in array traversal
• Exploit temporal locality with buffer pool
• Exploit spatial locality in sorted array search
Conclusion

• Exploiting temporal and spatial locality
• Quantifying cache friendliness of algorithm
• Exploit spatial locality with row-stores or column-stores
• Exploit spatial locality in array traversal
• Exploit temporal locality with buffer pool
• Exploit spatial locality in sorted array search

• Thanks for listening!