دراسة سرعة أداء أنظمة الربط العلائقي المستخدمة في تطوير التطبيقات

اعداد

م. ايلياس شلوف

اشراف

أ.د. ناصر أبو صالح

د. أسامة ناصر

الملخص

تشهد هندسة البرمجيات توسعاً مستمراً في استخدام نماذج الربط العلائقية ORMs عبر أطر عمل متعددة ولغات برمجة متنوعة، حيث تلعب هذه النماذج دوراً محورياً في تسهيل عمليات التفاعل مع قواعد البيانات العلائقية. تهدف هذه الدراسة إلى إجراء مقارنة شاملة بين أحدث وأبرز نماذج ORM المستخدمة حتى عام 2025، من خلال تحليل نظري وتقني لآليات البناء، أساليب التعامل مع الكيانات، وسهولة الاستخدام، إلى جانب تقييم عملي لأداء هذه النماذج عبر مجموعة من استعلامات القواعد البيانات الشائعة والمعقدة. بناءً على تجارب مقارنة مع عدد من أنظمة إدارة قواعد البيانات العلائقية الرائدة، أبرزت النتائج فروقات ملحوظة في سرعة التنفيذ وكفاءة التفاعل مع البيانات بين النماذج المختلفة. في الختام، تقدم هذه الدراسة توصيات مبنية على نتائجها لمساعدة المطورين في اختيار نموذج ORM الأمثل الذي يتناسب مع متطلباتهم التقنية ويساهم في تحسين أداء التطبيقات البرمجية.

الكلمات المفتاحية: قواعد بيانات، لغة الاستعلامات الهيكلية SQL، لغات برمجة، أطر عمل، أنظمة إدارة قواعد البيانات، استعلامات، هندسة برمجيات، ORM (نظام/نموذج الربط العلائقي).

Studying ORM models performance used in developing applications

Abstract

Software engineering has witnessed continuous growth in the use of Object Relational Mapping (ORM) models across various frameworks and programming languages, where these models play a pivotal role in facilitating interactions with relational databases. This study aims to conduct a comprehensive comparison of the latest and most prominent ORM models used up to 2025, through a theoretical and technical analysis of their construction mechanisms, entity handling approaches, and ease of use, alongside a practical evaluation of their performance across a range of common and complex database queries. Based on comparative experiments with several leading relational database management systems, the results revealed significant differences in execution speed and data interaction efficiency among the different models. In conclusion, this study provides recommendations grounded on its findings to assist developers in selecting the most suitable ORM model that aligns with their technical requirements and enhances software application performance.

Keywords: Databases, SQL, Programming languages, Frameworks, Database management systems, Queries, Software engineering, ORM (Object-Relational Mapping).

1. مقدمة

شهدت السنوات الأخيرة تحولًا ملحوظًا في مجال إدارة قواعد البيانات، حيث اتجهت المؤسسات نحو توزيع قواعد البيانات على خوادم متعددة وتحسين الوصول إليها حتى من مواقع جغرافية متباعدة، وذلك لتحسين سرعة الاستجابة وتوافر البيانات. في هذا السياق، برزت الحاجة إلى تسهيل عمليات التعامل مع قواعد البيانات العلائقية للمطورين، خاصةً من غير المتخصصين في كتابة استعلامات SQL معقدة، مما أدى إلى ظهور مفهوم نماذج الربط العلائقي ORMs.

تتمثل المشكلة الرئيسية التي يعالجها هذا البحث في التحديات المستمرة والمتجددة المتعلقة باختيار نموذج ORM الأمثل، خاصةً مع التطورات السريعة في لغات البرمجة وأُطُر العمل وأنظمة إدارة قواعد البيانات حتى عام 2025. فبينما توفر نماذج ORM مستوىً عاليًا من التجريد والتسهيل، إلا أن اختلافاتها في الأداء وسهولة الاستخدام وتوافقها مع متطلبات البيئات الحديثة تثير التساؤلات حول مدى ملاءمتها في سياق التطورات التقنية الراهنة.

لذلك، يسلط هذا البحث الضوء على فجوة معرفية قائمة في التقييم الحديث لنماذج ORM، من خلال دراسة مقارنة تجمع بين التحليل النظري واختبارات الأداء العملية، بهدف توفير توصيات مبنية على دلائل علمية تساعد المطورين على اتخاذ قرارات مدروسة في اختيار نماذج ORM التي تلبي متطلبات تطبيقاتهم في ظل التحولات المستمرة التي يشهدها المجال.

يُرجى العلم أن التفاصيل التعريفية والفنية المتعلقة بمكونات وبنية نماذج ORM ستُعرض بشكل مفصل في قسم "**الإطار النظري**" لاحقًا.

2. أهمية البحث وأهدافه

تلعب نماذج الربط العلائقي (Object–Relational Mapping – ORM) دورًا حيويًا في تطوير التطبيقات البرمجية الحديثة، حيث تُسهّل التفاعل بين البرمجة كائنية التوجه وقواعد البيانات العلائقية من خلال تجريد وتعقيد استعلامات SQL، مما يعزز إنتاجية المطورين ويسهّل صيانة الأكواد البرمجية. ومع تنامي تنوع نماذج وأطر ORM ، يواجه المطورون تحدي اختيار النموذج الأنسب لمتطلبات مشاريعهم، والتي تتفاوت بناءً على عوامل مثل لغة البرمجة المستخدمة، البيئات التطويرية، متطلبات الأداء، والميزات التقنية الخاصة بكل نموذج.

يركز هذا البحث على إعادة تقييم وتحديث نتائج الدراسات السابقة المتعلقة بنماذج ال ORM، انطلاقًا من تسارع التطور في لغات البرمجة، أُطُر العمل، محركات قواعد البيانات، ونماذج ORM نفسها حتى عام 2025، وذلك لضمان ملاءمتها للاستخدام الحالي في بيئات التطوير الحديثة.

يهدف البحث لتحقيق ما يلي:

1. **تحليل نظري:** دراسة معمقة لتطور نماذجORM ، خصائصها التشغيلية، وآليات التعامل مع الكيانات عبر أشهر الأُطُر البرمجية حتى عام 2025.
2. **تقييم عملي:** مقارنة الأداء التشغيلي لهذه النماذج من حيث سرعة تنفيذ الاستعلامات وكفاءتها عبر استعلامات متنوعة، باستخدام أنظمة إدارة قواعد البيانات العلائقية الأكثر انتشارًا.
3. **تقديم توصيات:** استخراج استنتاجات واضحة وأدلة تساعد المطورين في اختيار نموذج ORM الأمثل وفقاً لمتطلبات ومحددات مشاريعهم البرمجية.

3. مواد وطرق البحث

3.1. المفاهيم الأساسية

يستند هذا البحث إلى مجموعة من المفاهيم النظرية الأساسية التي تشكل جوهر دراسة نماذج الربط العلائقي ORM. من بين هذه المفاهيم نمطا Data Mapper و Active Record، حيث يمثل الأول أسلوبًا يُفصل بين طبقة الكود البرمجية وقاعدة البيانات بالكامل، بينما يمكّن الثاني كل كائن برمجي من التحكم المباشر في عملياته على الجداول المرتبطة به. فهم الفوارق بين هذه الأساليب يعد خطوة أولى محورية في تحليل مناهج العمل ونماذج الأداء في تقنيات ORM المتنوعة.

3.2. نماذج ORM المشهورة وأطر العمل المستخدمة

تغطي الدراسة مجموعة مختارة من أشهر نماذج الـ ORM المعتمدة في التطوير البرمجي حتى عام 2025، مثل Hibernate و Entity Framework Core و Hibernate و Dapper و Eloquent و Doctrine، بالإضافة إلى نماذج بيئة الاندرويد مثل Room, greenDAO. ويشمل التحليل مقارنة منهجية بين هذه النماذج من حيث التصميم، سهولة الربط مع قواعد البيانات، آليات التفاعل مع الكيانات، وتبني أنماط مختلفة لمعالجة استعلامات البيانات والتحكم في الأداء.

3.3. الأدوات والبيئة البرمجية

اعتمدت الدراسة على تحليل وتلخيص سبع أوراق بحثية مرجعية شكلت عينة ممثّلة للدراسات المقارنة في المجال. تمت الاستعانة بأطر عمل برمجية متنوعة (Java, .NET, PHP, Android) وباستخدام أحدث إصدارات بيئات التطوير وأنظمة إدارة قواعد البيانات العلائقية الرائدة حتى عام 2025. تضمن العمل رصد الأداء (زمن الاستجابة واستهلاك الموارد)، ودراسة سهولة الاستخدام، وموثوقية التطبيق العملي للنماذج في سيناريوهات متعددة. بالإضافة إلى ذلك، أجري تحليل نقدي لمواطن القوة والضعف في الدراسات السابقة، وتمت مراجعة مدى مواكبتها للتقنيات الحديثة وتوجهات التطوير الجارية.

4. الإطار النظري

تُعد نماذج الربط العلائقي ORM من التقنيات الأساسية التي تربط بين البرمجة كائنية التوجه وقواعد البيانات العلائقية. فهذه النماذج توفر طبقة تجريدية تُسهّل عملية التفاعل مع قواعد البيانات عن طريق تمثيل الجداول العلائقية ككائنات داخل لغات البرمجة، مما يقلل الحاجة إلى كتابة استعلامات SQL معقدة.

يتركز الإطار النظري لهذا البحث على دراسة المفاهيم الأساسية التي تقوم عليها نماذج ORM، بما في ذلك آليات تحويل البيانات بين النماذج الكائنية والعلائقية، وتصميم الأطر التي تدعم هذه النماذج، وكذلك تحليل مكونات نموذج ORM الرئيسي، حيث ان:

- الغرض الأساسي من هذا المفهوم كان تمثيل جداول البيانات ك صفوف غرضية في لغات البرمجة دون الحاجة الى معالجة كل بيانات الجداول في كل خطوة.
- ORM هي اختصار لـ Object–Relational–Mapping (ORM) وهي تقنية برمجية لتحويل البيانات بين قواعد البيانات العلائقية ولغات البرمجة الغرضية التوجه مثل Java من خلال جعل التعامل الأساسي مع الأغراض بدلاً من الجداول والاستعلامات.
- يستند هذا النموذج على نموذج الاتصال التقليدي مع قواعد البيانات
- يتم فيه إدارة المناقلات وانشاء المفاتيح بشكل تلقائي.
- يخفي تفاصيل استعلامات SQL من خلال جعل التعامل مع منطق البرمجة الغرضية التوجه.

يتكون نموذج ال ORM من الكيانات الأربعة التالية:

1. واجهة برمجة تطبيقات API لإجراء عمليات CRUD الأساسية على الكائنات/الأغراض.
2. لغة أو واجهة برمجة تطبيقات لتحديد الاستعلامات التي تشير إلى صفوف وخصائص الصفوف.
3. وسيلة قابلة للتكوين لتحديد الربط مع البيانات الوصفية.
4. أسلوب للتفاعل مع كائنات المعاملات لإجراء عمليات مثل: فحص التغييرات على البيانات، وجلب النتائج المتأخر، ووظائف تحسين أخرى.

5. الدراسات المرجعية

تتناول الدراسات السابقة نماذج الربط العلائقي (ORM) من زوايا متعددة تجمع بين التحليل النظري والتقييم العملي للأداء، مع اختلاف في نطاق الدراسة والبيئات المستخدمة، وهو ما يسمح برسم صورة شاملة عن تطور هذه النماذج حتى عام 2025.

في البداية، تقدم الدراسة الأولى [1] مراجعة تاريخية نظرية تغطي عشرين عاماً من تطور نماذج ORM في عدة لغات برمجة، مع التركيز على حلول مشكلة "عدم التطابق" بين النظم الغرضية وقواعد البيانات العلائقية. تبرز الورقة أهمية نماذج التصميم الأساسية التي تقوم عليها هذه النماذج وكيف تخدم آليات جلب البيانات المختلفة، مع توفير جدول ملخص

للمقارنة بين هذه النماذج. بالرغم من شمولية الدراسة، تظل محدودة في تقديم تقييم عملي أو نتائج أداء، مما يبرز الحاجة إلى دراسات تجريبية محدثة تأخذ في الاعتبار التقنيات والإصدارات الحديثة.

تكمن قوة الدراسة الثانية [2] في تقييم الأداء العملي لمجموعة من نماذج ORM داخل بيئةNET. ، خصوصاً Entity Framework Core، nHibernate، وDapper، عبر عمليات CRUD على مجموعات بيانات بأحجام متفاوتة. توضح نتائج الدراسة الاستخدام الجزئي الأفضل لكل نموذج حسب نوع العملية، ولا تبرز وجود نموذج متفوق بشكل مطلق؛ مما يؤكد أن الاختيار يعتمد على نوعية العمليات ومتطلبات المشروع. إلا أن الدراسة تفتقر إلى تغطية أطر عمل أخرى أو لغات برمجة غيرNET. ، كما أنها لا تتعامل مع تعقيدات الاستعلامات المعقدة أو التفاعلات متعددة الأنظمة، وهو ما تعد هذه الورقة البحثية الحالية محاولة لسده.

ركزت الدراسة الثالثة [3] على نموذج Eloquent ORM في بيئةPHP/ Laravel ، موضحة خصائصه في دعم أنواع متعددة من العلاقات بين الكيانات، وهو مفيد لفهم الدعم الوظيفي للنماذج في تطبيقات ويب محددة. لكنها تبقى دراسة تطبيقية ضيقة النطاق تفتقر إلى مقارنة أداء مع نماذج أخرى أو تقييم تجريبي متعدد البيئات. لذا تبرز الحاجة إلى استعراض شامل وأعمق يدمج هذه النماذج ضمن مقارنة أوسع وأدوات قياس أداء موحدة.

تناولت الدراسة الرابعة [4] نموذج Hibernate في بيئةJava ، مشيرة إلى متطلبات التهيئة، آلية عمل EntityManager، ولغة الاستعلام الخاصةHQL ، مع تسليط الضوء على ميزة التخزين المؤقت التي تحسن الأداء بشكل ملحوظ عند تفعيلها. تقدم هذه الدراسة خلفية تقنية ثرية، لكن تحليلها يقتصر على نموذج واحد فقط، ولا تقدم مقارنات مباشرة مع نماذج أخرى. كما تفتقر إلى اختبار عملي متكامل يشمل الاستعلامات المختلفة وحدود الأداء.

في مجال تطبيقات الأندرويد، قدمت الدراسة الخامسة [5] مقارنة عملية بين مكتبات Room وgreenDAO، فضلاً عن SQLiteOpenHelperالتقليدي. أظهرت النتائج تفوق greenDAO في معظم عمليات CRUD من حيث الأداء، فيما تدعم Room التتابع في الحذف والتعديل ولكن بتكاليف أكبر من حيث استخدام الموارد والوقت. هذه الدراسة تطرح تساؤلات هامة حول تبعية اختيار النموذج لسياق التطبيق ومستوى تعقيد العلاقات بين الكيانات، لكنها تنحصر في بيئة تطوير نموذجية، مما يحد من تعميم النتائج على أطر عمل ولغات برمجة أخرى.

أما الدراسة السادسة [6]، فهي تقدم تحليلاً لميزات ومزايا نموذج Doctrine المستخدم في بيئةPHP/Symfony ، مع استعراض لآليات توليد المخططات التلقائية، لغة الاستعلام الخاصةDQL ، والتخزين المؤقت. كما تشير إلى التحديات المرتبطة بزيادة التعقيد الطبقي نتيجة التجريد، واحتمالية ضعف الأداء مع الاستعلامات غير المحسنة. مع ذلك، الدراسة تعتمد أساساً على تقييم نظري وتجريبي محدود النطاق، ما يترك مجالاً لبحث أعمق مقارنة بالأطر المختلفة والتقنيات الحديثة.

أخيراً، تقدم الدراسة السابعة [7] مقارنة أداء مختصة بين نماذج ORM الثلاثة الأكثر شيوعاً فيJava: Hibernate ، Ebean، وTopLink، مستعينة بالمعيار Benchmark 007 لإجراء اختبارات زمن التنفيذ على مجموعة استعلامات CRUDمتنوعة. توضح النتائج تفوق Ebean عموماً بسبب واجهته الخالية من الجلسات، مع تميز Hibernate في إدراج البيانات، وتوصي الدراسة باستخدام Ebean للأداء الأفضل. رغم قدم هذه الدراسة (2012) وتخصيصها لبيئة Javaفقط، فإنها تعطي لمحة مهمة عن تطور الأداء عبر الأجيال وتبرز الحاجة لإعادة تقييم مماثل يشمل التحديثات والتقنيات الحديثة.

5.1. ملخص الدراسات السابقة

الرقم المرجعي	عنوان المدارسة	تاريخ النشر	بيئة العمل	المنهجية	أبرز النتائج	نقاط الضعف
1	**Twenty years of object–relational mapping**	**2017**	متعددة (Java, .NET, PHP…)	مراجعة نظرية وتحليلية	ملخص شامل لنماذج ORM وآليات التعامل مع عدم التطابق	يفتقر لتقييم عملي وحديث للأداء
2	**Performance Comparison of CRUD Methods using NET Object Relational Mappers**	**2020**	NET (EF, nHibernate, Dapper)	تجريبية (وقت التنفيذ، الذاكرة)	لا يوجد متفوق مطلق، الأداء يعتمد على نوع العملية وعدد الكيانات	محدود ببيئة NET. فقط، لا يشمل تعقيدات الاستعلام
3	**Eloquent Object Relational Mapping Models for Biodiversity Information System**	**2017**	PHP (Laravel)	دراسة تطبيقية	توضيح دعم العلاقات المعقدة بين الكيانات	عدم وجود مقارنة أداء أو بيئات أخرى

4	**Lessons in Persisting Object Data using Object–Relational Mapping**	**2019**	Java (Hibernate)	مراجعة تقنية مع أمثلة تطبيقية	شرح ميزات النموذج من التخزين المؤقت، HQL، ميزات الحماية	تركز على نموذج واحد فقط، لا توجد مقارنة تجريبية
5	**Room vs. greenDAO for Android**	**2022**	**اندرويد**	تجريبية (RAM, CPU, زمن التنفيذ)	GreenDAO أسرع، Room يدعم التتالي لكنه أبطأ ويستهلك موارد أكثر	حصر على بيئة أندرويد فقط، عدم تعميم النتائج
6	**Improving data processing performance for web applications using object–relational mapping**	**2015**	PHP (Symfony)	مراجعة نظرية وتجريبية محدودة	شرح الميزات من تبسيط إدارة البيانات، تحسين الأداء مع بعض العيوب	تقييم محدود الأبعاد والمقارنات
7	**Object–relational mapping tools and Hibernate**	**2017**	Java	تجريبية (Benchmark 007، زمن التنفيذ)	Ebean الأسرع، Hibernate الأفضل للإدراج، توصية باستخدام Ebean	قديمة لعام 2012، لا تشمل التطورات الأحدث

جدول 1 ملخص مقارنة الدراسات المرجعية

5.2. نقاط الضعف المشتركة والمبررات لإجراء البحث الحالي

على الرغم من القيمة العلمية والعملية لهذه الدراسات، إلا أن لديها عدة نقاط ضعف تبرر القيام بالبحث الحالي:

- **نقص التغطية الشاملة** :تميل معظم الدراسات إلى التركيز على لغات أو بيئات محددة (مثلNET. ، Java، PHP، Android)، مع ندرة مقارنة شاملة عبر بيئات تطوير وأطر متعددة في آن واحد.
- **تفاوت المنهجيات** :تختلف الدراسات بين النظرية الصرفة والتقييم العملي، مع غياب موحد لمنهجيات قياس الأداء والمعايير المستخدمة، مما يصعب استخلاص مقارنات موضوعية دقيقة.
- **تجاهل التطور التقني الحديث** :بعض الدراسات، مثل [7] سنة 2012 و [1] التي استعرضت التطور حتى 2016، لا تغطي أحدث نماذج ORM أو التطورات الجديدة في لغات البرمجة وأطر العمل حتى عام 2025، وهو أمر حيوي لفهم الفروقات الحالية.
- **غياب تقييم الأداء في حالات معقدة** :قلة الدراسات التي تدرس تأثير نماذج ORM على استعلامات معقدة، إدارة العلاقات المعقدة، والتعامل مع أحجام بيانات كبيرة بطرق موحدة.
- **عدم التوازن في التركيز** :بعض الدراسات تعطي تفصيلاً ناعماً (كما في دراسة [2]) فيما تختصر أخرى مواضيع هامة، مثل جوانب الحماية، الأمان، والتوافق مع أنظمة قواعد البيانات الحديثة.
- **الإغفال عن البيئة العملية الواقعية** :كالتأثير على حجم التطبيقات واستهلاك الموارد، خصوصاً في البيئات منخفضة الموارد كالأجهزة المحمولة، الذي أجازته دراسة [5] لكنه غير مغطى في بقية الدراسات.

بناءً على ذلك، يهدف البحث الحالي إلى سد هذه الفجوات من خلال مقارنة موسعة ومنهجية تجمع بين التحليل النظري ونماذج الأداء العملي عبر بيئات وأطر متعددة وحديثة، مع تدقيق في استعلامات الأداء المعقدة، واستهلاك الموارد، والتوافق مع متطلبات مطوري البرمجيات في عام 2025. تقدم الورقة توصيات دقيقة وموضوعية تساعد في اختيار نموذج ORM الأمثل وفقاً لاحتياجات الأنظمة المعاصرة، مع الاعتماد على بيئة تجريبية موحدة وأدوات قياس دقيقة تعزز من موثوقية النتائج.

6. النماذج المعمارية لنماذج ORM والمقارنة التطبيقية

في البداية يجب ان نذكر ان نماذج ال ORM الحالية أصبحت تعتمد على أكثر من معمارية في بنائها وهي التي تحدد سهولة الكتابة بالإضافة لسرعة التنفيذ والربط والمقابلة مع قواعد البيانات، وأشهر هذه البنى المعمارية هي:

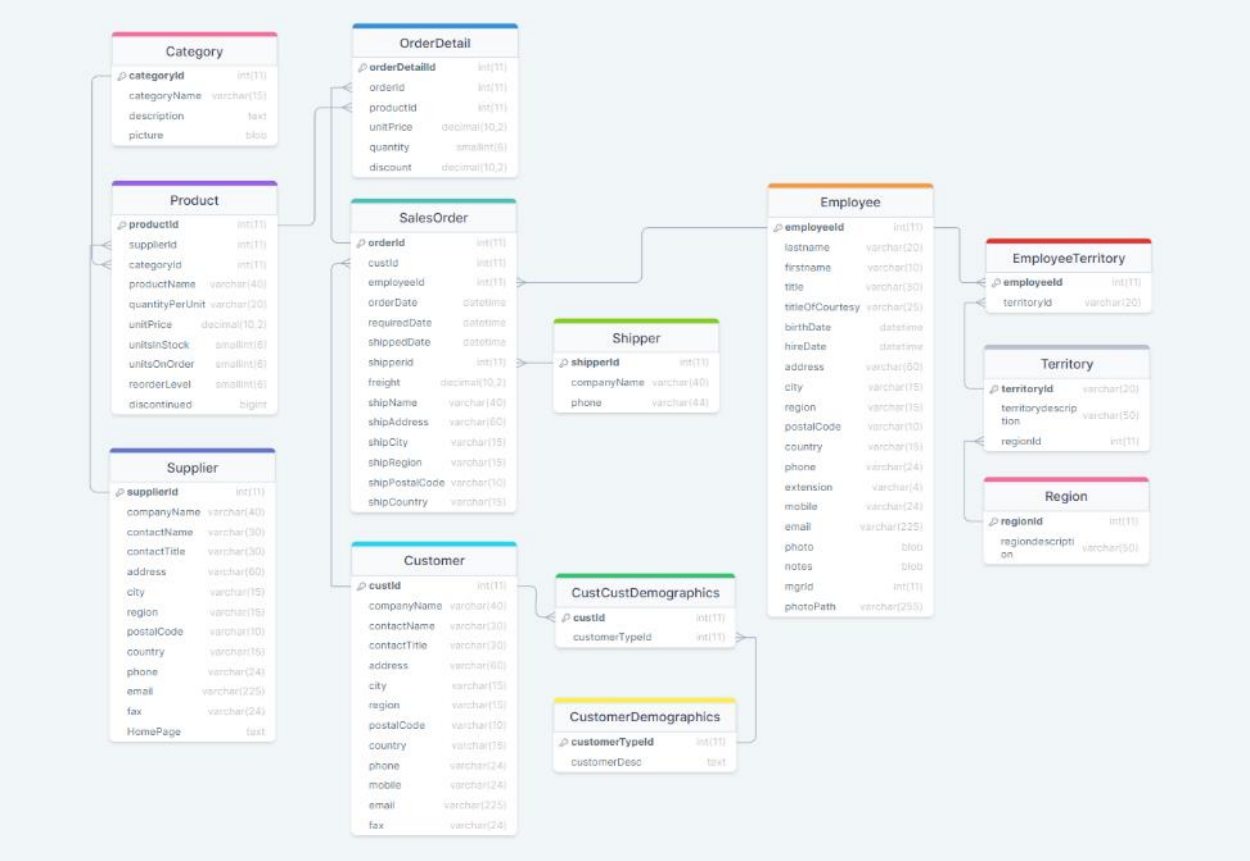
1. **Active Record**: في هذه البنية يقابل الصف بشكل مباشر جدول قاعدة البيانات ويستطيع أي غرض منه التفاعل معها والقيام بالعمليات بشكل مباشر مثل الادراج والحذف والتعديل وغيرها، وتتميز هذه البنية بسهولة وسرعة الكتابة للعمليات البسيطة لكنها غير مفضلة في العمليات المعقدة كونها تزيد من درجة الاعتمادية بين الصفوف، وامثلة عن هذه البنية هو ال Django ORM.
2. **Entity Mapper**: في هذه البنية يكون هذا الصف هو الوسيط بين صفوف النماذج التي تمثل الجداول و قاعدة البيانات حيث تكون الأغراض المنشأة من الصفوف هي فقط تحوي على بيانات وليس لها أي وصول لقاعدة البيانات، وتكون مسؤولية ال EM هنا هي مقابلة قيم الحقول في هذه الصفوف وجداول قاعدة البيانات، ومن النماذج التي تستخدم هذه البنية هو نموذج Ktorm.
3. **Entity Manager**: وهي بنية خاصة حصرية فقط ببعض النماذج مثل نموذج ال Hibernate تتحكم بشكل كامل بكل عمليات قاعدة البيانات من انشاء وحذف الجداول والحقول للقيام بالاستعلامات وغيرها واهم ما يميز هذه البنية هو الأداء السريع وتقنيات التخزين المؤقت والجلب الكسول للبيانات وغيرها.

والان بعد ذكر هذه المقارنة النظرية لتوضيح أهمية البنية المعمارية على سهولة كتابة وسرعة أداء النماذج المختلفة، سننتقل للمقارنة بين أشهر النماذج حاليا وهي:

1. **Java Spring Boot**: Hibernate, Eclipse Link, Native Connection
2. **Kotlin Ktor**: Ktorm
3. **PHP**: Eloquent + Laravel, Doctrine + Symfony
4. **Python**: Django ORM, SQL Alchemy + flask
5. **Node.js**: Sequelize, Prisma, TypeORM (typescript)
6. **C# .NET Core**: Entity Framework, Dapper (Micro ORM)

وتمت المقارنة من خلال تنفيذ 10 استعلامات وقياس زمن التنفيذ لأول استدعاء والمتوسط لتنفيذ كل استعلام 100 مرة متتالية بعد المرة الأولى لرؤية فيما إذا كان هنالك تأثير للتخزين المؤقت على سرعة الاستعلام وبالإضافة لمقارنة النتائج السابقة مع نتائج الاتصال التقليدي في جافا (كونه الأسرع بين باقي اللغات).

كما انه تم استخدام قاعدة بيانات شهيرة **northwind** على نظامي إدارة قواعد بيانات، الأول موجه لقاعدة بيانات **MySQL** والثاني ل **PostgreSQL**، والمخطط التالي يظهر بنية هذه القاعدة والعلاقات بين الجداول فيها.



الشكل 1 المخطط الهيكلي لقاعدة البيانات Northwind

7. بيئة التنفيذ والتجربة العملية

تمت المقارنة على حاسب شخصي وبشكل محلي لكي يكون القياس لزمن التنفيذ فقط دون وجود أي اتصال بعيد مع قواعد البيانات، ومواصفات هذا الحاسب موضحة بالجدول التالي:

Intel Core i7	Ram 16 GB	SSD 1 TB	Windows 11 Pro
12700H	DDR5 5200 MHZ	NVME PCIE 4.0	23H2

وتم تسجيل زمن التنفيذ تلقائياً من خلال صف تم بناؤه بمختلف اللغات البرمجية الموافقة لأطر العمل لقياس زمن تنفيذ كل الاستعلامات على المسلك الرئيسي دون وجود مسالك فرعية والوحدة المستخدمة للقياس هي الميكرو ثانية.

7.1. الاستعلامات المستخدمة للتجربة

1. جلب كل الموظفين (جدول صغير)

```
select * from employee
```

2. جلب كل تفاصيل الطلبات (جدول كبير)

```
select * from orderDetail
```

3. جلب موظف باستخدام رقمه التسلسلي (PK – index)

```
select * from employee where employeeId = ?
```

4. جلب موظف باستخدام اسمه الأول والأخير

```
select * from employee where concat(firstname, ' ', lastname) = ?
```

5. حذف عميل محدد باستخدام رقمه التسلسلي

```
delete from customer where custId = ?
```

6. جلب كل الأماكن ثم تفريغ الجدول ثم إعادة ادخال البيانات المحذوفة

```
select * from territory; // نخزنها في متحول list
delete from territory;
foreach record in list then:
        insert into territory (…) values (…record)
```

7. جلب كل عميل مع اجمالي الطلبات التي قام بدفعها

```
SELECT c.contactName as name, sum(od.unitPrice * od.quantity * (1 - od.discount))
as total
FROM customer c
join salesorder so on cast(so.custId as integer) = c.custId
join orderdetail od on od.orderId = so.orderId
GROUP by c.custId
ORDER by total DESC;
```

8. جلب كل الأماكن السكنية التي يقطنها العملاء (بلد – مدينة) مع اجمالي عدد العملاء القاطنين فيها

```
SELECT c.country, c.city, count(c.custId) as count
FROM customer c
GROUP by c.country, c.city
ORDER by count desc;
```

9. جلب اجمالي مبيعات كل الأصناف

```
SELECT p.productName as name, sum(od.unitPrice * od.quantity * (1 - od.discount))
as total
FROM product p
join orderdetail od on od.productId = p.productId
GROUP by p.productId
ORDER by total DESC;
```

10. جلب الطلبات التي قيمة كل منها بين 5000 و12000

```
SELECT a.* from (
    select so.*, sum(od.unitPrice * od.quantity * (1 - od.discount)) total
    FROM salesorder so
    join orderdetail od on od.orderId = so.orderId
    GROUP by so.orderId
    ORDER by total desc
) a where a.total BETWEEN 5000 and 12000
```

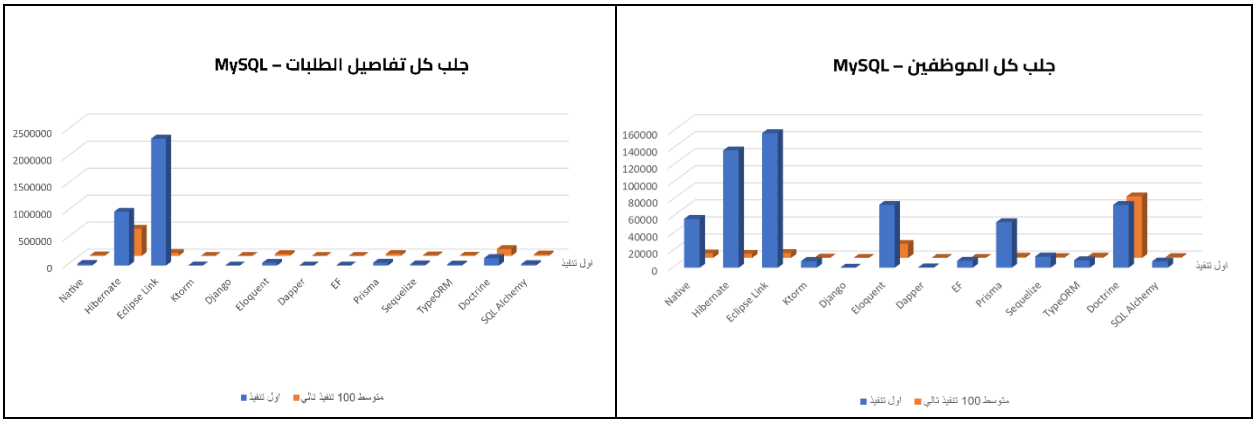
8. التجربة العملية

تم تنفيذ الاستعلامات السابقة على الجهاز الذي تم ذكر مواصفاته بشكل سابق باستخدام الصف البرمجي التالي والذي تم بناء نسخ منه بلغات البرمجة المختلفة لكي تناسب كل النماذج مع مختلف اطر العمل المدروسة، حيث تم تسجيل اول مدة تنفيذ والمتوسط الحسابي لتنفيذ نفس الاستعلام 100 مرة متتالية، وتم تسجيل النتائج مع قاعدتي البيانات وطرح النتائج في المخططات التالية، حيث يمثل المحور الشاقولي مدة التنفيذ بالميكرو ثانية والمحور الافقي يمثل النموذج الذي تم قياس مدة تنفيذ الاستعلام فيه.

```
public record Task (
    1 usage
    String name, Runnable action
) { }

3 usages
private TaskLogger taskLogger;
3 usages
private TimeUnit timeUnit;

1 usage  Elias Shallouf
public TaskRunner(TaskLogger taskLogger, TimeUnit timeUnit) {...}

1 usage  Elias Shallouf
public void execute(Task task) {
    long duration = System.nanoTime();

    task.action().run();

    duration = System.nanoTime() - duration;
    taskLogger.log(task,  comment: "exec 1", timeUnit.convert(duration));
}
```

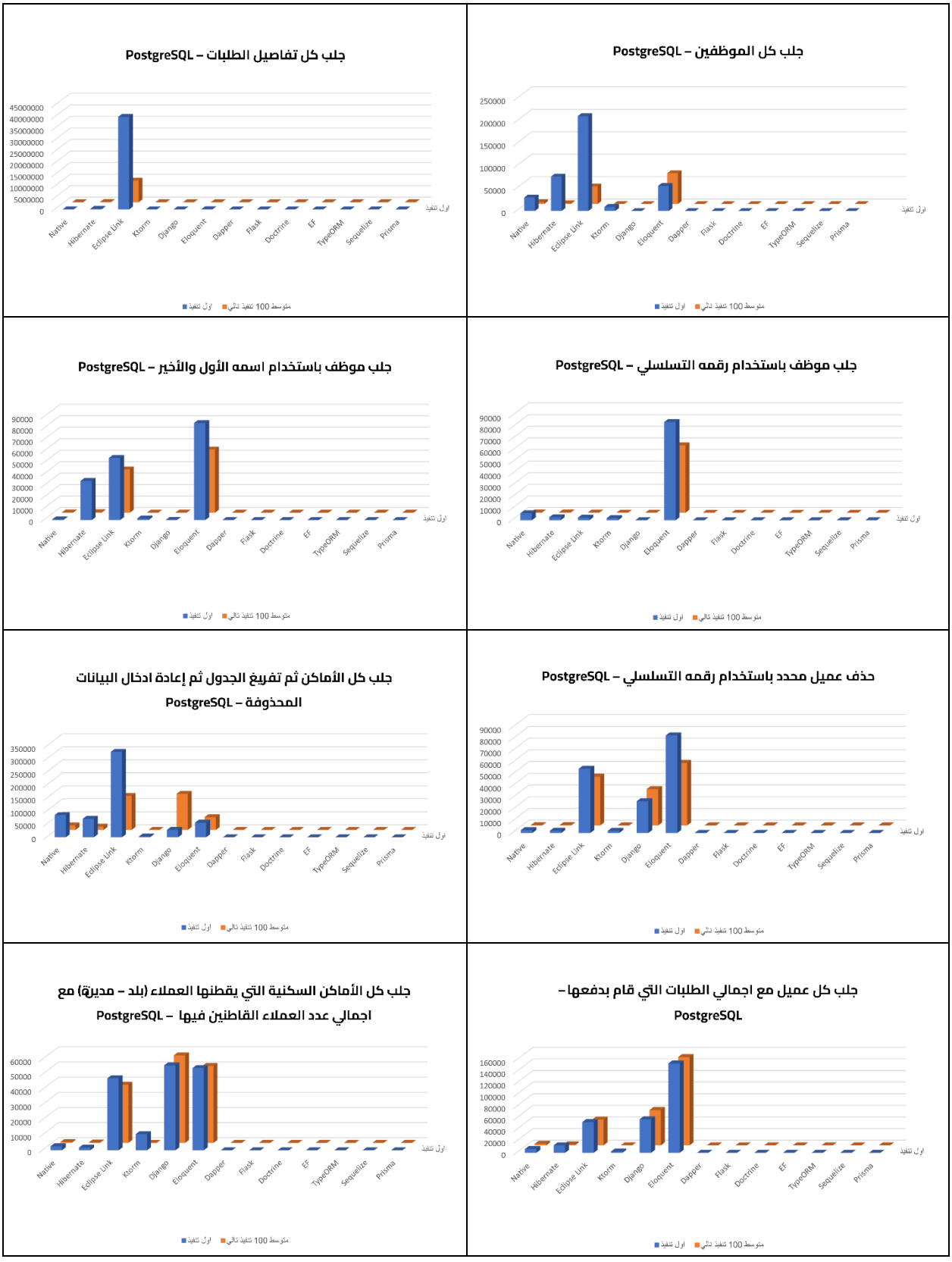
الشكل 2 صف TaskRunner المستخدم لقياس ازمنة التنفيذ

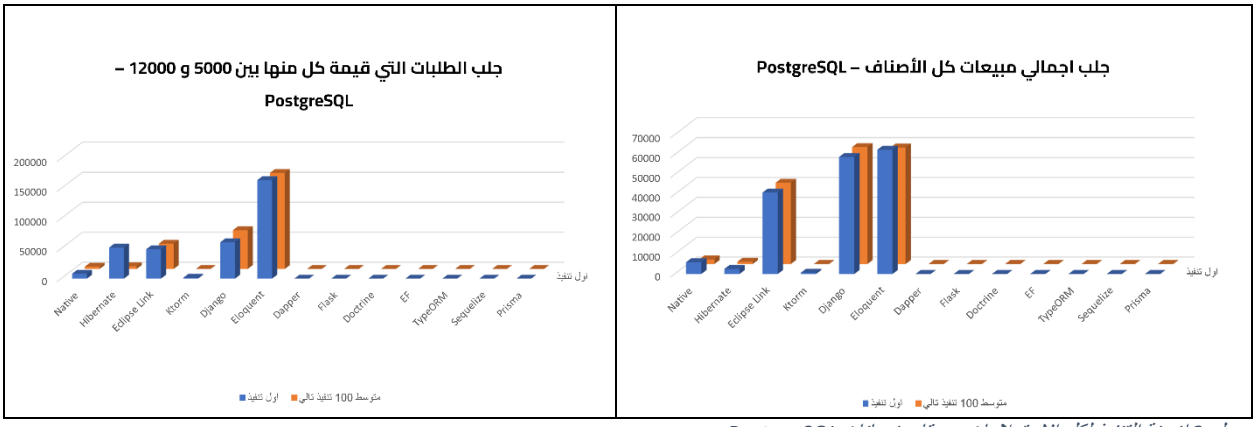
8.1. ازمنة التنفيذ في MySQL



جدول 2 ازمنة التنفيذ لكل الاستعلامات مع قاعدة بيانات MySQL

8.2. ازمنة التنفيذ في PostgreSQL





جدول 3 ازمنة التنفيذ لكل الاستعلامات مع قاعدة بيانات PostgreSQL

8.3. الملاحظات والنتائج المستخلصة بعد القياسات

بعد المقارنات السابقة نأتي للملاحظات عن الأداء مع ذكر اهم مميزات كل نموذج وسيئاته:

النموذج	الميزات	السلبيات	ملاحظات نظرية عن الاداء
Hibernate	معيار JPA، ميزات غنية (التخزين المؤقت، التحميل الكسول/المبكر)، مجتمع كبير	معقد، حمل زائد محتمل، إعدادات مطولة	حمل زائد محتمل، لكن أداء عالي مع الضبط (التخزين المؤقت، تحسين الاستعلامات)
Eclipse Link	معيار JPA، متوافق بدقة، غني بالميزات	معقد، حمل زائد محتمل	نفس الملاحظات عن الـ Hibernate؛ لكن النتائج الفعلية قد تختلف قليلاً
Ktorm	Kotlin DSL، امن كتابيا مرونة تشبه SQL في Kotlin	مجتمع أصغر من JPA، ميزات أقل شمولاً من ORMsالكاملة	أداء جيد؛ حمل أقل من ORMsالكاملة، أكثر من JDBC
Eloquent	سهل الاستخدام للغاية، تجربة مطور رائعة (Laravel) البساطة	قد يؤدي إلى مشكلة N+1 إذا لم يتم الحذر، مرتبط بشدة غالبا ب Laravel	سريع جداً لعمليات CRUD البسيطة؛ الأداء يعتمد على التحميل المبكر

Doctrine	غني بالميزات (التخزين المؤقت، التحميل الكسول/المبكر) المستودعات	قد يكون معقدًا، حمل زائد محتمل	**نفس ملاحظات Hibernate/EF Core؛ يعتمد على تحسين DQL والتخزين المؤقت للأداء**
Django ORM	تكامل ممتاز مع Django، هجرات مدمجة، إنتاجية عالية، واجهة برمجية موجهة للبايثون بامتياز	مرتبط بشدة بـ Django، احتمال مشكلة N+1 أقل مرونة من SQLAlchemy أحيانًا	جيد عموماً؛ الية جلب ال QuerySets الكسولة تساعد
SQL Alchemy	قوي ومرن تحكم دقيق (Core API)	قد يكون معقدًا بسبب المرونة	أداء عالي، خاصة Core API أداء ORM جيد وقابل للضبط
Sequelize	الأكثر استقرار ل node.js غير متزامن قائم على الوعود، دعم الهجرات	واجهة برمجة التطبيقات تعتبر أحيانًا مطولة/معقدة، البدائل الجديدة نوعا ما لاقت رواجا اكثر منه	مقبول عموماً؛ بعض المقارنات تظهره أبطأ من خيارات جديدة مثل Prisma
Prisma	تجربة مطور ممتازة وامن كتابيا، نظام تهجير رائع، عميل مُولد	نموذج مختلف يتطلب تكيفًا بالأخص لبناء ملف التخطيط	يهدف لأداء عالي؛ غالباً ممتاز في المقارنات
TypeORM	يدعم أنماطًا متعددة، قائم على المحددات مما يجعله الأكثر ملائمة كتابيا ل TypeScript	المحددات مثيرة للجدل المرونة تضيف تعقيدًا يوجد تعقيد في واجهته البرمجية	**جيد عموماً؛ الأداء يعتمد على نمط الاستخدام Active Record/Data Mapper والإعدادات**

EF Core	تكامل ممتاز مع LINQ مما يجعله امن كتابيا بالإضافة لأدوات مدمجة متعددة دعم من مايكروسوفت تهجير	يعمل أساسًا في بيئة .NET. قد يبدو معقدًا	أداء جيد جداً ومنافس في الإصدارات الحديثة؛ تحسن كبير عن EF6
Dapper	سريع للغاية، واجهة برمجية خفيفة/بسيطة تحكم كامل في SQL	يتطلب SQL خام (احتمالية الأخطاء)، لا يوجد تتبع تغييرات/وحدة عمل/تخزين مؤقت/تهجير	سرعة قريبة من ADO.NET الأصلي؛ من أسرع خيارات الوصول للبيانات في .NET

جدول 4 ميزات وسلبيات نماذج الربط العلائقي المدروسة مع ملاحظات نظرية عن الاداء

وانطلاقا من التجربة العملية للتنفيذ قد تم استنتاج بعض الملاحظات كالتالي:

1. نموذج **Eclipse Link** لا يفضل استخدامه ابدا نظرا لوجود مشاكل برمجية مزعجة اثناء عملية التطوير باستخدامه.
2. نموذج **Django** أسرع نموذج في جلب معلومات الجداول بشكل مباشر دون أي عملية ربط مع جداول أخرى لكن لا يقوم بالتخزين المؤقت لذلك سرعة التنفيذ لا تتأثر بعدد المرات التي يتم فيها تنفيذ الاستعلام.
3. نموذج **Ktorm** هو الاسهل في الكتابة من خلال واجهة الجداول المقابلة في Kotlin وافضل النماذج في عملية التخزين المؤقت.
4. **Hibernate** هو النموذج الأكثر استقرارا في جافا ويقدم سرعة تنفيذ قريبة واحيانا اسرع من الاتصال التقليدي.
5. **Eloquent** نموذج جيد لكن بيئة عمل Laravel أصبحت قديمة الطراز نسبيا ونتائجه ليست الأفضل.
6. ال **PostgreSQL** لديها مشكلة في الربط مع كل الأطر البرمجية وتحتاج لتسمية كل الصفوف والحقول بمحارف صغيرة ولحل هذه المشكلة بسهولة يجب استخدام ميزة إعادة التسمية في حال كان النموذج المستخدم يوفرها (**Lower Case Naming Convention | Strategy**).
7. كما انها لا تدعم العمليات البسيطة مثل الجمع بين الاعمدة ويتم التعويض عنها بالتوابع الموجودة مثل **concat**.

8. ال **Django ORM** في المدارسة السابقة قامت بتنفيذ تعليمات الجلب بزمن معدوم عند قياسه وذلك كونها تقوم بهذه العملية على مسلك فرعي وليس الرئيسي.

9. **Entity Framework** اسهل نموذج كتابي ويريح المطور من عبء التعامل مع ال SQL بشكل كامل.

10. **Dapper** هو عبارة عن **Micro ORM** أي فقط يستخدم للمقابلة عند عمليات الجلب (صفوف مع قيم بسيطة وليس علاقات بين الجداول على شكل اغراض) ولا يحتوي على أي ميزة لكن يفضل استخدامه لسرعته العالية التي تكافئ الاتصال التقليدي.

11. **Prisma** لا تدعم المقابلة المباشر من خلال الصفوف أي انها تحتاج الى ملف توصيف خاص بها يتم بعد كتابته تحويله باستخدام مفسر لها الى أغراض مكتوبة بلغة ال JavaScript.

12. كما انها لا تدعم التوابع الخاصة ب SQL وكذلك الامر العمليات الحسابية بين الاعمدة لكنها تدعم فقط العمليات التجميعية على عمود واحد مباشرة (مثل **(sum(price_)** لذلك عند الحاجة لأي استعلام شبه معقد يجب استخدام استعلامات مباشرة باستخدام **queryRaw**.

13. **Sequelize** هو نموذج سهل الكتابة ويدعم توابع ال SQL كما يدعم الكتابة المباشر ك SQL بحيث يمكن دمجها مع الاستعلامات المكتوبة فيها أيضا.

14. **TypeORM** ابسط من **Sequelize** ويدعم ال **SubQuery** مع حرية كتابة كبيرة دون الحاجة الى خبرة كبيرة مع ال SQL.

8.4. مقارنة آليات عمل نماذج ORM المدروسة

في ضوء النتائج العملية التي تم التوصل إليها، يصبح من الضروري التعمق في تحليل الجوانب التقنية والنظرية التي توضح أسباب تفاوت أداء نماذج ORM المختلفة. يعنى هذا القسم بدراسة معمقة للبنية المعمارية لكافة النماذج الخاضعة للتحليل، بداية من آليات بناء الاستعلامات، مرورًا بإدارة الكيانات وتحويل النتائج، وانتهاءً بمتطلبات سهولة الاستخدام والتكامل مع لغات البرمجة المختلفة. كما سيتم تسليط الضوء على التحديات التقنية التي تؤثر على الأداء والكفاءة التشغيلية لتلك النماذج، وذلك بتوظيف إطار مقارن يجمع بين النظرية والتطبيق، بما يدعم توجهات اختيار النموذج الأمثل في بيئات التطوير الحديثة.

8.4.1. Hibernate

- **نوع النموذج:** Data Mapper

- **طريقة العمل:** يعتمد على معيار .(Java Persistence API) JPA يستخدم EntityManager لإدارة الكيانات (Entities) ضمن سياق الثبات (Persistence Context) الذي يعمل كوحدة عمل ويتتبع التغييرات. يقوم بربط الجداول بـ (Plain Old Java Objects) POJOs باستخدام التعليقات التوضيحية (Annotations)أو XML . كما يدعم التحميل الكسول (Lazy Loading) والنشط(Eager Loading) والتخزين المؤقت (Caching) للمستوى الأول والثاني.

- **طريقة كتابة الصفوف والتعامل معها** :تُعرّف الكيانات كـ POJOs مع تعليقات JPA التوضيحية (مثل @Entity, @Table, @Id, @Column) وعلاقات مثل (@OneToMany, @ManyToOne) كما يمكن استخدام الوراثة.

- **طريقة تحويل النتائج:** يقوم بتحويل نتائج الاستعلامات تلقائيًا إلى كائنات Java بناءً على الربط المحدد. يدعم استرجاع أجزاء من الكيانات (Projections).

- **طريقة كتابة الاستعلامات**

  - HQL (Hibernate Query Language)
  - JPQL (Java Persistence Query Language)
  - Criteria API: استعلامات باستخدام كائنات مبرمجة بشكل مسبق.
  - SQL خام

- **سهولة الكتابة وضرورة معرفة SQL:** يوفر مستوى تجريد عالٍ يقلل الحاجة لكتابة SQL مباشرة في البداية. منحنى تعلم متوسط إلى مرتفع لاستغلال كافة الميزات. معرفة SQL مفيدة للاستعلامات المعقدة أو تحسين الأداء.

- **سرعة التنفيذ (ملاحظات نظرية):** متوسط كونه يمكن أن يكون هناك حمل زائد (overhead) بسبب التجريد وآلية تتبع التغييرات. التخزين المؤقت (Caching) يمكن أن يحسن الأداء بشكل كبير. الأداء يعتمد بشدة على تصميم الاستعلامات واستخدام نوع التحميل المناسب Lazy/Eager.

```java
@Entity
public class Customer implements Serializable {
    @Id Long custId;
    @Column String companyName;
    @Column String contactName;
    @Column String contactTitle;
    @Column String address;
    @Column String city;
    @Column String region;
    @Column String postalCode;
    @Column String country;
    @Column String phone;
    @Column String mobile;
    @Column String email;
    @Column String fax;
```

```java
public interface CustomerRepository extends CrudRepository<Customer, Long> {
    @Query("""
        SELECT new com.eliasshallouf.msc.seminar2.domain.model.helpers.CustomerWithTotal(
            c.contactName name,
            sum(od.unitPrice * od.quantity * (1 - od.discount)) total
        )
        FROM Customer c
        join SalesOrder so on so.customer.custId = c.custId
        join OrderDetail od on od.order.orderId = so.orderId
        GROUP by c.custId
        ORDER by total DESC
    """)
    List<CustomerWithTotal> getCustomersWithTotals();

    @Query("""
        SELECT new com.eliasshallouf.msc.seminar2.domain.model.helpers.CustomerLivePlace(
            c.country,
            c.city,
            count(c.custId) as count
        )
        FROM Customer c
        GROUP by c.country, c.city
        ORDER by count desc
    """)
    List<CustomerLivePlace> getCustomersLivePlaces();
}
```

الشكل 3 مثال عن طريقة تعريف الكيانات والاستعلامات في Hibernate

8.4.2 EclipseLink

- **نوع النموذج:** Data Mapper
- **طريقة العمل:** تطبيق أخر لمعيار ال JPA، لكن مشابه جدا ل Hibernate لكن يعتبر أحيانا أكثر توافقا مع معيار JPA بشكل صارم مقارنة ب Hibernate الذي يضيف ميزاته الخاصة.
- **طريقة كتابة الصفوف والتعامل معها:** مطابق ل Hibernate
- **طريقة تحويل النتائج:** تحويل تلقائي للنتائج إلى كائنات بناءً على الربط.
- **طريقة كتابة الاستعلامات:** JPQL و Criteria API بشكل أساسي. دعم لـ SQL الخام.
- **سهولة الكتابة وضرورة معرفة SQL:** مشابه لـ Hibernate. مستوى تجريد عالٍ، معرفة SQL مفيدة للحالات المتقدمة.
- **سرعة التنفيذ (ملاحظات نظرية):** مشابه لـ Hibernate نظرياً. قد تختلف الأرقام الفعلية في قياسات الأداء المحددة، لكن كلاهما يقع ضمن فئة ORMs كاملة الميزات مع حمل زائد محتمل مقارنة بالوصول المباشر.

```java
@Entity
public class Customer implements Serializable {
    @Id Long custId;
    @Column String companyName;
    @Column String contactName;
    @Column String contactTitle;
    @Column String address;
    @Column String city;
    @Column String region;
    @Column String postalCode;
    @Column String country;
    @Column String phone;
    @Column String mobile;
    @Column String email;
    @Column String fax;
```

```java
@Repository
public interface CustomerRepository extends JpaRepository<Customer, Long> {
    @Query(value = """
        SELECT c.contactName name, sum(od.unitPrice * od.quantity * (1 - od.discount)) total
        FROM Customer c
        join SalesOrder so on so.customer.custId = c.custId
        join OrderDetail od on od.order.orderId = so.orderId
        GROUP by c.custId
        ORDER by total DESC
    """)
    List<Object[]> getCustomersWithTotals();

    @Query(value = """
        SELECT c.country country, c.city city, count(c.custId) count
        FROM Customer c
        GROUP by c.country, c.city
        ORDER by count desc
    """, nativeQuery = true)
    List<Object[]> getCustomersLivePlaces();
}
```

الشكل 4 مثال عن طريقة تعريف الكيانات والاستعلامات في EclipseLink

8.4.3 Ktorm

- **نوع النموذج:** يمكن اعتباره Lightweight ORM Mapper / DSL-based SQL Builder (ليس نمط Data Mapper أو Active Record تقليدي صارم).
- **طريقة العمل:** يوفر DSL (Domain Specific Language) مكتوب بلغة Kotlin لبناء استعلامات SQL بطريقة آمنة النوع (type-safe) وبرمجية. يركز على بناء الاستعلامات وتحويل النتائج، مع تجريد أقل لوحدات العمل من النماذج السابقة.
- **طريقة كتابة الصفوف والتعامل معها:** يتم تعريف الكيانات عادة كـ data class أو interface في Kotlin. يستخدم Ktorm واجهات لتعريف الجداول وربطها بالكيانات.
- **طريقة تحويل النتائج:** يوفر آليات لتحويل نتائج الاستعلامات المبنية بالـ DSL إلى كائنات Kotlin.
- **طريقة كتابة الاستعلامات:** باستخدام ال DSL الخاص بـ Ktorm في كود Kotlin لبناء استعلامات SELECT, INSERT, UPDATE, DELETE. كما يدعم بناء استعلامات مثل الربط والتجميع والاستعلامات الفرعية.
- **سهولة الكتابة وضرورة معرفة SQL:** مصمم لمطوري Kotlin، يستفيد من ميزات اللغة. يتطلب فهم كيفية بناء الاستعلامات باستخدام DSL. معرفة ال SQL مفيدة لفهم ما يتم توليده لكن ليست ضرورية.
- **سرعة التنفيذ (ملاحظات نظرية):** أداؤه جيدًا، مع حمل زائد أقل من ORMs كاملة الميزات، ولكنه أعلى من استخدام ال JDBC المباشر، بسبب بناء الاستعلامات وتحويل النتائج.

```kotlin
interface Customer : Entity<Customer> {
    companion object : Entity.Factory<Customer>()

    var id: Int
    var companyName: String
    var contactName: String
    var contactTitle: String
    var address: String
    var city: String
    var region: String
    var postalCode: String
    var country: String
    var phone: String
    var mobile: String
    var email: String
    var fax: String
}

object Customers : Table<Customer>("customer") {
    val id = int("custid").primaryKey().bindTo { it.id }
    val companyName = varchar("companyname").bindTo { it.companyName }
    val contactName = varchar("contactname").bindTo { it.contactName }
    val contactTitle = varchar("contacttitle").bindTo { it.contactTitle }
    val address = varchar("address").bindTo { it.address }
    val city = varchar("city").bindTo { it.city }
    val region = varchar("region").bindTo { it.region }
    val postalCode = varchar("postalcode").bindTo { it.postalCode }
    val country = varchar("country").bindTo { it.country }
    val phone = varchar("phone").bindTo { it.phone }
    val mobile = varchar("mobile").bindTo { it.mobile }
    val email = varchar("email").bindTo { it.email }
    val fax = varchar("fax").bindTo { it.fax }
}
```

```kotlin
suspend fun getCustomersWithTotals() = withContext(Dispatchers.IO) {
    val name = Customers.contactName.aliased("name")
    val total = (
            sum(
                OrderDetails.unitPrice
                times
                OrderDetails.quantity
                times
                (1.0 minus OrderDetails.discount)
            )
        ).aliased("total")

    db.from(Customers)
        .leftJoin(SalesOrders, on = it.id.eq(SalesOrders.customer))
        .leftJoin(OrderDetails, on = SalesOrders.id.eq(OrderDetails.order))
        .select(name, total)
        .groupBy(it.id)
        .orderBy(total.desc())
        .map {
            CustomerWithTotal(
                it[name] ?: "",
                it[total] ?: 0.0
            )
        }
}
```

الشكل 5 مثال عن طريقة تعريف الكيانات والاستعلامات في Ktorm

8.4.4 Eloquent

- **نوع النموذج:** Active Record.
- **طريقة العمل:** لكل كائن نموذج (Model) يمثل جدولاً في قاعدة البيانات ويحتوي على المنطق اللازم للتعامل مع هذا الجدول بشكل مبني بشكل مسبق (مثل find(), delete(), save()).

- **طريقة كتابة الصفوف والتعامل معها:** تُعرّف النماذج (Models) كأصناف PHP ترث من *Illuminate\Database\Eloquent\Model*. يتم تعريف العلاقات (Relationships) كدوال داخل النموذج.
- **طريقة تحويل النتائج:** يتم تحويل النتائج تلقائيًا إلى كائنات من صنف النموذج.
- **طريقة كتابة الاستعلامات:** من خلال استخدام دوال ستاتيكية مرتبطة بالكائنات على النموذج نفسه (مثل User::find(1), User::where('active', 1)->get()). يوفر واجهة Query Builder. كما يدعم الـ SQL الخام.
- **سهولة الكتابة وضرورة معرفة SQL:** سهل جداً للبدء والاستخدام خاصة للمهام الشائعة. يتطلب معرفة أقل بـ SQL في البداية. معرفة SQL مفيدة لتحسين الاستعلامات المعقدة.
- **سرعة التنفيذ (ملاحظات نظرية):** متوسط السرعة بالأخص للعمليات البسيطة.

```php
class Customer extends Model
{
    use HasFactory;

    protected $table = 'customer';

    protected $fillable = [
        'custid', 'companyName', 'contactName',
        'contactTitle', 'address', 'city',
        'region', 'postalCode', 'country',
        'phone', 'mobile', 'email', 'fax'
    ];


    public function salesOrder()
    {
        return $this->hasMany(SalesOrder::class, 'custid', 'custid');
    }

    public function customerDemographics()
    {
        return $this->belongsToMany(
            CustomerDemographics::class,
            'custcustdemographics',
            'custid',
            'customertypeid'
        );
    }
}
```

```php
$customerPlaces = Customer::select(
        'country',
        'city',
        DB::raw('count(*) as total_customers')
    )
    ->groupBy('country', 'city')
    ->orderBy('total_customers', 'desc')
    ->get();
```

الشكل 6 مثال عن طريقة تعريف الكيانات والاستعلامات في Eloquent

8.4.5. Doctrine

- **نوع النموذج:** Data Mapper.
- **طريقة العمل:** مشابه لـ Hibernate/EclipseLink. يستخدم EntityManager لإدارة الكيانات. يعتمد على نمط المستودع (Repository pattern) لفصل منطق الاستعلامات. يربط الجداول بـ POPOs (Plain Old PHP Objects) باستخدام التعليقات التوضيحية (Annotations/Attributes) أو XML أو YAML. يدعم التحميل الكسول/النشط والتخزين المؤقت L1/L2.
- **طريقة كتابة الصفوف والتعامل معها:** تُعرّف الكيانات كـ POPOs مع تعليقات توضيحية أو إعدادات أخرى لتحديد الربط والعلاقات.

- **طريقة تحويل النتائج:** تحويل تلقائي للنتائج إلى كائنات بناءً على الربط.
- **طريقة كتابة الاستعلامات:**
  - DQL (Doctrine Query Language)
  - QueryBuilder API
  - SQL خام عبر NativeQuery
- **سهولة الكتابة وضرورة معرفة SQL:** مستوى تجريد عالٍ. يتطلب فهم مفاهيم مثل EntityManager والمستودعات. معرفة أقل بـ SQL مطلوبة للبدء. معرفة SQL مفيدة للحالات المتقدمة.
- **سرعة التنفيذ (ملاحظات نظرية):** مشابه لـ Hibernate/EclipseLink. الأداء يعتمد على الاستعلامات والتخزين المؤقت والتحميل. يمكن أن يكون هناك حمل زائد مقارنة بالوصول المباشر.

```php
#[ORM\Entity(repositoryClass: CustomerRepository::class)]
#[ORM\Table(name: 'customer')]
class Customer
{
    #[ORM\Id]
    #[ORM\GeneratedValue(strategy: 'AUTO')]
    #[ORM\Column(name: 'custId', type: 'bigint')]
    private ?int $custId = null;

    #[ORM\Column(name: 'companyName', type: 'string', length: 255)]
    private string $companyName;

    #[ORM\Column(name: 'contactName', type: 'string', length: 255, nullable: true)]
    private ?string $contactName = null;

    #[ORM\Column(name: 'contactTitle', type: 'string', length: 255, nullable: true)]
    private ?string $contactTitle = null;

    #[ORM\Column(type: 'string', length: 255, nullable: true)]
    private ?string $address = null;

    #[ORM\Column(type: 'string', length: 255, nullable: true)]
    private ?string $city = null;

    #[ORM\Column(type: 'string', length: 255, nullable: true)]
    private ?string $region = null;

    #[ORM\Column(name: 'postalCode', type: 'string', length: 255, nullable: true)]
    private ?string $postalCode = null;

    #[ORM\Column(type: 'string', length: 255, nullable: true)]
    private ?string $country = null;

    #[ORM\Column(type: 'string', length: 255, nullable: true)]
    private ?string $phone = null;

    #[ORM\Column(type: 'string', length: 255, nullable: true)]
    private ?string $mobile = null;
```

```php
$customerRepository = $this->entityManager->getRepository(Customer::class);

$query = $customerRepository->createQueryBuilder('c')
    ->select('c.country, c.city, COUNT(c.custId) as total_customers')
    ->groupBy('c.country, c.city')
    ->orderBy('total_customers', 'DESC')
    ->getQuery();

$customerPlaces = $query->getResult();
```

الشكل 7 مثال عن طريقة تعريف الكيانات والاستعلامات في Doctrine

8.4.6. Django

- **نوع النموذج:** يعتبر قريبًا من Active Record (النماذج تعرف بياناتها وسلوكها، ولكن الاستعلامات تتم غالبًا عبر Manager).
- **طريقة العمل:** يُعرّف كل نموذج (Model) هيكل جدول قاعدة البيانات. يوفر واجهة برمجة تطبيقات (API) عالية المستوى للاستعلامات (QuerySets) التي تعتبر كسولة (lazy) بطبيعتها (لا يتم تنفيذ الاستعلام إلا عند الحاجة للنتائج).
- **طريقة كتابة الصفوف والتعامل معها:** تُعرّف النماذج كأصناف Python ترث من *django.db.models.Model*. تُعرّف الحقول والعلاقات كخاصيات (attributes) على الصنف.
- **طريقة تحويل النتائج:** QuerySets تُرجع كائنات من صنف النموذج عند الطلب.

- **طريقة كتابة الاستعلامات:** باستخدام الـ Manager على النموذج (مثل Model.objects.filter(...), Model.objects.get(...)) لإنشاء QuerySets. دعم لـ SQL الأصلي.
- **سهولة الكتابة وضرورة معرفة SQL:** سهل التعلم والبدء به خاصة ضمن إطار Django. يقلل الحاجة لكتابة SQL بشكل كبير. معرفة SQL مفيدة لتحسين الأداء وفهم الاستعلامات المعقدة
- **سرعة التنفيذ (ملاحظات نظرية):** أداء جيد بشكل عام كون طبيعة الـ QuerySets الكسولة تساعد. يجب الانتباه لتجنب مشاكل N+1 عند استخدام select_related و prefetch_related التي تقوم بتحميل الأغراض المرتبطة بعلاقة مع الجدول الحالي باستخدام المفتاح الأجنبي.

```python
from django.db import models

class Customer(models.Model):
    custId = models.AutoField(primary_key=True, db_column="custid")
    companyName = models.CharField(max_length=40, db_column="companyname")
    contactName = models.CharField(max_length=30, db_column="contactname")
    contactTitle = models.CharField(max_length=30, db_column="contacttitle")
    address = models.CharField(max_length=60)
    city = models.CharField(max_length=15)
    region = models.CharField(max_length=15, blank=True, null=True)
    postalCode = models.CharField(max_length=10, db_column="postalcode")
    country = models.CharField(max_length=15)
    phone = models.CharField(max_length=24)
    email = models.EmailField(max_length=225)
    fax = models.CharField(max_length=24, blank=True, null=True)
    mobile = models.CharField(max_length=24)

    class Meta:
        db_table = 'customer'
```

```python
queryset = Customer.objects.values('country', 'city').annotate(
    count=Count('custId')
).order_by('-count')

results = list(queryset)
```

الشكل 8 مثال عن طريقة تعريف الكيانات والاستعلامات في Django

8.4.7 SQLAlechmy

- **نوع النموذج:** Data Mapper لكنه يوفر تجربة أقرب لـ SQL Builder.
- **طريقة العمل:** يُعرّف كل نموذج (Model) هيكل جدول قاعدة البيانات. كما يوفر بنية اتصال مباشر مع قاعدة البيانات تمثل الجلسة Session حيث تستخدم لبناء تعليمات الـ SQL بشكل برمجي.
- **طريقة كتابة الصفوف والتعامل معها:** تُعرّف النماذج كصفوف Python ترث من الصف *db.Model* حيث db هي غرض منشأ باستخدام الباني SQLAlchemy(). تُعرّف الحقول والعلاقات كخاصيات (attributes) على الصف ك متحولات من النوع db.Column.
- **طريقة تحويل النتائج:** تُرجع كائنات من صنف النموذج.
- **طريقة كتابة الاستعلامات:**
  - باستخدام الجلسة لانشاء تعليمات معقدة مثل الربط db.session.query(...)

- باستخدام صف النموذج بشكل مباشر من اجل عمليات الجلب ذات شروط المقارنة البسيطة Model.query.get(id), Model.query.filter_by(column=value…)
- خام SQL

- **سهولة الكتابة وضرورة معرفة SQL:** يتطلب فهم مفاهيم مثل Session ووحدة العمل. معرفة SQL مفيدة لتحسين الأداء وفهم الاستعلامات المعقدة
- **سرعة التنفيذ (ملاحظات نظرية):** أداءً يقترب من الوصول المباشر. أداء النموذج بشكل عام يعتمد على الاستخدام (مشابه لـ Hibernate/Doctrine).

```python
db = SQLAlchemy()

class Customer(db.Model):
    __tablename__ = 'Customer'
    custId = db.Column(db.Integer, primary_key=True)
    companyName = db.Column(db.String(40))
    contactName = db.Column(db.String(30))
    contactTitle = db.Column(db.String(30))
    address = db.Column(db.String(60))
    city = db.Column(db.String(15))
    region = db.Column(db.String(15), nullable=True)
    postalCode = db.Column(db.String(10))
    country = db.Column(db.String(15))
    phone = db.Column(db.String(24))
    email = db.Column(db.String(225))
    fax = db.Column(db.String(24), nullable=True)
    mobile = db.Column(db.String(24))
```

```python
def count_customers_by_city():
    results = db.session.query(
        Customer.country,
        Customer.city,
        func.count(Customer.custId).label('count')
    ).group_by(Customer.country, Customer.city
    ).order_by(func.count(Customer.custId).desc()
    ).all()
```

الشكل 9 مثال عن طريقة تعريف الكيانات والاستعلامات في SQLAlchemy

Sequelize .8.4.8

- **نوع النموذج:** يعتبر مزيجًا بين Active Record و Data Mapper (النماذج لديها دوال استعلام، ولكن هناك أيضًا تركيز على تعريف النموذج بشكل منفصل).
- **طريقة العمل:** يُعرّف النماذج (Models) التي تمثل الجداول. يوفر دوال على النماذج لإجراء عمليات CRUD والاستعلامات. يعتمد على الوعود (Promises) للعمليات غير المتزامنة (Async).
- **طريقة كتابة الصفوف والتعامل معها:** تُعرّف النماذج باستخدام sequelize.define() مع تحديد الأعمدة وأنواعها والعلاقات.
- **طريقة تحويل النتائج:** يحول النتائج إلى كائنات JavaScript (أو نُسخ من النموذج).
- **طريقة كتابة الاستعلامات:**
  - باستخدام دوال على النموذج (مثل Model.create(), Model.findAll()) كما يوفر خيارات لتحديد الشروط والربط بين الجداول والكتابة ك SQL داخل تعليمات الارجاع برمجيا باستخدام Sequelize.literal().
  - خام SQL.

- **سهولة الكتابة وضرورة معرفة SQL:** يتطلب معرفة SQL لبناء الاستعلامات المعقدة كما انه معقد الكتابة نوعيا.
- **سرعة التنفيذ (ملاحظات نظرية):** أداء مقبول عمومًا، ولكن كانت هناك بعض الانتقادات حول الأداء في بعض الحالات المعقدة أو تحت الحمل العالي مقارنة ببعض البدائل الأحدث.

```
const Customer = (sequelize) => sequelize.define('Customer', {
  custId: {
    type: DataTypes.BIGINT,
    primaryKey: true,
    autoIncrement: true,
    field: 'custid'
  },
  companyName: { ···
  },
  contactName: { ···
  },
  contactTitle: { ···
  },
  address: { ···
  },
  city: { ···
  },
  region: { ···
  },
  postalCode: { ···
  },
  country: { ···
  },
  phone: { ···
  },
  mobile: { ···
  },
  email: { ···
  },
  fax: { ···
  }
}, { tableName: 'customer', timestamps: false });
```

```
(await Customer.findAll({
      attributes: [
          'country',
          'city',
          [Sequelize.fn('count', Sequelize.col('custid')), 'count']
      ],
      group: ['country', 'city'],
      order: [[Sequelize.col('count'), 'DESC']]
})).map(r => r.toJSON())
```

الشكل 10 مثال عن طريقة تعريف الكيانات والاستعلامات في Sequelize

8.4.9 Prisma

- **نوع النموذج:** نوع جديد "Next-generation ORM" (ليس Data Mapper أو Active Record تقليدي).
- **طريقة العمل:** يعتمد على ملف مخطط (Schema file – schema.prisma) يمثل الهيكل لقاعدة البيانات ونماذج التطبيق. يقوم Prisma بتوليد عميل قاعدة بيانات (Prisma Client) آمن النوع (type-safe) بالكامل بناءً على هذا المخطط.
- **طريقة كتابة الصفوف والتعامل معها:** يتم تعريف النماذج في ملف schema.prisma. والعميل المولّد (Prisma Client) يوفر هذه النماذج ككائنات TypeScript/JavaScript.
- **طريقة تحويل النتائج:** Prisma Client يحول النتائج تلقائيًا إلى كائنات JavaScript/TypeScript آمنة النوع.
- **طريقة كتابة الاستعلامات:**

- باستخدام Prisma Client المولّد، والذي يوفر واجهة برمجة تطبيقات (API) واضحة، سلسلة، وآمنة النوع للاستعلامات (مثل (prisma.user.findMany({ where: ... })، حيث prisma هو غرض منشأ من الصف المولد PrismaClient.
- SQL خام.

- **سهولة الكتابة وضرورة معرفة SQL:** يتطلب معرفة SQL لبناء الاستعلامات المعقدة كون الطرق الموجودة فيه محدودة الخيارات كما انه معقد الكتابة بالأخص مرحلة بناء النماذج كونه يعتمد على الكتابة بلغة زائفة خاصة به.
- **سرعة التنفيذ (ملاحظات نظرية):** يهدف إلى أداء عالٍ من خلال تحسين الاستعلامات المولّدة واستخدام محرك استعلام مكتوب بلغة Rust. غالبًا ما يُعتبر أداؤه جيدًا جدًا ومنافسًا قويًا.

```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model Customer {
  custId BigInt @id @default(autoincrement()) @map("custid")
  companyName String @map("companyname")
  contactName String @map("contactname")
  contactTitle String? @map("contacttitle")
  address String? @map("address")
  city String? @map("city")
  region String? @map("region")
  postalCode String? @map("postalcode")
  country String? @map("country")
  phone String? @map("phone")
  mobile String? @map("mobile")
  email String? @map("email")
  fax String? @map("fax")
  SalesOrder SalesOrder[]
  customersDemographics CustomerCustomerDemographics[]

  @@map("customer")
}
```

```
await prisma.customer.groupBy({
  by: ['country', 'city'],
  _count: {
    custId: true
  },
  orderBy: {
    _count: {
      custId: 'desc'
    }
  }
})
```

الشكل 11 مثال عن طريقة تعريف الكيانات والاستعلامات في Prisma

10.3.8. TypeORM

- **نوع النموذج:** يدعم كلا النمطين: Data Mapper و Active Record (يمكن الاختيار)
- **طريقة العمل:** يمكن العمل به بشكل مشابه لـ Hibernate/Doctrine (باستخدام EntityManager والمستودعات – Data Mapper) أو بشكل مشابه لـ Eloquent/Django ORM (حيث تحتوي الكيانات على دوال الحفظ والحذف – Active Record). يستخدم المُزخرفات (Decorators) في TypeScript بشكل مكثف لتعريف الكيانات والربط.
- **طريقة كتابة الصفوف والتعامل معها:** تُعرّف الكيانات كأصناف TypeScript مع مُزخرفات مثل @ ,Entity @PrimaryGeneratedColumn, @Column، وعلاقات مثل @OneToMany.

- **طريقة تحويل النتائج:** تحويل تلقائي للنتائج إلى كائنات TypeScript.
- **طريقة كتابة الاستعلامات:**
  - باستخدام QueryBuilder يتم انشاءه باستخدام مستودع النموذج
  - دوال معرفة مسبقا على النماذج
  - SQL خام
- **سهولة الكتابة وضرورة معرفة SQL:** مرن جدًا ولكنه قد يكون معقدًا بعض الشيء بسبب دعمه لأنماط متعددة وخيارات الإعداد الكثيرة. كما ان معرفة SQL ضرورية في حالة استخدام باني الاستعلامات.
- **سرعة التنفيذ (ملاحظات نظرية):** أداء جيد بشكل عام يعتمد على كيفية كتابة الاستعلامات.

```typescript
@Entity()
export class Customer {
  @PrimaryGeneratedColumn('increment', { type: 'bigint' })
  custId!: number;

  @Column({ type: 'text', nullable: true })
  companyName?: string;

  @Column({ type: 'text', nullable: true })
  contactName?: string;

  @Column({ type: 'text', nullable: true })
  contactTitle?: string;

  @Column({ type: 'text', nullable: true })
  address?: string;

  @Column({ type: 'text', nullable: true })
  city?: string;

  @Column({ type: 'text', nullable: true })
  region?: string;

  @Column({ type: 'text', nullable: true })
  postalCode?: string;

  @Column({ type: 'text', nullable: true })
  country?: string;

  @Column({ type: 'text', nullable: true })
  phone?: string;
```

```typescript
let connection = await dataSource.initialize();

const entityManager = dataSource.manager;

const customerRepository = entityManager.getRepository(Customer);

await customerRepository
    .createQueryBuilder('customer')
    .select('customer.country as country')
    .select('customer.city as city')
    .select('COUNT(customer.custId) as count')
    .groupBy('customer.country, customer.city')
    .orderBy('count', 'DESC')
    .getRawAndEntities();
```

الشكل 12 مثال عن طريقة تعريف الكيانات والاستعلامات في TypeORM

11.3.8. Entity Framework Core

- **نوع النموذج:** Data Mapper.
- **طريقة العمل:** يستخدم DbContext كوحدة عمل (Unit of Work) ومستودع (Repository). يربط الجداول بأصناف (Plain Old CLR Objects) POCO باستخدام تعليقات توضيحية (Data Annotations) أو واجهة برمجة تطبيقات سلسلة (Fluent API) في OnModelCreating. يتتبع التغييرات تلقائيًا. يدعم التحميل الكسول/النشط.
- **طريقة كتابة الصفوف والتعامل معها:** تُعرّف الكيانات كـ POCOs. يتم الربط والعلاقات عبر Data Annotations أو Fluent API أي انه مشابه لطريقة تعريف الصفوف باستخدام معيار ال JPA في Java.
- **طريقة تحويل النتائج:** تحويل تلقائي للنتائج إلى كائنات POCO.

- **طريقة كتابة الاستعلامات:**
  - باستخدام LINQ (Language-Integrated Query) وهي الطريقة الأساسية والمفضلة، توفر استعلامات آمنة النوع ومدمجة باللغة (C# / VB.NET).
  - SQL خام.
- **سهولة الكتابة وضرورة معرفة SQL:** مستوى تجريد عالٍ جدًا بفضل LINQ. يقلل الحاجة لكتابة SQL بشكل كبير. منحنى تعلم متوسط لفهم DbContext و LINQ المتقدم. معرفة SQL مفيدة لتحسين الأداء أو الاستعلامات المعقدة جدًا.
- **سرعة التنفيذ (ملاحظات نظرية):** يعتبر أداؤه الآن جيدًا جدًا ومنافسًا، مع استمرار وجود بعض الحمل الزائد مقارنة بالوصول المباشر.

```
[Table("customer")]
0 references
public class Customer
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    0 references
    public int CustId { get; set; }

    0 references
    public string? CompanyName { get; set; }

    0 references
    public string? ContactName { get; set; }

    0 references
    public string? ContactTitle { get; set; }

    0 references
    public string Address { get; set; }

    0 references
    public string City { get; set; }

    0 references
    public string? Region { get; set; }

    0 references
    public string PostalCode { get; set; }

    0 references
    public string Country { get; set; }

    0 references
    public string Phone { get; set; }
```

```
public List<CustomerLivePlace> GetCustomersLivePlaces() {
    return [.. context.Customers
        .GroupBy(c => new { Country = c.Country, City = c.City })
        .Select(g => new
        {
            g.Key.Country,
            g.Key.City,
            Count = g.Count()
        })
        .OrderByDescending(x => x.Count)
        .Select(g => new CustomerLivePlace
            (
                g.Country,
                g.City,
                g.Count
            )
        )];
}
```

الشكل 13 مثال عن طريقة تعريف الكيانات والاستعلامات في Entity Framework Core

12.3.8. Dapper

- **نوع النموذج:** Micro ORM / Object Mapper (ليس ORM كامل بالمعنى التقليدي).
- **طريقة العمل:** مكتبة بسيطة تركز على شيء واحد: تنفيذ استعلامات SQL وتحويل نتائجها إلى كائنات POCO. لا يوفر تتبع تغييرات تلقائي، ولا وحدة عمل، ولا بناء استعلامات بلغة أخرى غير SQL، ولا تحميل كسول.

- **طريقة كتابة الصفوف والتعامل معها:** تُعرّف الكيانات كـ POCOs بسيطة. لا يتطلب Dapper أي تعليقات توضيحية أو إعدادات ربط معقدة؛ يعتمد على تطابق أسماء الأعمدة مع خصائص (حقول) الكائن (يمكن تخصيصه).
- **طريقة تحويل النتائج:** يقوم بتحويل النتائج بكفاءة عالية جدًا إلى كائنات POCO باستخدام دوال الإضافة (Extension Methods) على IDbConnection.
- **طريقة كتابة الاستعلامات:** كتابة SQL خام (Raw SQL) بشكل مباشر كسلاسل نصية، مع دعم ممتاز للمعاملات (Parameters) للحماية من SQL Injection.
- **سهولة الكتابة وضرورة معرفة SQL:** يتطلب معرفة ممتازة بـ SQL. يوفر تجريدًا بسيطًا جدًا فوق ADO.NET.
- **سرعة التنفيذ (ملاحظات نظرية):** سريع جدًا، أداؤه قريب جدًا من ADO.NET المباشر (مثل JDBC في Java). يعتبر من أسرع طرق الوصول للبيانات في .NET. بعد ADO.NET الخام بسبب قلة الحمل الزائد.

```
public class Customer
{
    0 references
    public int CustId { get; set; }
    0 references
    public string CompanyName { get; set; }
    0 references
    public string ContactName { get; set; }
    0 references
    public string ContactTitle { get; set; }
    0 references
    public string Address { get; set; }
    0 references
    public string City { get; set; }
    0 references
    public string Region { get; set; }
    0 references
    public string PostalCode { get; set; }
    0 references
    public string Country { get; set; }
    0 references
    public string Phone { get; set; }
    0 references
    public string Mobile { get; set; }
    0 references
    public string Email { get; set; }
    0 references
    public string Fax { get; set; }
}
```

```
public List<CustomerLivePlace> GetCustomersLivePlaces()
{
    return (List<CustomerLivePlace>) connection.Query<CustomerLivePlace>(@"
        SELECT c.country, c.city, count(c.custId) as count
        FROM customer c
        GROUP by c.country, c.city
        ORDER by count desc;
    ");
}
```

الشكل 14 مثال عن طريقة تعريف الكيانات والاستعلامات في Dapper

وبشكل ملخص، وبعد إتمام المقارنة العملية بين سرعة أداء كل من النماذج التي تمت دراستها وذكر التفاصيل النظرية لآلية عمل كل منها ونوع النموذج الأساسي فيها ومدى ضرورة الخبرة في لغة الاستعلامات الهيكلية للتعامل معا، سيتم تلخيص الفروقات بينها من حيث الية التصميم والملاحظات النظرية عنها في الجداول التالية:

النموذج	نوع النموذج	الية العمل الاساسية	الية الاستعلامات	الية تعريف الكيانات
Hibernate	Data Mapper (JPA)	EntityManager, Unit of Work, L1/L2 Cache	JPQL / HQL, Criteria API	POJOs + JPA Annotations/XML
Eclipse Link	Data Mapper (JPA)	EntityManager, Unit of Work, L1/L2 Cache	JPQL, Criteria API	POJOs + JPA Annotations/XML
Ktorm	SQL Builder / Lightweight ORM	Kotlin DSL for SQL يركز على الكتابة الامنة واقل تجريد من النماذج الكاملة	Kotlin DSL	Kotlin Data Classes + Table Definitions (Interfaces)
Eloquent	Active Record	نماذج تتضمن منطق الثبات (مثل save())، تعتمد على الاصطلاحات التلقائية وطبقة تجريد قاعدة البيانات DBAL	توابع النموذج QueryBuilder	Models extending base class, Conventions + Properties/Methods
Doctrine	Data Mapper	EntityManager, Unit of Work, L1/L2 Cache	DQL / QueryBuilder	**POJOs + JPA Annotations/XML**
Django ORM	اقرب ل Active Record	النماذج مع دوال حفظ ومدير النماذج مع خاصية objects للاستعلامات	QuerySet API	النماذج ترث صف أساسي والحقول تمثل كخصائص له
SQL Alchemy	Data Mapper	Session (Unit of Work) مقابلة بشكل تصريحي	Session Query / Core API	صفوف ترث صف النموذج الأساسي او صف مع ربط بشكل تصريحي
Sequelize	Active Record / Data Mapper	نماذج تتضمن توابع للحفظ وتوابع ثابتة للاستعلامات	توابع النموذج	عن طريق دالة sequelize.define()

Prisma	Next-gen ORM / Generator	مقاد بملف تخطيط قاعدة البيانات ويُنشئ عميلًا آمنًا للنوع ، يركز على تجربة المطور	Prisma Client API	ملف تخطيط "schema.prisma" يتم الوصول له من خلال صف العميل المولد
TypeORM	Data Mapper / Active Record	EntityManager/Repositories BaseEntity المقابلة باستخدام المحددات اتصالات غير متزامنة	QueryBuilder توابع النماذج	**صفوفTypeScript** +محددات Decorators (@Entity…)
EF Core	Data Mapper	DbContext (Unit of Work, Repository Gateway)	LINQ (Language Integrated Query)	POCOs (Plain Old CLR Object)s + Data Annotations/Fluent API Mapping
Dapper	Micro-ORM	ينفذ استعلامات ال SQL الخام فقط ويقوم بمقابلة النتائج بصفوف برمجية لا يوجد توليد للاستعلامات	SQL خام	POCOs

جدول 5 ملخص عن نماذج الربط العلائقي المدروسة

الخاتمة:

في الختام، تجدر الإشارة إلى أن جميع القياسات والتجارب التي أُجريت في هذا البحث تمت على جهاز حاسوبي بمواصفات فوق المتوسطة، واستخدمت اتصالاً مباشراً إلى قاعدة البيانات عبر مسلك رئيسي واحد دون الاعتماد على تقنيات التخزين المؤقت. كما أن بيئة الاختبار استندت إلى قاعدة بيانات صغيرة نسبيًا من حيث الحجم، مما يجعل النتائج عرضة للتغير إذا ما تم تطبيق نفس التجارب على قواعد بيانات أكثر تعقيدًا وحجمًا أكبر.

مع ذلك، يوفر هذا البحث إطارًا واضحًا وموجهًا للمطورين لاختيار نموذج الربط العلائقي Object Relational Mapping (ORM) الأمثل بناءً على متطلبات وأدوات لغات البرمجة الأشهر حتى تاريخ إعداد الدراسة. إن هذه الدراسة تمكّن من تقديم مقارنة موضوعية وشاملة بين النماذج المختلفة، مما يسهل على أصحاب المشاريع اتخاذ قرارات مدروسة تعزز جودة الأداء وتناسب بيئات التطوير المختلفة.

كما تفتح هذه الورقة الباب أمام توسيع الدراسات المستقبلية لتشمل بيئات بيانات أضخم وأكثر تعقيدًا، مع تضمين تأثيرات التخزين المؤقت، التوزيع الشبكي، وتعقيدات الاستعلامات، إضافة إلى اختبار النماذج في ظروف استخدام واقعية ومتنوعة. مما يسهم في تعزيز فهمنا لأداء نماذج ORM في سيناريوهات عملية متعددة ويؤهل المطورين لاتخاذ قرارات أكثر دقة وفاعلية في تطوير برمجياتهم.

المراجع:

[1] Alexandre Torres, Renata Galantea, Marcelo S. Pimentaa, Alexandre Jonatan B. Martins. (2017) - **Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design**. Information and Software Technology 82 (2017) 1 –18.

[2] Doina Zmaranda, Lucian-Laurentiu Pop-Fele, Cornelia Győrödi, Robert Győrödi, George Pecherle. (2020) - **Performance Comparison of CRUD Methods using NET Object Relational Mappers : A Case Study**. Department of Computer Science and Information Technology, University of Oradea, Oradea, Romania1.

[3] Edy Budiman, Muh Jamil, Ummul Hairah, Hario Jati, Rosmasari. (2017) - **Eloquent Object Relational Mapping Models for Biodiversity Information System**. International Conference on Computer Applications and Information Processing Technology (CAIPT).

[4] Gregory Vial. (2019) - **Lessons in Persisting Object Data using Object-Relational Mapping**. IEEE Software (Volume: 36, Issue: 6, Nov.- Dec. 2019).

[5] Bernard Che Longho. (2022) - **Room vs. greenDAO for Android : A comparative analysis of the performance of Room and greenDAO**. Mittuniversitetet MID Sweden University.

[6] Magdalena Borys, Beata Panczyk. (2015) – **Improving data processing performance for web applications using object–relational mapping**. Actual problems of economics #2(164).

[7] Bruno Baldez Correa, Yue Wang. (2017) – **Object–relational mapping tools and Hibernate**. University libre de Bruxelles.

[8] Shoaib Mahmood Bhatti, Zahid Hussain Abro, Farzana Rauf Abor. (2013) – **Performance Evaluation of Java Based Object Relational Mapping Tools**. Mehran University Research Journal of Engineering & Technology, Volume 32, No. 2, April.