تعلّم أساسيات قواعد البيانات من خلال بناء قاعدتك الخاصة

ما الذي ستتعلمه؟

تُبنى الأنظمة المعقدة مثل قواعد البيانات على عدد محدود من المبادئ البسيطة:

- **الذرية والاستمرارية**: قاعدة البيانات ليست مجرد ملفات عادية
  - ضمان استمرارية البيانات باستخدام fsync
  - استعادة النظام بعد الأعطال المفاجئة
- **نظام تخزين المفاتيح والقيم القائم على B-tree**
  - بنية بيانات محسّنة للتخزين على القرص
  - إدارة المساحة التخزينية بكفاءة
- **قاعدة بيانات علائقية مبنية على نظام المفاتيح والقيم**
  - آلية ربط الجداول والفهارس بأشجار B-tree
  - لغة استعلام مشابهة لـ SQL مع محلل نحوي ومفسّر
- **إدارة التزامن في المعاملات**

اكتب قاعدة بيانات كاملة في 3000 سطر برمجي فقط

الأمر المذهل أن موضوعاً بهذا التعقيد يمكن تغطيته في 3000 سطر برمجي، وهو حجم أصغر بكثير من أي قاعدة بيانات حقيقية، مع الاحتفاظ بجميع المفاهيم الأساسية. مما يجعله مشروعاً مثالياً لأي شخص يرغب في التعلم خلال وقت فراغه.

عدد الأسطر	المرحلة
366	بنية شجرة B+tree
601	نظام مفاتيح وقيم بالإضافة التدريجية
731	نظام عملي مع إدارة المساحة الحرة
1107	بناء الجداول فوق نظام المفاتيح والقيم
1294	استعلامات النطاق
1438	الفهارس الثانوية
1461	واجهات المعاملات

التعلم بالممارسة العملية وليس بحفظ المصطلحات

قال العالم فاينمان: "ما لا أستطيع بناءه بيدي، لا أفهمه حقاً". لكن هل يكفي قراءة الكتب لتتعلم بناء قواعد البيانات؟ للأسف، المراجع الأكاديمية في هذا المجال مليئة بالمصطلحات المبهمة والمعاني المتضاربة. لذلك التعلم من خلال التطبيق العملي هو الطريق الأمثل.

رغم أن المجال واسع، إلا أن جوهر قواعد البيانات يرتكز على عدد محدود من المبادئ الأساسية.

المبدأ الأول: الاستمرارية والذرية

ما المقصود بقاعدة البيانات؟

في عالم التسويق، يُطلق مسمى "قاعدة بيانات" على أي شيء تقريباً، بما في ذلك جداول Excel وخوادم الذاكرة المؤقتة. لكننا نركز هنا على قواعد البيانات الحقيقية مثل MySQL وPostgres وSQLite. ما الذي يجمع بينها؟

- تحفظ البيانات بشكل دائم على القرص الصلب
- مصممة للعمل مع بيانات أكبر من سعة الذاكرة العشوائية
- مطوّرة من الصفر، وليست مجرد واجهة لقواعد بيانات أخرى

لا يوجد سوى عدد محدود من المشاريع الناضجة التي تحقق هذه المعايير، وجميعها ضخم الحجم. على سبيل المثال، الكود المصدري لـ SQLite3 يُقاس بالميغابايت حتى بعد الضغط. لذلك لن تكون قاعدة بيانات حقيقية هي المكان الأمثل لتعلم المبادئ الأساسية. بدلاً من ذلك، سنبني قاعدة بيانات من الصفر في 3000 سطر فقط.

قاعدة البيانات أكثر من مجرد صيغة تخزين

الحفظ الدائم للبيانات هو المعيار الأهم لأي قاعدة بيانات تقليدية. لهذا السبب تستخدم الهواتف المحمولة SQLite كقاعدة بيانات مدمجة. لكن إن كانت قاعدة البيانات في النهاية مجرد ملف، لماذا لا نستخدم الملفات العادية مباشرة؟

السبب أن مفهوم "الحفظ الدائم" له متطلبات صارمة: في لحظة معينة، يجب أن تُضمن البيانات المضافة للنظام من الضياع حتى لو تعطل الجهاز بشكل مفاجئ بسبب انقطاع الكهرباء أو غيره.

هناك متطلبان رئيسيان: القدرة على النجاة من الأعطال المفاجئة، والقدرة على تأكيد حالة الاستمرارية. بما أن معظم قواعد البيانات تعمل فوق أنظمة الملفات، فإن هذه الأنظمة يجب أن تلبي نفس المتطلبات. لكن الفرق الجوهري أن استخدام الملفات العادية لا يوفر ضمانات الاستمرارية، مما قد يؤدي لفقدان أو تلف البيانات بعد انقطاع الكهرباء، بينما قواعد البيانات توفر هذا الضمان.

جعل الملفات مستمرة يعني أنك بنيت نصف قاعدة بيانات، وهذا ما سنتعلمه.

استدعاء النظام fsync

fsync هو الأمر الذي يضمن حفظ جميع البيانات المكتوبة بشكل دائم على القرص. تستخدمه قواعد البيانات لطلب وتأكيد الاستمرارية؛ حيث لا تعيد قاعدة البيانات رسالة النجاح للعميل إلا بعد تنفيذ fsync بنجاح.

لكن ماذا لو تعطل النظام قبل أو أثناء تنفيذ fsync؟ قد تُفقد آخر البيانات المضافة، لكن المهم ألا تُحفظ بشكل جزئي. هذا المبدأ "إما كل شيء أو لا شيء" يُسمى الذرية.

يجب أن تستطيع قاعدة البيانات الاستعادة من الأعطال إلى حالة منطقية وسليمة، وهذا أصعب بكثير من مجرد استخدام fsync.

المبدأ الثاني: بنى البيانات للفهرسة

التحكم في السرعة والتكلفة من خلال بنى البيانات

تحول قاعدة البيانات الاستعلام إلى نتيجة دون أن يعرف المستخدم الآلية الداخلية. لكن النتيجة ليست كل شيء - السرعة والتكلفة (من حيث الذاكرة والإدخال/الإخراج والمعالجة) مهمة جداً أيضاً. من هنا جاء التمييز بين قواعد البيانات التحليلية (OLAP) والمعاملاتية (OLTP):

- **OLAP (التحليلية)**: تتعامل مع كميات ضخمة من البيانات، وتجري عمليات تجميع ودمج معقدة. الفهرسة فيها محدودة أو معدومة. تعتمد على تخزين البيانات بشكل عمودي. تُستخدم لتنفيذ استعلامات مخصصة ومعقدة لا تتطلب سرعة فورية.
- **OLTP (المعاملاتية)**: تتعامل مع كميات صغيرة من البيانات باستخدام الفهارس. تتميز بالسرعة والتكلفة المنخفضة. تعتمد على بنى B+tree أو LSM-tree. تُستخدم لتنفيذ استعلامات محددة مسبقاً للمستخدمين النهائيين وتتطلب نتائج فورية.

مصطلح "معاملاتية" هنا لا علاقة له بالمعاملات في قواعد البيانات، بل هو مجرد تصنيف شائع.

قواعد البيانات التقليدية الثلاث (MySQL وPostgres وSQLite) جميعها معاملاتية (OLTP)، ويمكنها أيضاً التعامل مع بعض المهام التحليلية إذا كان حجم البيانات صغيراً. لكن الأفضل استخدام قواعد بيانات متخصصة لكل حالة استخدام. لذلك ظهرت قواعد بيانات حديثة مخصصة للتحليل فقط مثل ClickHouse وDuckDB وجميع قواعد بيانات "البيانات الضخمة".

يمثل OLAP وOLTP مسارين مختلفين. سنركز على OLTP ونهمل OLAP، مما يجعل بنى البيانات للفهرسة أساسية، بينما تصبح عمليات الدمج والتجميع غير ذات أهمية.

بنى البيانات في الذاكرة مقابل القرص

وضع بنية فهرسة على القرص يواجه تحديات إضافية. التحدي الأول هو اختيار البنية المناسبة. الذاكرة العشوائية والقرص لهما خصائص مختلفة تماماً، خاصة من حيث زمن الاستجابة. حتى مع أحدث أقراص SSD، يكون زمن الاستجابة أبطأ بألف مرة من الذاكرة العشوائية. دراسة هذه الخصائص توصلنا لبنيتين فقط: B+tree وLSM-tree. لذلك خيارات قواعد البيانات في بنى البيانات محدودة جداً.

نوع الوسط	زمن الاستجابة
الذاكرة العشوائية (RAM)	50-100 نانو ثانية

أقراص SSD	50,000-100,000 نانو ثانية
أقراص HDD	5,000,000-10,000,000 نانو ثانية

التحدي الثاني هو حفظ البيانات بشكل دائم. قاعدة البيانات في جوهرها هي بنية بيانات مخزنة على القرص. لذلك فهم بنى البيانات شرط أساسي (راجع كتابي "ابنِ Redis الخاص بك" لممارسة بنى البيانات). يمكنك تعلم بنى البيانات من الكتب، لكن الجزء المفقود هو كيفية تخزينها على القرص وتحديثها تدريجياً مع الحفاظ على الذرية والاستمرارية.

التحدي الثالث هو التزامن. بالنسبة للبيانات في الذاكرة، يكفي عادة استخدام قفل واحد (mutex) للتحكم في الوصول. أما للبيانات على القرص، فإن بطء عمليات الإدخال/الإخراج يجعل هذا الأسلوب غير عملي ويتطلب آليات أكثر تطوراً.

المبدأ الثالث: بناء قاعدة البيانات العلائقية فوق نظام المفاتيح والقيم

طبقتان من الواجهات

SQL تكاد تكون مرادفة لقواعد البيانات. لكن SQL هي مجرد واجهة مستخدم، وليست جوهر قاعدة البيانات. المهم هو الوظائف الأساسية التي تعمل خلف الكواليس.

هناك واجهة أبسط بكثير تُسمى المفتاح-القيمة (KV). يمكنك من خلالها قراءة أو تعيين أو حذف مفتاح واحد، والأهم من ذلك، الاستعلام عن نطاق من المفاتيح بترتيب محدد. نظام KV أبسط من SQL لأنه يعمل في طبقة أدنى. قواعد البيانات العلائقية تُبنى فوق واجهات شبيهة بـ KV تُسمى محركات التخزين.

```
type KV interface {
    // قراءة، تعيين، حذف
    Get(key []byte) (val []byte, ok bool)
    Set(key []byte, val []byte)
    Del(key []byte)
    // استعلام نطاق
    FindGreaterThan(key []byte) Iterator
    // ...
}
type Iterator interface {
    HasNext() bool
    Next() (key []byte, val []byte)
}
```

النصف الأول من الكتاب يبني نظام KV الذي يعتمد عليه النصف الثاني.

لغات الاستعلام: المحللات والمفسرات

الخطوة الأخيرة سهلة نسبياً رغم زيادة عدد الأسطر. المحلل النحوي والمفسر كلاهما يُبنى باستخدام التكرار (Recursion) فقط! هذا الدرس قابل للتطبيق على أي لغة برمجة تقريباً، أو حتى لبناء لغتك البرمجية الخاصة أو لغة نطاق محدد (DSL). راجع كتابي "من الكود المصدري إلى كود الآلة" لمزيد من التحديات في هذا المجال.

الأسئلة وبعض المفاهيم:
شرح تفصيلي لمفاهيم قواعد البيانات

لمحة عن المفاهيم الأساسية (5 أسطر لكل مفهوم)

fsync

أمر نظام يضمن كتابة البيانات من الذاكرة المؤقتة إلى القرص الصلب فعلياً. عندما تكتب برنامج بيانات إلى ملف، لا تُحفظ فوراً على القرص بل تبقى في ذاكرة مؤقتة (buffer) لتحسين الأداء. استدعاء fsync يجبر نظام التشغيل على نقل كل البيانات المعلقة من الذاكرة إلى القرص الفعلي. هذا ضروري لضمان عدم فقدان البيانات عند انقطاع الكهرباء المفاجئ. قواعد البيانات تستخدمه لتأكيد حفظ المعاملات بشكل دائم قبل إعلام المستخدم بنجاح العملية.

B-tree

بنية بيانات شجرية متوازنة مصممة خصيصاً للتخزين على القرص الصلب. كل عقدة (node) في الشجرة تحتوي على عدة مفاتيح وروابط (عادة مئات)، بدلاً من اثنين فقط كما في الأشجار الثنائية. هذا التصميم يقلل عدد عمليات القراءة من القرص بشكل كبير. B+tree (النسخة المحسّنة) تحتفظ بجميع البيانات في الأوراق فقط، مما يسهل عمليات المسح المتسلسل. تُستخدم في معظم قواعد البيانات التقليدية مثل MySQL وPostgres وSQLite لأنها توفر أداء ممتاز للقراءة والكتابة معاً.

LSM-tree

بنية بيانات حديثة تعطي أولوية للكتابة السريعة على حساب القراءة. تعمل بمبدأ الكتابة المتسلسلة (sequential writes) بدلاً من العشوائية، وهو أسرع بكثير على الأقراص. البيانات الجديدة تُكتب أولاً في الذاكرة (memtable)، ثم تُنقل لاحقاً إلى القرص في ملفات مرتبة. بمرور الوقت، تُدمج الملفات الصغيرة في ملفات أكبر (compaction). مناسبة للتطبيقات التي تكتب بيانات كثيرة مثل أنظمة التسجيل (logging) وقواعد البيانات الموزعة مثل Cassandra وRocksDB.

مصممة للعمل مع بيانات أكبر من سعة الذاكرة العشوائية

المشكلة

الذاكرة العشوائية (RAM) محدودة وغالية الثمن. قد يكون لديك 16 جيجابايت RAM لكن قاعدة بياناتك تحتوي على 500 جيجابايت أو عدة تيرابايت من البيانات.

الحل في قواعد البيانات

قواعد البيانات لا تحمّل كل البيانات في الذاكرة دفعة واحدة. بدلاً من ذلك:

1. **التخزين الأساسي على القرص**: البيانات الكاملة موجودة على القرص الصلب
2. **ذاكرة مؤقتة ذكية (Cache)**: فقط البيانات المستخدمة حديثاً أو المتوقع استخدامها تُحمّل في RAM
3. **الصفحات (Pages)**: البيانات مقسمة لوحدات صغيرة (4-16 كيلوبايت عادة)، تُحمّل حسب الحاجة
4. **خوارزميات الإحلال**: عندما تمتلئ الذاكرة، تُحذف البيانات الأقل استخداماً لإفساح المجال لبيانات جديدة

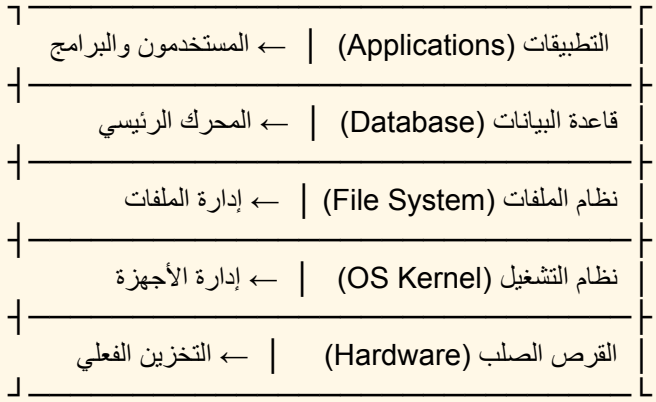
مثال عملي

تخيل أنك تبحث عن سجل موظف معين في قاعدة بيانات ضخمة:

- قاعدة البيانات تقرأ فقط الصفحة التي تحتوي على هذا السجل من القرص
- تضعها في الذاكرة المؤقتة
- تعيد لك النتيجة
- إذا طلبت نفس السجل مرة أخرى، ستجده في الذاكرة (أسرع بألف مرة)

مكان قواعد البيانات في النظام

البنية الطبقية



التفاصيل

1. موقع قاعدة البيانات

- تعمل كبرنامج عادي فوق نظام التشغيل
- ليست جزءاً من النظام (مثل Windows أو Linux)
- يمكن تثبيتها وإزالتها مثل أي تطبيق آخر

2. التفاعل مع نظام التشغيل

- تطلب من النظام فتح وقراءة وكتابة الملفات
- تستخدم استدعاءات النظام (system calls) مثل: open, read, write, fsync
- تحصل على مساحة في الذاكرة من النظام

3. أنواع التنفيذ

- **خادم منفصل** (MySQL, PostgreSQL): تعمل كخدمة مستقلة تستقبل الاتصالات عبر الشبكة
- **مدمجة** (SQLite): مكتبة تعمل داخل التطبيق نفسه، بدون خادم منفصل
- **سحابية** (AWS RDS): مستضافة على خوادم بعيدة يمكن الوصول إليها عبر الإنترنت

4. في الأجهزة المختلفة

- **الحواسيب**: عادة خادم منفصل أو مدمجة
- **الهواتف**: دائماً مدمجة (SQLite في Android/iOS)
- **الخوادم**: خوادم منفصلة عالية الأداء

شرح: استعادة قاعدة البيانات من الأعطال

لماذا هذا صعب؟

fsync وحده لا يكفي لأنه يحفظ البيانات فقط، لكن لا يضمن الاتساق المنطقي.

السيناريو الإشكالي

تخيل أنك تنقل 1000 ريال من حساب أحمد إلى حساب سارة:

1. قراءة رصيد أحمد: 5000 ريال
2. طرح 1000 من حساب أحمد ← 4000 ريال
3. كتابة على القرص: رصيد أحمد = 4000
4. [انقطاع الكهرباء هنا!] ⚡
5. لم يتم: قراءة رصيد سارة
6. لم يتم: إضافة 1000 لحساب سارة

النتيجة الكارثية:

- أحمد فقد 1000 ريال
- سارة لم تستلم شيئاً
- المال اختفى من النظام!

كيف تحل قواعد البيانات هذه المشكلة؟

1. سجل المعاملات (Write-Ahead Log - WAL)

قبل تنفيذ أي عملية، تُكتب النية في سجل:
"سأنقل 1000 ريال من أحمد إلى سارة"
↓
تنفيذ العملية الفعلية
↓
تأكيد الاكتمال في السجل

2. عند الاستعادة من العطل

- قاعدة البيانات تقرأ السجل
- إذا وجدت عملية غير مكتملة، لديها خيارين:
  - **إكمالها** (إذا كانت في مرحلة متقدمة)
  - **التراجع عنها** (إذا كانت في البداية)

3. مثال عملي للاستعادة

السجل يحتوي:
[✓] بدء معاملة #12345
[✓] رصيد أحمد القديم: 5000
[✓] رصيد أحمد الجديد: 4000
[✗] رصيد سارة القديم: لم يُسجل
[✗] رصيد سارة الجديد: لم يُسجل

القرار: إلغاء المعاملة بالكامل
→ إعادة رصيد أحمد إلى 5000 ريال

لماذا هذا أصعب من fsync؟

- fsync يحفظ البيانات فقط
- الاستعادة تحتاج: فهم المنطق + اتخاذ قرارات + تصحيح التناقضات

العلاقة بين بنى البيانات وقواعد البيانات

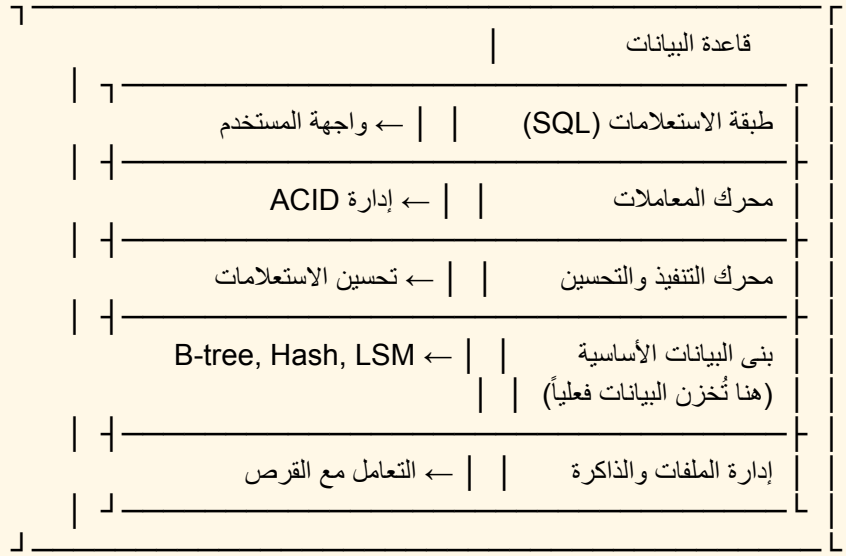
التعريفات

بنية البيانات (Data Structure) طريقة تنظيم وترتيب البيانات في الذاكرة أو القرص لتسهيل الوصول إليها ومعالجتها. مثل: المصفوفات، القوائم المترابطة، الأشجار، الجداول المبعثرة (Hash Tables).

قاعدة البيانات (Database) نظام كامل يدير البيانات، يتضمن: التخزين، الاسترجاع، الأمان، التزامن، الاستعلامات.

العلاقة التفصيلية

1. قاعدة البيانات = بنى بيانات + قواعد + آليات



2. بنى البيانات هي القلب النابض

كل عملية في قاعدة البيانات تعتمد على بنية بيانات:

العملية	البنية المستخدمة	السبب
البحث عن سجل بمفتاح محدد	B-tree أو Hash Table	سرعة البحث O(log n)
قراءة نطاق من السجلات	B+tree	المفاتيح مرتبة
الفهارس الثانوية	B-tree منفصلة	فهرس لكل عمود
الذاكرة المؤقتة	Hash Table + Linked List	LRU Cache
سجل المعاملات	قائمة متسلسلة	كتابة سريعة متتابعة

3. مثال ملموس: البحث عن موظف

```
SELECT * FROM employees WHERE id = 12345;
```

ما يحدث خلف الكواليس:

1. قاعدة البيانات تستقبل الاستعلام
↓
2. تحوّله إلى عملية بحث في B-tree
↓
3. B-tree تبدأ من الجذر:
- العقدة الجذرية: [1-10000] → اذهب لليسار
- العقدة الوسطى: [10001-20000] → اذهب لليسار
- الورقة: تحتوي على السجل 12345 ✓
↓
4. قراءة 3 صفحات فقط من القرص (بدلاً من فحص جميع السجلات)
↓
5. إرجاع النتيجة للمستخدم

4. لماذا لا تكفي بنى البيانات العادية؟

بنى البيانات في الكتب الدراسية مصممة للذاكرة العشوائية:

- ✗ لا تتعامل مع الأعطال المفاجئة
- ✗ لا تدعم المعاملات المتزامنة
- ✗ لا تدير الذاكرة والقرص بكفاءة
- ✗ لا توفر واجهات استعلام عالية المستوى

قواعد البيانات تأخذ هذه البنى وتضيف:

- ✓ ضمانات الاستمرارية (fsync)
- ✓ إدارة الأقفال والتزامن
- ✓ ذاكرة مؤقتة ذكية
- ✓ لغة استعلام (SQL)
- ✓ التحقق من الصلاحيات

أنواع قواعد البيانات المختلفة

ليس فقط OLAP و OLTP

هناك تصنيفات متعددة لقواعد البيانات:

التصنيف 1: حسب نموذج البيانات

1. علائقية (Relational)

- الأشهر والأقدم
- البيانات في جداول بصفوف وأعمدة

- مثل: MySQL, PostgreSQL, Oracle

2. وثائقية (Document)

- تخزن البيانات كوثائق JSON
- مرنة في البنية
- مثل: MongoDB, CouchDB

3. مفتاح-قيمة (Key-Value)

- أبسط نموذج: مفتاح ← قيمة
- سريعة جداً
- مثل: Redis, DynamoDB

4. أعمدة واسعة (Wide-Column)

- تخزن البيانات بشكل عمودي
- ممتازة للبيانات الضخمة
- مثل: Cassandra, HBase

5. رسومية (Graph)

- للبيانات المترابطة بعلاقات معقدة
- مثل الشبكات الاجتماعية
- مثل: Neo4j, ArangoDB

6. بحثية (Search)

- متخصصة في البحث النصي السريع
- مثل: Elasticsearch, Solr

7. سلاسل زمنية (Time-Series)

- للبيانات المرتبطة بالوقت
- مثل قياسات الأجهزة
- مثل: InfluxDB, TimescaleDB

التصنيف 2: حسب الاستخدام

OLTP (معاملاتية)

- معاملات سريعة ومتكررة
- بيانات صغيرة لكل عملية
- مثل: أنظمة البيع، البنوك

OLAP (تحليلية)

- استعلامات معقدة وكبيرة
- تحليل وتقارير
- مثل: مستودعات البيانات

HTAP (هجينة)

- تجمع بين OLTP و OLAP
- مثل: TiDB, MemSQL

التصنيف 3: حسب التوزيع

مركزية

- خادم واحد فقط
- مثل: SQLite, PostgreSQL (وضع واحد)

موزعة

- عدة خوادم تعمل معاً
- مثل: Cassandra, CockroachDB

هل توجد قواعد بيانات للتعدين؟

نعم! قواعد بيانات البلوكتشين (Blockchain Databases)

- تُستخدم في العملات الرقمية والتعدين
- مثل: Bitcoin (تستخدم LevelDB داخلياً)
- خصائصها:
  - موزعة بالكامل
  - غير قابلة للتعديل (immutable)
  - شفافة ومشفرة
  - بطيئة لكن آمنة جداً

لماذا التركيز على OLTP و OLAP؟

ليس لأنهما النوعان الوحيدان، بل لأنهما:

1. **الأكثر شيوعاً**: 90% من قواعد البيانات تندرج تحتهما
2. **الأساس التاريخي**: أول التصنيفات في الأدبيات
3. **التمييز الجوهري**: الفرق في المعمارية الداخلية واضح
4. **تعليمية**: أفضل نقطة بداية للتعلم

الأنواع الأخرى غالباً:

- إما متخصصة لحالات محددة
- أو مبنية فوق OLTP/OLAP بتعديلات

عملية التخزين والعلاقة مع بنى البيانات

كيف تُخزن البيانات؟ رحلة كاملة

السيناريو: إضافة موظف جديد

(INSERT INTO employees (id, name, salary
VALUES (12345, 'أحمد', 5000);

الخطوات التفصيلية

1. استقبال الاستعلام

التطبيق ← قاعدة البيانات
"أريد إضافة موظف جديد"

2. التحليل والتخطيط

- تحليل نحوي للأمر SQL
- التحقق من الصلاحيات
- اختيار خطة التنفيذ

3. سجل المعاملات (WAL) - قبل أي شيء

كتابة في السجل:
"معاملة #789: إضافة موظف id=12345"
↓
fsync للسجل (ضمان الحفظ)

4. التطبيق على بنية البيانات (في الذاكرة)

B-tree للجدول الرئيسي:
- إيجاد المكان المناسب لـ id=12345
- إدراج السجل الجديد في البنية
- قد يتطلب تقسيم عقدة إذا امتلأت

5. تحديث الفهارس

B-tree لفهرس الأسماء:

- إضافة: "أحمد" → 12345

B-tree لفهرس الرواتب:
- إضافة: 5000 → 12345

6. وضع علامة "متسخة" على الصفحات

الصفحات المعدلة → قائمة الصفحات المتسخة
(ستُكتب للقرص لاحقاً)

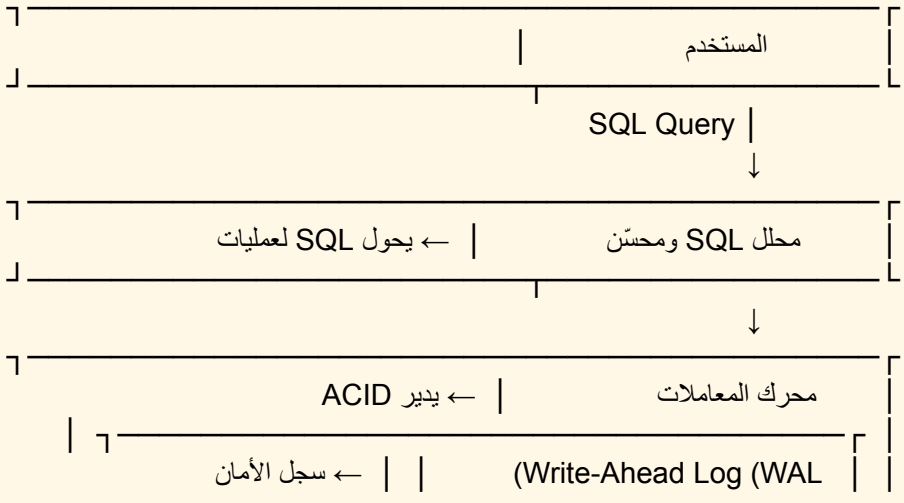
7. الكتابة الفعلية للقرص (غير فورية)

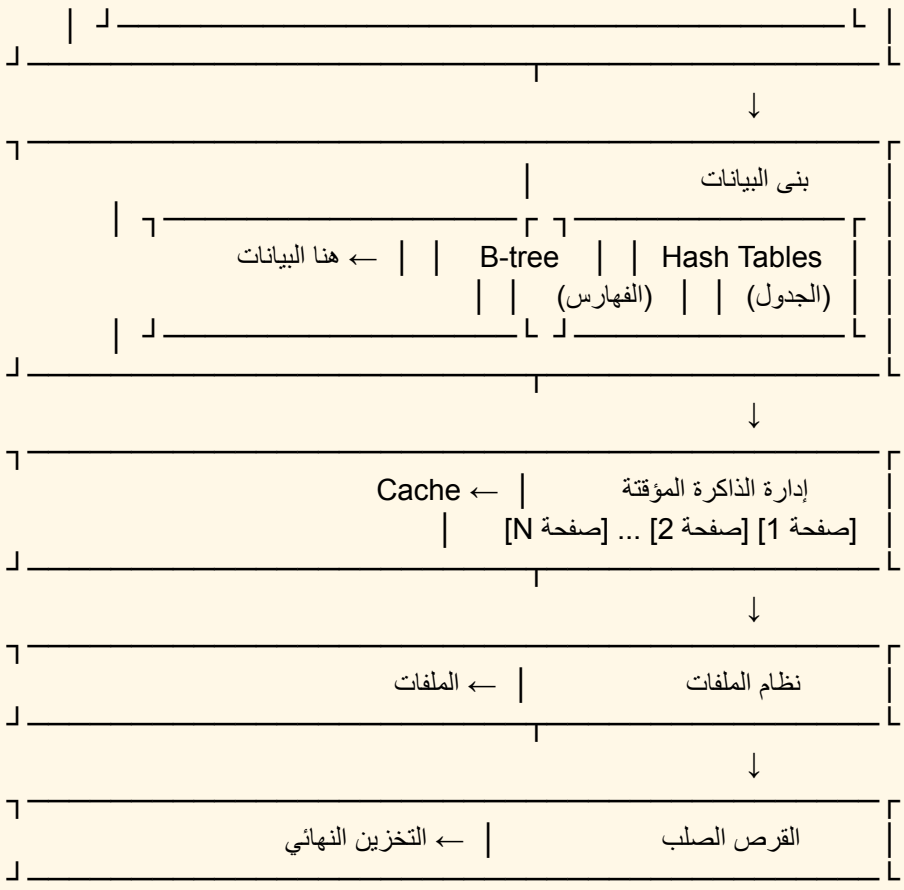
عملية في الخلفية (background process):
كل عدة ثوانٍ:
- أخذ الصفحات المتسخة
- كتابتها للقرص
- fsync نهائي

8. تأكيد المعاملة

كتابة في السجل:
"معاملة #789: مكتملة بنجاح"
↓
fsync للسجل
↓
إرجاع نجاح للمستخدم ✓

العلاقة بين المكونات





هل يمكن إيجاد بنى بيانات أفضل؟

الجواب: صعب جداً، لكن البحث مستمر

لماذا نستخدم B-tree و LSM-tree فقط؟

هذان الهيكلان ليسا عشوائيين، بل نتيجة **قيود فيزيائية** للأجهزة:

1. خصائص الذاكرة العشوائية (RAM)

- سريعة جداً: 100 نانو ثانية
- وصول عشوائي بنفس السرعة
- صغيرة ومكلفة
- → البنى المناسبة: Hash Tables, Arrays, Trees العادية

2. خصائص القرص الصلب

- بطيء: 100,000 نانو ثانية
- القراءة المتسلسلة أسرع 100x من العشوائية
- كبير ورخيص

- → البنى المناسبة: B-tree (قراءة/كتابة متوازنة), LSM-tree (كتابة سريعة)

3. أقراص SSD الحديثة

- وسط بين RAM والقرص التقليدي
- أدت لتطوير بنى هجينة
- مثل: Bw-tree (في SQL Server)

البحوث الحالية:

- **بنى تعلم الآلة (Learned Indexes)**: استخدام نماذج ML للتنبؤ بمواقع البيانات
- **بنى للذاكرة المستمرة (NVM)**: تقنية جديدة بين RAM والقرص
- **بنى موزعة**: مثل Raft-based trees

لكن...

- B-tree عمره 50 سنة وما زال الأفضل للاستخدام العام
- LSM-tree عمره 25 سنة وممتاز للكتابة الكثيفة
- **السبب**: القيود الفيزيائية لم تتغير جذرياً

التزامن: الفرق بين الذاكرة والقرص

المشكلة الأساسية

عندما يستخدم عدة مستخدمين قاعدة البيانات في نفس الوقت، نحتاج **منع التضارب**.

السيناريو الإشكالي

مستخدمان يحاولان تعديل نفس السجل:

المستخدم A: يقرأ رصيد أحمد = 1000 ريال
المستخدم B: يقرأ رصيد أحمد = 1000 ريال
المستخدم A: يضيف 500 → يكتب 1500
المستخدم B: يضيف 300 → يكتب 1300
النتيجة النهائية: 1300 ريال ❌
(المفروض 1800 ريال!)

الحل 1: قفل واحد (Mutex) - بسيط للذاكرة

في البيانات بالذاكرة:

```
lock = threading.Lock()
```

```
def add_money(account, amount):
    with lock:  # قفل واحد لكل شيء
        balance = accounts[account]
        balance += amount
        accounts[account] = balance
    # الآن المستخدم الآخر يمكنه الدخول
```

لماذا يعمل جيداً في الذاكرة؟

- العملية سريعة جداً (ميكروثانية)
- المستخدم الآخر ينتظر قليلاً فقط
- لا مشكلة في التأخير

الحل 2: للقرص - نحتاج شيئاً أذكى

المشكلة مع القرص:

```
lock = threading.Lock()

def add_money(account, amount):
    with lock:  # قفل كامل
        # قراءة من القرص - 10 مللي ثانية! ⏳
        balance = read_from_disk(account)
        balance += amount
        # كتابة للقرص - 10 مللي ثانية أخرى! ⏳
        write_to_disk(account, balance)
        # fsync - 20 مللي ثانية إضافية! ⏳
        fsync_disk()
    # المجموع: 40 مللي ثانية والنظام مقفل بالكامل!
```

لماذا هذا كارثي؟

- كل مستخدم يحجز النظام لـ 40 مللي ثانية
- مع 1000 مستخدم متزامن = 40 ثانية انتظار!
- النظام يصبح عديم الفائدة

الحلول المتقدمة للقرص

1. الأقفال الدقيقة (Fine-Grained Locks)

بدلاً من قفل واحد، قفل لكل سجل أو صفحة:

```
locks = {}  # قفل لكل حساب

def add_money(account, amount):
    if account not in locks:
        locks[account] = threading.Lock()

    with locks[account]:  # قفل هذا الحساب فقط
        balance = read_from_disk(account)
        balance += amount
        write_to_disk(account, balance)
```

الفائدة:

- المستخدم A يعدّل حساب أحمد
- المستخدم B يعدّل حساب سارة في نفس الوقت
- لا تعارض! كل واحد يعمل على بيانات مختلفة

2. التزامن التفاؤلي (Optimistic Concurrency)

```
def add_money(account, amount):
    while True:
        # قراءة بدون قفل + رقم النسخة
        balance, version = read_with_version(account)
        new_balance = balance + amount

        # محاولة الكتابة بشرط
        success = write_if_version_matches(
            account, new_balance, version
        )

        if success:
            break  # نجحت!
        else:
            # شخص آخر عدّل البيانات، أعد المحاولة
            continue
```

الفائدة:

- لا أقفال أثناء القراءة (أسرع)
- فقط عند الكتابة نتحقق من التعارض
- مناسب عندما التعارضات نادرة

3. MVCC (Multi-Version Concurrency Control)

تقنية متقدمة تستخدمها PostgreSQL وOracle:

المستخدم A يبدأ معاملة في الساعة 10:00
المستخدم B يبدأ معاملة في الساعة 10:05

عند قراءة نفس السجل:
- A يرى النسخة كما كانت الساعة 10:00
- B يرى النسخة كما كانت الساعة 10:05
- كل واحد يرى "نسخته" من البيانات
- لا أقفال نهائياً للقراءة!

كيف تعمل؟

- قاعدة البيانات تحتفظ بنسخ متعددة من كل سجل
- كل معاملة ترى "لقطة" (snapshot) من وقت بدايتها
- الكتابة فقط تحتاج أقفال

مثال واقعي مقارن

موقع تجارة إلكترونية، 10,000 مستخدم متزامن:

قفل واحد (Mutex):

السرعة: 25 عملية/ثانية
زمن الاستجابة: 400 مللي ثانية
النتيجة: موقع بطيء جداً ❌

أقفال دقيقة:

السرعة: 5,000 عملية/ثانية
زمن الاستجابة: 20 مللي ثانية
النتيجة: أداء مقبول ✓

MVCC:

السرعة: 50,000 عملية/ثانية
زمن الاستجابة: 2 مللي ثانية
النتيجة: أداء ممتاز ✓✓

الخلاصة

الجانب	الذاكرة	القرص
سرعة العملية	ميكروثانية	مللي ثانية
القفل البسيط	يكفي ✓	كارثي ❌
الحل الأمثل	قفل واحد	أقفال دقيقة + MVCC
التعقيد	بسيط	معقد

واجهة المفتاح-القيمة (Key-Value)

ما هي؟

أبسط واجهة لقاعدة بيانات، تعمل مثل **قاموس أو جدول مبعثر**:

```
مفتاح → قيمة
"user:123" → "{name: 'أحمد', age: 30}"
"product:456" → "{name: 'هاتف', price: 2000}"
"counter:views" → "15234"
```

العمليات الأساسية

```
# 1. التخزين (Set)
db.set("user:123", "أحمد")

# 2. القراءة (Get)
name = db.get("user:123")  # → "أحمد"

# 3. الحذف (Delete)
db.delete("user:123")

# 4. الاستعلام عن نطاق (Range Query)
# جميع المستخدمين من user:100 إلى user:200
users = db.range("user:100", "user:200")
```

لماذا هي طبقة أدنى من SQL؟

SQL (عالية المستوى):

```
SELECT name, salary
FROM employees
'WHERE department = 'IT
AND salary > 5000
;ORDER BY salary DESC
```

يتحول داخلياً إلى عمليات KV (منخفضة المستوى):

```
# 1. استخدام فهرس القسم
("it_employees = db.range("idx_dept:IT:0", "idx_dept:IT:999999

# 2. لكل موظف، تحقق من الراتب
[] = results
:for emp_id in it_employees
    ("{emp_data = db.get(f"emp:{emp_id
    :if emp_data['salary'] > 5000
        (results.append(emp_data

# 3. ترتيب النتائج
(results.sort(key=lambda x: x['salary'], reverse=True
```

كيف تُبنى الجداول فوق KV؟

مثال: جدول موظفين

```
) CREATE TABLE employees (
    ,id INT PRIMARY KEY
    ,(name VARCHAR(100
    ,salary INT
    (department VARCHAR(50
);

;('IT' ,6000 ,'أحمد' ,INSERT INTO employees VALUES (123
```

التخزين الفعلي في KV:

1. البيانات الرئيسية (الجدول):
مفتاح: "emp:123"
قيمة: {"name": "أحمد", "dept": "IT" ,"salary": 6000}

2. الفهرس على القسم:
مفتاح: "idx_dept:IT:123"

قيمة: null (فقط لإثبات الوجود)

3. الفهرس على الراتب:
مفتاح: "idx_salary:6000:123"
قيمة: null

لماذا KV أبسط؟

1. لا توجد أعمدة أو صفوف

- فقط: مفتاح ← قيمة
- التطبيق يفهم معنى القيمة

2. لا توجد استعلامات معقدة

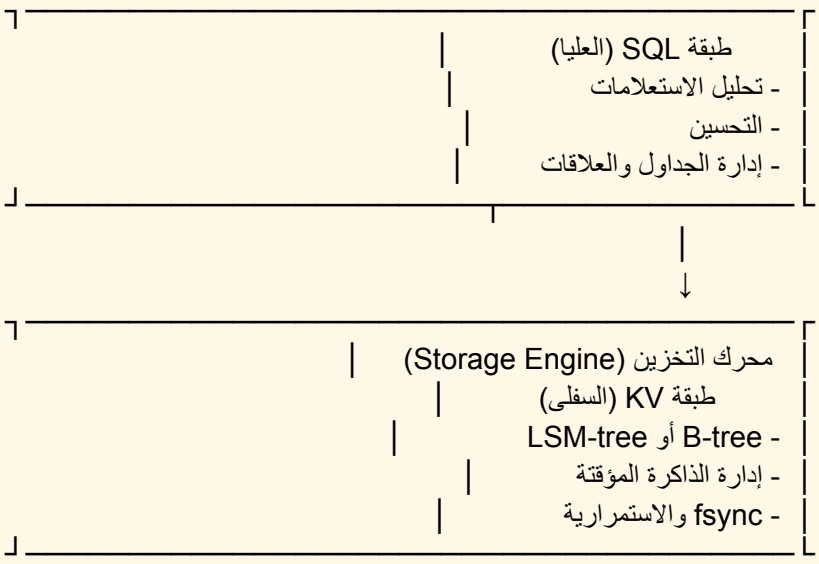
- لا joins
- لا group by
- لا aggregations
- فقط: ضع، اقرأ، احذف

3. المسؤولية على المطور

- KV لا تفهم معنى البيانات
- أنت تصمم كيف تُخزن
- أنت تدير الفهارس

محرك التخزين (Storage Engine)

هذا هو الاسم الرسمي لطبقة KV في قواعد البيانات:



أمثلة واقعية

1. MySQL

- يمكنك اختيار محرك التخزين:
  - InnoDB (B-tree)
  - MyISAM (B-tree بسيط)
  - RocksDB (LSM-tree)
- كل واحد = تطبيق مختلف لـ KV

2. PostgreSQL

- محرك واحد داخلي
- مبني على B-tree
- جميع الجداول ← عمليات KV

3. قواعد بيانات KV النقية

- Redis: KV في الذاكرة
- LevelDB: KV على القرص
- لا توجد طبقة SQL فوقها

لماذا هذا التصميم المتطبق؟

1. فصل المسؤوليات

- SQL تهتم بالمنطق والاستعلامات
- KV تهتم بالتخزين والأداء

2. قابلية التبديل

- يمكن تغيير محرك التخزين دون تغيير SQL
- MySQL يدعم محركات متعددة

3. إعادة الاستخدام

- يمكن استخدام نفس محرك KV في تطبيقات مختلفة
- مثل RocksDB يُستخدم في Facebook وBitcoin وغيرها

الخلاصة المهمة

SQL = واجهة سهلة للمستخدم

- يكتب استعلامات واضحة ومعبرة
- لا يهتم بالتخزين الداخلي

KV = المحرك القوي خلف الكواليس

- يهتم بالأداء والاستمرارية
- مسؤول عن التخزين الفعلي

العلاقة:

مستخدم يكتب SQL
→ قاعدة البيانات تحولها لعمليات KV
→ محرك KV ينفذها على القرص
→ النتيجة تُعاد للمستخدم

بناءً على طلبك، سنقوم الآن بإعادة صياغة شاملة ومعمقة، نربط فيها بين **التفاصيل التقنية الدقيقة** (Under the Hood)، **الأمثلة الواقعية**، و**الروابط العلمية** مع التخصصات الأخرى، لتكوين صورة بانورامية لهذه المفاهيم.

سنبدأ من الوحدة الأساسية (Thread) وصولاً إلى أعقد أنظمة إدارة البيانات (MVCC).

---

1. الـ Thread (خيط المعالجة)

المفهوم التقني:

الـ Thread هو أصغر وحدة تنفيذية يمكن لنظام التشغيل جدولتها. يختلف عن "العملية" (Process) في كونه خفيف الوزن (Lightweight)؛ حيث تشترك جميع الخيوط التابعة لنفس العملية في نفس مساحة الذاكرة (Memory Address Space)، مما يجعل التبديل بينها (Context Switching) أسرع، وتواصلها فيما بينها أسهل، ولكنه أخطر.

الآلية (Under the Hood):

- لكل Thread مكدس خاص (Stack) وسجلات معالج (Registers).
- يشتركون في الـ Heap (الذاكرة الديناميكية) والـ Global Variables.

المثال الواقعي:

تخيل فريق جراحي (Process) داخل غرفة العمليات.

- الجراح، المساعد، والمخدر هم الـ (Threads).
- جميعهم يعملون على نفس المريض (البيانات المشتركة).
- جميعهم يستخدمون نفس الأدوات الموضوعة على الطاولة (الذاكرة المشتركة).

الرابط العلمي (هندسة المعالجات والفيزياء):

يرتبط بمفهوم Hyper-threading في الفيزياء الإلكترونية، حيث يتم استغلال الفراغات الزمنية النانومترية في أنابيب المعالجة (Pipeline) لتمرير تعليمات خيط آخر بينما ينتظر الخيط الأول جلب بيانات من الذاكرة.

---

2. Mutex (Mutual Exclusion) - الاستبعاد المتبادل

المفهوم التقني:

هو كائن برمجي (Object) يضمن الذرية (Atomicity). يعتمد على عمليات انخفاض مستوى (Low-level) في المعالج مثل Test-and-Set أو Compare-and-Swap. يضع الـ Thread في حالة "نوم" (Blocked state) إذا وجد المورد مشغولاً، مما يوفر استهلاك المعالج مقارنة بـ Spinlock (الذي يظل يدور في حلقة مفرغة).

المثال الواقعي:

مدرج هبوط الطائرات في المطار.

- المدرج هو المورد المشترك (Critical Section).
- الطائرات هي الـ (Threads).
- برج المراقبة هو الـ (Mutex). يسمح لطائرة واحدة فقط بالهبوط، ويجب على البقية الدوران في الجو (Wait Queue) حتى تخلو القناة.

الرابط العلمي (الرياضيات ونظرية المخططات - Graph Theory):

عندما نستخدم عدة Mutexes، نصبح عرضة لمشكلة Deadlock (الاستعصاء). علمياً، يتم تمثيل ذلك بمخطط يسمى Resource Allocation Graph. إذا تشكلت "دورة مغلقة" (Cycle) في هذا المخطط، فإن النظام قد توقف رياضياً ومنطقياً.

---

3. Fine-Grained Locks (الأقفال دقيقة الحبيبات)

المفهوم التقني:

هو تحسين لنموذج القفل. بدلاً من استخدام قفل واحد كبير (Coarse-grained) يحمي هيكل البيانات بالكامل، نستخدم مصفوفة من الأقفال أو قفل لكل عقدة (Node) في القائمة المترابطة أو الشجرة.

المثال الواقعي:

نظام إشارات المرور في المدينة.

- **Coarse-grained:** إشارة مرور واحدة توقف المدينة بأكملها لمرور سيارة إسعاف. (آمن لكنه يشل الحركة).
- **Fine-Grained:** إشارات ذكية عند كل تقاطع. توقف فقط التقاطعات التي ستمر منها الإسعاف، بينما تستمر الحركة في باقي المدينة بشكل طبيعي.

الرابط العلمي (بحوث العمليات - Operations Research):

يرتبط هذا بمفهوم Parallel Speedup وقانون أمدال (Amdahl's Law). الهدف العلمي هنا هو تقليل الجزء المتسلسل (Serial Portion) في البرنامج لتعظيم الاستفادة من المعالجات المتعددة.

---

4. Optimistic Concurrency (التزامن المتفائل)

المفهوم التقني:

تقنية "لا قفلية" (Lock-free approach). تفترض أن التصادم (Conflict) نادر.

- **المراحل:**
  1. **Read:** قراءة البيانات مع رقم إصدار (Timestamp/Version).
  2. **Compute:** إجراء العمليات محلياً.
  3. **Validate:** التحقق هل تغير رقم الإصدار في قاعدة البيانات؟
  4. **Commit/Rollback:** الحفظ إذا لم يتغير، أو الإلغاء إذا تغير.

المثال الواقعي:

تعديل مقالة على ويكيبيديا.

أنت تفتح المقالة لتعديلها. شخص آخر يفتحها أيضاً. الشخص الأول يضغط "حفظ" فينجح. الشخص الثاني يضغط "حفظ" بعده بثانية، فيخبره النظام: "حدث تضارب في التعديل، شخص ما عدل الصفحة أثناء عملك، يرجى دمج التغييرات". النظام لم يمنعك من القراءة والكتابة، لكنه منع الحفظ الفاسد.

الرابط العلمي (نظرية الاحتمالات - Probability Theory):

فعالية هذا النموذج تعتمد على دالة احتمالية $P(collision)$.

- إذا كان P منخفضاً (Low Contention): الأداء يكون خارقاً.
- إذا كان P مرتفعاً (High Contention): الأداء ينهار بسبب كثرة إعادة المحاولة (Retries)، مما يستهلك طاقة المعالج بلا طائل.

---

5. MVCC (Multi-Version Concurrency Control)

المفهوم التقني:

هو قمة هرم إدارة البيانات. الفكرة الجوهرية: "للحاضر عدة وجوه".

بدلاً من تحديث البيانات في مكانها (In-place update)، كل عملية تحديث تنشئ صفاً جديداً (Tuple) مع طابع زمني لبداية ونهاية الصلاحية.

- القراء يقرأون النسخة التي تتوافق مع وقت بدء استعلامهم (Consistent Snapshot).
- الكتاب ينشئون نسخاً جديدة.

المثال الواقعي:

نظام المحاسبة البنكية.

تخيل أنك تطلب كشف حساب للسنة الماضية (عملية قراءة ثقيلة). في نفس اللحظة، تصلك حوالة جديدة (عملية كتابة).

- في الأنظمة القديمة: الحوالة تنتظر حتى ينتهي كشف الحساب (بطء).
- في **MVCC**: كشف الحساب يقرأ "نسخة" البيانات كما كانت لحظة الطلب. الحوالة الجديدة تُكتب في "نسخة مستقبلية". كلاهما يعمل بأقصى سرعة دون أن يرى أحدهما الآخر.

الرابط العلمي (الفيزياء النسبية - Relativity):

يرتبط هذا بمفهوم الإطار المرجعي (Reference Frame) في الفيزياء.

لا يوجد "زمن مطلق" للبيانات. كل عملية (Transaction) ترى البيانات من منظور "مخروط الضوء" الخاص بها. ما هو "حقيقي" لعملية بدأت الساعة 10:00 يختلف عما هو "حقيقي" لعملية بدأت 10:01، وكلاهما صحيح في إطاره الزمني.

جدول الترابط العلمي والعملي الشامل

المفهوم	الجوهر التقني	المثال الواقعي	الرابط العلمي/النظري
Threads	Shared Memory Execution	فريق جراحي واحد	هندسة الإلكترونيات (Pipeline Efficiency)
Mutex	Atomicity & Locking	مدرج طائرات واحد	نظرية المخططات (Deadlock Cycles)
Fine-Grained	Granularity Reduction	إشارات مرور ذكية	قانون أمدال (Parallel Scaling)

Optimistic	Validation Phase	تحرير ويكيبيديا	نظرية الاحتمالات (Collision Rate)
MVCC	Snapshot Isolation	كشف حساب بنكي	الفيزياء النسبية (Reference Frames)
