# 02. هياكل بيانات الفهرسة (Indexing Data Structures)

## 2.1 أنواع الاستعلامات (Types of Queries)

قبل مناقشة تحديات قواعد البيانات بالتفصيل، نحتاج لاختيار هيكل بيانات، لأن هيكل البيانات يحدد أنواع الاستعلامات المدعومة. تبدو لغة SQL وكأنها تدعم استعلامات عشوائية، لكن فقط مجموعة صغيرة من الاستعلامات يمكنها الاستفادة من هياكل البيانات، لذلك يركز الكتاب على OLTP.

استعلامات OLTP الحقيقية تنقسم إلى 3 أنواع:

1. **المسح الكامل (Full Scan)**: فحص مجموعة البيانات كاملة بدون فهرس إذا كان الجدول صغيراً
2. **استعلام نقطي (Point Query)**: استعلام الفهرس بمفتاح واحد
3. **استعلام نطاقي (Range Query)**: استعلام نطاق من المفاتيح بترتيب معين

**شرح المصطلحات:**

- **OLTP** (Online Transaction Processing): معالجة المعاملات المباشرة - مثل إضافة طلب شراء أو تحديث حساب
- **OLAP** (Online Analytical Processing): معالجة تحليلية - مثل حساب إجمالي المبيعات الشهرية
- **Index (الفهرس)**: هيكل بيانات مساعد يسرع البحث، مثل فهرس الكتاب

**مثال عملي:**

جدول المنتجات:
ID | الاسم | السعر
1 | لابتوب | 3000
2 | ماوس | 50
3 | شاشة | 800

- استعلام نقطي: "أعطني المنتج رقم 2"
- استعلام نطاقي: "أعطني كل المنتجات بسعر بين 100 و 1000"

الاستعلامات النطاقية تتطلب ترتيب المفاتيح، وتدعم عمليتين:

- **Seek (البحث)**: إيجاد مفتاح البداية
- **Iterate (التكرار)**: زيارة المفتاح السابق/التالي بالترتيب

---

## 2.2 جداول التجزئة (Hash Tables)

جداول التجزئة مناسبة فقط للاستعلامات النقطية (get, set, del)، لكننا لن نهتم بها بسبب عدم وجود ترتيب.

**شرح جداول التجزئة:** جدول التجزئة يحول المفتاح إلى رقم (hash) ويستخدمه كعنوان للتخزين.

**مثال:**

المفتاح "أحمد" ← دالة التجزئة ← 42 ← التخزين في موقع 42
المفتاح "فاطمة" ← دالة التجزئة ← 17 ← التخزين في موقع 17

**التحديات:**

- **كيف ننمّي الجدول؟** عندما يمتلئ الجدول، يجب نقل المفاتيح لجدول أكبر. نقل كل شيء مرة واحدة مكلف O(N)، لذلك يجب إعادة التجزئة تدريجياً

---

## 2.3 المصفوفات المرتبة (Sorted Arrays)

أبسط هيكل بيانات مرتب هو المصفوفة المرتبة. يمكن البحث فيها بتعقيد O(log N) باستخدام البحث الثنائي.

**مثال:**

مصفوفة مرتبة: [5, 12, 23, 45, 67, 89]
البحث عن 45:
- المنتصف: 23 (صغير جداً)
- المنتصف الأيمن: 67 (كبير جداً)
- بين 23 و 67: نجد 45 ✓

**المشكلة:** التحديث O(N) - مكلف جداً! لإضافة رقم 30، يجب تحريك كل الأرقام بعد 23.

**الحلول:**

1. **B+Tree**: تقسيم المصفوفة لمصفوفات صغيرة متداخلة
2. **LSM-Tree**: تخزين التحديثات في مصفوفة صغيرة ثم دمجها مع المصفوفة الرئيسية

---

## 2.4 شجرة B (B-Tree)

شجرة B هي شجرة متوازنة n-أرية (n-ary)، مشابهة للأشجار الثنائية لكن كل عقدة تخزن عدة مفاتيح (حتى n حيث n > 2).

**شرح المصطلحات:**

- **Node (عقدة)**: جزء من الشجرة يحتوي على مفاتيح وروابط
- **Leaf Node (عقدة ورقية)**: عقدة نهائية بدون أطفال
- **Internal Node (عقدة داخلية)**: عقدة تحتوي على روابط لعقد أخرى

**مثال شجرة B:**

```
        [30, 60]          ← عقدة داخلية (مستوى 1)
       /   |   \
[10,20] [40,50] [70,80]   ← عقد ورقية (مستوى 2)
```

## تقليل الوصول العشوائي بأشجار أقصر

القرص يمكنه فقط تنفيذ عدد محدود من عمليات I/O في الثانية (IOPS). كل مستوى في الشجرة = قراءة قرص واحدة.

**مقارنة:**

- شجرة ثنائية: $\log_2(1000) = \sim 10$ مستويات
- شجرة 10-أرية: $\log_{10}(1000) = 3$ مستويات فقط!

**المفاضلات:**

- n كبيرة ← عقد أكبر ← تحديث أبطأ
- n كبيرة ← زمن قراءة أطول

في الممارسة العملية، حجم العقدة = بضع صفحات من نظام التشغيل (4KB لكل صفحة).

## عمليات I/O بوحدة الصفحات

**المصطلحات:**

- **Sector (قطاع)**: 512 بايت على الأقراص القديمة
- **Page (صفحة)**: 4KB وحدة ذاكرة في نظام التشغيل
- **Page Cache**: ذاكرة تخزين مؤقت للقرص

الحد الأدنى لوحدة I/O يعني أن عقد الشجرة يجب أن تُخصص بمضاعفات الوحدة - نصف وحدة مستخدم = نصف I/O مهدر!

## متغير B+Tree

في سياق قواعد البيانات، B-tree تعني B+tree. في B+tree:

- العقد الداخلية لا تخزن القيم
- القيم موجودة فقط في العقد الورقية
- هذا يؤدي لشجرة أقصر لأن العقد الداخلية تملك مساحة أكبر للفروع

**مثال B+Tree:**

```
                [60, 30]          ← بدون قيم، فقط مفاتيح للتوجيه
               /   |   \
[A,20:B] [40:C,50:D] [70:E,80:F:10]   ← القيم هنا فقط
```

**تكلفة المساحة**

الأشجار الثنائية غير عملية بسبب عدد المؤشرات - كل مفتاح له مؤشر قادم من العقدة الأم. في B+tree، عدة مفاتيح في عقدة ورقية تشارك مؤشراً واحداً قادماً.

---

# 2.5 التخزين بنمط السجل (Log-Structured Storage)

### التحديث بالدمج: توزيع التكلفة

المثال الأكثر شيوعاً هو LSM-tree (Log-Structured Merge Tree). الفكرة الرئيسية ليست "log" ولا "tree"، بل "merge" (الدمج)!

**الفكرة الأساسية:**

ملف صغير (تحديثات حديثة) + ملف كبير (بقية البيانات)

```
الكتابات → | تحديثات جديدة | ⇒ | بيانات متراكمة |
             ملف 1              ملف 2
```

عندما يصل الملف الصغير لعتبة معينة، يُدمج مع الملف الكبير.

**مثال عملي:**

الملف الصغير: [زيد:100, أحمد:200]
الملف الكبير: [سارة:50, فاطمة:150, محمد:300]

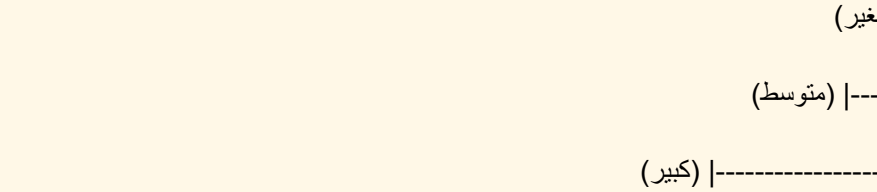
بعد الدمج:
[أحمد:200, زيد:100, سارة:50, فاطمة:150, محمد:300] ← مرتب ومدمج

الدمج تعقيده O(N) لكن يمكن تنفيذه بالتوازي مع القراء والكاتبين.

### تقليل تضخم الكتابة بمستويات متعددة

**تضخم الكتابة (Write Amplification)**: كتابة بيانات أكثر من البيانات الفعلية المطلوبة.

**مثال 3 مستويات:**




المستويات تنمو بشكل أسي (exponential)، والنمو بقوة 2 ينتج أقل تضخم كتابة.

## فهرسة LSM-tree

كل مستوى يحتوي على هياكل فهرسة، يمكن أن تكون ببساطة مصفوفة مرتبة. لكن خيار منطقي هو استخدام B-tree داخل كل مستوى - هذا هو جزء "tree" من LSM-tree.

## استعلامات LSM-tree

المفاتيح يمكن أن تكون في أي مستوى، لذلك للاستعلام:

- **استعلام نقطي**: استخدام Bloom Filters لتقليل عدد المستويات المبحوثة
- **استعلام نطاقي**: دمج النتائج من كل مستوى

**Bloom Filter**: هيكل بيانات احتمالي يخبرك "ربما موجود" أو "بالتأكيد غير موجود".

**المستويات الأحدث لها أولوية** لأنها تحتوي على أحدث نسخ من المفاتيح. المفاتيح المحذوفة تُعلّم بعلامة خاصة (tombstones/شواهد قبور).

عملية الدمج تستعيد المساحة من المفاتيح القديمة أو المحذوفة (تُسمى compaction/الضغط).

## LSM-tree الحقيقية: SSTable, MemTable و Log

**المصطلحات:**

- **SSTable** (Sorted String Table): المستويات مقسمة لملفات متعددة غير متداخلة
- **MemTable**: فهرس في الذاكرة للمستوى الأول
- **Log**: سجل للمستوى الأول

**لماذا SSTable؟**

- الدمج يمكن أن يكون تدريجياً
- يقلل متطلبات المساحة الحرة
- عملية الدمج موزعة عبر الزمن

**لماذا MemTable؟** حتى لو كان السجل صغيراً، نحتاج فهرس مناسب. بيانات السجل مكررة في فهرس بالذاكرة (MemTable)، يمكن أن يكون B-tree أو skiplist، مع ميزة تسريع قراءة التحديثات الحديثة.

---

## 2.6 ملخص هياكل بيانات الفهرسة

هناك خياران رئيسيان:

1. **B+Tree**: شجرة متوازنة، تحديث مباشر في مكانه
2. **LSM-Tree**: دمج متعدد المستويات، تحديث مؤجل

**المقارنة:**

| الميزة | B+Tree | LSM-Tree |
|---|---|---|
| القراءة | سريعة جداً | بطيئة نسبياً (بحث متعدد المستويات) |
| الكتابة | بطيئة (I/O عشوائي) | سريعة (I/O متسلسل) |
| المساحة | كفاءة عالية | تحتاج مساحة للدمج |
| التعقيد | معقدة (تحديث في المكان) | أبسط نسبياً |

LSM-tree تحل العديد من التحديات مثل كيفية تحديث هياكل البيانات على القرص وإعادة استخدام المساحة. بينما تبقى هذه التحديات لـ B+tree.

---

## الملخص العام

**هياكل الفهرسة في قواعد البيانات:**

1. **الهدف**: دعم الاستعلامات النقطية والنطاقية بكفاءة
2. **الخيارات الرئيسية**:
   - جداول التجزئة: سريعة لكن بدون ترتيب ❌
   - المصفوفات المرتبة: بسيطة لكن التحديث مكلف ❌
   - **B+Tree**: شجرة متوازنة، الأفضل للقراءة ✓
   - **LSM-Tree**: دمج متعدد المستويات، الأفضل للكتابة ✓
3. **المفاهيم المهمة**:
   - عمليات I/O بوحدة الصفحات (4KB)
   - تضخم الكتابة وكيفية تقليله

- المفاضلة بين سرعة القراءة والكتابة
- إعادة استخدام المساحة

4. **التطبيقات**:

- (**B+Tree**: MySQL, PostgreSQL (InnoDB
- **LSM-Tree**: Cassandra, RocksDB, LevelDB

الاختيار بين B+Tree و LSM-Tree يعتمد على نمط الاستخدام - هل التطبيق قراءة-مكثفة أم كتابة-مكثفة؟