Hashing

เป็นวิธีการจัดเก็บข้อมูล (ซึ่งมักเป็นดรรชนีหรือ key) เพื่อการสืบค้นที่รวดเร็ว ซึ่งจัดว่ารวดเร็วที่สุดในจำนวนโครงสร้างข้อมูลที่กล่าวมาทั้งหมด เพราะใช้เวลาคงที่ในการสืบค้นตำแหน่งของข้อมูล หรือ O(1) (จะเจอหรือไม่ขึ้นอยู่กับวิธีการจัดเก็บข้อมูล) แต่ก็ต้องสิ้นเปลืองหน่วยความจำมากในอันที่จะได้มาซึ่งความรวดเร็วของการประมวลผล ด้วยสาเหตุและความจำเป็นซึ่งจะกล่าวต่อไป

คุณลักษณะที่สำคัญๆ คือ

- ทำงานบน fixed size array (มักเรียกว่า table)
- ค่าของ key จะถูกแปลง (mapped) ด้วย hashing function เพื่อกำหนดช่องที่แน่นอนในการเก็บ
- ขนาดของ table นิยมใช้เป็นเลขจำเพาะ
- มีระบบจัดการ collision resolution เพื่อรองรับวิธี hashing แบบต่างๆ

Collision เกิดจากการซ้อนทับของข้อมูล (key) ที่ผ่านกระบวนการ hashing ลงสู่ตำแหน่งเดียวกันใน table เช่น สมมุติให้ hash function เป็น modulo (%) ของ table size (ขนาด 17) ถ้าข้อมูลเป็น 25 และ 59 จะถูก mapped ลงในตำแหน่งเดียวกัน คือช่องที่ 8 ของ table ข้อมูลที่อ่านเข้ามาทีหลังจะซ้อนทับข้อมูลแรก วิธีแก้ปัญหาดังกล่าว (เรียกว่า collision resolution) คือ หาตำแหน่งใหม่ให้แก่ข้อมูลตัวหลัง เพื่อไม่ให้ซ้อนทับตัวแรก ในที่นี้จะกล่าวถึงวิธีง่ายๆ ที่นิยมใช้กัน 2 วิธี คือ separate chaining และ open addressing

ก่อนที่จะพูดถึง collision resolution ทั้งสองวิธี จะขอกล่าวถึงปัจจัยที่ใช้ในการพิจารณาประสิทธิภาพการทำงานก่อน ปัจจัยที่ว่าคือ load factor (λ) ซึ่งเป็นอัตราส่วนระหว่างจำนวนข้อมูลใน table ต่อขนาดของ table หรืออีกนัยหนึ่งคือ ความถี่ (ปริมาณ) ของข้อมูลใน table เช่น ถ้าค่าของ λ เป็น 0.8 หมายถึง table มีข้อมูลหนาแน่นถึง 80% เมื่อเทียบกับขนาดของตาราง λ ยิ่งสูง การสืบค้นก็จะยิ่งช้าลง จึงไม่ต้องสงสัยเลยว่า load factor เป็นตัวชี้ที่สำคัญในการพิจารณาการปรับวิธีการทำงานหรือกระบวนการแก้ปัญหา collision ข้างต้น เพื่อประสิทธิภาพของการทำงาน (ความเร็ว) สูงสุด

Separate chaining

วิธีนี้เป็นการสร้าง singly linked list ที่มีตำแหน่งหลักใน hash table เป็นฐาน (หัว) ของ chain (หนังสือบางเล่มเรียกว่า bucket) แต่ละตำแหน่งหลักจึงมี list ของตนเองที่เก็บข้อมูลที่ซ้ำกันอันเป็นผลพวงมาจากการ mapped หรือ hashed เข้าสู่ตำแหน่งหลักเดียวกัน จากรูปที่ 5.6 หน้า

153 ถ้า 25 และ 59 ในตัวอย่างข้างต้น mapped ลงในตารางของรูป ก็จะเกิด chain ในตำแหน่งที่ 8 เก็บค่าของ 25 และ 59 ตามลำดับ (สมมุติว่าอ่าน 25 ก่อน 59)

จากตัวอย่างดังกล่าว สามารถสรุปได้ดังนี้

- ความยาวเฉลี่ยของ list = λ
- unsuccessful search = λ (เพราะต้อง traverse ถึง λ links กว่าจะรู้ว่าไม่เจอ)
- successful search = $1 + \lambda/2$ (1 ครั้งเป็นอย่างน้อยและเฉลี่ย $\lambda/2$ ครั้ง)

Open addressing

วิธีแรกมีข้อด้อยในส่วนของการใช้ linked list ซึ่งต้องใช้เวลาในการ traverse ไปตาม chain ถึง O (n) ทั้งนี้ยังไม่รวม overhead ที่เกิดจาก dynamic allocation ระหว่างการสร้าง linked list ให้เป็น chain จากตำแหน่งหลักและตำแหน่งต่อๆ ไป วิธีหลังนี้ จึงใช้ตำแหน่งใน table แทน chain ในการเก็บข้อมูลเพื่อความเร็วในการจัดเก็บและสืบค้น ค่าของ λ จึงเป็นตัวบอกความถี่ของข้อมูลในตารางอย่างแท้จริง และมีค่าไม่เกิน 1.0 ในขณะที่วิธี separate chaining อาจเกิดกรณีที่ค่า λ มากกว่า 1.0 ถ้า chain หรือ linked list มีความยาวมากเป็นพิเศษ

ในที่นี้จะกล่าวถึงการหาตำแหน่งตามวิธี open addressing สามรูปแบบ คือ linear probing, quadratic probing และ double hashing

Linear probing ประยุกต์ฟังชันเชิงเส้นในการหาตำแหน่งใหม่ เช่น การเลือก F(i) = i เป็นการเสาะหาตำแหน่งถัดจากที่เกิด collision (ดูรูป 5.11 หน้า 158) เพื่อใช้เป็นที่เก็บข้อมูล หากตำแหน่งถัดไปถูกจับจองโดยข้อมูลที่มีอยู่ก่อน ก็จะขยับไปตำแหน่งถัดไปอีก 1 หนึ่งช่อง (ในลักษณะการขยับไปเรื่อยๆ แบบเชิงเส้น) เช่นนี้เรื่อยไปจนกว่าจะเจอตำแหน่งว่าง จึงจะเก็บข้อมูลลงในช่องดังกล่าว จะเห็นว่าวิธีนี้ล่าช้าทั้งในระหว่างการสืบค้นเพื่อเพิ่มข้อมูล (insert) หรือลดข้อมูล (delete) ซึ่งจะได้ว่า

Unsuccessful search[1] = $1/2 \left[(1 + 1/(1 - \lambda)^2) \right]$

Successful search = $1/2 \left[1 + 1/(1 - \lambda) \right]$

Quadratic probing เพื่อแก้ปัญหาความล่าช้าของวิธี linear probing จึงมีการเลือก collision resolution function เพื่อลดความล่าช้าในการหาตำแหน่งใหม่หลังจากที่เกิด collision ฟังชันที่ใช้ก็อยู่ในรูปแบบเดียวกับ linear probing คือ $F(i) = i^2$ โดย i แทนครั้งที่หา (probe)

[1] ต้องการ 1 ครั้งสำหรับการไปที่ base location และ $(1- \lambda)$ สำหรับ unsuccessful search และ insert จึงได้กำลังสอง

ตำแหน่งใหม่ เช่น สมมุติว่าข้อมูลเป็น 31 % 11 (ขนาดของ table) ได้ตำแหน่ง 9 ซึ่งมีข้อมูลอื่นเก็บอยู่ก่อนแล้ว การหาตำแหน่งใหม่ครั้งแรก สามารถคำนวณจาก

ตำแหน่งใหม่(1) = $h(31) + 1^2$ = 31%11 + 1 = 10%11 = 10

หากมีการชนกันอีก ก็จะหาค่าตำแหน่งใหม่ (ครั้งที่ 2 และ 3) จาก

ตำแหน่งใหม่(2) = $h(31) + 2^2$ = 31%11 + 4 = 13%11 = 2

ตำแหน่งใหม่(3) = $h(31) + 3^2$ = 31%11 + 9 = 18%11 = 7

เช่นนี้เรื่อยไป จนกว่าจะเจอตำแหน่งว่างที่จะบรรจุข้อมูลลงไป จะเห็นว่า วิธีนี้ใช้หลักกระโดดทีละหลายๆ ตำแหน่ง เริ่มจาก 1, 4, 9 เรื่อยไปโดยหวังว่าจะไม่เกิดการชนเช่นวิธี linear probing ซึ่งขยับทีละตำแหน่ง ทำให้มีโอกาส (ความน่าจะเป็น) ของการชนกับข้อมูลอื่นสูง แต่ในทางปฏิบัติกลับไม่เกิดผลดีเท่าที่ควร เพราะการก้าวกระโดดจะข้ามตำแหน่งในระยะคงที่เสมอ คือ 1, 4, 9 ตำแหน่งเช่นนี้เรื่อยไป วิธีแก้คือพยายามให้ค่า λ ต่ำเพื่อช่วยเพิ่มโอกาสของการกระโดดเจอตำแหน่งว่างได้โดยง่าย (ทฤษฎี 5.1 หน้า 160)

ในการ implement จริง ฟังชัน F(i) ทำงานช้า[2] จึงมีการหาวิธีลัดเพื่อเพิ่มประสิทธิภาพของการคำนวณ hash function ที่รวดเร็วและสิ้นเปลืองน้อย จากขั้นตอนข้างต้น จะเห็นว่า ค่าของ collision resolution function หรือ F(i) ที่คำนวณได้คือ 1, 4, 9, 16, 25 ตามลำดับ วิธีลัดคือเลี่ยงการยกกำลัง แต่อาศัยการคูณและบวกจากความสัมพันธ์

$$F(i) = F(i-1) + 2*i - 1$$

โดยที่ F(0) = 0 เมื่อแทนค่าในสมการความสัมพันธ์ข้างต้น จะได้ว่า

$$F(1) = F(0) + 2*1 - 1 = 1$$

$$F(2) = F(1) + 2*2 - 1 = 4$$

$$F(3) = F(2) + 2*3 - 1 = 9$$

ซึ่งการคูณจำนวนใดๆ ด้วยสองในระดับฮาร์ดแวร์คือการ shift bit ของจำนวนที่ต้องการคูณสองไปทางซ้าย 1 ครั้ง ทำให้การคำนวณค่าของ F(i) สามารถกระทำได้อย่างรวดเร็ว เพราะ operations ที่ใช้ก็เพียงค่าเก่าของ F(i) (คือ F(i-1)) การ shift bit และการลบหนึ่ง[3] เท่านั้น

Double hashing เป็นความพยายามแก้ปัญหาของ linear และ quadratic probing ที่เสาะหาในตำแหน่งที่ซ้ำๆ อันเป็นสาเหตุของการชนครั้งแล้วครั้งเล่า โดยอาศัย randomness ของ hash function ในการหาตำแหน่งใหม่หลังจากเกิดการชนกันขึ้น collision resolution function ก็

---

[2] คำว่าช้าในที่นี้หมายถึงสิ้นเปลืองทรัพยากรที่ใช้ในการคำนวณเลขยกกำลังสองมาก ซึ่งเวลาก็เป็นทรัพยากรหลักอันหนึ่ง

[3] หรือบวกด้วย 2's complement ของ -1 นั่นเอง

สร้างในรูปแบบตรงไปตรงมา ที่นิยมกันมาก คือ $F(i) = i * h_2(x)$ ซึ่ง $h_2(x)$ เป็นฟังชันใหม่ที่ไม่ซ้ำกับ hash function เดิม ดังตัวอย่าง

$$h_2(x) = R - (x \bmod R)$$

R เป็น prime ที่น้อยกว่า TableSize (ดูตัวอย่างในรูป 5.18 หน้า 165) หากขนาดของ TableSize เป็น prime ที่เหมาะสมแล้ว double hashing จะทำงานได้ดี

Rehashing

ทั้งสามวิธีของ open addressing ที่กล่าวข้างต้นล้วนมีปัญหาการชนกันครั้งแล้วครั้งเล่า โดยเฉพาะวิธี quadratic probing ซึ่งอาจจะเกิดปรากฏการณ์ "เด้ง" ไปมาระหว่าง probe ตำแหน่งใหม่ (เริ่มจาก 1, 4, 9 ฯลฯ) อันเนื่องมาจาก λ มีค่าสูง ทำให้การ insert ข้อมูลใหม่เข้าในตารางช้าลง จึงมีความจำเป็นที่จะต้อง "ขยาย" ตารางให้ใหญ่ขึ้นเพื่อลดการชน และเป็นการลดค่า λ ลงด้วย นิยมขยายตารางให้ใหญ่เป็นสองเท่าจากเดิม จากตัวอย่างรูป 5.19-5.21 หน้า 166-167 จะเห็นว่า ด้วยข้อมูล 13, 15, 6, 24, 23 แม้ว่าจะมีการชนกันเพียงครั้งเดียวสำหรับตารางขนาด 7 ช่อง เมื่อเทียบกับตารางขนาด 17 ช่องที่ชนกันถึงสองครั้งขณะทำการ rehash โดย 23 ชนกับ 6 ในตำแหน่งที่ 6 และ 24 ชน 23 ในตำแหน่งที่ 7 ก่อนที่จะเจอตำแหน่งว่างในช่องที่ 8 แต่ตารางหลังสามารถรองรับข้อมูลที่เพิ่มขึ้น (เช่น 14) ด้วยจำนวนครั้งของการชนและ probe ที่น้อยกว่าตารางแรกมากเพราะค่า λ น้อย

คำถามที่ตามมาคือ เราใช้เกณฑ์อะไรในการตัดสินใจทำ rehashing และควรทำเมื่อใด แนวทางที่เป็นไปได้ คือ rehash เมื่อ

1. $\lambda >= 0.5$ หรือ
2. insert ข้อมูลไม่ได้ หรือ
3. λ = ค่า load factor ที่กำหนด (เช่น 0.7)

โดยปกติจะ rehash อัตโนมัติภายในโปรแกรมเอง แต่กระบวนการ rehashing สิ้นเปลืองทรัพยากรมาก อีกทั้งยังกระทบต่อการทำงานของการใช้ตารางอีกด้วย จึงมักไม่ rehash บ่อยนักถ้าไม่จำเป็น

Extendible hashing

เป็นแนวคิดที่จะช่วยแบ่งเบาภาระของการ rehashing ข้างต้นส่วนหนึ่ง แต่เหตุผลหลักคือขนาดของตารางเมื่อเทียบกับหน่วยความจำที่มี ถ้าขนาดของตารางใหญ่เกินกว่าที่จะบรรจุลงในหน่วยความจำปฐมภูมิ หรือเกิดการชนกันบ่อยมาก ทำให้ต้อง access ดิกส์หลายครั้งกว่าที่จะได้ข้อมูลที่ต้องการ ซึ่งเสียเวลามาก

แนวคิดของ extendible hashing อาศัยหลักการสร้างดรรชนี (หรือ directory) ที่ชี้ไปยังกลุ่มข้อมูล (หรือเทียบได้กับ leaf ของ B-Tree) ชนิดเดียวกัน เพื่อลดปริมาณ disk access ให้น้อยลง จากรูป 5.23 หน้า 169 ข้อมูลในแต่ละกลุ่มมีส่วนที่คล้ายกันคือ สองบิทแรกเหมือนกัน ดังนั้นในการสืบค้นข้อมูลใดๆ จะเริ่มโดยพิจารณาสองบิทแรกก่อนเพื่อทำการดึงเฉพาะตารางที่เกี่ยวข้องมาค้นหาเท่านั้น ไม่ probe ทั้งหมดเหมือนดังตัวอย่างที่กล่าวข้างต้น

โครงสร้างก็จะใช้ B-Tree ที่มี root node ทำหน้าที่เป็นดรรชนีมีขนาด (order) 2^D โดยที่ D เป็นจำนวนบิทของดรรชนีที่ใช้ในการจำแนกข้อมูล จากตัวอย่าง D = 2 ดรรชนีจึงมีขนาด 2^2 หรือ 4 ช่อง

ความได้เปรียบอย่างหนึ่งของ extendible hashing คือการขยายตาราง สามารถทำในเฉพาะกลุ่มข้อมูลที่ต้องการเท่านั้น แทนที่จะขยายทั้งตารางใหญ่ดังเช่นใน rehashing จากรูป 5.24 หน้า 170 จะเห็นว่า เฉพาะกลุ่มข้อมูลหรือ leaf ที่สามจากซ้ายสุดเท่านั้นที่มีการแตกออกเป็นสองตาราง (เทียบได้กับการขยายตารางในหัวข้อที่แล้ว) พร้อมทั้งแยกข้อมูลตามกลุ่ม และ update ค่าของดรรชนีใน root node เป็นสามบิท แต่ leaf อื่นยังคงใช้ดรรชนีเพียงสองบิทก็เพียงพอในการสืบค้น ในทำนองเดียวกัน การแทรก 000000 ทำให้ leaf ซ้ายสุดต้องถูกแตกออกเป็นสองตารางย่อยอีก และใช้สามบิทในการแยกแยะข้อมูล

ยังมีอีกกรณีที่อาจเกิดใน extendible hashing คือ leaf หรือตารางย่อยใดอาจมีจำนวนข้อมูลมากเป็นพิเศษ เช่น leaf อื่นมีไม่เกิน 4 ดังรูป แต่ leaf หนึ่งมีถึง 9 (สมมุติ) เราอาจประยุกต์วิธี rehashing ข้างต้นเพื่อรองรับปริมาณข้อมูล “ชนิดเดียวกัน” จำนวนมากๆ (กรณีนี้การแตก root node อาจจะไม่ช่วย เพราะเหตุใด)

ความลึกของ B-Tree ที่มีโครงสร้างย่อยในระดับ leaf เป็น hash table นั้นยังคงเหมือนกับ B-Tree ทุกประการ นั่นคือ $O(ceil[\log_{ceil[m/2]} n])$ หากเรามอง B-Tree เป็น ADT ที่สามารถรองรับโครงสร้างข้อมูลชนิดใดๆ ก็ได้ แม้กระทั่ง B-Tree ด้วยกันเอง ก็จะเห็นว่าหลักการ extendible hashing เป็นการรวมโครงสร้างข้อมูลมากกว่าหนึ่งชนิดให้เป็นโครงสร้างข้อมูลใหม่ที่มีความยืดหยุ่นและประสิทธิผลสูง

ตัวอย่างข้างต้นใช้ข้อมูลตัวเลขแบบง่ายๆ เพื่อแสดงให้เห็นถึงศักยภาพของโครงสร้างข้อมูลแบบ hash table แต่ในงานประยุกต์จริงมักเป็น string มากกว่าตัวเลข ซึ่งก็ใช้หลักการเดียวกัน เพียงแต่รายละเอียดของการประยุกต์อาจแตกต่างกันไปตามความเหมาะสมกับลักษณะและความสัมพันธ์ระหว่าง field ย่อยของข้อมูล เช่น อาจมีการสร้าง primary key และ secondary key (foreign key) สำหรับอ้างอิง (cross-reference) เพื่อรักษาไว้ซึ่งความถูกต้องของข้อมูล เป็นต้น

Application

มักจะใช้ในการทำ table-lookup ในระดับฮาร์ดแวร์ อันเป็นส่วนหนึ่งของ associative memory สำหรับ caching ในหน่วยประมวลผลกลาง หรือ translation lookaside buffer (TLB) ที่สนับสนุนการแปลง address ของ page table entry (PTE) ใน virtual memory management ส่วน extendible hashing เป็นโครงสร้างหลักในการทำดรรชนีระบบฐานข้อมูลขนาดใหญ่