03. شجرة B واستعادة البيانات بعد الأعطال

(B-Tree & Crash Recovery)

3.1 شجرة B كشجرة n-أرية متوازنة

شجرة متوازنة الارتفاع (Height-Balanced Tree)

العديد من الأشجار الثنائية العملية، مثل شجرة AVL أو شجرة RB (الحمراء-السوداء)، تُسمى أشجار متوازنة الارتفاع، بمعنى أن ارتفاع الشجرة (من الجذر إلى الأوراق) محدود بـ O(log N)، لذلك البحث يكون O(log N).

شجرة B أيضاً متوازنة الارتفاع؛ الارتفاع هو نفسه لجميع العقد الورقية.

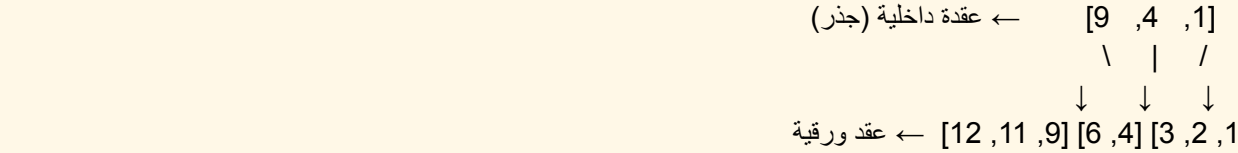
شرح المصطلحات:

- **Height (الارتفاع)**: عدد المستويات من الجذر إلى الأوراق
- **Balanced (متوازن)**: جميع المسارات لها نفس الطول
- **AVL Tree**: شجرة ثنائية بحث متوازنة تلقائياً
- **RB Tree (Red-Black Tree)**: شجرة حمراء-سوداء متوازنة

تعميم الأشجار الثنائية (Generalizing Binary Trees)

الأشجار n-أرية يمكن تعميمها من الأشجار الثنائية (والعكس). مثال على ذلك شجرة 2-3-4، وهي شجرة B حيث كل عقدة يمكن أن يكون لها 2 أو 3 أو 4 أطفال. شجرة 2-3-4 مكافئة لشجرة RB.

تصور شجرة B+tree من مستويين لتسلسل مرتب [1, 2, 3, 4, 6, 9, 11, 12]:



شرح: في شجرة B+tree، فقط العقد الورقية تحتوي على القيم، والمفاتيح مكررة في العقد الداخلية للإشارة إلى نطاق المفاتيح في الشجرة الفرعية.

في هذا المثال، العقدة [1, 4, 9] تشير إلى أن أشجارها الفرعية الثلاثة موجودة ضمن الفترات:

- **[1, 4)**: من 1 إلى 4 (لا تشمل 4)
- **[4, 9)**: من 4 إلى 9 (لا تشمل 9)

- **[9, +∞)**: من 9 إلى ما لا نهاية

ومع ذلك، نحتاج فقط إلى مفتاحين لثلاث فترات، لذلك يمكن حذف المفتاح الأول (1) وتصبح الفترات الثلاث:

- **(-∞, 4)**: من سالب ما لا نهاية إلى 4
- **[4, 9)**: من 4 إلى 9
- **[9, +∞)**: من 9 إلى ما لا نهاية

3.2 شجرة B كمصفوفات متداخلة

مصفوفات متداخلة من مستويين (Two-Level Nested Arrays)

بدون معرفة تفاصيل شجرة RB أو شجرة 2-3-4، يمكن فهم شجرة B من المصفوفات المرتبة.

المشكلة في المصفوفات المرتبة: التحديث O(N) مكلف جداً!

الحل: إذا قسمنا المصفوفة إلى m مصفوفة أصغر غير متداخلة، يصبح التحديث O(N/m).

لكن يجب علينا معرفة أي مصفوفة صغيرة نحدث/نستعلم أولاً. لذلك نحتاج مصفوفة مرتبة أخرى من المراجع إلى المصفوفات الأصغر - هذه هي العقد الداخلية في B+tree.

[[1,2,3], [4,6], [9,11,12]]

التكلفة:

- البحث لا يزال O(log N) مع بحثين ثنائيين
- إذا اخترنا m = √N، يصبح التحديث O(√N)

مثال عملي:

لنفرض لدينا 100 عنصر:
- مصفوفة واحدة: التحديث O(100)
- 10 مصفوفات × 10 عناصر: التحديث O(10)

مستويات متعددة من المصفوفات المتداخلة

O(√N) غير مقبول لقواعد البيانات، لكن إذا أضفنا مستويات أكثر بتقسيم المصفوفات أكثر، تنخفض التكلفة أكثر.

لنفترض أننا نستمر في تقسيم المستويات حتى تصبح جميع المصفوفات لا تزيد عن ثابت s:

- ننتهي بـ log(N/s) مستويات
- تكلفة البحث: O(log N) = O(log(N/s) + log(s))

للإدراج والحذف: بعد إيجاد العقدة الورقية، تحديث العقدة الورقية يكون ثابتاً O(s) في معظم الأحيان.

المشكلة المتبقية: الحفاظ على الثوابت التي تقول:

- العقد ليست أكبر من s
- العقد ليست فارغة

3.3 صيانة شجرة B+tree

3 ثوابت يجب الحفاظ عليها عند تحديث B+tree:

1. **نفس الارتفاع لجميع العقد الورقية**
2. **حجم العقدة محدود بثابت**
3. **العقدة ليست فارغة**

تنمية شجرة B بتقسيم العقد (Growing by Splitting)

الثابت الثاني يُنتهك بالإدراج في عقدة ورقية، ويُستعاد بتقسيم العقدة إلى عقد أصغر.

مثال على التقسيم:

قبل التقسيم:

بعد التقسيم:

بعد تقسيم عقدة ورقية، تحصل العقدة الأم على فرع جديد، مما قد يتجاوز أيضاً حد الحجم، لذلك قد تحتاج هي أيضاً إلى التقسيم.

انتشار التقسيم إلى الجذر:

قبل:

L1 L2 L6

بعد (زيادة الارتفاع):
new_root
\ /
N1 N2
\ | | /
L1 L3 L4 L6

تقسيم العقدة يمكن أن ينتشر إلى عقدة الجذر، مما يزيد الارتفاع بمقدار 1.

هذا يحفظ الثابت الأول، لأن جميع الأوراق تكتسب ارتفاعاً بمقدار 1 في نفس الوقت.

تقليص شجرة B بدمج العقد (Shrinking by Merging)

الحذف قد يؤدي إلى عقد فارغة. الثابت الثالث يُستعاد بدمج العقد الفارغة في عقدة شقيقة.

الدمج عكس التقسيم:

- يمكن أن ينتشر أيضاً إلى عقدة الجذر
- يمكن أن ينخفض ارتفاع الشجرة

تحسين: يمكن القيام بالدمج مبكراً لتقليل المساحة المهدرة - يمكنك دمج عقدة غير فارغة عندما يصل حجمها إلى حد أدنى.

3.4 شجرة B على القرص

يمكنك الآن كتابة كود شجرة B في الذاكرة باستخدام هذه المبادئ. لكن شجرة B على القرص تتطلب اعتبارات إضافية.

التخصيص القائم على الكتل (Block-Based Allocation)

التفصيل المفقود: كيف نحدد حجم العقدة؟

في الذاكرة:

- يمكنك تحديد العدد الأقصى للمفاتيح في عقدة
- حجم العقدة بالبايتات ليس مصدر قلق
- يمكنك تخصيص أي عدد من البايتات حسب الحاجة

على القرص:

- لا يوجد malloc/free أو garbage collector
- تخصيص المساحة وإعادة الاستخدام متروك لنا بالكامل

الحل: إعادة استخدام المساحة يمكن أن تتم بقائمة حرة (free list) إذا كانت جميع التخصيصات بنفس الحجم. لذلك، جميع عقد شجرة B لها نفس الحجم.

شجرة B بـ Copy-on-Write للتحديثات الآمنة

رأينا 3 طرق مقاومة للأعطال لتحديث بيانات القرص:

1. **إعادة تسمية الملفات**
2. **السجلات (Logs)**
3. **أشجار LSM**

الدرس: لا تدمر أي بيانات قديمة أثناء التحديث!

تطبيق الفكرة على الأشجار: اصنع نسخة من العقدة وعدّل النسخة بدلاً من الأصل.

كيف يعمل Copy-on-Write؟

الشجرة الأصلية:

```
      d
     / \
    b   e
   / \
  a   c
```

بعد تحديث c:

```
      D*        ← جذر جديد
     / \
    B*  e       ← عقد منسوخة (B*, D*)
   / \
  a*  C*        ← الورقة المحدثة
```

الأحرف الكبيرة (*) = عقد منسوخة
الأحرف الصغيرة = عقد مشتركة (لم تتغير)

العملية:

1. الإدراج أو الحذف يبدأ عند عقدة ورقية
2. بعد عمل نسخة مع التعديل، يجب تحديث العقدة الأم للإشارة إلى العقدة الجديدة
3. هذا أيضاً يتم على نسخة من العقدة الأم
4. النسخ ينتشر إلى عقدة الجذر، مما ينتج جذر شجرة جديد

النتيجة:

- الشجرة الأصلية تبقى سليمة ويمكن الوصول إليها من الجذر القديم
- الجذر الجديد، مع النسخ المحدثة طوال الطريق إلى الورقة، يشارك جميع العقد الأخرى مع الشجرة الأصلية

المصطلحات البديلة:

- **Immutable (غير قابل للتغيير)**
- **Append-only (إضافة فقط)** - ليس حرفياً
- **Persistent (مستمر)** - غير متعلق بالمتانة

⚠️ **تحذير:** مصطلحات قواعد البيانات ليس لها معاني متسقة!

مشكلتان متبقيتان لـ Copy-on-Write B-tree:

1. **كيف نجد جذر الشجرة؟** يتغير بعد كل تحديث!
    - مشكلة الأمان من الأعطال تُختزل إلى تحديث مؤشر واحد (سنحلها لاحقاً)
2. **كيف نعيد استخدام العقد من النسخ القديمة؟**
    - هذه وظيفة القائمة الحرة (free list)

مزايا Copy-on-Write B-tree

1. عزل اللقطات (Snapshot Isolation) مجاناً:

- المعاملة تبدأ بنسخة من الشجرة
- لن ترى التغييرات من النسخ الأخرى

2. استعادة البيانات بدون جهد:

- فقط استخدم آخر نسخة قديمة

3. نموذج التزامن متعدد القراء-كاتب واحد:

- القراء لا يحجبون الكاتب

مثال عملي:

النسخة 1: [أحمد:100, سارة:200]
النسخة 2: [أحمد:100, سارة:200, علي:300] ← إضافة علي
النسخة 3: [أحمد:150, سارة:200, علي:300] ← تحديث أحمد

القارئ 1 يرى النسخة 1
القارئ 2 يرى النسخة 2
الكاتب يعمل على النسخة 3

البديل: التحديث في المكان مع الكتابة المزدوجة (Double-Write)

بينما استعادة البيانات واضحة في هياكل Copy-on-Write، قد تكون غير مرغوب فيها بسبب **تضخم الكتابة المرتفع**.

المشكلة:

- كل تحديث ينسخ المسار الكامل O(log N)
- معظم التحديثات في المكان تلمس عقدة ورقية واحدة فقط

الحل: Double-Write

من الممكن القيام بتحديثات في المكان مع استعادة البيانات بدون Copy-on-Write:

الخطوات:

1 احفظ نسخة من جميع العقد المحدثة في مكان ما
| a=1 b=2 | → نسخة: | a=2 b=4 |

2 fsync النسخ المحفوظة
(يمكن الرد على العميل عند هذه النقطة)

3 حدّث هيكل البيانات فعلياً في المكان

4 fsync التحديثات

ماذا بعد العطل؟

الحالة بعد العطل:
| a=2 b=4 | + | ??????? |
البيانات (سيئة) النسخة (جيدة)

الاستعادة: طبّق النسخة المحفوظة
| a=2 b=4 | + | a=2 b=4 |
البيانات (جديدة) عديمة الفائدة الآن

بعد العطل، قد يكون هيكل البيانات محدثاً جزئياً، لكننا لا نعرف حقاً. ما نفعله هو تطبيق النسخ المحفوظة بشكل أعمى، بحيث ينتهي هيكل البيانات بالحالة المحدثة، بغض النظر عن الحالة الحالية.

المصطلح: النسخ المحدثة المحفوظة تُسمى **double-write** في مصطلحات MySQL.

ماذا لو كانت الكتابة المزدوجة تالفة؟

يتم التعامل معها بنفس طريقة السجلات: **Checksum (المجموع الاختباري)**

- **إذا اكتشف Checksum كتابة مزدوجة سيئة:** تجاهلها
  - إنها قبل أول fsync، لذلك البيانات الرئيسية في حالة جيدة وقديمة

- **إذا كانت الكتابة المزدوجة جيدة:** تطبيقها سينتج دائماً بيانات رئيسية جيدة

التسجيل الفيزيائي (Physical Logging)

بعض قواعد البيانات تخزن فعلياً الكتابات المزدوجة في السجلات، يُسمى **physical logging**.

نوعان من التسجيل:

1. **Logical Logging (تسجيل منطقي):**
   - يصف عمليات عالية المستوى مثل "إدراج مفتاح"
   - يمكن تطبيقه فقط عندما تكون قاعدة البيانات في حالة جيدة
2. **Physical Logging (تسجيل فيزيائي):**
   - تحديثات صفحات القرص منخفضة المستوى
   - مفيد للاستعادة لأنه يعمل مع أي حالة

مبدأ استعادة البيانات (Crash Recovery Principle)

مقارنة Double-Write مع Copy-on-Write:

Double-Write:

- يجعل التحديثات **Idempotent (متكافئة)**
- قاعدة البيانات يمكنها إعادة محاولة التحديث بتطبيق النسخ المحفوظة
- تطبيق العملية مرتين = تطبيقها مرة واحدة

Copy-on-Write:

- يبدل كل شيء للنسخة الجديدة **Atomically (ذرياً)**
- الكل أو لا شيء

الأفكار الأساسية:

Double-Write:

- يضمن معلومات كافية لإنتاج **النسخة الجديدة**

Copy-on-Write:

- يضمن الحفاظ على **النسخة القديمة**

الطريقة الثالثة:

ماذا لو حفظنا العقد الأصلية بدلاً من العقد المحدثة مع Double-Write؟

- تستعيد إلى النسخة القديمة مثل Copy-on-Write
- يمكننا دمج الطرق الثلاث في فكرة واحدة:

المبدأ الموحد:

هناك معلومات كافية للحصول على **الحالة القديمة أو الحالة الجديدة** في أي نقطة

ملاحظة: بعض النسخ مطلوب دائماً، لذلك العقد الأكبر أبطأ في التحديث.

اختيارنا: سنستخدم Copy-on-Write لأنه أبسط، لكن يمكنك الانحراف هنا.

3.5 ما تعلمناه

مبادئ B+tree:

✅ **شجرة n-أرية:** حجم العقدة محدود بثابت

✅ **نفس الارتفاع لجميع الأوراق:** شجرة متوازنة

✅ **التقسيم والدمج:** للإدراج والحذف

هياكل البيانات القائمة على القرص:

✅ **Copy-on-Write:** لا تدمر البيانات القديمة

✅ **Double-Write:** للاستعادة من الأعطال

🎯 خطوات الترميز

يمكننا البدء في الترميز الآن! 3 خطوات لإنشاء KV مستمر بناءً على B+tree:

1. **ترميز هيكل بيانات B+tree**
2. **نقل B+tree إلى القرص**
3. **إضافة قائمة حرة (Free List)**

📊 ملخص المقارنة

الميزة	Copy-on-Write	Double-Write
الفكرة	حفظ النسخة القديمة	حفظ النسخة الجديدة
الاستعادة	استخدم النسخة القديمة	أعد تطبيق التحديثات
تضخم الكتابة	مرتفع O(log N)	منخفض O(1)
البساطة	أبسط ✓	أكثر تعقيداً
Snapshot Isolation	مجاناً ✓	يحتاج عمل إضافي
الاستخدام	SQLite, LMDB	MySQL InnoDB

💡 نصائح عملية

متى تستخدم B+tree؟

- قراءات كثيرة، كتابات قليلة
- حاجة لـ range queries
- استعلامات OLTP

متى تستخدم Copy-on-Write؟

- تحتاج Snapshot Isolation
- استعادة بيانات سهلة
- تزامن متعدد القراء

متى تستخدم Double-Write؟

- تقليل تضخم الكتابة مهم
- أداء الكتابة حرج
- لديك خبرة في إدارة التعقيد