Hash Performance

Larry Strickland, Ph.D.
Chief Product Officer
Data Kinetics
Overview

- DataKinetics – a well kept secret
- Where is Hash used?
- What does Hash do?
- Hash for Indexing
- Hash algorithms, generic steps
- Performance Impacts of the steps
- Some results
DataKinetics - Company Highlights

- Solving IT problems and reducing IT costs for the Fortune 500 Who’s Who for over 35 Years
- We have the world’s largest companies as clients:
  - 5 of the top 7 US banks
  - 2 of the top 3 credit card companies
  - 5 of the top 8 US property & casualty insurance companies
  - The top 2 US health insurance companies
  - 8 of the top 22 in the Fortune 500
- We have been working with these companies for a combined 1000+ years

© 2014 DataKinetics Ltd. All rights reserved.
DataKinetics Products

• tableBASE Family – very fast in memory table manager. Accelerating batch and Online processing

• SQData – Heterogeneous Change Data Replication

• DB2 Tools – Quality Assurance and Performance Analysis tools for DB2

• AutoSoftCapping – dynamically optimizing Defined Capacity to minimize SCRT R4HA (MLC $)
Where is Hash used?

- Hash tables
  - Finding duplicates
  - Faster lookup (e.g. number versus arbitrary string)
- Bloom Filters
- Dictionary
- Indexing
- Verifying Integrity
  - Signature
  - CRC
- Etc, etc, etc
What does Hash do?

- Maps space to another space
  - One way
  - Typically shrinks (doesn’t have to)
  - Arbitrary bytes to number
  - Can encrypt
When using Hash to index

• Hash is used to calculate a slot
  – Slot calculated can simply be a pointer to the key (if in memory)
Important aspects of a Hash

• Repeatable process
• Folding/Consuming
  – An arbitrary length variable can be mapped to a fixed length number
• Dividing (remainder)
  – Ability to determine the number of points in mapped space
• Use of Primes?
• Collisions
  – Because it maps larger spaces to smaller ones – collisions can and will occur
Folding/Consuming

- Folding is used to deal with arbitrary length starting point.
- Broken in to blocks (normally bytes) and combined to produce number
Dividing

• To map to arbitrary sized space, the number from the folding is divided by the size of the space, and the remainder is the hash output
  – Modulo N (where N is the size of the space we are mapping to)
Use of Primes

- Most Hash algorithms provide a better distributing across the target space when primes are used.
- The cause is related to keys with patterns in them, where the pattern has a relationship (multiple) of one of the factors.

<table>
<thead>
<tr>
<th>Key</th>
<th>MOD 8</th>
<th>MOD 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>24</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>
Handling collisions

• Mapping to a small space means collisions are inevitable.
  – Two common methods for handling this
    1. Use next available empty slot
      – Advantage – doesn’t need more space (the size of N)
      – Disadvantage – increased collisions – clustering
     » 1. a) Use second HASH to skip that many slots – increased collisions, but reduces clustering
    2. Use link list to have list of items that collide
      – Advantage – Fewer collisions
      – Disadvantage – Space requirements unpredictable (need to consider 100% collisions)
Density

• Density refers to the amount of slots available for the amount of elements to be stored
  – E.g 100 items to be stored. If we allocate a Hash with 500 slots, then we have a density of 20%

• Should it be big or small
  – Probability of a collision is greater, the higher the density (approximately half the density)
  – The lower the density, the more space we require.

• Note, if we are handling collisions by using the next available slot, we must keep the density below 100%
Performance components

• There are two major components of the performance
  – The hash function
  – Collision handling

• Obviously higher collision numbers impact performance. Collisions primarily come from
  – Density
  – Effectiveness of the Hash
What does a well performing Hash behave?

• Low collision rate
  – Independent of Key type, length and distribution

• Easy to calculate (minimize CPU cycles)

• So which is more important
  – It depends
  – Damn!
Example

• Consider Hash algorithms with different effectiveness
  • Cost to handle collisions given a weight of $x$
  • Ideal collision rate is $i$
    – A
      • Cost to calculate Hash is $4x$
      • Collision rate is $i$
    – B
      • Cost to calculate Hash is $3x$
      • Collision rate is $6i$
    – C
      • Cost to calculate Hash is $x$
      • Collision rate is $10i$
Double the key length??

Hash Cost versus Density

Density

<table>
<thead>
<tr>
<th>Density</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0.2</td>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>0.3</td>
<td>4</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>0.4</td>
<td>6</td>
<td>16</td>
<td>24</td>
</tr>
<tr>
<td>0.5</td>
<td>8</td>
<td>32</td>
<td>48</td>
</tr>
<tr>
<td>0.6</td>
<td>10</td>
<td>64</td>
<td>96</td>
</tr>
<tr>
<td>0.7</td>
<td>12</td>
<td>128</td>
<td>192</td>
</tr>
<tr>
<td>0.8</td>
<td>14</td>
<td>256</td>
<td>384</td>
</tr>
<tr>
<td>0.9</td>
<td>16</td>
<td>512</td>
<td>640</td>
</tr>
<tr>
<td>1.0</td>
<td>18</td>
<td>1024</td>
<td>1280</td>
</tr>
</tbody>
</table>
So when choosing a Hash algorithm ....

• What we know
  – Length of Key
  – Type of key (numeric, ascii etc)

• What we my not know
  – Number or rows
  – Distribution of Key

These can all effect the probability of a collision!
Ideal Hash algorithm

- Easy to calculate (processing power)
- Low collisions
- Good distribution to target space, irrespective of distribution within source space
Example results – number of keys

• Relatively cheap to calculate hash
  – Well distributed data
Some results

- Same algorithm (non-ideal key distribution)
Looking at some alternatives
So where does this leave us?

• If we don’t know much – should use a highly efficient Hash
  – I recommend the Fowler-Noll-Vo Hash Function (FNV)

• But, if we know
  – Well distributed key
  – Small number of keys
  – V. Low Density

….. we may consider a cheaper to calculate Hash
Specific Hashes

• With some knowledge of a key, we can create some very effective (high performance, low collisions) Hashes.

• E.g. Canadian Postcodes e.g K1A 3M2
  – Letters D, F, I, O, Q or U are not used
  – Letters W, or Z are not used in first position
  – 6 bytes have 300,000,000,000 combinations
  – Can limit to 7,400,000 with knowledge of distribution
  – Only about 830,000 in use
• There are standard Hash algorithms out there
  – Linux 32 and 64 bit Hash algorithm
    \[ F(\text{byte}[\ ] = (\sum_{k=0}^{n-1} p^k \cdot \text{byte}[k]) \mod (2^b) \]
  where \( p \) is a prime (31), and
  \( n \) is the number of bytes
  \( b \) is 32 or 64
  – Maps any string to either 32 or 64bit number

• Doesn’t behave well with high densities

• However, combinations \( 2^{32} \) or \( 2^{64} \) - so low densities should be guaranteed
References

• Use of Primes

• Fowler-Noll-Vo Hash function (FNV)
  – http://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function