ARC: A Self-Tuning, Low Overhead Replacement Cache

Presented by Naju Mancheril
February 23, 2005

Outline

• Introduction and Motivation
  – Why manage buffers?
  – Why not use LRU or DBMin?
• Frequency and Recency
  – LRU-K
  – ARC
• Comparisons and Conclusions

Why Use Buffers?

• Size, speed, cost ... pick any two.
  You cannot have all three.
  – General rule for computer memories.
  – Motivates use of caches.
• Cache Hierarchy
  – Use small, fast, memories to temporarily store segments of larger, smaller memories.
  – Programs with temporal and spatial locality run as if we have a large fast memory.
Why Use Buffers?

- Many databases are HUGE.
- All databases need non-volatile storage (think ‘D’ in ACID).
- **Solution**: Use a disk to store pages.
- **Downside**: Disk I/O is slowwwwww…
- **Compromise**: Use memory (RAM) to store active pages.

**Five-Minute Rule:**

“We are willing to pay more for memory buffers up to a certain point, in order to reduce the cost of disk arms for a system.”

-- Gray and Putzolu

---

Basic Buffer Management

- **Transfer data in pages.**
  - page = smallest chunk copied in and out of RAM.
  - LARGE enough to improve sequential access
  - Small enough to improve random access (index traversal, leaves of non-clustered index).
- **Buffer Pool**
  - Don’t go through OS virtual memory model.
  - Let the DBMS manage its own memory and get better performance.
  - 1 page per buffer.

---

Basic Buffer Management

- **On-line Demand Paging**
  - When a page we want is not in memory, load it into our buffer pool.
- **Online Pre-fetching**
  - Predict future page accesses.
  - Not that crazy since DBMS can look at the page in context.
- **Offline Optimal**
  - Provides upper bounds for performance of online algorithms.
Basic Buffer Management

• On-line Demand Paging
  – When a page we want is not in memory, load it into our buffer pool.

• Online Pre-fetching
  – Predict future page accesses.
  – Not that crazy since DBMS can look at the page in context.

• Offline Optimal
  – Provides upper bounds for performance of online algorithms.

Basic Buffer Management

• Buffer pool has limited number of buffers.
  – Need load/unload policy.
  – Lazy unload: don’t flush a page from buffer pool except to make room for a new page
    • Since most database systems are dedicated, underutilizing the buffer pool offers NO ADVANTAGE.

• Load
  – If we have room, simply copy page into an empty buffer.
  – If all buffers occupied, select a victim page to replace.

• Unload
  – Only copy page back to disk if we have made changes.

Evaluating Buffer Management

• Cache Hit Ratio

\[
C = \frac{h}{T}
\]

The ratio of buffer pool hits to total number of page references.
Every researcher tries to measure C once the algorithm reaches a “quasi-stable” state ... It is not always clear what this means.
Replacement Policy

• Under the demand paging model, the selection procedure for the victim page differentiates one buffer management system from another.
• Optimal Hit Ratio with Belady’s MIN:
  – At time $t$, replace the page that will referenced furthest in the future.

Replacement Policy

• Optimal Hit Ratio with Belady’s MIN:
  – At time $t$, replace the page that will referenced furthest in the future.
  – Only one problem … we need to predict the future.
  – MIN is an offline algorithm.
  – But it does provide an upper bound on the hit ratio for a given reference pattern.

Approximating MIN: LRU

• LRU replaces the page that has not been accessed for the longest time.
  – Replace page accessed furthest in the past
  – Intuition:
    • If we haven’t accessed a page in a long time, maybe we don’t need to look at it again.
  – Upside:
    • Implemented in most commercial systems.
  – Downside:
    • Looping scans (LS) can lead to hit ratio of 0.
    • Even if we start with frequently accessed pages in buffers, a sequential scan will corrupt the pool.
Approximating MIN: MRU

- MRU replaces the page that has been accessed most recently.
  - Intuition:
    - This performs really well on looping scans.
  - Upside:
    - Looping scans: \( m \) buffers, \( m-1 \) hits per loop.
  - Downside:
    - Does not favor temporal locality.
    - MRU implemented selectively, if DBMS knows it is in a looping sequential scan.
    - Scan resistant: once frequently used pages are in buffers, a sequential scan cannot corrupt the pool.

Can We Do Better?

- We want to keep frequently accessed pages in buffers so most requests will be satisfied quickly.
Can We Do Better?

• We want to keep frequently accessed pages in buffers so most requests will be satisfied quickly.

• But LRU does not take into account page access frequency!
• It only tracks how recently a page was accessed.

DBMin

• Query optimizer uses the query plan's access pattern to estimate how often each page will be accessed.
• Frequently accessed pages (such as inner relations) are kept buffers.
  – **Upside:**
    • Got pretty good performance.
  – **Downside:**
    • Very complicated bookkeeping.
    • High overhead.

LFU: Least Frequently Used

• LFU replaces the page that has been accessed the least frequently.
  – **Upside:**
    • Low overhead: just track an access count.
  – **Downside:**
    • Often need to maintain statistics on pages outside the buffer pool.
  – **Optimizations:**
    • Timeout window: after page p is out for 50 references, stop tracking its frequency.
LFU: Least Frequently Used

- LFU replaces the page that has been accessed the least frequently.
  - **Upside:**
    - We force the frequently used pages into buffer pool.
    - Optimal under Independent Reference Model.
    - Scan resistant.
  - **Downside:**
    - Has a hard time adapting to changing access patterns since we keep whole history.
  - **Optimizations:**
    - Truncate history: the latest 1,000,000 references to a table are probably better indicators of what will be popular than the previous 1,000,000 references.

---

**LRU-K**

- Track the $K$ recent accesses to a page.
- **Backward Distance**
  
  Given a reference sequence $r_1, r_2, \ldots, r_t$

  \[
  b(p, i) = \begin{cases} 
  \infty & \text{if } p \text{ does not appear at least } i \text{ times in } r_{t-1}, r_t \\
  \inf \{j \mid r_{t-j} = p \text{ and exactly } i \text{ other values } j' \text{ with } r_{t-j'} \neq p, \text{ where } j' > p \}
  \end{cases}
  \]

- **Example:** the current access sequence has been $a, b, c, a, a, d$. Now we access $r_t = b$ (not in buffer pool) and need to evict a page.

  \[
  \begin{align*}
  b(a, 1) &= 2 \\
  b(a, 2) &= 3 \\
  b(a, 3) &= 6 \\
  b(b, 1) &= 0 \\
  b(b, 2) &= 5 \\
  b(c, 1) &= 4 \\
  b(c, 2) &= \infty \\
  b(d, 1) &= 1 \\
  b(d, 2) &= \infty
  \end{align*}
  \]
**LRU-K**

- When it comes time to select a victim page, simply remove the page with the largest backward-K distance.

- That is, select a page $p$ that satisfies
  \[ b_t(p,K) = \max_{\rho \in \text{pool}} \{ b_t(p,\rho) \} \]

- Why is this reasonable?
  - If we let $K=1$ and evict page with largest backward 1-distance, we are doing LRU!
  - LRU: evict $p$ if $p$ satisfies
  \[ b_t(p,1) = \max_{\rho \in \text{pool}} \{ b_t(p,\rho) \} \]
  - LRU == LRU-1
  - LRU-K: consider a more general approach.

**LRU-2**

- When it comes time to select a victim page, remove any page that has only been accessed once. It will have infinite $b_t(p,2)$
- If every page has been accessed at least twice, select page that’s first sighting
- That is, select a page $p$ that’s first access was furthest in the past.
LRU-K

- Reward frequent access.
- Remove pages that have been accessed 1x before removing pages that have been accessed 2x.
- Remove pages that have been accessed 2x before removing pages that have been accessed 3x.
- Remove pages that have been accessed \( K-1 \) times before removing pages that have been accessed \( K \) times.

... if all else fails, apply LRU

Advantages of LRU-K

- Whenever LRU reads an infrequently accessed page, it keeps it for a long time. LRU-K can detect a backward-\( K \) distance of infinity and drop immediately.
- Since we are only keeping history of \( K \) accesses, LRU-K adapts faster than naïve LFU to changes in access frequency.
- In multi-user environments, query optimizer plans can start to overlap and interfere – not clear how to deal with these
  - But LRU-K does not need to track buffers at the query, or even the user level. It is a more global replacement policy.

Probabilistic Interpretation

- Can estimate inter-access period, even with LRU-2.
- Why is this important?
  - When \( \Pr(t_r = r) \) is independent of \( t \) (true under Independent Reference Model) the optimal algorithm \( A_0 \) evicts the page with lowest reference probability.
  - This page has the highest expected inter-access period.
  - LRU-K:
    - Page with highest backward \( K \)-distance has the highest estimated inter-access period.
    - Page with highest backward \( K \)-distance has the lowest estimated reference probability.
    - If probabilities reach steady-state, LRU-K is an approximation of \( A_0 \).
Other Quirks of LRU-K

• To avoid "metronome errors" we don't drop a page's history after we flush it to disk.
• We can store the history with the page.
  – After all, the only time we need to access the history is when we access the page (at which point it is in memory).
  – Or to decide when to evict it (at which point it is in memory)
• Only problem is that now we always need to write pages back to disk... can't simply drop them, even if they were only read by query.
• To avoid this, keep page history in memory.

Page Access History

• To avoid having to always write back flushed pages, keep page history in memory.
• It is infeasible to keep the page forever. How long should we go before we can drop this history?

• System Parameter: Retained Information Period
  • Use max Backward K-distance of all pages we want to ensure to be memory resident. Authors recommend 200 seconds.
Other Quirks of LRU-K

- Queries often perform correlated access to a given page.
  - To read value, then modify it (i.e. increment)
  - Retry an aborted operation
  - Batch-processing: only touch ½ page at a time.
- Correlation leads to 2-3 quick accesses.
  - We don’t want these creating short K-distance intervals and throwing off LRU-K algorithm.
  - We need a way to treat a set of correlated accesses as one access.
  - We also want to account for the possibility that a correlated access will occur … don’t immediately drop a page once we’ve read it.

Correlated Access

- How can we deal with correlated access?

  - System Parameter: Correlated Reference Period

  - When a page is loaded, don’t make it a candidate for replacement until CRP has passed.
  - If we get another access within this period, extend CRP.
    - Requires slight modification of those timestamps we’ve been saving for page p – we need to shrink the correlated reference period down to a point in history – add | CRP | to old timestamps.
Cache Hit Ratio

- Two pools of 100, and 10,000 pages
- Access pattern alternates between two pools.
  - I1, R1, I2, R2, I3, R3

Cache Hit Ratio

- Single pool of 1000 pages
- 80-20 data skew

Cache Hit Ratio

- OLTP trace
- 470,000 page references.
The Case for ARC

- There are two main problems with LRU-K
  - Requires $O(\log n)$ time to select victim page since all page histories are stored in a binary search tree. In practice, this does not scale well.
  - Requires a database administrator to set values for Retained Information Period and Correlated Reference Period.
- So?

The Case for ARC

- The choice for CIP can dramatically affect performance of LRU-2. How good is your DB administrator?

<table>
<thead>
<tr>
<th>x</th>
<th>0.01</th>
<th>0.05</th>
<th>0.1</th>
<th>0.25</th>
<th>0.5</th>
<th>0.75</th>
<th>0.9</th>
<th>0.95</th>
<th>0.99</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0.87</td>
<td>1.63</td>
<td>2.41</td>
<td>3.09</td>
<td>3.77</td>
<td>4.00</td>
<td>4.02</td>
<td>4.04</td>
<td>4.06</td>
</tr>
<tr>
<td>2048</td>
<td>1.38</td>
<td>2.58</td>
<td>2.87</td>
<td>4.22</td>
<td>4.82</td>
<td>4.77</td>
<td>4.90</td>
<td>5.00</td>
<td>5.00</td>
</tr>
<tr>
<td>4096</td>
<td>2.84</td>
<td>4.05</td>
<td>1.38</td>
<td>3.42</td>
<td>5.79</td>
<td>5.70</td>
<td>5.61</td>
<td>5.61</td>
<td>5.61</td>
</tr>
<tr>
<td>8192</td>
<td>5.86</td>
<td>7.02</td>
<td>7.28</td>
<td>7.76</td>
<td>8.38</td>
<td>8.47</td>
<td>8.36</td>
<td>8.25</td>
<td>8.25</td>
</tr>
<tr>
<td>16384</td>
<td>9.75</td>
<td>10.57</td>
<td>10.47</td>
<td>10.47</td>
<td>9.41</td>
<td>8.98</td>
<td>7.80</td>
<td>7.28</td>
<td>7.15</td>
</tr>
<tr>
<td>32768</td>
<td>15.85</td>
<td>15.47</td>
<td>16.21</td>
<td>16.18</td>
<td>15.32</td>
<td>15.46</td>
<td>15.45</td>
<td>9.82</td>
<td>9.50</td>
</tr>
<tr>
<td>131072</td>
<td>39.20</td>
<td>40.86</td>
<td>40.31</td>
<td>39.46</td>
<td>39.12</td>
<td>32.21</td>
<td>30.79</td>
<td>29.70</td>
<td>29.98</td>
</tr>
<tr>
<td>262144</td>
<td>56.47</td>
<td>55.64</td>
<td>53.80</td>
<td>57.75</td>
<td>40.42</td>
<td>48.70</td>
<td>46.20</td>
<td>40.14</td>
<td>40.14</td>
</tr>
<tr>
<td>524288</td>
<td>89.57</td>
<td>83.14</td>
<td>65.94</td>
<td>62.90</td>
<td>62.90</td>
<td>62.73</td>
<td>60.40</td>
<td>60.37</td>
<td>60.82</td>
</tr>
</tbody>
</table>

The Case for ARC

- Other constant time algorithms:
  - 2Q has two tunable parameters $K_i$ and $K_{out}$
  - LIRS has tunable parameter $L_{RIS}$
  - LRFU has tunable parameter $k$ that determines whether it behaves like LFU ($k=0$) or LRU ($k=1$).
- We can create classes of replacement policies...
- A contains B if all pages tracked by B are tracked by A.
- Perform poorly if tuned poorly.
- OK for stable workloads, but need tuning every time workload characteristics change.
- We need on-line adaptation.
**DBL(2c)**

- **Main idea:**
  - Keep 2 LRU lists of pages.
  - L1 tracks pages seen once in order of access (LRU)
  - L2 tracks pages that have been accessed more than once (high frequency pages)

For a cache of size 2c, replace the LRU page in L1 if L1 contains c pages. Otherwise, replace LRU page of L2. We have the following invariants:

\[
0 \leq |L_1| + |L_2| \leq 2c \\
0 \leq |L_1| \leq c \\
0 \leq |L_2| \leq 2c
\]

**π(c)**

- Divide L1 and L2 into top and bottom
  - L1 = T1 . B1
  - L2 = T2 . B2
- More formally...

**ARC**

- Divide L1 and L2 into top and bottom
  - L1 = T1 . B1
  - L2 = T2 . B2
- Keep p pages from T1 and c-p pages from T2 in cache.

- B.1 If |T1,p| > p, replace the LRU page in T1.p
- B.2 If |T1,p| < p, replace the LRU page in T2.p

- B.3 is arbitrary. We can vary p to favor T1 or T2.
ARC

- Divide L1 and L2 into top and bottom
  - L1 = T1 . B1
  - L2 = T2 . B2
- A hit in B1 should result in us favoring L1 (increase \( p \)). A hit in B2 should let us favor L2 (decrease \( p \)).
- When \(|B1| > |B2|\)
  - Hit in B1 leads to increment of \( p \) by 1.
  - Hit in B2 decrement of \( p \) by \(|B1|/|B2|\).
- When \(|B1| < |B2|\)
  - Hit in B1 leads to increment of \( p \) by \(|B2|/|B1|\)
  - Hit in B2 decrement of \( p \) by 1.

ARC

- Hit ratios

<table>
<thead>
<tr>
<th>( e )</th>
<th>LRU</th>
<th>MQ</th>
<th>ARC</th>
<th>( \Pi1 )</th>
<th>LRU-2</th>
<th>LRU-3</th>
<th>LRU-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1052</td>
<td>0.97</td>
<td>0.95</td>
<td>1.22</td>
<td>0.94</td>
<td>1.03</td>
<td>0.89</td>
<td>0.70</td>
</tr>
<tr>
<td>2048</td>
<td>0.45</td>
<td>0.45</td>
<td>2.43</td>
<td>2.27</td>
<td>3.01</td>
<td>2.18</td>
<td>1.71</td>
</tr>
<tr>
<td>4096</td>
<td>0.73</td>
<td>0.81</td>
<td>4.28</td>
<td>3.83</td>
<td>5.09</td>
<td>3.53</td>
<td>3.62</td>
</tr>
<tr>
<td>8192</td>
<td>2.30</td>
<td>2.82</td>
<td>9.38</td>
<td>10.27</td>
<td>9.87</td>
<td>7.58</td>
<td>7.67</td>
</tr>
<tr>
<td>16384</td>
<td>7.37</td>
<td>9.44</td>
<td>16.68</td>
<td>15.78</td>
<td>17.18</td>
<td>14.83</td>
<td>15.79</td>
</tr>
<tr>
<td>32768</td>
<td>17.18</td>
<td>25.75</td>
<td>37.63</td>
<td>31.33</td>
<td>28.86</td>
<td>28.37</td>
<td>27.39</td>
</tr>
<tr>
<td>65536</td>
<td>96.16</td>
<td>48.76</td>
<td>43.62</td>
<td>47.91</td>
<td>45.77</td>
<td>48.72</td>
<td>43.93</td>
</tr>
<tr>
<td>131072</td>
<td>62.10</td>
<td>69.70</td>
<td>69.35</td>
<td>69.45</td>
<td>67.56</td>
<td>66.60</td>
<td>69.65</td>
</tr>
<tr>
<td>262144</td>
<td>80.38</td>
<td>90.47</td>
<td>89.38</td>
<td>88.03</td>
<td>90.59</td>
<td>90.32</td>
<td>89.78</td>
</tr>
<tr>
<td>524288</td>
<td>96.77</td>
<td>96.83</td>
<td>97.30</td>
<td>96.86</td>
<td>97.32</td>
<td>97.38</td>
<td>97.21</td>
</tr>
</tbody>
</table>

ARC

- Hit ratios

<table>
<thead>
<tr>
<th>( e )</th>
<th>LRU</th>
<th>MQ</th>
<th>ARC</th>
<th>( \Pi1 )</th>
<th>LRU-2</th>
<th>LRU-3</th>
<th>LRU-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1052</td>
<td>0.97</td>
<td>0.95</td>
<td>1.16</td>
<td>1.13</td>
<td>2.07</td>
<td>4.09</td>
<td>4.08</td>
</tr>
<tr>
<td>2048</td>
<td>4.84</td>
<td>4.83</td>
<td>4.89</td>
<td>4.89</td>
<td>4.83</td>
<td>4.84</td>
<td>4.83</td>
</tr>
<tr>
<td>4096</td>
<td>5.61</td>
<td>5.61</td>
<td>5.76</td>
<td>5.78</td>
<td>5.81</td>
<td>5.61</td>
<td>5.61</td>
</tr>
<tr>
<td>8192</td>
<td>6.22</td>
<td>6.23</td>
<td>7.14</td>
<td>7.52</td>
<td>7.54</td>
<td>7.29</td>
<td>6.81</td>
</tr>
<tr>
<td>16384</td>
<td>7.00</td>
<td>7.11</td>
<td>10.12</td>
<td>11.65</td>
<td>10.67</td>
<td>11.01</td>
<td>9.29</td>
</tr>
<tr>
<td>32768</td>
<td>8.93</td>
<td>9.56</td>
<td>13.94</td>
<td>15.89</td>
<td>16.36</td>
<td>16.35</td>
<td>11.15</td>
</tr>
<tr>
<td>65536</td>
<td>14.48</td>
<td>20.82</td>
<td>26.09</td>
<td>27.46</td>
<td>25.79</td>
<td>25.35</td>
<td>25.85</td>
</tr>
<tr>
<td>131072</td>
<td>28.21</td>
<td>31.78</td>
<td>38.65</td>
<td>41.69</td>
<td>39.78</td>
<td>39.78</td>
<td>40.57</td>
</tr>
<tr>
<td>262144</td>
<td>48.91</td>
<td>51.56</td>
<td>53.47</td>
<td>51.31</td>
<td>53.43</td>
<td>54.56</td>
<td>53.85</td>
</tr>
<tr>
<td>524288</td>
<td>50.31</td>
<td>61.35</td>
<td>62.79</td>
<td>61.64</td>
<td>63.17</td>
<td>63.13</td>
<td>63.89</td>
</tr>
</tbody>
</table>
Conclusions

- ARC
  - Has constant overhead (two queues).
  - Hit rate at least as good as LRU.
  - Hit rate approaches optimal tuning parameters of other constant-time algorithms.
  - May be smarter than most DB administrators…

Conclusions

- How quickly does it adapt?