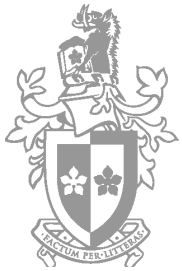
An assessment of performance and operability of ORM and SQL querying in Document-Oriented, Object-Relational and Relational DBMSs

COS30009–Database Programming

Swinburne University of Technology

Alex Cummaudo 1744070

Semester 2, 2014



Abstract

Database programmers have a variety of tools at their fingertips. Which of these tools provide the greatest operability and flexibility? How well do these tools operate with high loads of data, and is there an adverse affect to performance for the sake of greater ease of programming? What kinds of queries on the database perform better than others? By developing a simple database testing tool over MySQL, PostgreSQL and MongoDb DBMSs, differently sized datasets were tested under different query patterns, which demonstrated poor performance and operability with MongoDb against MySQL or PostgreSQL, though ORM for MySQL and PostgreSQL tended to perform better than raw SQL queries. However, such a varied distribution of results concluded that performance will vary over dataset size.

Contents

1 Introduction

1.1 Background

Over recent years, a number of database technologies have come about that have consistently the way in which programmers can utilise databases in their programs. These have been optimised over the years, from initial Relational Database Management Systems such as MySQL developed in the late 1970s, to NoSQL, Document-Based data stores such as MongoDb almost thirty years later in the late 2000s.

While most of the different approaches in database management have a fundamental goal, to maintain the integrity and state of the data they contain, there are multiple factors that distinguish most from each-other.

1.1.1 Data Store Technology

Throughout this research report, Relational Database Management Systems, Object-Relational Database Management Systems and NoSQL (Document-Oriented) Data Stores will be investigated. The different database management technologies bring with them their own advantages and disadvantages, as Sabău (2007) and Vaish (2013) elaborate:

Relational Database Management Systems (RDBMS)

- RDBMSs strive at separating data from application programs. Data exists and can be modified independently of the way they are used in applications.
- RDBMSs store data only, as well as stored procedures, unlike ORDBMSs which store objects in the traditional encapsulation sense (data *and* methods).
- Relatively simple via rows, tulles, and tables.
- Complex joins help ensure data tables remain independent from each-other, but are still relateable.
- Performance is usually related directly to the complexity of the data structure.

Object-Relational Database Management Systems (ORDBMS)

- ORDBMSs are an extension of the RDBMS, with a mechanism that provides object encapsulation.
- ORDBMSs store objects, including their methods; a direct mapping to the way in which the program uses the data (i.e., a tight coupling between application use and the database).
- Extends the SQL by allowing OO-features to be included within the database, such as BLOBs (Binary Large Objects).
- Relationships between objects can be naturally accessed; rather than using joins between several tables, simply use dot-notation syntax and the ORDBMS understands: e.g., `customer.address.city` would translate to an appropriate match of `INNER JOIN`s
- Useful in applications that process a large number of ad-hoc queries on data items which can have a complex data structure.

NoSQL (Document-Oriented) Data Stores

- While the term NoSQL has grown from the simple "I do not want to use SQL to access my Database", NoSQL systems do *not* follow traditional RDBMS principles, chiefly violating the constraints set by the ACID[1] nature.
- Data stored in a NoSQL data store is non-relational, and does not use SQL as a query language. Instead, functions can be declared using simple scripting languages such as JavaScript to query data.
- While the focus on (O)RBDMS's is ACID, the focus in NoSQL data stores is BASE (Vaish, 2013, p.10), a term which focuses on:
  - *Basic Availability*—Regardless of success or failure, all requests to the data store receive a response.

[1] Atomicity, Consistency, Isolation and Durability; a set of "tightly related criteria that a well-behaved transaction processing system must meet... with regards to completing transactions atomically (single-units of work), consistently (error-safe) isolated (transaction-independence) and durable (permanently changeable)." (Zaitsev et al., 2008)

  - *Soft State*—System state can change over time, even without any input.
  - *Eventual Consistency*—The data store can be inconsistent momentarily, so long as it will become consistent eventually.

- NoSQL data stores are schemaless; a data entity does not have to be set in stone before it is persisted.

- Since there are no relational joins in a NoSQL data store, Vaish (2013, p.21) suggests that "one simple and logical solution is to–at times–duplicate the data across the tables".

1.1.2 Data Access Technology

Programmers have a mix of data access techniques at their disposal to map their application's entities to a database. Direct SQL injection within applications on a database connection has been improved upon using prepared statements. The prepared statement preprocessed statement handler to provide a *performance boost*; the "server need only analyse the statement once, not each time it is executed" (DuBois, 2013, pp.377-378), whereby the database can store fewer query-plans while still providing the client with as many, reusable queries (by the form of parameters) without ever hindering on database performance.

This said, newer technologies such as ORM (Object-Relational Mapping) has become very appealing to developers due to its flexible nature with the data store it sources its data from and its auto-generated schemas. However, performance-related issues have arisen that begins to question the use of ORM technologies in large-scale applications, which Zaitsev et al. discusses:

> "The [ORM] design may appeal to developers, because it lets them work in an object-oriented fashion without needing to think about how the data is stored. However, applications that "hide complexity from developers" usually don't scale very well... think carefully before trading performance for developer productivity, and always test on a realistically large dataset, so you don't discover performance problems too late."
>
> - (Zaitsev et al., 2008, p.96)

Hence, a division between the operability of the code that developers work with and the scalability of database performance exists. This is especially the case for a divide between using ORM and not using ORM, and is a topic which this research report will investigate (refer to Section 1.3).

1.2 Mapping Technology

For the purposes of this research article, the following technologies will be mapped with each other during performance tests to determine any noticeable performance and operability affects:

- **ORM**: *NoSQL Data Stores and ORDBMSs.*
  Both technologies each allow for a direct mapping of objects between the data store and the applications by which they are used in. Either by using schemas (ORDBMS) or direct manipulation on the data store (NoSQL), these ORM framework will help assist the link between data store and application-domain, and is therefore most suitable between these technologies.

- **Direct SQL Querying:** *RDBMSs and ORDBMSs*
  NoSQL data stores, as the name suggests, does not query its data with the SQL. Relational database management systems, whether they be object-oriented or otherwise, can accept SQL queries in order to select specific kinds of datasets and scalar query values (e.g., counts, summations and so on). This therefore excludes the NoSQL data store technology, thereby leaving the relational database management tools.

1.3 Aims and Goals

By first defining what the technologies are and how they differ, there now exists two primary questions to answer, which can then be divided further:

1. How does the performance of the technologies differ:
   (a) over a different DBMS?
   (b) over a different dataset size?

(c) when querying for returning datasets using joins, or just scalar querying?

2. What is the operability of the technologies?

   (a) What is the syntactical difference when using a SQL-based Data Access Layer?

   (b) How does this differ from using a ORM-based Data Model Layer?

These aims help reveal what happens to programming operability when the syntactical elements of database entities change, as well as the difference in selection performance of both kinds of technology.

1.4 Research Methodology

1.4.1 Consideration Factors

The performance of variant query selection of data in large, medium and small data sets will help indicate which mix of technologies handle best with differently sized datasets, using different queries to assess a range of selection criteria. The selection of different types of queries will be considered, such as:

- a selection of all entities in a given model,
- a scalar query for counting or summing values in the data store,
- a selection of a combination of relatable entities over various models,
- a calculation of values over a combination of relatable entities over multiple models, and
- a top 3 selection of the highest calculations in the aforementioned calculation query.

These queries will not only assess how well the group of technologies code performance-wise, but will also give an indication as to how *easy* it is to create these queries, run them and assess their results from an operability perspective. For example, if it is easier for developers to just "map" their objects to a persistent state (i.e., by letting the database handle the persistent-enabling technology), then this assumes that the implications on *retrieving* and *using* those values over the queries above must be as easy as it is to create and persist those objects.

Due to the limitations (see Section 4.2) of this research article, the scope of performance handling will be limited to querying only, and will exclude updating and insertion performance and operability factors.

1.4.2 Technology Selection

Using MongoDb, MySQL and PostgreSQL MySQL is a particularly popular RBDMS since it is free, well supported by the open source community and is easy to install and use on almost all systems. MySQL is both a contender for ORM and Direct SQL manipulation as many development platforms provide a library to its database adapter interface. In addition, PostgreSQL is free ORDBMS, supported by the open-source community, that offer many of the same features that MySQL does.

However, "PostgreSQL users claim it's faster and more stable and offers more features... PostgreSQL follows ISO SQL standards more correctly whereas MySQL is more pragmatic" (Cooper, 2007, p.271). Hence, the validity of this statement from PostgreSQL can also be examined, in addition to its operability with ORM and SQL manipulation.

Lastly, MongoDb, is one of the most leading popular document-oriented NoSQL data stores. As Vaish (2013, p.31) describes, its schemaless creation of entities prove to be "very useful in web-based applications where there is a need for storing different types of content that may evolve over time", never needing a database to be 'created' before data is used as underlying *collections* of entities are made on the fly—eliminating many of the errors often faced when inserting new data into traditional RDBMSs.

MySQL, PostgreSQL and MongoDb fully support database indexing[2], which will significantly improve the performance of querying from the get-go.

Using Ruby on Rails Widely-used RDBMS systems such as MySQL, PostgreSQL, SQLite and Oracle are all fully-operable with the Ruby programming language, without the need for third-party tools or technologies that may contend with each other. The Ruby on Rails framework comes with a ORM framework, *ActiveRecord*, which "ties database tables to classes [by] working as you would any other Ruby object and all changes will be stored in the relevant database table automatically." (Cooper, 2007, p.390). The automatic mapping is achieved by on-the-fly generation of SQL statements and queries that can ease the oper-

[2]Specifically, indexing built on a B-Tree structure (DuBois, 2013, pp.98-99)

ability of selection, insertion and updates of Ruby objects to their persistent state on the database.

The third-party ORM framework, *MongoMapper*, can co-exist with ActiveRecord classes in order to provide the same functionality that ActiveRecord does but to a MongoDb database (obviously, without using any SQL). It uses a similar method to achieve this functionality, but using the MongoDb aggregation and selection querying language to achieve much of the same functionality.

As such, the suitability of Ruby on Rails will provide quick, iterative development on the test criteria needed to satisfy the consideration factors defined in Section 1.4.1 above. It is therefore a suitable candidate for the purposes of this report's primary language-base.

1.4.3 Metric Selection

Selection of dataset sizes A definition on the *sizes* of datasets that need to be produced will be indicative of the relative performance differences that will be tested among the different technologies. As such, the dataset sizes themselves should be defined with using a 5^n exponential scaling approach, whereby, using a base a dataset size of x entities:

- a small dataset would be defined with a scale factor $5x^1$,
- a medium dataset would be defined with a scale factor $5x^2$, and
- a large dataset would be defined with a scale factor $5x^3$.

This exponential approach will scale data to increasingly proportions, thereby giving a good factor as to how database performance will differ over variant dataset sizes.

Measuring performance Performance metrics will be defined as the difference in speed to complete query tests when operating with the datasets within a query test suite. This will be measured using the Ruby *Benchmark* class, which will observe the real time taken to run a test in milliseconds. To cater for noticeable affects in performance, any outliers in the results will factored out, allowing for a direct contrast between non-outlying data for better performance indicators.

Measuring operability To observe how the operability of code differs between ORM and direct SQL querying, the count of code lines that is required in both the ORM data models and SQL data access layers will be used as a measure for ease of writing code. In addition to this, direct contrasts between code flexibility in both ActiveRecord and MongoMapper ORM classes, and seeing if this is easier or harder than writing raw SQL statements instead. This will help determine how the code operates, and whether any additional increases in performance are hindered by 'uglier' code in any of the mixed technology used.

2 Method

2.1 Case Study

For the purposes of this research, a simple case study involving a typical store with orders, customers and items to be purchased will be implemented, that is:

- A store has a number of products,
- A product can be purchased multiple times,
- A customer purchases a product, recorded as a purchase.

This model will help assess different types of queries (see Section 2.3) as will also be easily scalable for the variant data sets that are to be tested, according to the data sets scalability factors as determined in Section 1.4.3.

As each of the entities can be easily mapped to both table schemas as well as Object-Oriented classes, both SQL and ORM technologies can be tested with ease on this case study, whereby each field can be mapped to either a property of the class, a column of the table or a field in the JSON files. An entity relationship diagram of this case study is illustrated in Figure 2.1.

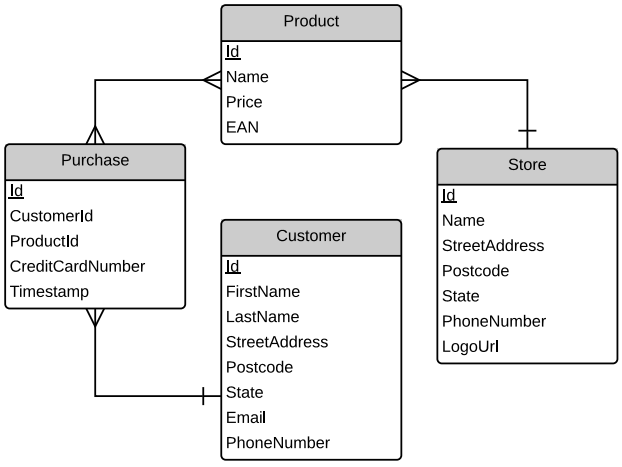


Figure 2.1: An ERD of the case study to be used for the test cases.

2.2 Database Population

Data populating the database will be implemented using the Ruby *Faker* class, which can generate random information, including product names and prices, business details, customer details and so on. This class suits the use case very well, and therefore can be easily included into a Database Population Script (see Section A.4 for the script implementation).

To ensure that data is populated with the *same* information between each of the three DBMSs assessed, it must be ensured that the randomisation seed is set as a constant depending on the dataset size to be used. If not, there may be a bias inside the actual data that exists within either the MySQL, PostgreSQL or MongoDb databases—this discrepancy will be amended by enforcing a randomisation seed in the population script itself.

For the purposes of this research, the seeds of each of the three databases were set by the scale factor mentioned in Section 1.4.3, that is:

- For the **Small Dataset**, the seed is set using `Random.srand(`5^1`)`
- For the **Medium Dataset**, the seed is set using `Random.srand(`5^2`)`
- For the **Large Dataset**, the seed is set using `Random.srand(`5^3`)`

2.3 Test Cases

To test different types of queries, the following test cases have been developed that will examine how the performance varies when the query type changes:

- Fetching of datasets
- Counting sizes of datasets
- Complex, case-study specific queries (i.e., calculated queries)

The first set of queries simply fetch all records that exist in each entity. This is equivalent to a `SELECT *` SQL query or an `Entity.all` MongoMapper or ActiveRecord class method.

Similarly, the second query type of queries (a scalar count query) will simply return the count, rather than the records themselves. This is equivelant to a `SELECT COUNT(*)` SQL

query or an `Entity.count` MongoMapper or ActiveRecord class method on the entity being counted.

The more complex queries to be developed are summarised below. The SQL query case, MongoMapper code and ActiveRecord code that implement these test cases are given in Section 3.

Store Revenue A store's revenue can be calculated from the sum of all the products that have been purchased from this store (i.e., from the purchases entity). This test case also applies to all store revenue as well, where there is no clause that limits the calculation to a specific clause.

Customer Expenditure The amount a customer has spent over all stores as well as at a specific store will also be tested.

Top Spenders The biggest spenders over all stores, as well as at a specific store, can be calculated from the customer expenditure of a number of stores. This can be limited to, say, the top three spenders.

2.4 Implementation Specifics

2.4.1 Benchmarking

To test performance benchmarking, the Ruby *Benchmark* class is required in the test case classes. All test cases inherit from an abstract `TestExec` class (refer to Section A), by which all test case methods themselves are marked as `private` and begin with the suffix `t_`. From this, a `run_tests` method is used to run all the test cases in a subclassed test executor, which establishes an appropriate connection to the database, executes the method under a benchmark method, and returns the time results. Listing 1 shows the code developed to run these tests below.

Listing 1: This methods runs all test case methods marked in the class under a Ruby benchmark time analysis to assess how long the method takes to complete.

```
#
# Executes all tests in this executor
#
```

```
  def run_tests
    # Disable the logger
    old_logger = ActiveRecord::Base.logger
    ActiveRecord::Base.logger = nil

    # Ensure we're connected to the right database
    DataAccess::Connector.setup(@dbms, @dataset)

    ret = {}

    # Execute all private methods
    private_methods(false).each do | method |
      next if not method.to_s.starts_with?("t_")
      result = nil
      puts "[#{@dbms}/#{@dataset}]\tRunning Test:\t#{method}"
      # Time how long (in ms) it takes to execute the test using benchmark
      time = (Benchmark.realtime { result = self.send(method) } * 1000).round(4)
      puts "[#{@dbms}/#{@dataset}]\tTest Ended:\t#{method} in #{time}ms"
      # remove the t_
      ret[method.to_s[2..-1]] = time
    end

    # Renable the logger
    ActiveRecord::Base.logger = old_logger

    # Return the result of the method
    ret
  end
end
```

2.4.2 SQL Operability

To improve the SQL operability of prepared statements in the test cases, a `DataAcess` module class, `Sql`, was developed to improve the ease at which prepared statements can be executed on any SQL-supporting database. This class implements an `exec` method, which can take in any SQL query, with associated query parameters (i.e., bind variables).

The code for this method checks for query parameters, and ensures that query parameters are optimised for either a MySQL connection or PostgreSQL connection—that is, query text contains a question-mark style parameter input:

```
SELECT * FROM Foo WHERE id = ?  AND cost > ?
```

The method will then *automatically* replace these question marks to a PostgreSQL standard:

```
SELECT * FROM Foo WHERE id = $1 AND cost > $2.
```

The code that implements this functionality is provided in Listing 2 below.

Listing 2: Execution of ActiveRecord SQL

```
#
# Execute a raw sql query via ActiveRecord. If providing sql parameters, use
# the MySQL standard for query parameters (i.e., SELECT * FROM table WHERE id = ?)
# If the dbms is Postgres, these values will be converted into appropriate Postgres
# instead (i.e., SELECT * FROM table WHERE id = ? becomes WHERE id = $1)
#
def self.exec(sql, params = nil)
  if Connector.dbms == :mongo
    raise "Can only be executed when the current dbms is not a NoSQL database"
  end

  # No parameters?
  if params.nil?
    ret = ActiveRecord::Base.connection.execute(sql)
    # PostgreSQL conversion from PG::Result to Array
    if ret.is_a? PG::Result
      return ret.values
    else
      return ret.to_a
    end
  # Else need to make a prepared statement
  else
    # MySQL Prepared Statements
    if Connector.dbms == :mysql
      return self.mysql_prepared_statement(sql, params)
    # PostgreSQL Prepared Statement
    elsif Connector.dbms == :postgresql
      # convert all the ? to $x for compatibility with postgresql
      sql = sql.gsub(/\s\?([\s\,\n\;])?/).with_index { | m, i | " $#{i+1}#{$1}" }
      return self.postgresql_prepared_statement(sql, params)
    end
  end
end
```

Also, note the slight difference in MySQL and PostgreSQL in using prepared statements; while MySQL must call a prepare method on the connection itself, PostgreSQL simply executes this in a once-off process. This is contrasted between Listings 3 and 4 below.

Listing 3: MySQL Prepared Statement Execution

```
#
# Execute a MySQL prepared statement
#
def self.mysql_prepared_statement(sql, params)
  self.check_prepared_statement(sql, params, /\s\?[\s\,\n\;]?/)
  pstmt = ActiveRecord::Base.connection.raw_connection.prepare(sql)
  pstmt.execute(*params).to_a
end
```

Listing 4: PostgreSQL Prepared Statement Execution.

```
#
# Execute a PostgreSQL prepared statement
#
def self.postgresql_prepared_statement(sql, params)
  self.check_prepared_statement(sql, params, /\s\$\d[\s\,\n\;]?/)
  ActiveRecord::Base.connection.raw_connection.exec_params(sql, params).values.to_a
end
```

3 Results

3.1 Performance Results

3.1.1 Dataset Querying

Figures 3.1 and 3.2 illustrate the time taken to select all of the customers over each dataset. As shown, there is a clear outlier with the large MongoDb dataset, which exponentially grows over the three sets, and is greatly outperformed by all other database technologies.

As such, Figure 3.2—which excludes MongoDb altogether—shows that the best selection technology is both ORM implementations with a significant improved performance over the SQL query, while out of the two SQL-only queries, MySQL outperforms PostgreSQL. The exponential rise in time is consistent with the increased dataset sizes, which is expected.

The best performer for this query is the PostgreSQL ORM implementation, with a consistent 0.031ms over both the small and medium and large datasets, with an increase to just 0.1ms in the larger dataset.

Figures 3.3 and 3.4 are consistent with the results found for the Customers dataset selection—MongoDb has once again been drastically outperformed by its Relational Database counterparts MySQL and PostgreSQL, leading to the non-outlier graph of Figure 3.4. This second graph is also consistent with the Customer results; ORM implementations are faster than SQL-only implementations, with PostgreSQL being the fastest out of MySQL and PostgreSQL.

Purchases, being the largest dataset in the entire case study, took the longest to load (as expected). Results shown in Figures 3.5 and 3.6 illustrate the query time for loading all purchases. Results essentially follow the same pattern as per Customer and Products, however, interestingly, the MySQL outperformed or equalled the PostgreSQL ORM loading

times, and is therefore the most optimal in this case.

Lastly, loading in all stores (Figures 3.7 and 3.8) follows the pattern which has been discovered over loading all of the data entities; essentially MongoDb is outperformed in all cases by a significant proportion, and the performance decreases exponentially over all dataset sizes. In the stores, the fastest response times came from PostgreSQL ORM implementations of test cases.

3.1.2 Scalar Querying (Counts)

In the counting of dataset sizes, no MongoDb outliers were found; in fact, over Figures 3.9 to 3.12, the following tends were noted:

- MongoDb was often the fastest for scalar count selection, often performing in under 1ms
- MySQL under raw SQL querying was usually the second fastest, however there are some notable outliers in large datasets sizes for product counts, as well as in all purchases, where other technologies often were quicker.

It is notable that out of the ORM technologies, PostgreSQL was often faster, though there are some exceptions for customer counting.

3.1.3 Scalar Querying (Calculations)

Calculated queries were often the most slow out of all of the technologies assessed. MongoDb was the slowest to perform in all of the calculated queries.Figure 3.13 was the only result that showed that there was no *significant* outlier in MongoDb; ORM PostgresQL was often the fastest to calculate a store's revenue in the larger datasets, while MySQL ORM was faster in smaller datasets. An interesting exception to previous trends was that PostgreSQL under SQL spiked up in poor performance for only medium datasets, and then reduced back down to normal ranges in the large dataset.

When assessing *all* of the store's revenue, sorted, MongoDb was again showing outlying data of significant proportions; Figure 3.14 shows this outlier, with the outlier being removed in Figure 3.15. When the outlier is removed, an interesting observation is found; PostgreSQL under ORM is the *slowest* in all cases (unlike the previous test). In fact, all ORM tests are outperformed by SQL-only tests, which are significantly faster (for MySQL especially).

When calculating a single customer's expenditure (Figures 3.16 and 3.19), the outlier show a dramatic outperformance in the large dataset; scaling up by a factor of approximately 33. Removing this outlier shows that the medium dataset spike in PostgreSQL under SQL still exists. This said, the MySQL under SQL is the *fastest* of all technologies, with the ORM implementations having their fastest performance under PostgreSQL. These results show a similar pattern with the top spender category of tests (refer to Figures 3.20 and 3.21).

The only other significant discrepancy is for the sorted top spenders at a single store; SQL-only tests prevail over all other tests, especially under MySQL. All ORM tests perform very poorly, with the MongoDb version being the worst (as usual) and the PostgreSQL and MySQL ORM coming in second best for performance.

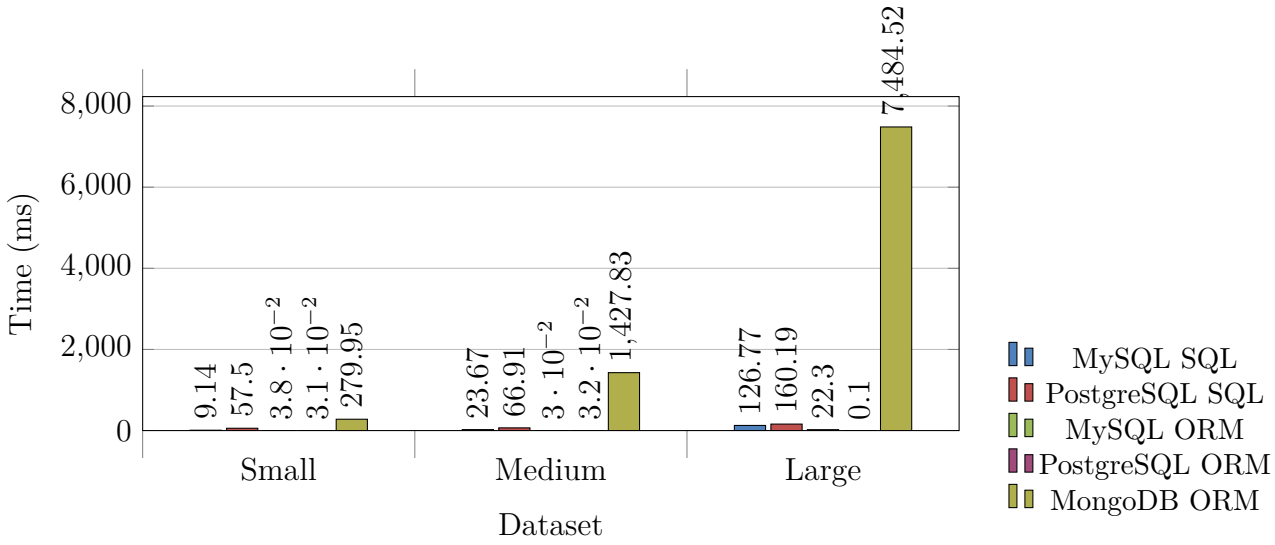


Figure 3.1 – Query time of all customers.(Test Identifier: `t_all_customers`)

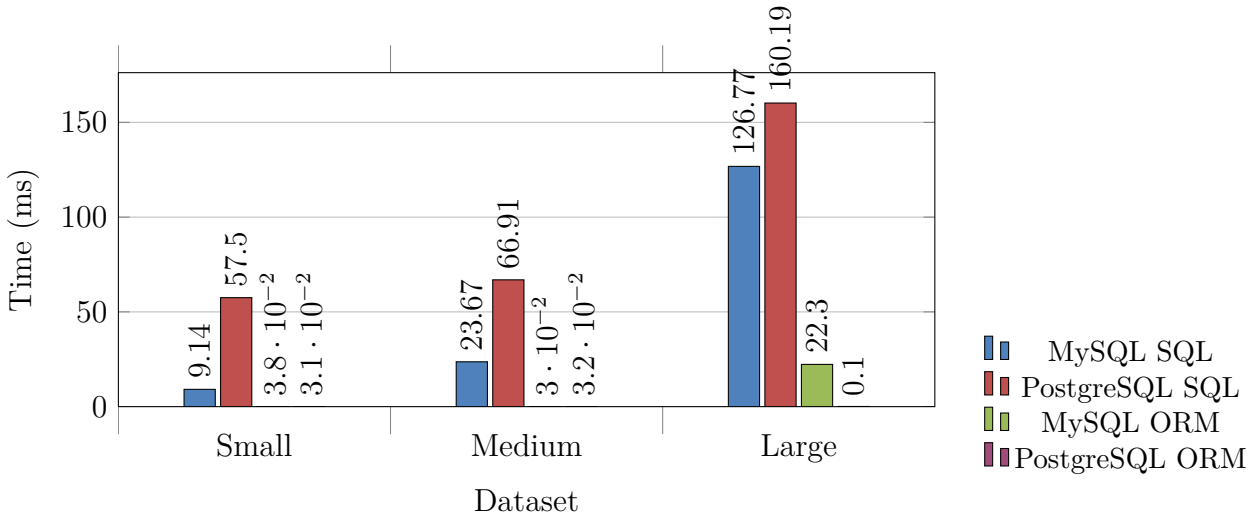


Figure 3.2 – Query time of all customers. **Excludes MongoDb outlier**.
(Test Identifier: `t_all_customers`)

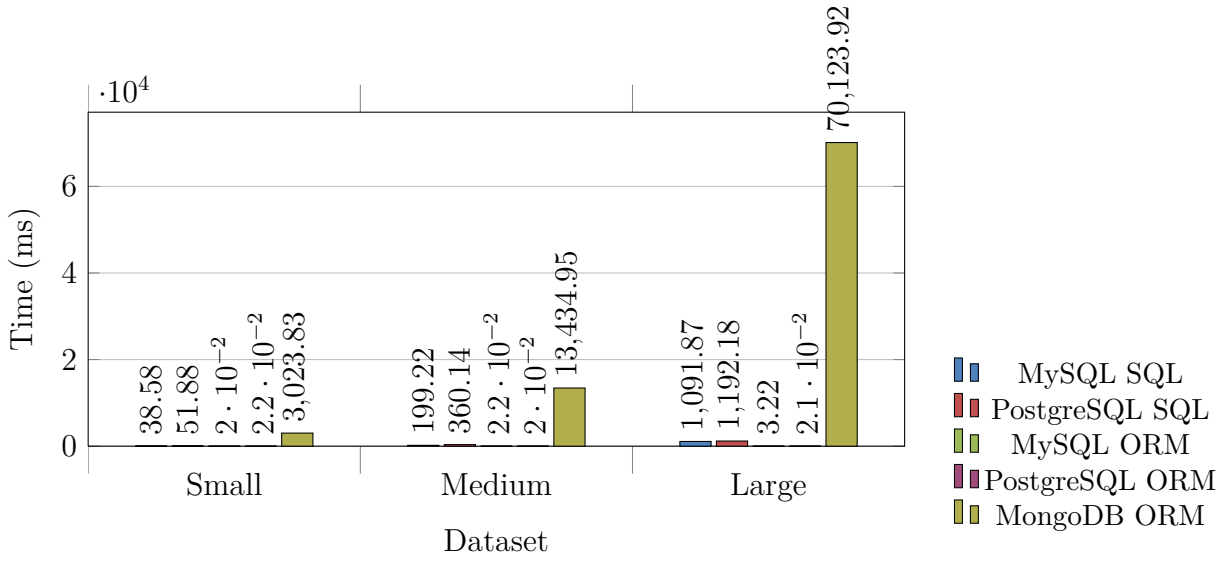


Figure 3.3 – Query time of all products.
(Test Identifier: `t_all_products`)

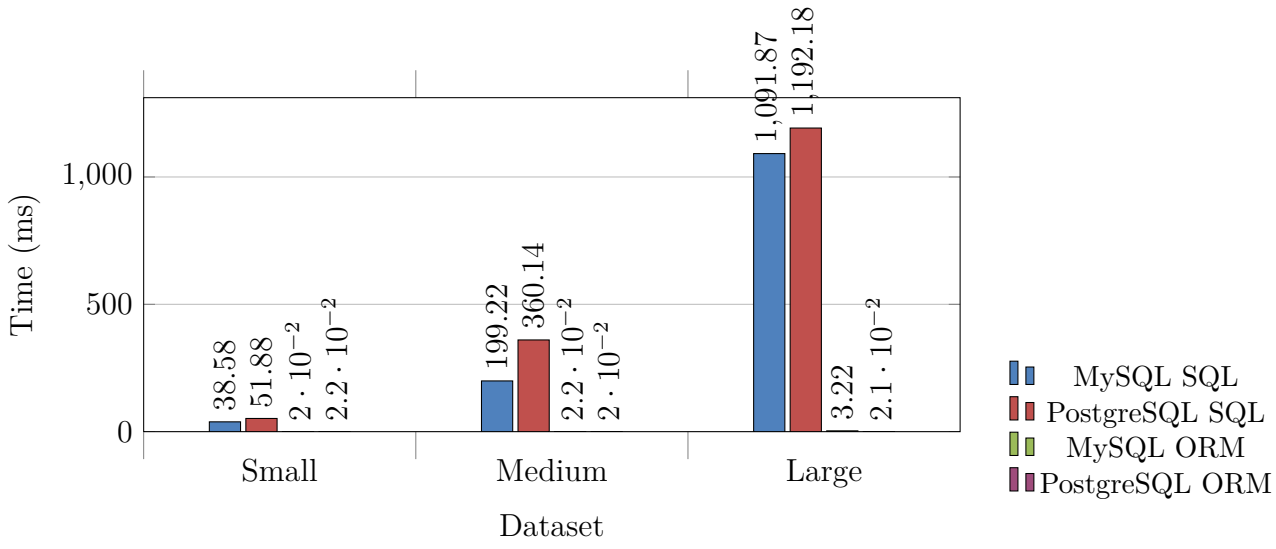


Figure 3.4 – Query time of all products. **Excludes MongoDb outlier**.
(Test Identifier: `t_all_products`)

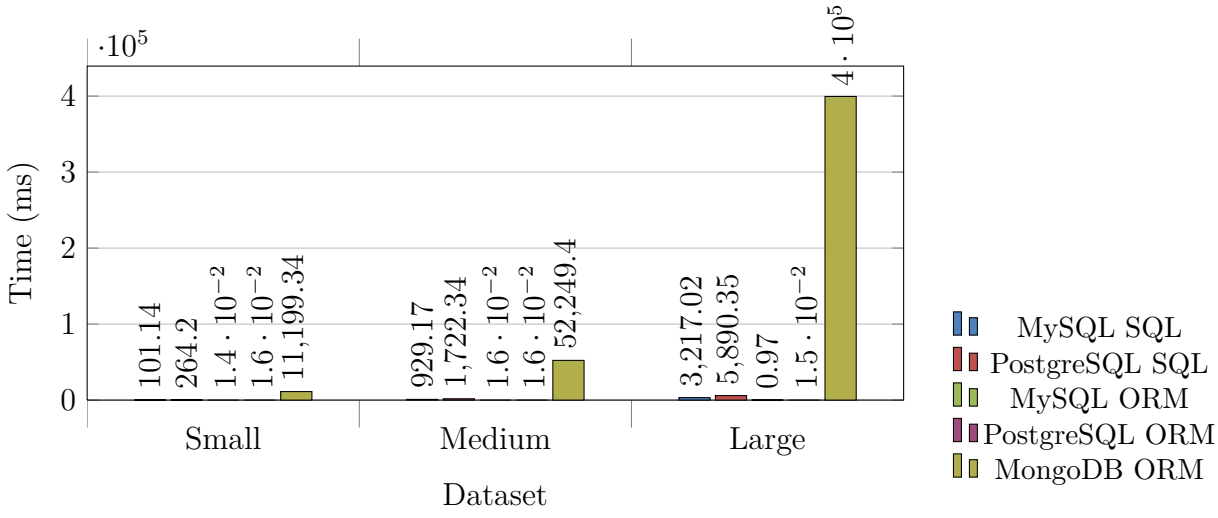


Figure 3.5 – Query time of all purchases.
(Test Identifier: `t_all_purchases`)

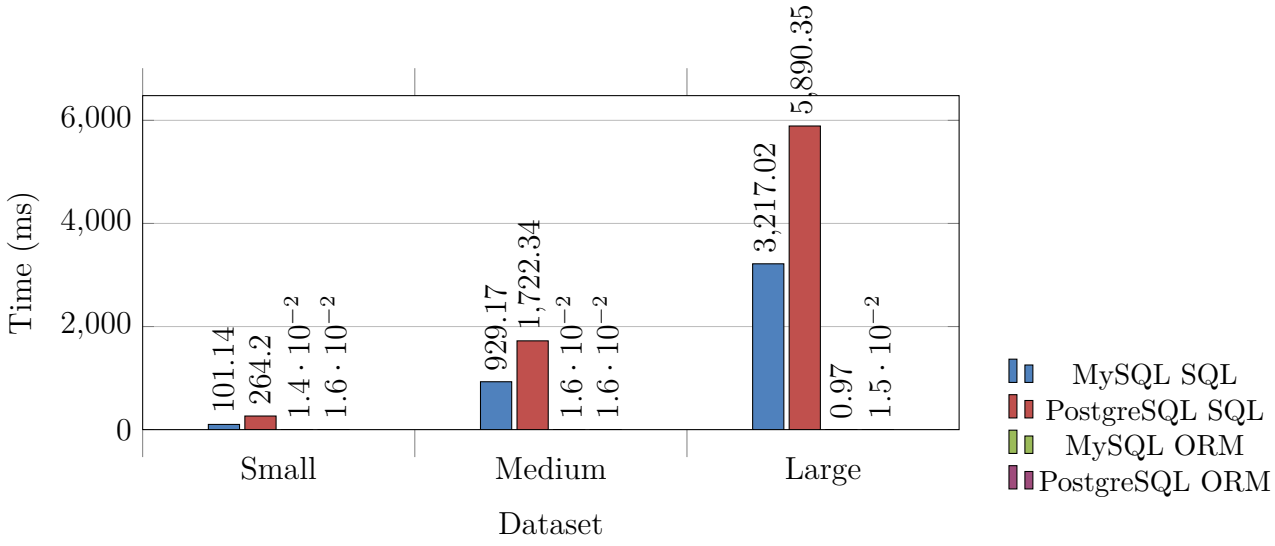


Figure 3.6 – Query time of all purchases. **Excludes MongoDb outlier**.
(Test Identifier: `t_all_purchases`)

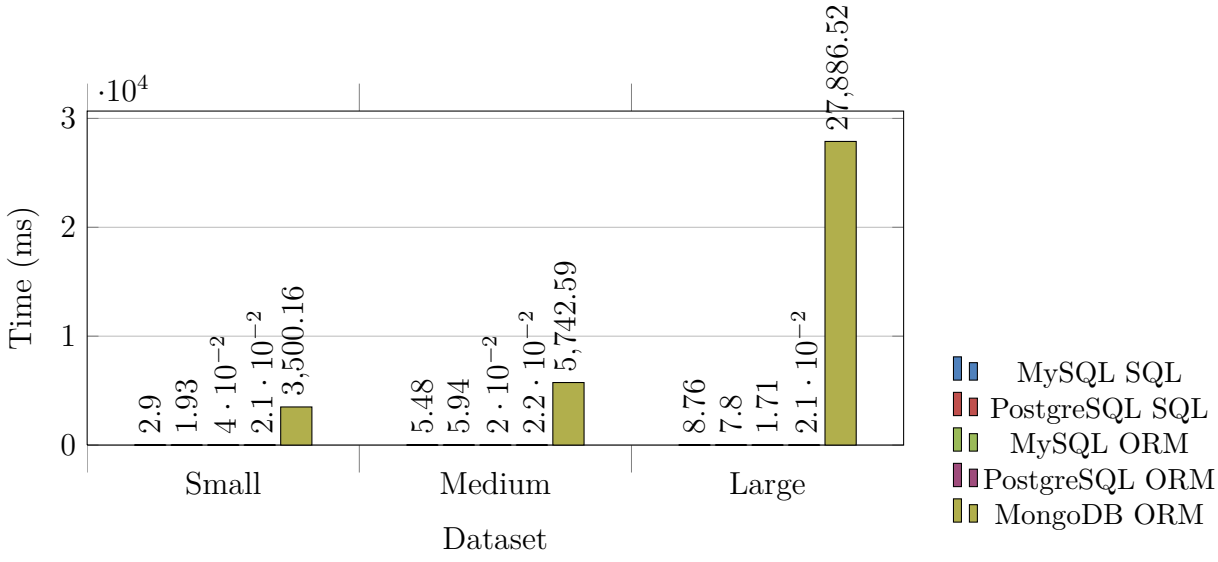


Figure 3.7 – Query time of all stores.
(Test Identifier: `t_all_stores`)

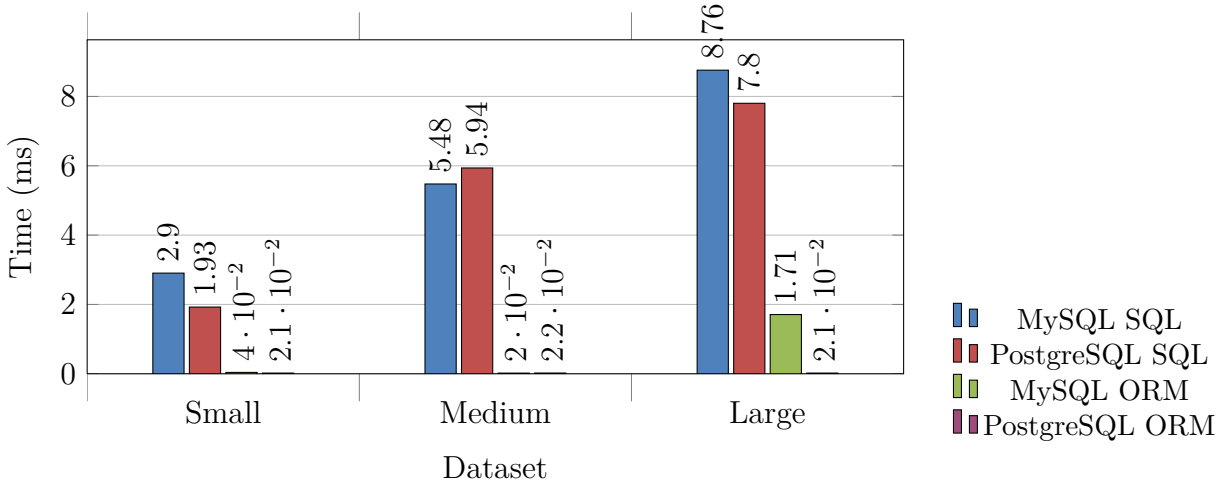


Figure 3.8 – Query time of all stores. **Excludes MongoDb outlier**.
(Test Identifier: `t_all_stores`)

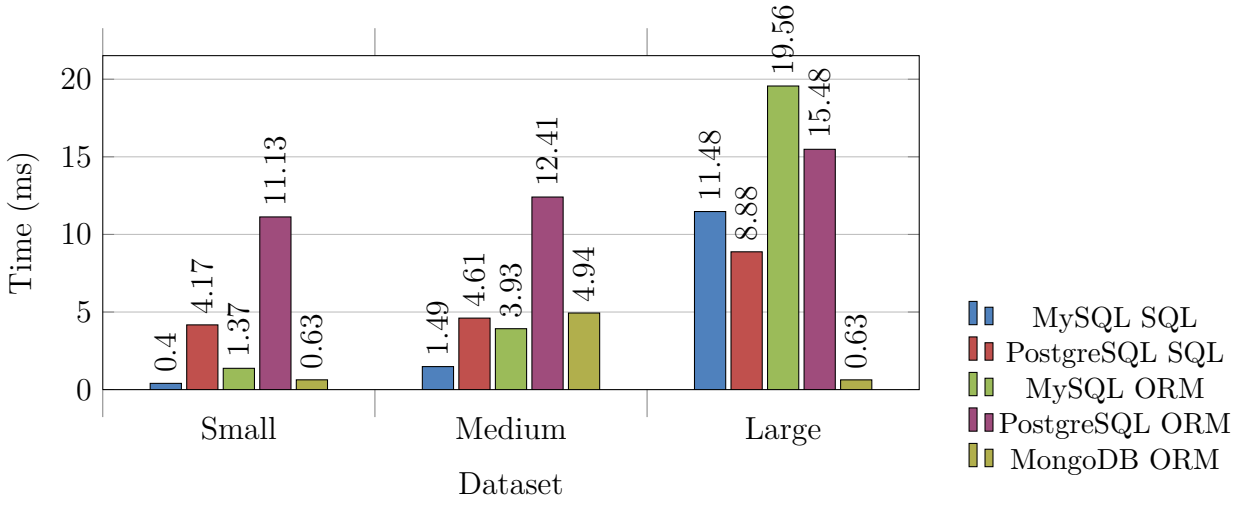


Figure 3.9 – Count query time of all customers.
(Test Identifier: `t_count_customers`)

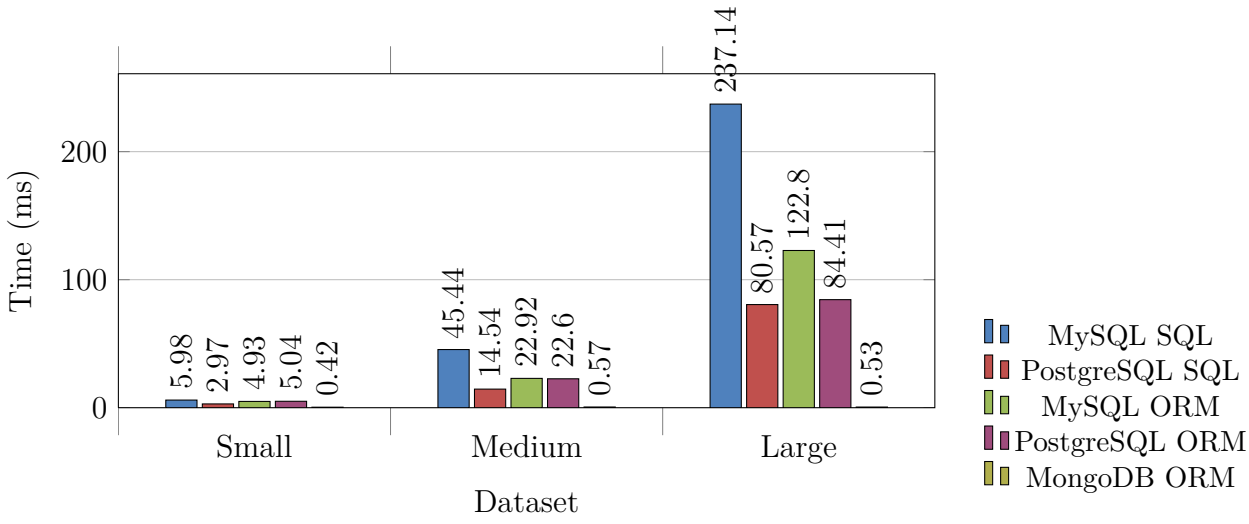


Figure 3.10 – Count query time of all products.
(Test Identifier: `t_count_products`)

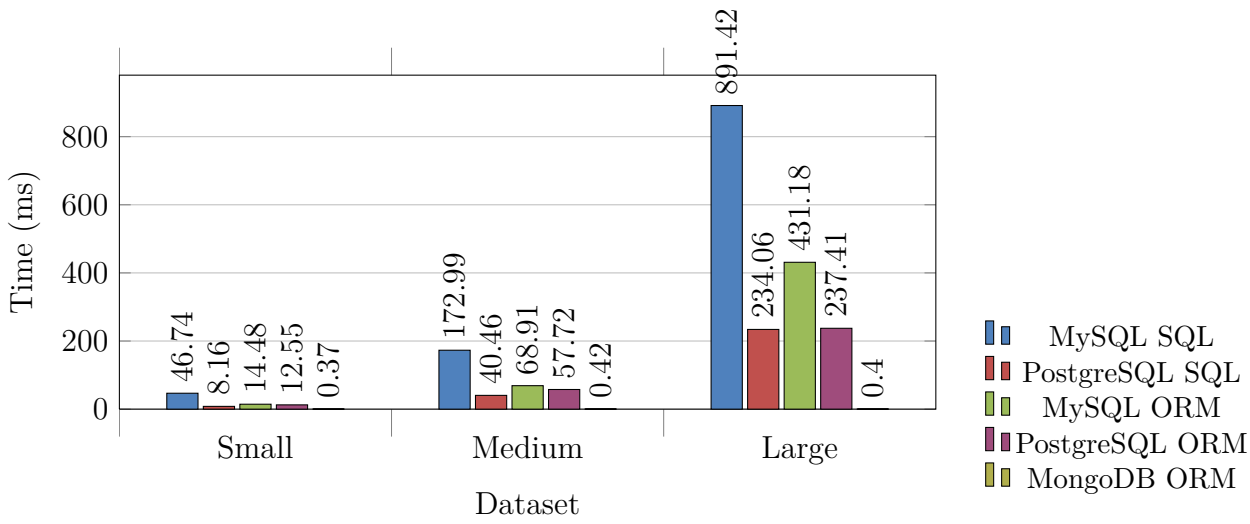


Figure 3.11 – Count query time of all purchases.
(Test Identifier: `t_count_purchases`)

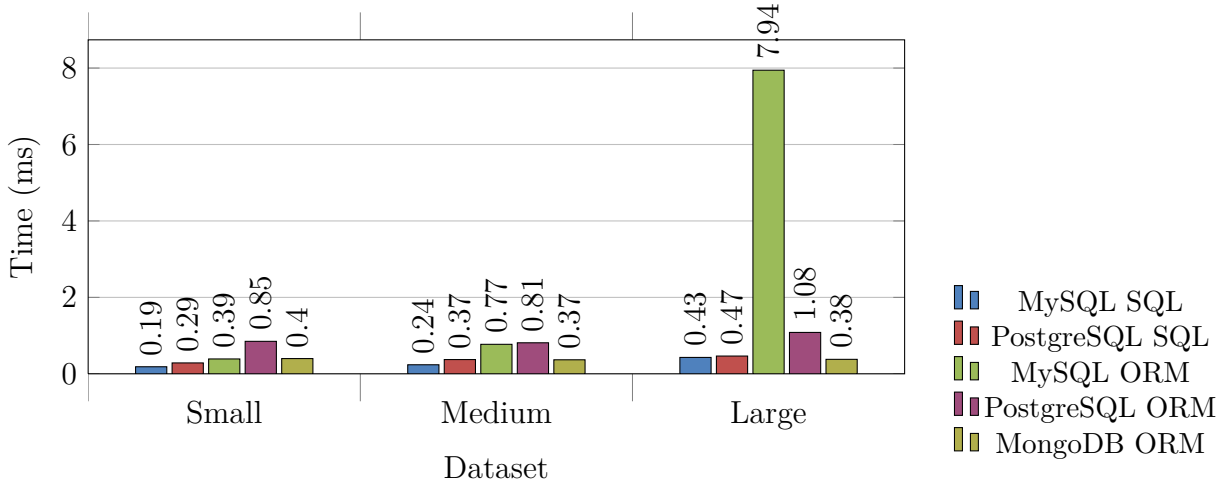


Figure 3.12 – Count query time of all purchases.
(Test Identifier: `t_count_stores`)

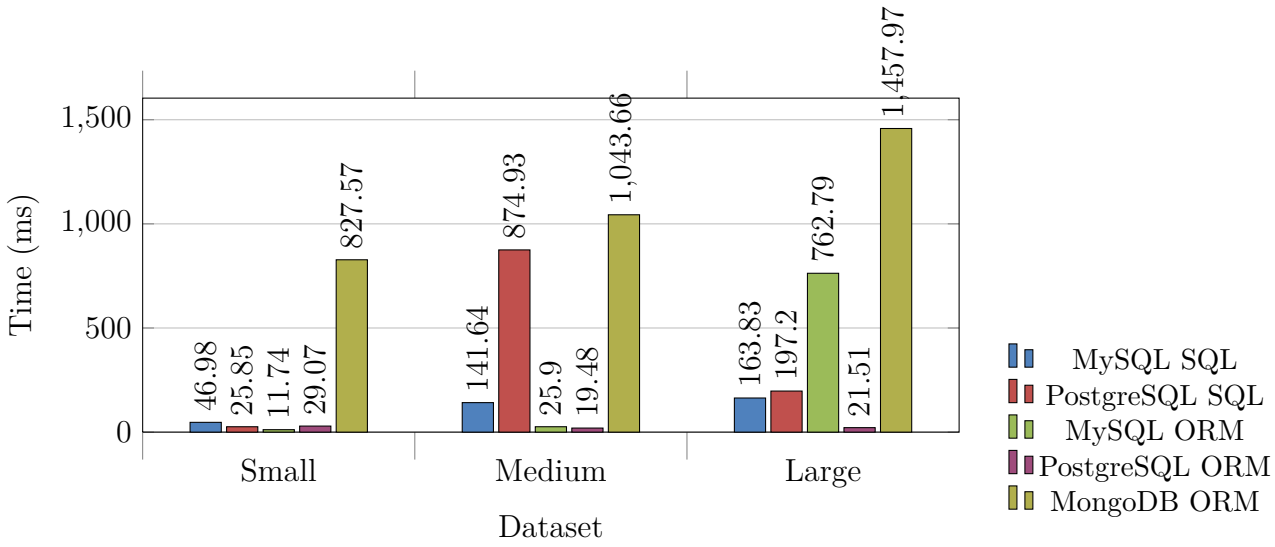


Figure 3.13 – Revenue calculation query time of a single store.
(Test Identifier: `t_single_store_revenue`)

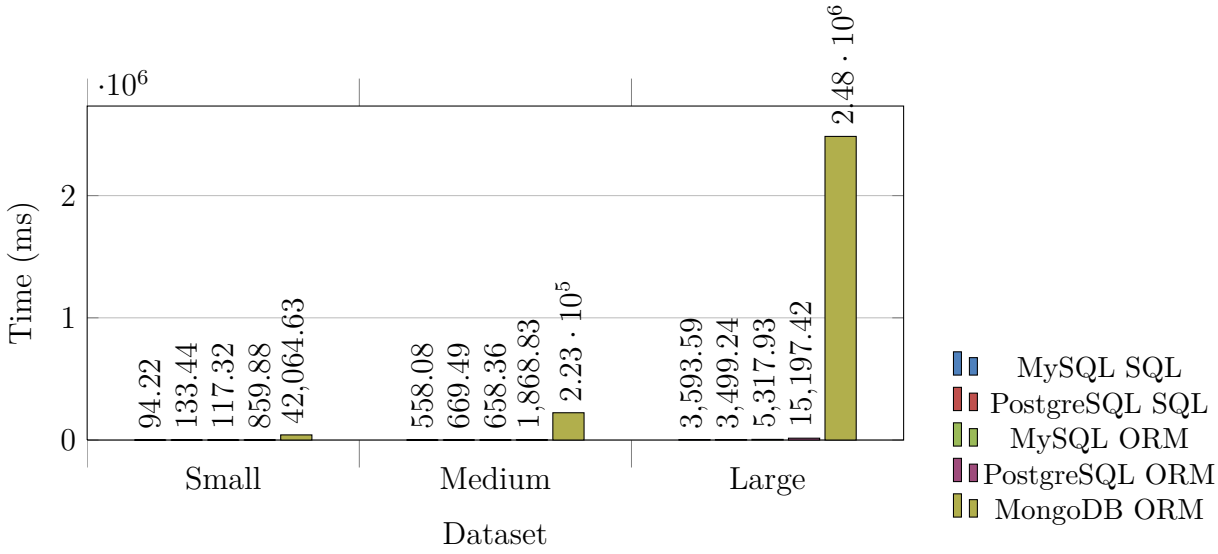


Figure 3.14 – Revenue calculation query time of all stores, sorted by most profitable.
(Test Identifier: `t_all_stores_revenue`)

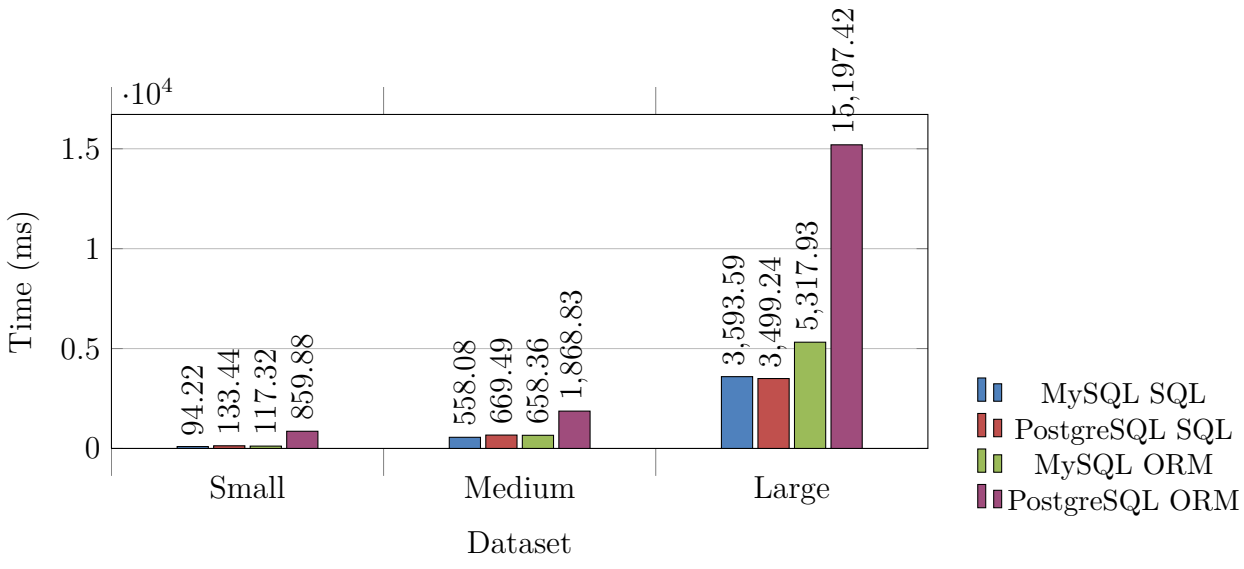


Figure 3.15 – Revenue calculation query time of all stores, sorted by most profitable. **Excludes MongoDb outlier**.
(Test Identifier: `t_all_stores_revenue`)

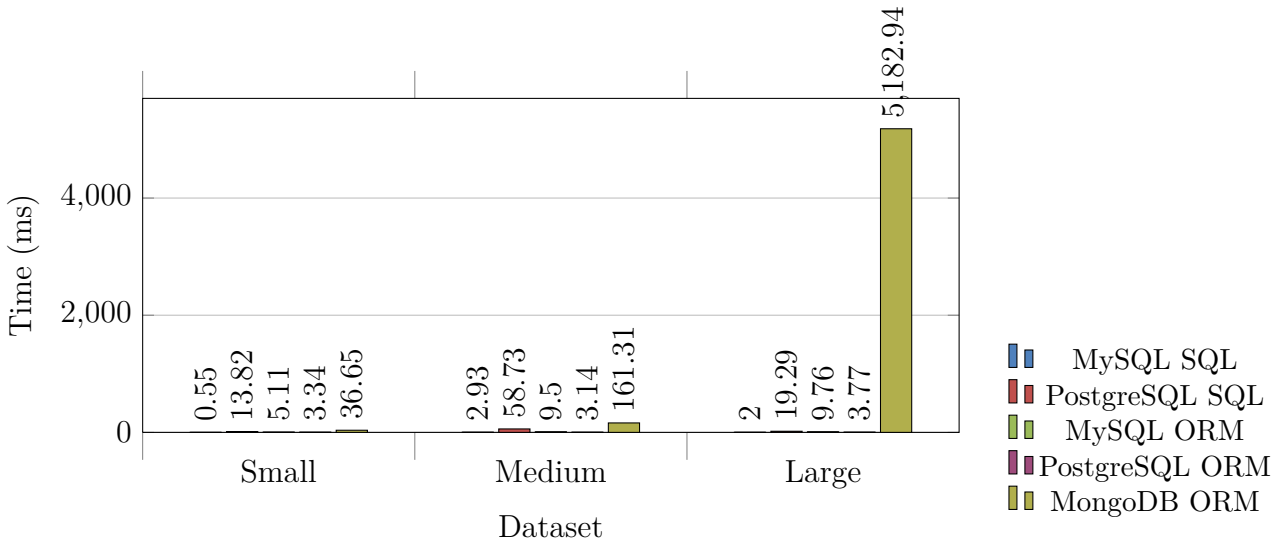


Figure 3.16 – Expenditure calculation query time of a single customer over all stores.
(Test Identifier: `t_single_customer_expenditure`)

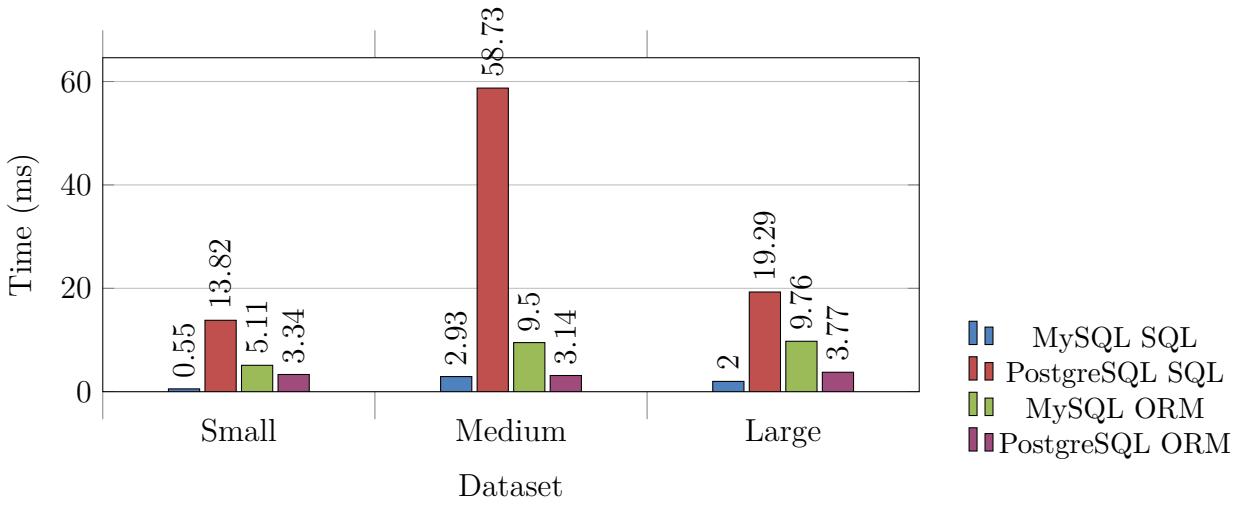


Figure 3.17 – Expenditure calculation query time of a single customer over all stores. **Excludes MongoDb outlier**.
(Test Identifier: `t_single_customer_expenditure`)

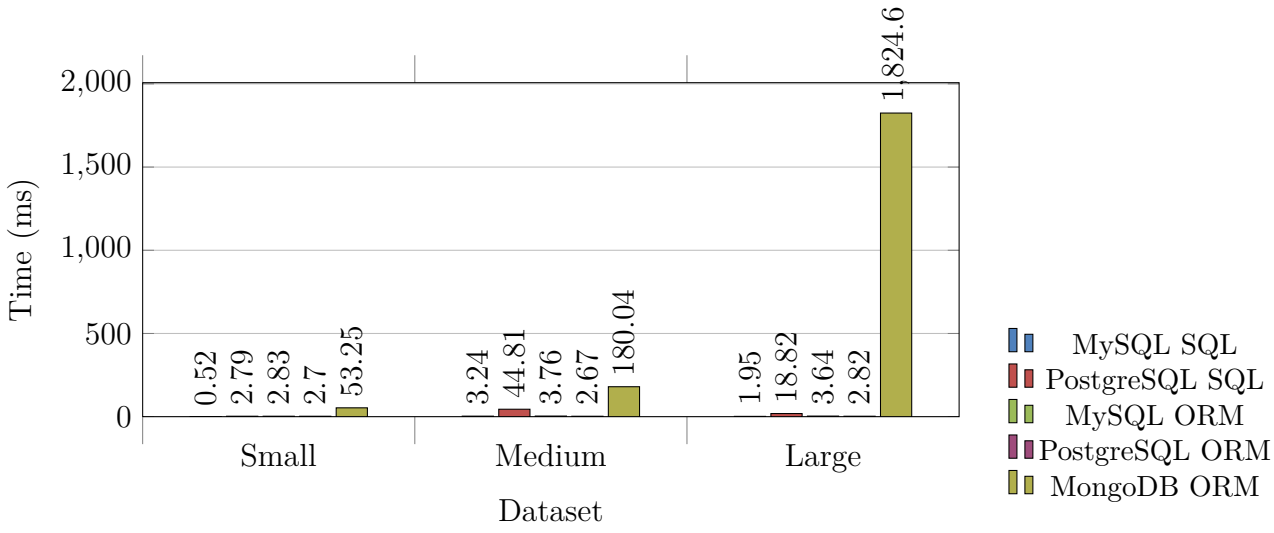


Figure 3.18 – Expenditure calculation query time of all customers at a particular store.
(Test Identifier: `t_single_customer_expenditure_at_store`)

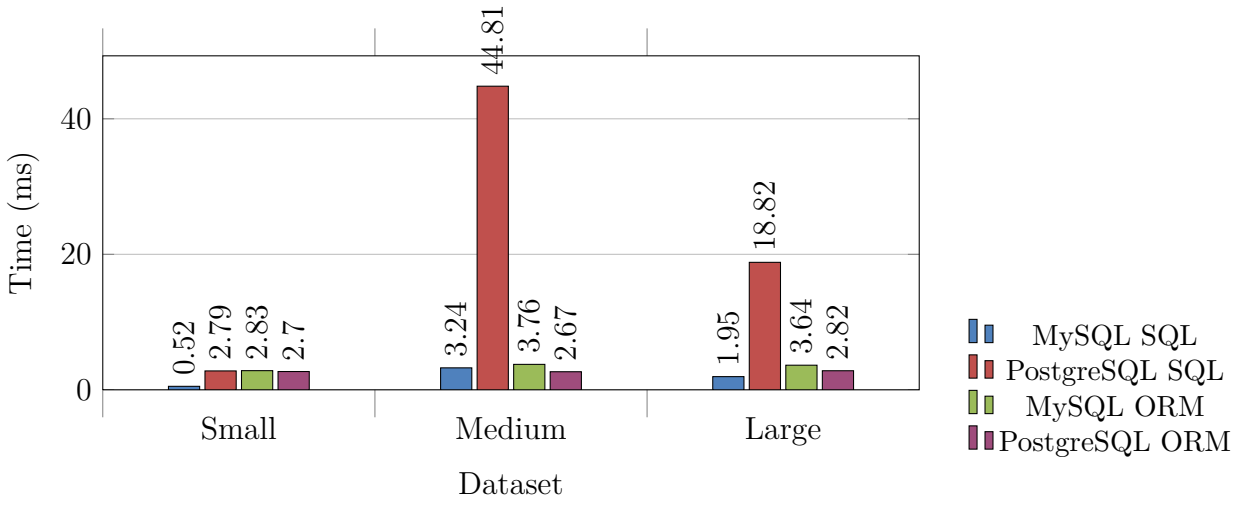


Figure 3.19 – Expenditure calculation query time of all customers at a particular store. **Excludes MongoDb outlier**.
(Test Identifier: `t_single_customer_expenditure_at_store`)

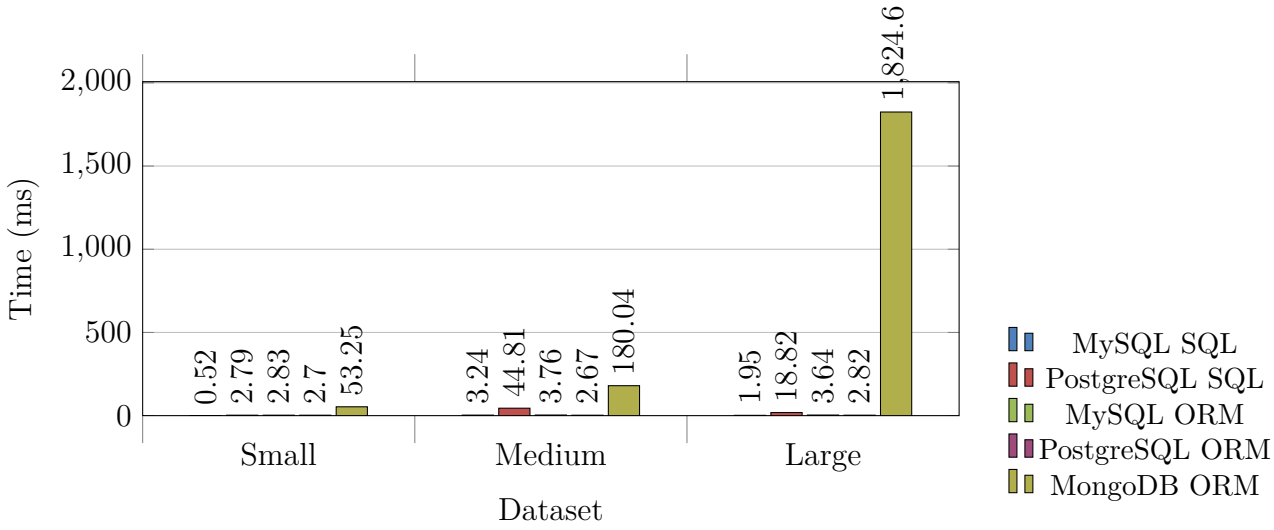


Figure 3.20 – Expenditure calculation query time of all customers, sorted by most expended and limited to top 3 customers.
(Test Identifier: `t_top_spenders`)

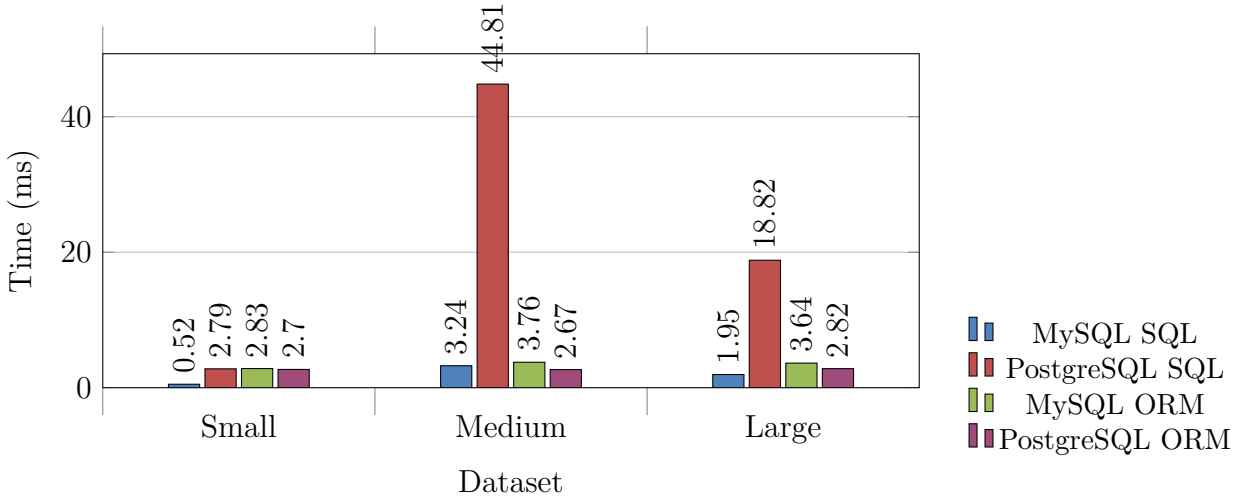


Figure 3.21 – Expenditure calculation query time of all customers, sorted by most expended and limited to top 3 customers. **Excludes MongoDb outlier**.
(Test Identifier: `t_top_spenders`)

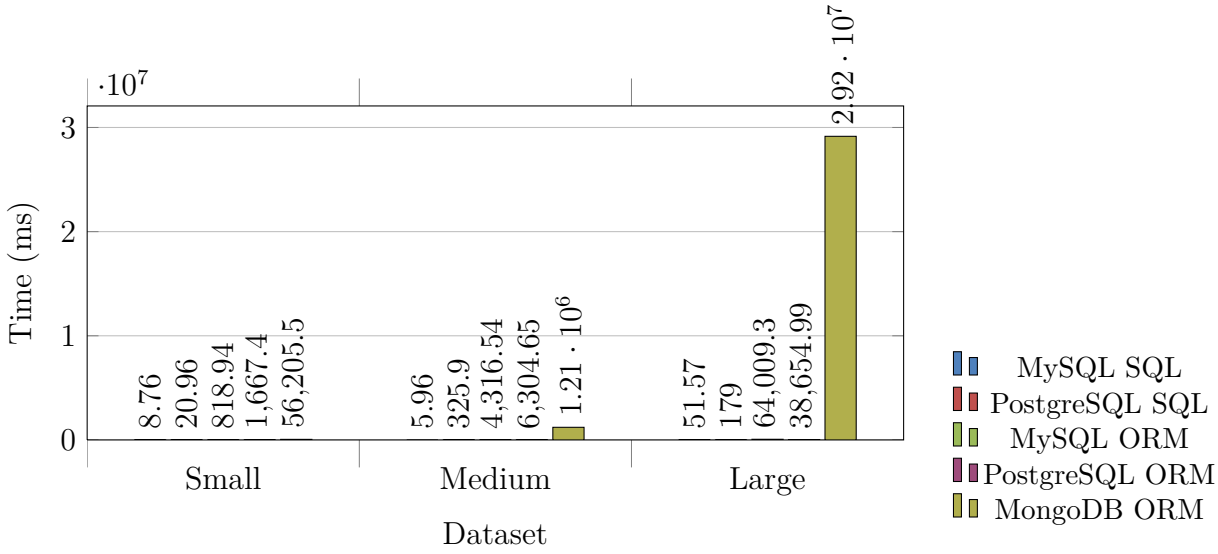


Figure 3.22 – Expenditure calculation query time of all customers at a single store, sorted by most expended and limited to top 3 customers. (Test Identifier: `t_top_spenders_at_store`)

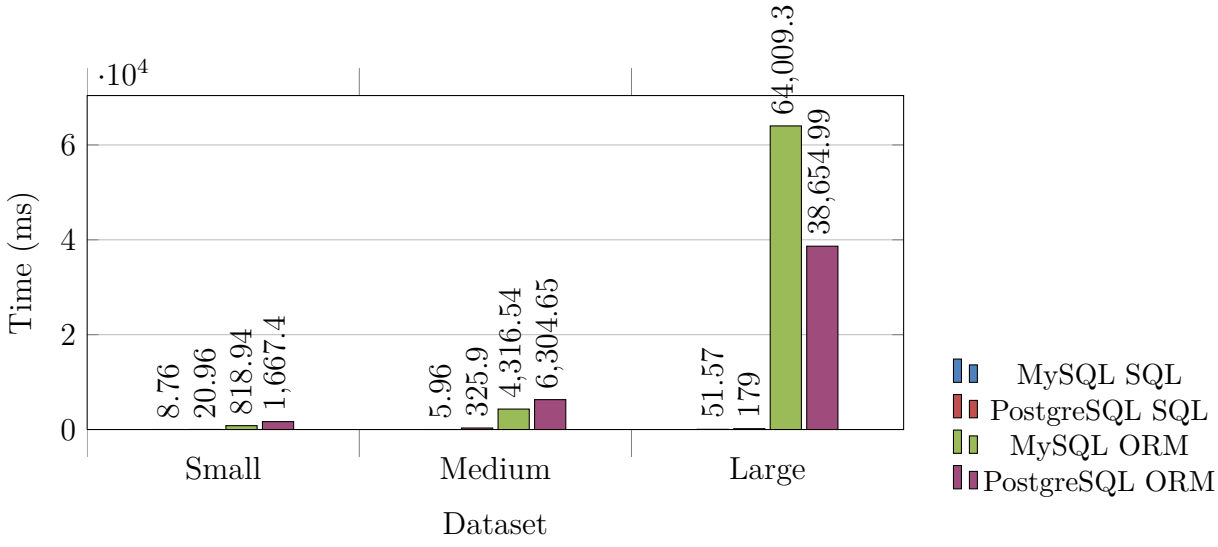


Figure 3.23 – Expenditure calculation query time of all customers at a single store, sorted by most expended and limited to top 3 customers. **Excludes MongoDb outlier**. (Test Identifier: `t_top_spenders_at_store`)

3.2 Operability Results

Code snippets from each of the *calculated* test case implementations are given below. These are given to show how complex calculations can vary between the implemented technologies.

3.2.1 Store Revenues

Listing 5: SQL test query that executes the calculation of a store's revenue. All stores revenue removes the `WHERE` clause.

```
# Store revenues
def t_single_store_revenue
  sql = 'SELECT SUM(products.price) AS "revenue" FROM purchases
         INNER JOIN products ON purchases.product_id = products.id
         INNER JOIN stores ON products.store_id = stores.id
         WHERE stores.id = ?;'
  DataAccess::Sql.exec(sql, [21])
end
def t_all_stores_revenue
  sql = 'SELECT stores.id AS "store_id", SUM(products.price) AS "revenue" FROM purchases
         INNER JOIN products
         ON purchases.product_id = products.id
         INNER JOIN stores ON products.store_id = stores.id
         GROUP BY stores.id
         ORDER BY revenue ASC;'
  DataAccess::Sql.exec(sql)
end
```

Listing 6: ORM mapping in the `Store` class for both a store's revenue and all stores revenue

```
  #
  # Get all stores sorted by the one with the most revenue
  #
  def self.sorted_by_revenue
    self.all.sort_by { | store |
      store.revenue
    }
  end

  #
  # Calculate the revenue for this store
  #
  def revenue
    purchases.sum(:price, :group => :product_id).to_f.round(2)
  end
```

Listing 7: ORM mapping in the `Store` class for both a store's revenue and all stores revenue sorted in MongoMapper.

```
#
# Get all stores sorted by the one with the most revenue
#
def self.sorted_by_revenue
  self.all.sort_by { | store |
    store.revenue
  }.reverse!
end

#
# Calculate the revenue for this store
#
def revenue
  return purchases.map { | purchase | purchase.product.price }.inject(:+).round(2)
end
```

3.2.2 Customer Expenditures

Listing 8: SQL test query that executes the calculation of customer expenditure. Where at a specific store an extended `WHERE` clause is added.

```
  sql = 'SELECT stores.id AS "store_id", SUM(products.price) AS "revenue" FROM purchases
         INNER JOIN products
         ON purchases.product_id = products.id
         INNER JOIN stores ON products.store_id = stores.id
         GROUP BY stores.id
         ORDER BY revenue ASC;'
  DataAccess::Sql.exec(sql)
end
# Customer Expenditure
def t_single_customer_expenditure
  sql = 'SELECT SUM(products.price) AS "money_spent" FROM purchases
         INNER JOIN products ON purchases.product_id = products.id
         INNER JOIN stores ON products.store_id = stores.id
         INNER JOIN customers ON purchases.customer_id = customers.id
         WHERE customers.id = ?;'
  DataAccess::Sql.exec(sql, [33])
end
```

Listing 9: ORM mapping in the `Customer` class for customer expenditure in ActiveRecord.

```
#
# Gets the total amount of money spent by this customer
```

```
#
def money_spent(store = nil)
  if store.nil?
    scope = purchases.all
  else
    scope = purchases.joins(:product).where(["products.store_id = :store", { store: store.id }])
  end

  scope.joins(:product).sum(:price, :group => :product_id).to_f.round(2)
end
```

Listing 10: ORM mapping in the `Customer` class for customer expenditure in MongoMapper. Note the method `product_ids_purchased` to fetch the ids needed for these calculations due to the lack of joining in MongoMapper.

```
#
# Gets the total amount of money spent by this customer
#
def money_spent(store = nil)
  productIds = product_ids_purchased()

  # Searching for a store
  if not store.nil?
    # Only match the product ids
    scope = { "$match" =>
      { "$and" => [
        { "_id" => { "$in" => productIds }},
        { "store_id" => { "$eq" => store.id }}
      ]}
    }
  else
    # Only match the product ids
    scope = { "$match" => { "_id" => { "$in" => productIds } } }
  end

  # Sum their price stage
  sumThePriceStage = {
    "$group" => {
      "_id" => nil,
      "total" => {
        "$sum" => "$price"
      }
    }
  }

  res = MongoDb::Product.collection.aggregate ([scope, sumThePriceStage])

  # No results?
  if res.first.nil?
    return 0.0
  end
```

```
    res.first['total'].round(2)
  end

  private

  #
  # Gets all the products purchased by this customer
  #
  def product_ids_purchased
    # Only get purchases from this customer
    purchaseMatchStage = { "$match" => { "customer_id" => id } }
    # Only select the ids of the purchase ids only
    productIdsOnlyStage = { "$project" => { "product_id" => 1 } }

    purchaseStages = [purchaseMatchStage, productIdsOnlyStage]

    # Get the productIds out of purchases
    productIds = purchases.collection.aggregate(purchaseStages).map { | result |
      result["product_id"]
    }
  end
```

3.2.3 Top Spenders

Listing 11: SQL test query that executes the calculation of the 3 top spenders. Where at a specific store an extended `WHERE` clause is added.

```
# Top Spenders
def t_top_spenders
  sql = 'SELECT customers.id AS "customer_id", SUM(products.price) AS "money_spent" FROM purchases
         INNER JOIN products ON purchases.product_id = products.id
         INNER JOIN stores ON products.store_id = stores.id
         INNER JOIN customers ON purchases.customer_id = customers.id
         GROUP BY customers.id
         ORDER BY money_spent DESC
         LIMIT 3;'
  DataAccess::Sql.exec(sql)
end
def t_top_spenders_at_store
  sql = 'SELECT customers.id AS "customer_id", SUM(products.price) AS "money_spent" FROM purchases
         INNER JOIN products ON purchases.product_id = products.id
         INNER JOIN stores ON products.store_id = stores.id
         INNER JOIN customers ON purchases.customer_id = customers.id
         WHERE stores.id = ?
         GROUP BY customers.id
         ORDER BY money_spent DESC
         LIMIT 3;'
  DataAccess::Sql.exec(sql, [19])
end
```

Listing 12: ORM mapping in the `Customer` class for customer expenditure in both MongoMapper and ActiveRecord.

```
#
# Gets the top spenders
#
def self.top_spenders(limit, store = nil)
  self.all.sort_by { | customer |
    customer.money_spent(store)
  }.reverse!.first (limit)
end
```

4 Discussion

4.1 Analysis

4.1.1 Performance Analysis

Basic Fetch Queries Over each of the technologies used, a persistent trend with ORM-based queries when fetching data was found—MySQL would only outperform PostgreSQL for *large base datasets*. This is especially true for the largest base size of a dataset (i.e., purchases) where `t_all_purchases` had a considerately faster fetch time when contrasting MySQL with PostgreSQL. This said, the smaller base dataset sizes, such as those in `t_all_products` or `t_all_customers`, PostgreSQL was considerably faster than MySQL.

In all of the basic fetch queries, MongoDB was always the slowest, regardless of the dataset size. The lag in the response time was proportional to the size of the dataset, suggesting that MongoDB has a linear speed in performance response times (i.e., there is no trend as to whether the speed slows down or improves as a dataset size varies in MongoDB).

Lastly, while significantly faster than MongoDB but slower than ORM, the raw-SQL queries consistently showed that MySQL is faster to load data when compared to PostgreSQL.

This suggests that the best performer in this case study, with regards to a basic fetch operation, is ORM backed by PostgreSQL on smaller base dataset sizes. When dealing with massive datasets, such as tens of millions of records, then MySQL is shown to be faster in these cases—as per the large purchases dataset.

Basic Count Queries Count queries were shown to be the only performance-leading query by MongoDB; MongoDB was significantly faster at loading the count of the records contained in its database when compared to the other ORM technologies assessed. Most importantly, MongoDB performed very well when dealing with medium- to large-dataset sizes, whereby each count query was able to return a scalar result in less than a millisecond *for all tests*. This often beat the raw SQL query selection time. However, an outlier existed for MongoDB ORM counts in the medium dataset for customer counts.

Interestingly, the other technologies did not seem to have the consistency in this query factor than MongoDB did; often, query time would fluctuate between over the different technologies, especially when looking at the large dataset sizes.

Typically MySQL under ORM was outperformed by PostgreSQL under ORM when selecting from a large dataset size, but then there is a discrepancy that contrasts with this suggestion with conflicting evidence in the `t_count_customers` test. Hence, as a *general rule of thumb*, the results suggest that if counting under ORM, PostgreSQL would yield the best performance. This said, however, if not considering the best performer of MongoDB, the next contender for a better performer would be PostgreSQL under raw-SQL, since this typically outperformed all other technologies (excluding MongoDb).

Complex Query Calculations As discussed in Section 3.1, MongoDB was the worst performer for calculations needed in this case study. Especially for the larger datasets, the time taken to calculate these results were often unreasonable and took minutes, if not hours, to generate the same result as the other technologies would in only a few seconds. While this may be a result of a biased case study towards Relational Databases (see Section 4.2), for the purposes of this research, MongoDb should not be considered for calculations of complex queries that involve the use of many joins between tables—which is understandable considering its non-relational nature.

For the remaining technologies, aside from the `t_single_store_revenue` test, MySQL under raw-SQL was the best performer, typically yielding results for small data sets in a fraction of the time as the other datasets. PostgreSQL was not as consistent in this regard, as time would typically spike for the medium-sized datasets; this said the ORM implementation of PostgreSQL would often outperform its MySQL equivalent, so there is a discrepancy between PostgreSQL performance under ORM and raw-SQL.

Overall, for each of the tests identified, MySQL under raw-SQL would yield the best

performance of all the technologies assessed, and while there are some fluctuations in this statement relative to the dataset sizes and complexity of the query, it was still much faster *on average*, as its results were *consistent* among all of the complexity query tests.

4.1.2 Operability Analysis

For all raw-SQL testing, a custom SQL execution class to better handle bind variables in Ruby was developed (refer to Section 2.4.2). As such, this class helped the operability of raw-SQL execution greatly, since from a client-perspective all that needed to be done was to insert the SQL query string in with question mark parameters and the class would handle the rest of the prepared statement—refer to Listings 5, 8, 11. In these examples, bind variables can simply be inserted with '?', and the `exec` method would handle the query, and parameters passed to it and execute the prepared statement for either PostgreSQL and MySQL bind variable types.

Since ActiveRecord is a feature that is at the very core of Ruby on Rails, interoperability with ActiveRecord was much easier than that of MongoMapper. Consider the steps needed to join in `Customer`'s `money_spent` method for ActiveRecord (Listing 13) when compared to MongoMapper (Listing 14); as there are no utilities in MongoMapper to easily make joins between entities in MongoMapper, this task is far more laborious:

Listing 13: Example of joining across tables with ActiveRecord.

```
scope.joins(:product).sum(:price, :group => :product_id).to_f.round(2)
```

Listing 14: Example of joining across tables with MongoDb.

```
# Sum their price stage
sumThePriceStage = {
  "$group" => {
    "_id" => nil,
    "total" => {
      "$sum" => "$price"
    }
  }
}

res = MongoDb::Product.collection.aggregate ([scope, sumThePriceStage])
```

Immediately, direct MongoDb syntax is required to achieve the same code as per Listing 13, which does not suit the expressive nature that ActiveRecord has—compare the one line of code with the ten lines of code. The `$group` and `$sum` aggregators are essentially equivalent to the programmer having to write SQL as if this were ActiveRecord, since this code is what would be used on the MongoDb console to achieve the same results.

Hence, there is no sense of easier operability when writing MongoDb ORM code with regards to complex queries (though this is debatable)—the level of expressiveness when joining and grouping does not match that of ActiveRecord's, which can just use `:group` symbol in its `sum` method; all expressible over just one line of code that uses dot notation to append functionality to the end of previous method calls. This differs from the MongoDb way of achieving the same functionality, which requires the definition of multiple 'stages' of the query—for example, the scope stage (not shown in Listing 14) must come before the `sumThePriceStage`, then used in an array when called upon the `Product`'s `collection`'s `aggregate` method.

This said, this code is functionally equivalent in MongoDb over all implemented languages (i.e., this is the *style* of MongoDb for queries). Achieving the same functionality in *another* ORM mapped to a PostgreSQL or MySQL database in another language would require a different ORM technology and thus differing syntax between language (compare ActiveRecord with, say, Hibernate in Java or Linq to SQL in C#).

Therefore, while MongoDb has a level of consistency between all languages when expressing queries which does not largely change (it is used almost exclusively as an array of hashes), ActiveRecord is only at the heart of Ruby *on Rails* and can therefore its level of expressiveness not be mapped to another language without additional effort. This is simply a matter of consistency and ubiquitousness among languages (i.e, MongoDb query syntax), or using a single language that not only supports but promotes the query syntax being used (ActiveRecord and Ruby).

4.2 Limitations

4.2.1 Case Study Bias

A significant problem with the results found in Section 3, especially the poor correlation of complex queries and fetch query performance for MongoDb, is the bias that exists in this case study for relational databases against document-oriented databases. The case study

chosen for this research is heavily geared towards relational databases, with many joins over many tables and entities to achieve the most basic of queries. While this proves an easy task for relational databases (as they are designed to handle these types of case studies with lots of joining), the document-oriented database chosen for this research struggles—as demonstrated.

More-specifically, Hoff (2010) makes specific mentions as to which use cases are *inappropriate* for a NoSQL data-store, specific those where there are complex relationships between entities (such as those in this report's use case), as well as ad-hoc, calculated queries (e.g., the calculated queries assessed in this use case also). This said, Hoff also mentions the following use cases which would be more suitable for assessing performance:

- Use cases that involve big data and large scalability, e.g. 7 TB/data *per day*
- Use cases that need massive write performance where lots of data needs to be written
- Use cases that require a flexible schema and flexible data-types
- Use cases that require parallel computing
- Use cases that require ease of interface with the data via a JSON interface

Future studies from this research would be benefited by applying *two* case studies—perhaps four—where half of the case studies can be geared toward relational databases and the other half can be geared toward document-oriented databases. By doing this, there will be no bias in the case studies chosen, and therefore a better assessment of both Document-Oriented and Relational Databases and their query performance can be made. The results from MongoDb in this research are greatly outside acceptable ranges, which questions the validity of these results for MongoDb performance—if these results were wholly valid (another limitation which a future benefit would benefit by resolving this issue), then there would be little reason to explain the popularity of MongoDb as the result times are simply unreasonable.

4.2.2 Language Bias

Whilst the Ruby language and Rails framework implement a robust database-connection and ORM environment, for the purposes of this report, it would be ideal if a range of

suitable languages and frameworks were used to eliminate any biases in just using Ruby on Rails. The report would therefore benefit by extending the test program and writing it in other languages and technologies that support database connections and ORM (e.g., C# and Entity Framework, Java and Hibernate etc.) to see if a change in language may have any affect on the results found and heuristics proposed by this report. Efficient languages such as C would be an ideal test environment to extend the findings of this report; a non-object oriented language may also improve performance by reducing object overhead, using structures or records to map object information instead. This will help deduce if any changes to languages also has any affect to the speed, operability and efficiency of the implementation of the test cases.

4.2.3 Limited Performance Scope

The scope of this research report examined basic fetch queries. This said, there was no assessment into the write, update or delete operations—only read operations. This clearly limits the scope of what may be better performance quality factors which was deduced in Section 4. In fact, Tezer summarises this notable discrepancy below:

> "NoSQL databases are usually faster—and sometimes extremely speedier—when it comes to *writes*. *Reads* can also be very fast depending on the type of NoSQL database and data being queried."
>
> - (Tezer, 2010)

Hence, future studies may benefit by assessing not just query performance, but also other kinds of Database performance factors between (O)RDBMSs and NoSQL data stores. That way a better assessment on *overall* performance can be deduced.

5 Conclusion

NoSQL data stores and Relational Database Management Systems both have opposing benefits and cons. Database programmers have developed tools to aid integrating data sourced from these stores into their client-side applications, with the aim that these tools improve the operability of the code that they work with. This said, the improved operability may

come at a performance cost, and the affect of this performance cost over various database technologies and sizes were investigated.

By using datasets of different sizes, various fluctuations in the performance metrics gathered for basic fetch queries, count queries and more complex count queries were found. The biggest outlier in the results was the extremely poor performance or ORM technology when mapped to a MongoDb data-store, taking, on average, four times longer to return a result than all of the other technologies. This said, it was the *fastest* to return count queries, which suggests that its performance is definable variable depending on the factor assessing that performance metric. Also notable was the relative fetch time of ORM-based technologies under MySQL and PostgreSQL, which often returned results much faster than raw SQL queries. On average, PostgreSQL was often faster at returning these batch results, usually within a time frame of 0.021ms.

As aforementioned, scalar count queries were often best performed by MongoDb, with the ORM technologies typically taking slightly longer than the raw-SQL technology, where all MySQL-backed queries, regardless of ORM or raw SQL, usually took longer to count than PostgreSQL, regardless of dataset size.

Lastly, the complex scalar queries proved to show the most interesting results, whereby MySQL was the faster responder when queried using raw SQL, though this was only true for small datasets, and not for larger ones. Therefore, it can be concluded that while MySQL can handle complex calculations for smaller datasets, PostgreSQL (under either ORM or raw SQL) is typically faster for *larger* dataset sizes, though these results were not always consistent.

There are many limitations posed by this research report; operability of the code was largely biased by the Ruby on Rails framework, and therefore future investigations would benefit by using a wider scope of both implementation languages, use cases as well as performance factors. The ActiveRecord library, built into the heart of Rails, makes the code operability considerably more favourable in the (O)RDBMS technologies when it comes to ORM mapping, while MongoMapper for MongoDb was not so easy to use when directly contrasted against ActiveRecord.

Ultimately, there is no 'right' answer that can be applied to all use cases, since there is just too much inconsistency between the results found. While there may be some general rules-of-thumb, such as the improved performance of MySQL for smaller-sized datasets over larger-sized ones, the area is still murky. Using SQL over ORM may be beneficial for *some*

queries but widely speaking, ORM and (O)RDBMs, usually, provide the best value for developers—with good performance and code operability, there is no reason not to use the two complementing technologies until the other technologies, notably MongoDb, mature.

References

P. Cooper. *Beginning Ruby: From Novice to Professional*. Apresspod Series. Apress, 2007. ISBN 9781590597668.

B. Deterling. How to execute a raw update sql with dynamic binding in rails. Published online via StackOverflow, December 2010. URL `http://stackoverflow.com/a/4484549/519967`. Cited September 28 2014.

P. DuBois. *MySQL*. Developer's library. Addison-Wesley, 2013. ISBN 9780321833877.

E. Emily. When to use Embedded Documents? Online via StackOverflow, January 2010. URL `http://stackoverflow.com/q/1993425`. Cited September 24 2014.

M. Gaidica. Parameterizing SQL Queries with ActiveRecord. Online, January 2010. URL `http://bytesofpi.com/post/23666298953/parameterizing-sql-queries-with-activerecord`. Cited October 4 2014.

J. L. Harrington. *Object-oriented database design clearly explained*. Morgan Kaufmann, 2000.

T. Hoff. What The Heck Are You Actually Using NoSQL For? Online, December 2010. URL `http://highscalability.com/blog/2010/12/6/what-the-heck-are-you-actually-using-nosql-for.html`. Cited September 20 2014.

E. Howe. Prepared Statement on Postgresql in Rails. Online via StackOverflow, December 2012. URL `http://stackoverflow.com/q/13805627`. Cited September 28 2014.

J. Kern. How to get a random record with Mongo Mapper. Online via Google Groups, October 2010. URL `https://groups.google.com/forum/#!msg/mongomapper/JT5Gx6scy9Q/DjMXFyRH7WAJ`. Cited September 24 2014.

MongoDB, Inc. Ruby MongoDB Driver. Online, October 2011. URL `http://docs.mongodb.org/ecosystem/drivers/ruby/`. Cited September 20 2014.

MongoDb, Inc. SQL to Aggregation Mapping Chart. Online, December 2010., a. URL `http://docs.mongodb.org/manual/reference/sql-aggregation-comparison/`. Cited September 28 2014.

MongoDb, Inc. SQL to MongoDb Chart. Online, March 2010., b. URL `http://docs.mongodb.org/manual/reference/sql-comparison/`. Cited September 28 2014.

J. Nunemaker. Querying in MonogMapper. Online, January 2010. URL `http://mongomapper.com/documentation/plugins/querying.html`. Cited September 24 2014.

G. Sabău. Comparison of rdbms, oodbms and ordbms. *Revista InformaticaEconomică,(44)*, 2007.

O. Tezer. A Comparison Of NoSQL Database Management Systems And Models? Online, February 2014. URL `https://www.digitalocean.com/community/tutorials/a-comparison-of-nosql-database-management-systems-and-models`. Cited October 21 2014.

G. Vaish. *Getting Started with Nosql*. Packt Publishing, 2013. ISBN 9781849694995.

P. Zaitsev, V. Tkachenko, J. Zawodny, A. Lentz, and D. Balling. *High Performance MySQL: Optimization, Backups, Replication, and More*. O'Reilly Media, 2008. ISBN 9780596554750.

A Source Code

A.1 Data Access Layer

Listing 15: The connector class establishes connections to the database under a particular DBMS and dataset size.

```
#
# Research Project
# COS30009 - Database Programming
#
# Alex Cummaudo, 1744070
# 24/09/2014
#

#
# The data access module maintains connections and raw
# queries to the database
#
module DataAccess
  #
  # Defines the connection to the database
  #
  class Connector
    @@remote_location = "localhost"

    # What is the current dbms?
    @@current_dbms = :mysql
    def self.dbms
      @@current_dbms
    end

    # What is the dataset size?
    @@dataset_size = :small
    def self.dataset_size
      @@dataset_size
    end

    #
    # Sets the database to either small dataset, medium dataset or large dataset
    #
    def self.dataset_size=(newSet)
      # Small med or large only!
      if [:small, :medium, :large].find_index(newSet).nil?
        raise "Invalid dataset (only accept symbols :small, :medium, :large)"
      end

      @@dataset_size = newSet
    end

    #
    # Sets the database to either mysql, postgres or mongodb
```

```
    #
    def self.dbms=(newDb)
      # Small med or large only!
      if [:mysql, :postgresql, :mongo].find_index(newDb).nil?
        raise "Invalid dbms (only accept symbols :mysql, :postgresql, :mongo)"
      end

      @@current_dbms = newDb
    end

    #
    # Sets up the connector with a dbms and dataset provided
    #
    def self.setup(dbms, dataset)
      self.dbms = dbms
      self.dataset_size = dataset
      self.establish_connection
    end

    #
    # Establishes a new connection to the database
    #
    def self.establish_connection
      # Need an ActiveRecord connection (i.e., not Mongo)
      if @@current_dbms != :mongo
        ActiveRecord::Base.establish_connection(
          :adapter => @@current_dbms,
          :host => @@remote_location,
          :user => "root",
          :password => "300594",
          :database => "possys_#{@@dataset_size}"
        )
      # Else, use mongo
      else
        MongoMapper.connection = Mongo::Connection.new(@@remote_location)
        MongoMapper.database = "possys-#{@@dataset_size}"
        if defined?(PhusionPassenger)
          PhusionPassenger.on_event(:starting_worker_process) do |forked|
            MongoMapper.connection.connect if forked
          end
        end
      end
    end

  end
end
```

Listing 16: The SQL class allows for raw execution of SQL into a Database using prepared statements as necessary.

```
#
# Research Project
```

```
# COS30009 - Database Programming
#
# Alex Cummaudo, 1744070
# 24/09/2014
#

#
# The data access module maintains connections and raw
# queries to the database
#
module DataAccess
  #
  # The SQL class allows us to execute raw sql commands to the current ActiveRecord database
  #
  class Sql
    #
    # Execute a raw sql query via ActiveRecord. If providing sql parameters, use
    # the MySQL standard for query parameters (i.e., SELECT * FROM table WHERE id = ?)
    # If the dbms is Postgres, these values will be converted into appropriate Postgres
    # instead (i.e., SELECT * FROM table WHERE id = ? becomes WHERE id = $1)
    #
    def self.exec(sql, params = nil)
      if Connector.dbms == :mongo
        raise "Can only be executed when the current dbms is not a NoSQL database"
      end

      # No parameters?
      if params.nil?
        ret = ActiveRecord::Base.connection.execute(sql)
        # PostgreSQL conversion from PG::Result to Array
        if ret.is_a? PG::Result
          return ret.values
        else
          return ret.to_a
        end
      # Else need to make a prepared statement
      else
        # MySQL Prepared Statements
        if Connector.dbms == :mysql
          return self.mysql_prepared_statement(sql, params)
        # PostgreSQL Prepared Statement
        elsif Connector.dbms == :postgresql
          # convert all the ? to $x for compatibility with postgresql
          sql = sql.gsub(/\s\?([\s\,\n\;])?/).with_index { | m, i | " $#{i+1}#{$1}" }
          return self.postgresql_prepared_statement(sql, params)
        end
      end
    end

    private

    #
    # Ensures the preparaed statement is suitably prepared using the given paramater definition (i.e., $1
        or ?)
    #
```

```
    def self.check_prepared_statement(sql, params, paramdef)
      sqlParamsCount = sql.scan(paramdef).count
      # Ensure params == number of $ in the sql
      if sqlParamsCount != params.count or sqlParamsCount == 0
        if sqlParamsCount != 0
          raise "Number of parameters mistmatch in prepared statement: sql = '#{sql}'; params = #{params}"
        else
          raise "Need at least one parameter to support prepared statement"
        end
      end
    end

    #
    # Execute a MySQL prepared statement
    #
    def self.mysql_prepared_statement(sql, params)
      self.check_prepared_statement(sql, params, /\s\?[\s\,\n\;]?/)
      pstmt = ActiveRecord::Base.connection.raw_connection.prepare(sql)
      pstmt.execute(*params).to_a
    end

    #
    # Execute a PostgreSQL prepared statement
    #
    def self.postgresql_prepared_statement(sql, params)
      self.check_prepared_statement(sql, params, /\s\$\d[\s\,\n\;]?/)
      ActiveRecord::Base.connection.raw_connection.exec_params(sql, params).values.to_a
    end
  end
end
```

A.2 Data Model Layer

A.2.1 ActiveRecord Models (MySQL and PostgreSQL mapping)

Listing 17: The customer ORM model class modelled under ActiveRecord.

```
#
# Research Project
# COS30009 - Database Programming
# Ruby ORM Implementation
#
# Alex Cummaudo, 1744070
# 20/09/2014
#

#
# The data models module contains ORM Entities
#
```

```ruby
module DataModels
  #
  # The ORM Entities made possible by ActiveRecord
  #
  module ActiveRecord
    #
    # The customer class represents customers as an ORM Entity
    #
    class Customer < ::ActiveRecord::Base
      #
      # Originating store name
      #
      self.table_name = :customers

      #
      # Model associations
      #
      has_many :purchases, foreign_key: :customer_id

      #
      # Get's the n'th entity
      #
      def self.find_n(n)
        limit(1).offset(n-1).first
      end

      #
      # Gets the top spenders
      #
      def self.top_spenders(limit, store = nil)
        self.all.sort_by { | customer |
          customer.money_spent(store)
        }.reverse!.first (limit)
      end

      #
      # Shorthand accessor for name
      #
      def name
        "#{first_name} #{last_name}"
      end

      #
      # Gets all the stores customers have purchased products from
      #
      def stores_purchased_from
        purchases.map { | purchase |
          purchase.product.store
        }.uniq
      end

      #
      # Gets the total amount of money spent by this customer
      #
      def money_spent(store = nil)
```

```
        if store.nil?
          scope = purchases.all
        else
          scope = purchases.joins(:product).where(["products.store_id = :store", { store: store.id }])
        end

        scope.joins(:product).sum(:price, :group => :product_id).to_f.round(2)
      end
    end
  end
end
```

Listing 18: The product ORM model class modelled under ActiveRecord.

```
#
# Research Project
# COS30009 - Database Programming
# Ruby ORM Implementation
#
# Alex Cummaudo, 1744070
# 20/09/2014
#

#
# The data models module contains ORM Entities
#
module DataModels
  #
  # The ORM Entities made possible by ActiveRecord
  #
  module ActiveRecord
    #
    # The product class represents products as an ORM Entity
    #
    class Product < ::ActiveRecord::Base
      #
      # Originating store name
      #
      self.table_name = :products

      #
      # Model associations
      #
      belongs_to :store, foreign_key: :store_id
      has_many :purchases, foreign_key: :product_id

      #
      # Get's the n'th entity
      #
      def self.find_n(n)
        limit(1).offset(n-1).first
      end
```

```
    end
  end
end
```

Listing 19: The purchase ORM model class modelled under ActiveRecord.

```
#
# Research Project
# COS30009 - Database Programming
# Ruby ORM Implementation
#
# Alex Cummaudo, 1744070
# 20/09/2014
#

#
# The data models module contains ORM Entities
#
module DataModels
  #
  # The ORM Entities made possible by ActiveRecord
  #
  module ActiveRecord
    #
    # The purchase class represents purchases as an ORM Entity
    #
    class Purchase < ::ActiveRecord::Base
      #
      # Originating store name
      #
      self.table_name = :purchases

      #
      # Model associations
      #
      belongs_to :product, foreign_key: :product_id
      belongs_to :customer, foreign_key: :customer_id

      #
      # Get's the n'th entity
      #
      def self.find_n(n)
        limit(1).offset(n-1).first
      end
    end
  end
end
```

Listing 20: The store ORM model class modelled under ActiveRecord.

```ruby
#
# Research Project
# COS30009 - Database Programming
# Ruby ORM Implementation
#
# Alex Cummaudo, 1744070
# 20/09/2014
#

#
# The data models module contains ORM Entities
#
module DataModels
  #
  # The ORM Entities made possible by ActiveRecord
  #
  module ActiveRecord
    #
    # The store class represents stores as an ORM Entity
    #
    class Store < ::ActiveRecord::Base
      #
      # Originating store name
      #
      self.table_name = :stores

      #
      # Model associations
      #
      has_many :products, foreign_key: :store_id
      has_many :purchases, through: :products

      #
      # Get's the n'th entity
      #
      def self.find_n(n)
        limit(1).offset(n-1).first
      end

      #
      # Get all stores sorted by the one with the most revenue
      #
      def self.sorted_by_revenue
        self.all.sort_by { | store |
          store.revenue
        }
      end

      #
      # Calculate the revenue for this store
      #
      def revenue
        purchases.sum(:price, :group => :product_id).to_f.round(2)
      end
    end
```

```
  end
end
```

A.2.2 MongoMapper Models (MongoDb mapping)

Listing 21: The customer ORM model class modelled under MongoMapper.

```
#
# Research Project
# COS30009 - Database Programming
# Ruby ORM Implementation
#
# Alex Cummaudo, 1744070
# 20/09/2014
#

#
# The data models module contains ORM Entities
#
module DataModels
  #
  # The ORM Entities made possible by MongoMapper
  #
  module MongoDb
    #
    # The customer class represents customers as an ORM Entity
    #
    class Customer
      include ::MongoMapper::Document

      #
      # Define keys
      #
      key :first_name, String
      key :last_name, String
      key :street_address, String
      key :city, String
      key :postcode, String
      key :state, String
      key :email, String
      key :phone_number, String

      #
      # Model associations
      #
      many :purchases, :class_name => "DataModels::MongoDb::Purchase", :foreign_key => :customer_id

      #
      # Get's the n'th entity
      #
```

```
def self.find_n(n)
  skip(n-1).first
end

#
# Gets the top spenders
#
def self.top_spenders(limit, store = nil)
  self.all.sort_by { | customer |
    customer.money_spent(store)
  }
end

#
# Shorthand accessor for name
#
def name
  "#{first_name} #{last_name}"
end

#
# Gets all the stores customers have purchased products from
#
def stores_purchased_from
  purchases.map { | purchase |
    purchase.product.store
  }.uniq
end

#
# Gets the total amount of money spent by this customer
#
def money_spent(store = nil)
  productIds = product_ids_purchased()

  # Searching for a store
  if not store.nil?
    # Only match the product ids
    scope = { "$match" =>
      { "$and" => [
        { "_id" => { "$in" => productIds }},
        { "store_id" => { "$eq" => store.id }}
      ]}
    }
  else
    # Only match the product ids
    scope = { "$match" => { "_id" => { "$in" => productIds } } }
  end

  # Sum their price stage
  sumThePriceStage = {
    "$group" => {
      "_id" => nil,
      "total" => {
        "$sum" => "$price"
```

```
          }
        }
      }

      res = MongoDb::Product.collection.aggregate ([scope, sumThePriceStage])

      # No results?
      if res.first.nil?
        return 0.0
      end

      res.first['total'].round(2)
    end

    private

    #
    # Gets all the products purchased by this customer
    #
    def product_ids_purchased
      # Only get purchases from this customer
      purchaseMatchStage = { "$match" => { "customer_id" => id } }
      # Only select the ids of the purchase ids only
      productIdsOnlyStage = { "$project" => { "product_id" => 1 } }

      purchaseStages = [purchaseMatchStage, productIdsOnlyStage]

      # Get the productIds out of purchases
      productIds = purchases.collection.aggregate(purchaseStages).map { | result |
        result["product_id"]
      }
    end
  end
 end
end
```

Listing 22: The product ORM model class modelled under MongoMapper.

```
#
# Research Project
# COS30009 - Database Programming
# Ruby ORM Implementation
#
# Alex Cummaudo, 1744070
# 20/09/2014
#

#
# The data models module contains ORM Entities
#
module DataModels
  #
```

```
  # The ORM Entities made possible by MongoMapper
  #
  module MongoDb
    #
    # The product class represents products as an ORM Entity
    #
    class Product
      include ::MongoMapper::Document

      #
      # Define keys
      #
      key :store_id, Integer
      key :name, String
      key :price, Float
      key :ean, String
      key :store_id, ObjectId

      #
      # Model associations
      #
      belongs_to :store, :class_name => "DataModels::MongoDb::Store", :foreign_key => :store_id
      many :purchases, :class_name => "DataModels::MongoDb::Purchase", :foreign_key => :product_id

      #
      # Get's the n'th entity
      #
      def self.find_n(n)
        skip(n-1).first
      end
    end
  end
end
```

Listing 23: The purchase ORM model class modelled under MongoMapper.

```
#
# Research Project
# COS30009 - Database Programming
# Ruby ORM Implementation
#
# Alex Cummaudo, 1744070
# 20/09/2014
#

#
# The data models module contains ORM Entities
#
module DataModels
  #
  # The ORM Entities made possible by MongoMapper
  #
```

```
  module MongoDb
    #
    # The purchase class represents purchases as an ORM Entity
    #
    class Purchase
      include ::MongoMapper::Document

      #
      # Define keys
      #
      key :product_id, Integer
      key :customer_id, Integer
      key :credit_card_number, String
      key :purchase_datetime, DateTime
      key :product_id, ObjectId
      key :customer_id, ObjectId

      #
      # Model associations
      #
      belongs_to :product, :class_name => "DataModels::MongoDb::Product", :foreign_key => :product_id
      belongs_to :customer, :class_name => "DataModels::MongoDb::Customer", :foreign_key => :customer_id

      #
      # Get's the n'th entity
      #
      def self.find_n(n)
        skip(n-1).first
      end
    end
  end
end
```

Listing 24: The store ORM model class modelled under MongoMapper.

```
#
# Research Project
# COS30009 - Database Programming
# Ruby ORM Implementation
#
# Alex Cummaudo, 1744070
# 20/09/2014
#

#
# The data models module contains ORM Entities
#
module DataModels
  #
  # The ORM Entities made possible by MongoMapper
  #
  module MongoDb
```

```
    #
    # The store class represents stores as an ORM Entity
    #
    class Store
      include ::MongoMapper::Document

      #
      # Define keys
      #
      key :name, String
      key :street_address, String
      key :city, String
      key :postcode, String
      key :state, String
      key :phone_number, String
      key :logo_url, String
      key :purchase_ids, Array# needed for a "many through"

      #
      # Model associations
      #
      many :products, :class_name => "DataModels::MongoDb::Product", :foreign_key => :store_id
      many :purchases, :class_name => "DataModels::MongoDb::Purchase", :in => :purchase_ids

      #
      # Get's the n'th entity
      #
      def self.find_n(n)
        skip(n-1).first
      end

      #
      # Get all stores sorted by the one with the most revenue
      #
      def self.sorted_by_revenue
        self.all.sort_by { | store |
          store.revenue
        }.reverse!
      end

      #
      # Calculate the revenue for this store
      #
      def revenue
        return purchases.map { | purchase | purchase.product.price }.inject(:+).round(2)
      end
    end
  end
end
```

A.3 Test Cases

Listing 25: The class which defines the test classes and the types of tests that can be run.

```
#
# Research Project
# COS30009 - Database Programming
# Ruby ORM Implementation
#
# Alex Cummaudo, 1744070
# 28/09/2014
#

#
# This module is a helper for the Runner page
#
module TesterHelper
  include DataAccess
  include DataModels
  require 'benchmark'

  #
  # The general executor for the database tests
  #
  class TestExec
    #
    # Class initaliser
    #
    def initialize(dbms, dataset)
      @dbms = dbms
      @dataset = dataset
    end

    #
    # Executes all tests in this executor
    #
    def run_tests
      # Disable the logger
      old_logger = ActiveRecord::Base.logger
      ActiveRecord::Base.logger = nil

      # Ensure we're connected to the right database
      DataAccess::Connector.setup(@dbms, @dataset)

      ret = {}

      # Execute all private methods
      private_methods(false).each do | method |
        next if not method.to_s.starts_with?("t_")
        result = nil
        puts "[#{@dbms}/#{@dataset}]\tRunning Test:\t#{method}"
        # Time how long (in ms) it takes to execute the test using benchmark
        time = (Benchmark.realtime { result = self.send(method) } * 1000).round(4)
        puts "[#{@dbms}/#{@dataset}]\tTest Ended:\t#{method} in #{time}ms"
        # remove the t_
        ret[method.to_s[2..-1]] = time
      end
```

```
    # Renable the logger
    ActiveRecord::Base.logger = old_logger

    # Return the result of the method
    ret
  end
end

#
# An SQL-Only Test Executor
#
class SqlTestExec < TestExec
  #
  # Class initaliser
  #
  def initialize(dbms, dataset)
    if [:mysql, :postgresql].find_index(dbms).nil?
      raise "SQL Test Executor tester only supports MySQL and Postgres"
    end
    super(dbms, dataset)
  end

  # Declare all tests as private under t_
  private

  # Result Sets
  def t_all_customers
    DataAccess::Sql.exec("SELECT * FROM Customers")
  end
  def t_all_products
    DataAccess::Sql.exec("SELECT * FROM Products")
  end
  def t_all_purchases
    DataAccess::Sql.exec("SELECT * FROM Purchases")
  end
  def t_all_stores
    DataAccess::Sql.exec("SELECT * FROM Stores")
  end
  # Scalar Count Tests
  def t_count_customers
    DataAccess::Sql.exec("SELECT COUNT(*) FROM Customers")
  end
  def t_count_products
    DataAccess::Sql.exec("SELECT COUNT(*) FROM Products")
  end
  def t_count_purchases
    DataAccess::Sql.exec("SELECT COUNT(*) FROM Purchases")
  end
  def t_count_stores
    DataAccess::Sql.exec("SELECT COUNT(*) FROM Stores")
  end
  # Store revenues
  def t_single_store_revenue
    sql = 'SELECT SUM(products.price) AS "revenue" FROM purchases
```

```
            INNER JOIN products ON purchases.product_id = products.id
            INNER JOIN stores ON products.store_id = stores.id
            WHERE stores.id = ?;'
    DataAccess::Sql.exec(sql, [21])
  end
  def t_all_stores_revenue
    sql = 'SELECT stores.id AS "store_id", SUM(products.price) AS "revenue" FROM purchases
            INNER JOIN products
            ON purchases.product_id = products.id
            INNER JOIN stores ON products.store_id = stores.id
            GROUP BY stores.id
            ORDER BY revenue ASC;'
    DataAccess::Sql.exec(sql)
  end
  # Customer Expenditure
  def t_single_customer_expenditure
    sql = 'SELECT SUM(products.price) AS "money_spent" FROM purchases
            INNER JOIN products ON purchases.product_id = products.id
            INNER JOIN stores ON products.store_id = stores.id
            INNER JOIN customers ON purchases.customer_id = customers.id
            WHERE customers.id = ?;'
    DataAccess::Sql.exec(sql, [33])
  end
  def t_single_customer_expenditure_at_store
    sql = 'SELECT SUM(products.price) AS "money_spent" FROM purchases
            INNER JOIN products ON purchases.product_id = products.id
            INNER JOIN stores ON products.store_id = stores.id
            INNER JOIN customers ON purchases.customer_id = customers.id
            WHERE customers.id = ? AND stores.id = ?;'
    DataAccess::Sql.exec(sql, [71, 50])
  end
  # Top Spenders
  def t_top_spenders
    sql = 'SELECT customers.id AS "customer_id", SUM(products.price) AS "money_spent" FROM purchases
            INNER JOIN products ON purchases.product_id = products.id
            INNER JOIN stores ON products.store_id = stores.id
            INNER JOIN customers ON purchases.customer_id = customers.id
            GROUP BY customers.id
            ORDER BY money_spent DESC
            LIMIT 3;'
    DataAccess::Sql.exec(sql)
  end
  def t_top_spenders_at_store
    sql = 'SELECT customers.id AS "customer_id", SUM(products.price) AS "money_spent" FROM purchases
            INNER JOIN products ON purchases.product_id = products.id
            INNER JOIN stores ON products.store_id = stores.id
            INNER JOIN customers ON purchases.customer_id = customers.id
            WHERE stores.id = ?
            GROUP BY customers.id
            ORDER BY money_spent DESC
            LIMIT 3;'
    DataAccess::Sql.exec(sql, [19])
  end
end
```

```
#
# A ORM SQL backend Test Executor (i.e., ActiveRecord models)
#
class OrmTestExec < TestExec
  #
  # Class initaliser
  #
  def initialize(dbms, dataset)
    if [:mysql, :postgresql, :mongo].find_index(dbms).nil?
      raise "ORM Test Executor tester only supports :mysql, :postgresql and :mongo dbms"
    end
    if dbms == :mongo
      # Alias models to the MongoDb models
      @models = ::DataModels::MongoDb
    else
      # Alias models to the ActiveRecord models
      @models = ::DataModels::ActiveRecord
    end
    super(dbms, dataset)
  end

  # Declare all tests as private under t_
  private

  # Result Sets
  def t_all_customers
    @models::Customer.all
  end
  def t_all_products
    @models::Product.all
  end
  def t_all_purchases
    @models::Purchase.all
  end
  def t_all_stores
    @models::Store.all
  end
  # Scalar Count Tests
  def t_count_customers
    @models::Customer.count
  end
  def t_count_products
    @models::Product.count
  end
  def t_count_purchases
    @models::Purchase.count
  end
  def t_count_stores
    @models::Store.count
  end
  # Store revenues
  def t_single_store_revenue
    @models::Store.find_n(21).revenue
  end
  def t_all_stores_revenue
```

```
      @models::Store.sorted_by_revenue
    end
    # Customer Expenditure
    def t_single_customer_expenditure
      @models::Customer.find_n(33).money_spent
    end
    def t_single_customer_expenditure_at_store
      store = @models::Store.find_n(50)
      @models::Customer.find_n(71).money_spent(store)
    end
    # Top Spenders
    def t_top_spenders
      @models::Customer.top_spenders(3)
    end
    def t_top_spenders_at_store
      store = @models::Store.find_n(19)
      @models::Customer.top_spenders(3, store)
    end
  end

end
```

Listing 26: The class which is run to execute all tests.

```
#
# Research Project
# COS30009 - Database Programming
# Ruby ORM Implementation
#
# Alex Cummaudo, 1744070
# 28/09/2014
#

#
# The controller links up with the view
#
class TesterController < ApplicationController
  helper TesterHelper

  def runner
    testers = []

    testers << TesterHelper::SqlTestExec.new(:mysql, :large)
    testers << TesterHelper::SqlTestExec.new(:mysql, :medium)
    testers << TesterHelper::SqlTestExec.new(:mysql, :small)

    testers << TesterHelper::SqlTestExec.new(:postgresql, :large)
    testers << TesterHelper::SqlTestExec.new(:postgresql, :medium)
    testers << TesterHelper::SqlTestExec.new(:postgresql, :small)

    testers << TesterHelper::OrmTestExec.new(:mysql, :large)
    testers << TesterHelper::OrmTestExec.new(:mysql, :medium)
```

```
    testers << TesterHelper::OrmTestExec.new(:mysql, :small)

    testers << TesterHelper::OrmTestExec.new(:postgresql, :large)
    testers << TesterHelper::OrmTestExec.new(:postgresql, :medium)
    testers << TesterHelper::OrmTestExec.new(:postgresql, :small)

    testers << TesterHelper::OrmTestExec.new(:mongo, :large)
    testers << TesterHelper::OrmTestExec.new(:mongo, :medium)
    testers << TesterHelper::OrmTestExec.new(:mongo, :small)

    testers.each { | t | t.run_tests }
  end
end
```

A.4 Database Population Script

Listing 27: The database population script.

```
#
# Research Project
# COS30009 - Database Programming
#
# Alex Cummaudo, 1744070
# 24/09/2014
#
namespace :db do

  desc "Initialises everything!"
  task supersetup: [:drop] do
    # Kill the old stuff
    system "RAILS_ENV=mysql_large rake db:setup"
    system "RAILS_ENV=mysql_medium rake db:setup"
    system "RAILS_ENV=mysql_small rake db:setup"
    system "RAILS_ENV=postgres_large rake db:setup"
    system "RAILS_ENV=postgres_medium rake db:setup"
    system "RAILS_ENV=postgres_small rake db:setup"
    #system "rm -rf ~/data/mongodb/dbp/possys/*"
    # # Repopulate the new stuff, multi-threaded of course!
    cfg = [ { :dbms => "mysql", :size => "small" },
            { :dbms => "mysql", :size => "medium" },
            { :dbms => "mysql", :size => "large" },
            { :dbms => "postgresql", :size => "large" },
            { :dbms => "postgresql", :size => "medium" },
            { :dbms => "postgresql", :size => "small" },
            { :dbms => "mongo", :size => "large" },
            { :dbms => "mongo", :size => "medium" },
            { :dbms => "mongo", :size => "small" } ]
    9.times do | idx |
      Thread.new do
        rails_path = Rails.root.to_s
```

```
      dbms = cfg[idx][:dbms]
      size = cfg[idx][:size]
      script = "cd \\\"#{rails_path}\\\" && RAILS_ENV=mysql_large rake db:populate dbms=#{dbms}
          size=#{size}"
      system "osascript -e 'tell application \"Terminal\" to do script \"#{script}\"'"
    end
  end
end

desc "Populates the database and fill with test data"
task :populate => :environment do
  require 'populator'
  require 'faker'
  require 'active_record'

  # set faker opts
  Faker::Config.locale = 'en-AU'

  def rand_time(from, to=Time.now)
    Time.at(rand_in_range(from.to_f, to.to_f))
  end

  def rand_in_range(from, to)
    rand * (to - from) + from
  end

  data = {}

  data[:dbms ] = ENV["dbms"].to_sym
  data[:scale] = ENV["size"].to_sym

  isMongo = data[:dbms] == :mongo

  unless DataAccess::Connector.dbms = data[:dbms]
    puts "=> Failed to change dbms to #{ENV['dbms']}"
    puts "=> FAILED"
    return
  end
  unless DataAccess::Connector.dataset_size = data[:scale]
    puts "=> Failed to change dbms to #{ENV['dbms']}"
    puts "=> FAILED"
    return
  end
  DataAccess::Connector.establish_connection

  # dynamically use naming for ActiveRecord or MongoDb
  if isMongo
    Store = DataModels::MongoDb::Store
    Customer = DataModels::MongoDb::Customer
    Product = DataModels::MongoDb::Product
    Purchase = DataModels::MongoDb::Purchase
    # Clear all previous...
    Store.destroy_all
    Customer.destroy_all
    Product.destroy_all
```

```
  Purchase.destroy_all
else
  Store = DataModels::ActiveRecord::Store
  Customer = DataModels::ActiveRecord::Customer
  Product = DataModels::ActiveRecord::Product
  Purchase = DataModels::ActiveRecord::Purchase
end

# define scale opts here
if data[:scale] == :large then
  scale_fctr = 125
end
if data[:scale] == :medium then
  scale_fctr = 25
end
if data[:scale] == :small then
  scale_fctr = 5
end

# Consistency: ensure data is the SAME regardless of db---base randomisation on scale
Random.srand(scale_fctr)

data[:no_stores] = 10 * scale_fctr
data[:no_products] = 5000 * scale_fctr
data[:no_customers] = 300 * scale_fctr
data[:no_purchases] = (data[:no_customers] * data[:no_products]) / (data[:no_stores] * 10)

data.each do | key, val |
  puts "#{key} is #{val}"
end

# create all the stores
puts "=> creating stores"
i = 0
data[:no_stores].times do
  store = Store.create(
    name: Faker::Company.name,
    street_address: Faker::Address.street_address,
    city: Faker::Address.city,
    postcode: Faker::Address.postcode,
    state: Faker::Address.state_abbr,
    phone_number: Faker::PhoneNumber.phone_number,
    logo_url: Faker::Company.logo
  )
  puts "==> (#{data[:dbms]}/#{data[:scale]}) created store [#{i+=1}/#{data[:no_stores]}]:
       #{store.name} (#{store.street_address}, #{store.city} #{store.postcode}, #{store.state})"
end

# create all the products
puts "=> creating products"
store_ids = if isMongo then Store.all.map { |p| p.id }
                    else Store.pluck (:id) end
i = 0
data[:no_products].times do
  rand_store_id = store_ids.sample
```

```
  product = Product.create(
    store_id: rand_store_id,
    name: Faker::Commerce.product_name,
    price: Faker::Commerce.price,
    ean: Faker::Code.ean
  )
  puts "==> (#{data[:dbms]}/#{data[:scale]}) created product [#{i+=1}/#{data[:no_products]}]:
      #{product.name} (Only $#{product.price} at #{product.store.name}!)"
end

# create all the customers
puts "=> creating customers"
i = 0
data[:no_customers].times do
  customer = Customer.create(
    first_name: Faker::Name.first_name,
    last_name: Faker::Name.last_name,
    street_address: Faker::Address.street_address,
    city: Faker::Address.city,
    postcode: Faker::Address.postcode,
    state: Faker::Address.state_abbr,
    phone_number: Faker::PhoneNumber.phone_number,
    email: Faker::Internet.free_email
  )
  puts "==> (#{data[:dbms]}/#{data[:scale]}) created customer [#{i+=1}/#{data[:no_customers]}]:
      #{customer.first_name} #{customer.last_name}"
end

# create all the purchases
puts "=> creating purchases"
product_ids = if isMongo then Product.all.map { |p| p.id }
                      else Product.pluck (:id) end
customer_ids = if isMongo then Customer.all.map { |p| p.id }
                       else Customer.pluck (:id) end
i = 0
data[:no_purchases].times do
  rand_product_id = product_ids.sample
  rand_customer_id = customer_ids.sample
  purchase = Purchase.create(
    product_id: rand_product_id,
    customer_id: rand_customer_id,
    credit_card_number: Faker::Business.credit_card_number,
    purchase_datetime: rand_time(5.years.ago)
  )
  # need to add to purchase_ids of store if is mongo
  if isMongo
    owningStore = Product.find(rand_product_id).store
    owningStore.purchase_ids << purchase.id
    # update owningStore
    owningStore.save
  end
  puts "==> (#{data[:dbms]}/#{data[:scale]}) created purchase [#{i+=1}/#{data[:no_purchases]}]:
      #{purchase.product.name} by #{purchase.customer.first_name} #{purchase.customer.last_name} on
      #{purchase.purchase_datetime}"
end
```

```
    puts "done!"

  end
end
```

A.5 Raw SQL Queries

Listing 28: SQL Queries for raw SQL tests

```
-- SQL for t_all_customers
SELECT * FROM Customers;

-- SQL for t_all_products
SELECT * FROM Products;

-- SQL for t_all_purchases
SELECT * FROM Purchases;

-- SQL for t_all_stores
SELECT * FROM Stores;

-- SQL for t_count_customers
SELECT COUNT(*) FROM Customers;

-- SQL for t_count_products
SELECT COUNT(*) FROM Products;

-- SQL for t_count_purchases
SELECT COUNT(*) FROM Purchases;

-- SQL for t_count_stores
SELECT COUNT(*) FROM Stores;

-- SQL for t_single_store_revenue (with parameters)
SELECT SUM(products.price) AS "revenue" FROM purchases
INNER JOIN products ON purchases.product_id = products.id
INNER JOIN stores ON products.store_id = stores.id
WHERE stores.id = ?;

-- SQL for t_all_stores_revenue
SELECT stores.id AS "store_id", SUM(products.price) AS "revenue" FROM purchases
INNER JOIN products
ON purchases.product_id = products.id
INNER JOIN stores ON products.store_id = stores.id
GROUP BY stores.id
ORDER BY revenue ASC;

-- SQL for t_single_customer_expenditure (with parameters)
SELECT SUM(products.price) AS "money_spent" FROM purchases
```

```
INNER JOIN products ON purchases.product_id = products.id
INNER JOIN stores ON products.store_id = stores.id
INNER JOIN customers ON purchases.customer_id = customers.id
WHERE customers.id = ?;

-- SQL for t_single_customer_expenditure_at_store (with parameters)
SELECT SUM(products.price) AS "money_spent" FROM purchases
INNER JOIN products ON purchases.product_id = products.id
INNER JOIN stores ON products.store_id = stores.id
INNER JOIN customers ON purchases.customer_id = customers.id
WHERE customers.id = ? AND stores.id = ?;

-- SQL for t_top_spenders (with parameters)
SELECT customers.id AS "customer_id", SUM(products.price) AS "money_spent" FROM purchases
INNER JOIN products ON purchases.product_id = products.id
INNER JOIN stores ON products.store_id = stores.id
INNER JOIN customers ON purchases.customer_id = customers.id
GROUP BY customers.id
ORDER BY money_spent DESC
LIMIT 3;

-- SQL for t_top_spenders_at_store (with parameters)
SELECT customers.id AS "customer_id", SUM(products.price) AS "money_spent" FROM purchases
INNER JOIN products ON purchases.product_id = products.id
INNER JOIN stores ON products.store_id = stores.id
INNER JOIN customers ON purchases.customer_id = customers.id
WHERE stores.id = ?
GROUP BY customers.id
ORDER BY money_spent DESC
LIMIT 3;
```

A.6 Ruby ActiveRecord Schema

Listing 29: The database schema for ActiveRecord database mappings.

```
#
# Research Project
# COS30009 - Database Programming
# ActiveRecord Schema
#
# Alex Cummaudo, 1744070
# 28/09/2014
#

ActiveRecord::Schema.define(version: 20140920122551) do

  create_table "customers", force: true do |t|
    t.string "first_name"
    t.string "last_name"
    t.string "street_address"
```

```
    t.string "city"
    t.string "postcode"
    t.string "state"
    t.string "email"
    t.string "phone_number"
  end

  create_table "products", force: true do |t|
    t.integer "store_id"
    t.string "name"
    t.decimal "price", precision: 10, scale: 2
    t.string "ean"
  end

  add_index "products", ["store_id"], name: "index_products_on_store_id", using: :btree

  create_table "purchases", force: true do |t|
    t.integer "product_id"
    t.integer "customer_id"
    t.string "credit_card_number"
    t.datetime "purchase_datetime"
  end

  add_index "purchases", ["customer_id"], name: "index_purchases_on_customer_id", using: :btree
  add_index "purchases", ["product_id"], name: "index_purchases_on_product_id", using: :btree

  create_table "stores", force: true do |t|
    t.string "name"
    t.string "street_address"
    t.string "city"
    t.string "postcode"
    t.string "state"
    t.string "phone_number"
    t.string "logo_url"
  end

end
```

B Raw Results

B.1 Tabulated Results

Below lists the tables of raw results gathered from each DBMS as tests were ran for each dataset. Dataset 3 refers to the large dataset, 2 refers to the medium dataset and 1 refers to the small dataset. All measurements in the below tables are made in **milliseconds**.

Table B.1: Query time of all customers.
(Test Identifier: `t_all_customers`)

DS	SQL MySQL	SQL PostgreSQL	ORM MySQL	ORM PostgreSQL	MongoDB
3	126.77	160.19	22.3	0.1	7,484.52
2	23.67	66.91	$3 \cdot 10^{-2}$	$3.2 \cdot 10^{-2}$	1,427.83
1	9.14	57.5	$3.8 \cdot 10^{-2}$	$3.1 \cdot 10^{-2}$	279.95

Table B.2: Query time of all products.
(Test Identifier: `t_all_products`)

DS	SQL MySQL	SQL PostgreSQL	ORM MySQL	ORM PostgreSQL	MongoDB
3	1,091.87	1,192.18	3.22	$2.1 \cdot 10^{-2}$	70,123.92
2	199.22	360.14	$2.2 \cdot 10^{-2}$	$2 \cdot 10^{-2}$	13,434.95
1	38.58	51.88	$2 \cdot 10^{-2}$	$2.2 \cdot 10^{-2}$	3,023.83

Table B.3: Query time of all purchases.
(Test Identifier: `t_all_purchases`)

DS	SQL MySQL	SQL PostgreSQL	ORM MySQL	ORM PostgreSQL	MongoDB
3	3,217.02	5,890.35	0.97	$1.5 \cdot 10^{-2}$	$4 \cdot 10^{5}$
2	929.17	1,722.34	$1.6 \cdot 10^{-2}$	$1.6 \cdot 10^{-2}$	52,249.4
1	101.14	264.2	$1.4 \cdot 10^{-2}$	$1.6 \cdot 10^{-2}$	11,199.34

Table B.4: Query time of all stores.
(Test Identifier: `t_all_stores`)

DS	SQL MySQL	SQL PostgreSQL	ORM MySQL	ORM PostgreSQL	MongoDB
3	8.76	7.8	1.71	$2.1 \cdot 10^{-2}$	27,886.52
2	5.48	5.94	$2 \cdot 10^{-2}$	$2.2 \cdot 10^{-2}$	5,742.59
1	2.9	1.93	$4 \cdot 10^{-2}$	$2.1 \cdot 10^{-2}$	3,500.16

Table B.5: Count query time of all customers.
(Test Identifier: `t_count_customers`)

DS	SQL MySQL	SQL PostgreSQL	ORM MySQL	ORM PostgreSQL	MongoDB
3	11.48	8.88	19.56	15.48	0.63
2	1.49	4.61	3.93	12.41	4.94
1	0.4	4.17	1.37	11.13	0.63

Table B.6: Count query time of all products.
(Test Identifier: `t_count_products`)

DS	SQL MySQL	SQL PostgreSQL	ORM MySQL	ORM PostgreSQL	MongoDB
3	237.14	80.57	122.8	84.41	0.53
2	45.44	14.54	22.92	22.6	0.57
1	5.98	2.97	4.93	5.04	0.42

Table B.7: Count query time of all purchases.
(Test Identifier: `t_count_purchases`)

DS	SQL MySQL	SQL PostgreSQL	ORM MySQL	ORM PostgreSQL	MongoDB
3	891.42	234.06	431.18	237.41	0.4
2	172.99	40.46	68.91	57.72	0.42
1	46.74	8.16	14.48	12.55	0.37

Table B.8: Count query time of all purchases.
(Test Identifier: `t_count_stores`)

DS	SQL MySQL	SQL PostgreSQL	ORM MySQL	ORM PostgreSQL	MongoDB
3	0.43	0.47	7.94	1.08	0.38
2	0.24	0.37	0.77	0.81	0.37
1	0.19	0.29	0.39	0.85	0.4

Table B.9: Revenue calculation query time of a single store.
(Test Identifier: `t_single_store_revenue`)

DS	SQL MySQL	SQL PostgreSQL	ORM MySQL	ORM PostgreSQL	MongoDB
3	163.83	197.2	762.79	21.51	1,457.97
2	141.64	874.93	25.9	19.48	1,043.66
1	46.98	25.85	11.74	29.07	827.57

Table B.10: Revenue calculation query time of all stores, sorted by most profitable.
(Test Identifier: `t_all_stores_revenue`)

DS	SQL MySQL	SQL PostgreSQL	ORM MySQL	ORM PostgreSQL	MongoDB
3	3,593.59	3,499.24	5,317.93	15,197.42	$2.48 \cdot 10^6$
2	558.08	669.49	658.36	1,868.83	$2.23 \cdot 10^5$
1	94.22	133.44	117.32	859.88	42,064.63

Table B.11: Expenditure calculation query time of a single customer over all stores.
(Test Identifier: `t_single_customer_expenditure`)

DS	SQL MySQL	SQL PostgreSQL	ORM MySQL	ORM PostgreSQL	MongoDB
3	2	19.29	9.76	3.77	5,182.94
2	2.93	58.73	9.5	3.14	161.31
1	0.55	13.82	5.11	3.34	36.65

Table B.12: Expenditure calculation query time of all customers at a particular store.
(Test Identifier: `t_single_customer_expenditure_at_store`)

DS	SQL MySQL	SQL PostgreSQL	ORM MySQL	ORM PostgreSQL	MongoDB
3	1.95	18.82	3.64	2.82	1,824.6
2	3.24	44.81	3.76	2.67	180.04
1	0.52	2.79	2.83	2.7	53.25

Table B.13: Expenditure calculation query time of all customers, sorted by most expended and limited to top 3 customers.
(Test Identifier: `t_top_spenders`)

DS	SQL MySQL	SQL PostgreSQL	ORM MySQL	ORM PostgreSQL	MongoDB
3	1.95	18.82	3.64	2.82	1,824.6
2	3.24	44.81	3.76	2.67	180.04
1	0.52	2.79	2.83	2.7	53.25

Table B.14: Expenditure calculation query time of all customers at a single store, sorted by most expended and limited to top 3 customers.
(Test Identifier: `t_top_spenders_at_store`)

DS	SQL MySQL	SQL PostgreSQL	ORM MySQL	ORM PostgreSQL	MongoDB
3	51.57	179	64,009.3	38,654.99	$2.92 \cdot 10^7$
2	5.96	325.9	4,316.54	6,304.65	$1.21 \cdot 10^6$
1	8.76	20.96	818.94	1,667.4	56,205.5

B.2 Test Result Output

The following log displays the raw output of all test cases that were made throughout the test.

Listing 30: Raw Output

```
[sql/mysql/large] Running Test: t_all_customers
[sql/mysql/large] Test Ended: t_all_customers in 126.769ms
[sql/mysql/large] Running Test: t_all_products
[sql/mysql/large] Test Ended: t_all_products in 1091.866ms
[sql/mysql/large] Running Test: t_all_purchases
[sql/mysql/large] Test Ended: t_all_purchases in 3217.018ms
[sql/mysql/large] Running Test: t_all_stores
[sql/mysql/large] Test Ended: t_all_stores in 8.758ms
[sql/mysql/large] Running Test: t_count_customers
[sql/mysql/large] Test Ended: t_count_customers in 11.475ms
[sql/mysql/large] Running Test: t_count_products
[sql/mysql/large] Test Ended: t_count_products in 237.137ms
[sql/mysql/large] Running Test: t_count_purchases
[sql/mysql/large] Test Ended: t_count_purchases in 891.424ms
[sql/mysql/large] Running Test: t_count_stores
[sql/mysql/large] Test Ended: t_count_stores in 0.429ms
[sql/mysql/large] Running Test: t_single_store_revenue
[sql/mysql/large] Test Ended: t_single_store_revenue in 163.83ms
[sql/mysql/large] Running Test: t_all_stores_revenue
[sql/mysql/large] Test Ended: t_all_stores_revenue in 3593.589ms
[sql/mysql/large] Running Test: t_single_customer_expenditure
[sql/mysql/large] Test Ended: t_single_customer_expenditure in 2.001ms
[sql/mysql/large] Running Test: t_single_customer_expenditure_at_store
[sql/mysql/large] Test Ended: t_single_customer_expenditure_at_store in 1.949ms
[sql/mysql/large] Running Test: t_top_spenders
[sql/mysql/large] Test Ended: t_top_spenders in 60931.109ms
[sql/mysql/large] Running Test: t_top_spenders_at_store
[sql/mysql/large] Test Ended: t_top_spenders_at_store in 51.573ms
[sql/mysql/medium]  Running Test: t_all_customers
[sql/mysql/medium]  Test Ended: t_all_customers in 23.668ms
[sql/mysql/medium]  Running Test: t_all_products
[sql/mysql/medium]  Test Ended: t_all_products in 199.221ms
[sql/mysql/medium]  Running Test: t_all_purchases
[sql/mysql/medium]  Test Ended: t_all_purchases in 929.168ms
[sql/mysql/medium]  Running Test: t_all_stores
[sql/mysql/medium]  Test Ended: t_all_stores in 5.475ms
[sql/mysql/medium]  Running Test: t_count_customers
[sql/mysql/medium]  Test Ended: t_count_customers in 1.485ms
[sql/mysql/medium]  Running Test: t_count_products
[sql/mysql/medium]  Test Ended: t_count_products in 45.438ms
[sql/mysql/medium]  Running Test: t_count_purchases
[sql/mysql/medium]  Test Ended: t_count_purchases in 172.993ms
[sql/mysql/medium]  Running Test: t_count_stores
[sql/mysql/medium]  Test Ended: t_count_stores in 0.237ms
[sql/mysql/medium]  Running Test: t_single_store_revenue
[sql/mysql/medium]  Test Ended: t_single_store_revenue in 141.638ms
[sql/mysql/medium]  Running Test: t_all_stores_revenue
[sql/mysql/medium]  Test Ended: t_all_stores_revenue in 558.079ms
[sql/mysql/medium]  Running Test: t_single_customer_expenditure
[sql/mysql/medium]  Test Ended: t_single_customer_expenditure in 2.932ms
[sql/mysql/medium]  Running Test: t_single_customer_expenditure_at_store
[sql/mysql/medium]  Test Ended: t_single_customer_expenditure_at_store in 3.244ms
[sql/mysql/medium]  Running Test: t_top_spenders
[sql/mysql/medium]  Test Ended: t_top_spenders in 1640.037ms
[sql/mysql/medium]  Running Test: t_top_spenders_at_store
```

```
[sql/mysql/medium]  Test Ended: t_top_spenders_at_store in 5.961ms
[sql/mysql/small] Running Test: t_all_customers
[sql/mysql/small] Test Ended: t_all_customers in 9.142ms
[sql/mysql/small] Running Test: t_all_products
[sql/mysql/small] Test Ended: t_all_products in 38.584ms
[sql/mysql/small] Running Test: t_all_purchases
[sql/mysql/small] Test Ended: t_all_purchases in 101.141ms
[sql/mysql/small] Running Test: t_all_stores
[sql/mysql/small] Test Ended: t_all_stores in 2.904ms
[sql/mysql/small] Running Test: t_count_customers
[sql/mysql/small] Test Ended: t_count_customers in 0.404ms
[sql/mysql/small] Running Test: t_count_products
[sql/mysql/small] Test Ended: t_count_products in 5.98ms
[sql/mysql/small] Running Test: t_count_purchases
[sql/mysql/small] Test Ended: t_count_purchases in 46.741ms
[sql/mysql/small] Running Test: t_count_stores
[sql/mysql/small] Test Ended: t_count_stores in 0.185ms
[sql/mysql/small] Running Test: t_single_store_revenue
[sql/mysql/small] Test Ended: t_single_store_revenue in 46.978ms
[sql/mysql/small] Running Test: t_all_stores_revenue
[sql/mysql/small] Test Ended: t_all_stores_revenue in 94.219ms
[sql/mysql/small] Running Test: t_single_customer_expenditure
[sql/mysql/small] Test Ended: t_single_customer_expenditure in 0.55ms
[sql/mysql/small] Running Test: t_single_customer_expenditure_at_store
[sql/mysql/small] Test Ended: t_single_customer_expenditure_at_store in 0.518ms
[sql/mysql/small] Running Test: t_top_spenders
[sql/mysql/small] Test Ended: t_top_spenders in 313.423ms
[sql/mysql/small] Running Test: t_top_spenders_at_store
[sql/mysql/small] Test Ended: t_top_spenders_at_store in 8.76ms
[sql/postgresql/large]  Running Test: t_all_customers
[sql/postgresql/large]  Test Ended: t_all_customers in 160.188ms
[sql/postgresql/large]  Running Test: t_all_products
[sql/postgresql/large]  Test Ended: t_all_products in 1192.175ms
[sql/postgresql/large]  Running Test: t_all_purchases
[sql/postgresql/large]  Test Ended: t_all_purchases in 5890.348ms
[sql/postgresql/large]  Running Test: t_all_stores
[sql/postgresql/large]  Test Ended: t_all_stores in 7.803ms
[sql/postgresql/large]  Running Test: t_count_customers
[sql/postgresql/large]  Test Ended: t_count_customers in 8.88ms
[sql/postgresql/large]  Running Test: t_count_products
[sql/postgresql/large]  Test Ended: t_count_products in 80.565ms
[sql/postgresql/large]  Running Test: t_count_purchases
[sql/postgresql/large]  Test Ended: t_count_purchases in 234.059ms
[sql/postgresql/large]  Running Test: t_count_stores
[sql/postgresql/large]  Test Ended: t_count_stores in 0.465ms
[sql/postgresql/large]  Running Test: t_single_store_revenue
[sql/postgresql/large]  Test Ended: t_single_store_revenue in 197.198ms
[sql/postgresql/large]  Running Test: t_all_stores_revenue
[sql/postgresql/large]  Test Ended: t_all_stores_revenue in 3499.238ms
[sql/postgresql/large]  Running Test: t_single_customer_expenditure
[sql/postgresql/large]  Test Ended: t_single_customer_expenditure in 19.288ms
[sql/postgresql/large]  Running Test: t_single_customer_expenditure_at_store
[sql/postgresql/large]  Test Ended: t_single_customer_expenditure_at_store in 18.817ms
[sql/postgresql/large]  Running Test: t_top_spenders
[sql/postgresql/large]  Test Ended: t_top_spenders in 6909.031ms
```

```
[sql/postgresql/large]  Running Test: t_top_spenders_at_store
[sql/postgresql/large]  Test Ended: t_top_spenders_at_store in 179.0ms
[sql/postgresql/medium] Running Test: t_all_customers
[sql/postgresql/medium] Test Ended: t_all_customers in 66.912ms
[sql/postgresql/medium] Running Test: t_all_products
[sql/postgresql/medium] Test Ended: t_all_products in 360.136ms
[sql/postgresql/medium] Running Test: t_all_purchases
[sql/postgresql/medium] Test Ended: t_all_purchases in 1722.338ms
[sql/postgresql/medium] Running Test: t_all_stores
[sql/postgresql/medium] Test Ended: t_all_stores in 5.937ms
[sql/postgresql/medium] Running Test: t_count_customers
[sql/postgresql/medium] Test Ended: t_count_customers in 4.608ms
[sql/postgresql/medium] Running Test: t_count_products
[sql/postgresql/medium] Test Ended: t_count_products in 14.54ms
[sql/postgresql/medium] Running Test: t_count_purchases
[sql/postgresql/medium] Test Ended: t_count_purchases in 40.455ms
[sql/postgresql/medium] Running Test: t_count_stores
[sql/postgresql/medium] Test Ended: t_count_stores in 0.373ms
[sql/postgresql/medium] Running Test: t_single_store_revenue
[sql/postgresql/medium] Test Ended: t_single_store_revenue in 874.926ms
[sql/postgresql/medium] Running Test: t_all_stores_revenue
[sql/postgresql/medium] Test Ended: t_all_stores_revenue in 669.493ms
[sql/postgresql/medium] Running Test: t_single_customer_expenditure
[sql/postgresql/medium] Test Ended: t_single_customer_expenditure in 58.728ms
[sql/postgresql/medium] Running Test: t_single_customer_expenditure_at_store
[sql/postgresql/medium] Test Ended: t_single_customer_expenditure_at_store in 44.813ms
[sql/postgresql/medium] Running Test: t_top_spenders
[sql/postgresql/medium] Test Ended: t_top_spenders in 866.863ms
[sql/postgresql/medium] Running Test: t_top_spenders_at_store
[sql/postgresql/medium] Test Ended: t_top_spenders_at_store in 325.898ms
[sql/postgresql/small]  Running Test: t_all_customers
[sql/postgresql/small]  Test Ended: t_all_customers in 57.504ms
[sql/postgresql/small]  Running Test: t_all_products
[sql/postgresql/small]  Test Ended: t_all_products in 51.879ms
[sql/postgresql/small]  Running Test: t_all_purchases
[sql/postgresql/small]  Test Ended: t_all_purchases in 264.197ms
[sql/postgresql/small]  Running Test: t_all_stores
[sql/postgresql/small]  Test Ended: t_all_stores in 1.925ms
[sql/postgresql/small]  Running Test: t_count_customers
[sql/postgresql/small]  Test Ended: t_count_customers in 4.172ms
[sql/postgresql/small]  Running Test: t_count_products
[sql/postgresql/small]  Test Ended: t_count_products in 2.968ms
[sql/postgresql/small]  Running Test: t_count_purchases
[sql/postgresql/small]  Test Ended: t_count_purchases in 8.155ms
[sql/postgresql/small]  Running Test: t_count_stores
[sql/postgresql/small]  Test Ended: t_count_stores in 0.285ms
[sql/postgresql/small]  Running Test: t_single_store_revenue
[sql/postgresql/small]  Test Ended: t_single_store_revenue in 25.85ms
[sql/postgresql/small]  Running Test: t_all_stores_revenue
[sql/postgresql/small]  Test Ended: t_all_stores_revenue in 133.435ms
[sql/postgresql/small]  Running Test: t_single_customer_expenditure
[sql/postgresql/small]  Test Ended: t_single_customer_expenditure in 13.82ms
[sql/postgresql/small]  Running Test: t_single_customer_expenditure_at_store
[sql/postgresql/small]  Test Ended: t_single_customer_expenditure_at_store in 2.79ms
[sql/postgresql/small]  Running Test: t_top_spenders
```

```
[sql/postgresql/small]  Test Ended: t_top_spenders in 153.21ms
[sql/postgresql/small]  Running Test: t_top_spenders_at_store
[sql/postgresql/small]  Test Ended: t_top_spenders_at_store in 20.961ms
[orm/mysql/large] Running Test: t_all_customers
[orm/mysql/large] Test Ended: t_all_customers in 22.299ms
[orm/mysql/large] Running Test: t_all_products
[orm/mysql/large] Test Ended: t_all_products in 3.22ms
[orm/mysql/large] Running Test: t_all_purchases
[orm/mysql/large] Test Ended: t_all_purchases in 0.965ms
[orm/mysql/large] Running Test: t_all_stores
[orm/mysql/large] Test Ended: t_all_stores in 1.709ms
[orm/mysql/large] Running Test: t_count_customers
[orm/mysql/large] Test Ended: t_count_customers in 19.562ms
[orm/mysql/large] Running Test: t_count_products
[orm/mysql/large] Test Ended: t_count_products in 122.802ms
[orm/mysql/large] Running Test: t_count_purchases
[orm/mysql/large] Test Ended: t_count_purchases in 431.177ms
[orm/mysql/large] Running Test: t_count_stores
[orm/mysql/large] Test Ended: t_count_stores in 7.944ms
[orm/mysql/large] Running Test: t_single_store_revenue
[orm/mysql/large] Test Ended: t_single_store_revenue in 762.789ms
[orm/mysql/large] Running Test: t_all_stores_revenue
[orm/mysql/large] Test Ended: t_all_stores_revenue in 5317.934ms
[orm/mysql/large] Running Test: t_single_customer_expenditure
[orm/mysql/large] Test Ended: t_single_customer_expenditure in 9.761ms
[orm/mysql/large] Running Test: t_single_customer_expenditure_at_store
[orm/mysql/large] Test Ended: t_single_customer_expenditure_at_store in 3.635ms
[orm/mysql/large] Running Test: t_top_spenders
[orm/mysql/large] Test Ended: t_top_spenders in 62083.091ms
[orm/mysql/large] Running Test: t_top_spenders_at_store
[orm/mysql/large] Test Ended: t_top_spenders_at_store in 64009.299ms
[orm/mysql/medium]  Running Test: t_all_customers
[orm/mysql/medium]  Test Ended: t_all_customers in 0.03ms
[orm/mysql/medium]  Running Test: t_all_products
[orm/mysql/medium]  Test Ended: t_all_products in 0.022ms
[orm/mysql/medium]  Running Test: t_all_purchases
[orm/mysql/medium]  Test Ended: t_all_purchases in 0.016ms
[orm/mysql/medium]  Running Test: t_all_stores
[orm/mysql/medium]  Test Ended: t_all_stores in 0.02ms
[orm/mysql/medium]  Running Test: t_count_customers
[orm/mysql/medium]  Test Ended: t_count_customers in 3.926ms
[orm/mysql/medium]  Running Test: t_count_products
[orm/mysql/medium]  Test Ended: t_count_products in 22.919ms
[orm/mysql/medium]  Running Test: t_count_purchases
[orm/mysql/medium]  Test Ended: t_count_purchases in 68.912ms
[orm/mysql/medium]  Running Test: t_count_stores
[orm/mysql/medium]  Test Ended: t_count_stores in 0.772ms
[orm/mysql/medium]  Running Test: t_single_store_revenue
[orm/mysql/medium]  Test Ended: t_single_store_revenue in 25.904ms
[orm/mysql/medium]  Running Test: t_all_stores_revenue
[orm/mysql/medium]  Test Ended: t_all_stores_revenue in 658.357ms
[orm/mysql/medium]  Running Test: t_single_customer_expenditure
[orm/mysql/medium]  Test Ended: t_single_customer_expenditure in 9.502ms
[orm/mysql/medium]  Running Test: t_single_customer_expenditure_at_store
[orm/mysql/medium]  Test Ended: t_single_customer_expenditure_at_store in 3.758ms
```

```
[orm/mysql/medium]  Running Test: t_top_spenders
[orm/mysql/medium]  Test Ended: t_top_spenders in 5035.192ms
[orm/mysql/medium]  Running Test: t_top_spenders_at_store
[orm/mysql/medium]  Test Ended: t_top_spenders_at_store in 4316.536ms
[orm/mysql/small] Running Test: t_all_customers
[orm/mysql/small] Test Ended: t_all_customers in 0.038ms
[orm/mysql/small] Running Test: t_all_products
[orm/mysql/small] Test Ended: t_all_products in 0.02ms
[orm/mysql/small] Running Test: t_all_purchases
[orm/mysql/small] Test Ended: t_all_purchases in 0.014ms
[orm/mysql/small] Running Test: t_all_stores
[orm/mysql/small] Test Ended: t_all_stores in 0.04ms
[orm/mysql/small] Running Test: t_count_customers
[orm/mysql/small] Test Ended: t_count_customers in 1.374ms
[orm/mysql/small] Running Test: t_count_products
[orm/mysql/small] Test Ended: t_count_products in 4.925ms
[orm/mysql/small] Running Test: t_count_purchases
[orm/mysql/small] Test Ended: t_count_purchases in 14.478ms
[orm/mysql/small] Running Test: t_count_stores
[orm/mysql/small] Test Ended: t_count_stores in 0.389ms
[orm/mysql/small] Running Test: t_single_store_revenue
[orm/mysql/small] Test Ended: t_single_store_revenue in 11.741ms
[orm/mysql/small] Running Test: t_all_stores_revenue
[orm/mysql/small] Test Ended: t_all_stores_revenue in 117.324ms
[orm/mysql/small] Running Test: t_single_customer_expenditure
[orm/mysql/small] Test Ended: t_single_customer_expenditure in 5.111ms
[orm/mysql/small] Running Test: t_single_customer_expenditure_at_store
[orm/mysql/small] Test Ended: t_single_customer_expenditure_at_store in 2.831ms
[orm/mysql/small] Running Test: t_top_spenders
[orm/mysql/small] Test Ended: t_top_spenders in 987.56ms
[orm/mysql/small] Running Test: t_top_spenders_at_store
[orm/mysql/small] Test Ended: t_top_spenders_at_store in 818.939ms
[orm/postgresql/large]  Running Test: t_all_customers
[orm/postgresql/large]  Test Ended: t_all_customers in 0.101ms
[orm/postgresql/large]  Running Test: t_all_products
[orm/postgresql/large]  Test Ended: t_all_products in 0.021ms
[orm/postgresql/large]  Running Test: t_all_purchases
[orm/postgresql/large]  Test Ended: t_all_purchases in 0.015ms
[orm/postgresql/large]  Running Test: t_all_stores
[orm/postgresql/large]  Test Ended: t_all_stores in 0.021ms
[orm/postgresql/large]  Running Test: t_count_customers
[orm/postgresql/large]  Test Ended: t_count_customers in 15.484ms
[orm/postgresql/large]  Running Test: t_count_products
[orm/postgresql/large]  Test Ended: t_count_products in 84.41ms
[orm/postgresql/large]  Running Test: t_count_purchases
[orm/postgresql/large]  Test Ended: t_count_purchases in 237.414ms
[orm/postgresql/large]  Running Test: t_count_stores
[orm/postgresql/large]  Test Ended: t_count_stores in 1.083ms
[orm/postgresql/large]  Running Test: t_single_store_revenue
[orm/postgresql/large]  Test Ended: t_single_store_revenue in 21.512ms
[orm/postgresql/large]  Running Test: t_all_stores_revenue
[orm/postgresql/large]  Test Ended: t_all_stores_revenue in 15197.415ms
[orm/postgresql/large]  Running Test: t_single_customer_expenditure
[orm/postgresql/large]  Test Ended: t_single_customer_expenditure in 3.769ms
[orm/postgresql/large]  Running Test: t_single_customer_expenditure_at_store
```

```
[orm/postgresql/large]  Test Ended: t_single_customer_expenditure_at_store in 2.817ms
[orm/postgresql/large]  Running Test: t_top_spenders
[orm/postgresql/large]  Test Ended: t_top_spenders in 42510.035ms
[orm/postgresql/large]  Running Test: t_top_spenders_at_store
[orm/postgresql/large]  Test Ended: t_top_spenders_at_store in 38654.985ms
[orm/postgresql/medium] Running Test: t_all_customers
[orm/postgresql/medium] Test Ended: t_all_customers in 0.032ms
[orm/postgresql/medium] Running Test: t_all_products
[orm/postgresql/medium] Test Ended: t_all_products in 0.02ms
[orm/postgresql/medium] Running Test: t_all_purchases
[orm/postgresql/medium] Test Ended: t_all_purchases in 0.016ms
[orm/postgresql/medium] Running Test: t_all_stores
[orm/postgresql/medium] Test Ended: t_all_stores in 0.022ms
[orm/postgresql/medium] Running Test: t_count_customers
[orm/postgresql/medium] Test Ended: t_count_customers in 12.409ms
[orm/postgresql/medium] Running Test: t_count_products
[orm/postgresql/medium] Test Ended: t_count_products in 22.599ms
[orm/postgresql/medium] Running Test: t_count_purchases
[orm/postgresql/medium] Test Ended: t_count_purchases in 57.721ms
[orm/postgresql/medium] Running Test: t_count_stores
[orm/postgresql/medium] Test Ended: t_count_stores in 0.812ms
[orm/postgresql/medium] Running Test: t_single_store_revenue
[orm/postgresql/medium] Test Ended: t_single_store_revenue in 19.478ms
[orm/postgresql/medium] Running Test: t_all_stores_revenue
[orm/postgresql/medium] Test Ended: t_all_stores_revenue in 1868.829ms
[orm/postgresql/medium] Running Test: t_single_customer_expenditure
[orm/postgresql/medium] Test Ended: t_single_customer_expenditure in 3.138ms
[orm/postgresql/medium] Running Test: t_single_customer_expenditure_at_store
[orm/postgresql/medium] Test Ended: t_single_customer_expenditure_at_store in 2.671ms
[orm/postgresql/medium] Running Test: t_top_spenders
[orm/postgresql/medium] Test Ended: t_top_spenders in 7924.876ms
[orm/postgresql/medium] Running Test: t_top_spenders_at_store
[orm/postgresql/medium] Test Ended: t_top_spenders_at_store in 6304.654ms
[orm/postgresql/small]  Running Test: t_all_customers
[orm/postgresql/small]  Test Ended: t_all_customers in 0.031ms
[orm/postgresql/small]  Running Test: t_all_products
[orm/postgresql/small]  Test Ended: t_all_products in 0.022ms
[orm/postgresql/small]  Running Test: t_all_purchases
[orm/postgresql/small]  Test Ended: t_all_purchases in 0.016ms
[orm/postgresql/small]  Running Test: t_all_stores
[orm/postgresql/small]  Test Ended: t_all_stores in 0.021ms
[orm/postgresql/small]  Running Test: t_count_customers
[orm/postgresql/small]  Test Ended: t_count_customers in 11.127ms
[orm/postgresql/small]  Running Test: t_count_products
[orm/postgresql/small]  Test Ended: t_count_products in 5.035ms
[orm/postgresql/small]  Running Test: t_count_purchases
[orm/postgresql/small]  Test Ended: t_count_purchases in 12.551ms
[orm/postgresql/small]  Running Test: t_count_stores
[orm/postgresql/small]  Test Ended: t_count_stores in 0.85ms
[orm/postgresql/small]  Running Test: t_single_store_revenue
[orm/postgresql/small]  Test Ended: t_single_store_revenue in 29.069ms
[orm/postgresql/small]  Running Test: t_all_stores_revenue
[orm/postgresql/small]  Test Ended: t_all_stores_revenue in 859.881ms
[orm/postgresql/small]  Running Test: t_single_customer_expenditure
[orm/postgresql/small]  Test Ended: t_single_customer_expenditure in 3.335ms
```

```
[orm/postgresql/small]  Running Test: t_single_customer_expenditure_at_store
[orm/postgresql/small]  Test Ended: t_single_customer_expenditure_at_store in 2.702ms
[orm/postgresql/small]  Running Test: t_top_spenders
[orm/postgresql/small]  Test Ended: t_top_spenders in 1237.173ms
[orm/postgresql/small]  Running Test: t_top_spenders_at_store
[orm/postgresql/small]  Test Ended: t_top_spenders_at_store in 1667.404ms
[orm/mongo/large] Running Test: t_all_customers
[orm/mongo/large] Test Ended: t_all_customers in 7484.523ms
[orm/mongo/large] Running Test: t_all_products
[orm/mongo/large] Test Ended: t_all_products in 70123.922ms
[orm/mongo/large] Running Test: t_all_purchases
[orm/mongo/large] Test Ended: t_all_purchases in 399510.916ms
[orm/mongo/large] Running Test: t_all_stores
[orm/mongo/large] Test Ended: t_all_stores in 27886.522ms
[orm/mongo/large] Running Test: t_count_customers
[orm/mongo/large] Test Ended: t_count_customers in 0.63ms
[orm/mongo/large] Running Test: t_count_products
[orm/mongo/large] Test Ended: t_count_products in 0.526ms
[orm/mongo/large] Running Test: t_count_purchases
[orm/mongo/large] Test Ended: t_count_purchases in 0.399ms
[orm/mongo/large] Running Test: t_count_stores
[orm/mongo/large] Test Ended: t_count_stores in 0.379ms
[orm/mongo/large] Running Test: t_single_store_revenue
[orm/mongo/large] Test Ended: t_single_store_revenue in 1457.965ms
[orm/mongo/large] Running Test: t_all_stores_revenue
[orm/mongo/large] Test Ended: t_all_stores_revenue in 2484307.501ms
[orm/mongo/large] Running Test: t_single_customer_expenditure
[orm/mongo/large] Test Ended: t_single_customer_expenditure in 5182.936ms
[orm/mongo/large] Running Test: t_single_customer_expenditure_at_store
[orm/mongo/large] Test Ended: t_single_customer_expenditure_at_store in 1824.599ms
[orm/mongo/large] Running Test: t_top_spenders
[orm/mongo/large] Test Ended: t_top_spenders in 29120345.539ms
[orm/mongo/large] Running Test: t_top_spenders_at_store
xorm/d[mongo/large] Test Ended: t_top_spenders_at_store in 29151839.022ms
[orm/mongo/medium]  Running Test: t_all_customers
[orm/mongo/medium]  Test Ended: t_all_customers in 1427.832ms
[orm/mongo/medium]  Running Test: t_all_products
[orm/mongo/medium]  Test Ended: t_all_products in 13434.947ms
[orm/mongo/medium]  Running Test: t_all_purchases
[orm/mongo/medium]  Test Ended: t_all_purchases in 52249.397ms
[orm/mongo/medium]  Running Test: t_all_stores
[orm/mongo/medium]  Test Ended: t_all_stores in 5742.585ms
[orm/mongo/medium]  Running Test: t_count_customers
[orm/mongo/medium]  Test Ended: t_count_customers in 4.937ms
[orm/mongo/medium]  Running Test: t_count_products
[orm/mongo/medium]  Test Ended: t_count_products in 0.568ms
[orm/mongo/medium]  Running Test: t_count_purchases
[orm/mongo/medium]  Test Ended: t_count_purchases in 0.423ms
[orm/mongo/medium]  Running Test: t_count_stores
[orm/mongo/medium]  Test Ended: t_count_stores in 0.367ms
[orm/mongo/medium]  Running Test: t_single_store_revenue
[orm/mongo/medium]  Test Ended: t_single_store_revenue in 1043.662ms
[orm/mongo/medium]  Running Test: t_all_stores_revenue
[orm/mongo/medium]  Test Ended: t_all_stores_revenue in 223312.334ms
[orm/mongo/medium]  Running Test: t_single_customer_expenditure
```

```
[orm/mongo/medium]  Test Ended: t_single_customer_expenditure in 161.307ms
[orm/mongo/medium]  Running Test: t_single_customer_expenditure_at_store
[orm/mongo/medium]  Test Ended: t_single_customer_expenditure_at_store in 180.044ms
[orm/mongo/medium]  Running Test: t_top_spenders
[orm/mongo/medium]  Test Ended: t_top_spenders in 1220236.079ms
[orm/mongo/medium]  Running Test: t_top_spenders_at_store
[orm/mongo/medium]  Test Ended: t_top_spenders_at_store in 1213900.848ms
[orm/mongo/small] Running Test: t_all_customers
[orm/mongo/small] Test Ended: t_all_customers in 279.951ms
[orm/mongo/small] Running Test: t_all_products
[orm/mongo/small] Test Ended: t_all_products in 3023.829ms
[orm/mongo/small] Running Test: t_all_purchases
[orm/mongo/small] Test Ended: t_all_purchases in 11199.337ms
[orm/mongo/small] Running Test: t_all_stores
[orm/mongo/small] Test Ended: t_all_stores in 3500.16ms
[orm/mongo/small] Running Test: t_count_customers
[orm/mongo/small] Test Ended: t_count_customers in 0.632ms
[orm/mongo/small] Running Test: t_count_products
[orm/mongo/small] Test Ended: t_count_products in 0.415ms
[orm/mongo/small] Running Test: t_count_purchases
[orm/mongo/small] Test Ended: t_count_purchases in 0.369ms
[orm/mongo/small] Running Test: t_count_stores
[orm/mongo/small] Test Ended: t_count_stores in 0.399ms
[orm/mongo/small] Running Test: t_single_store_revenue
[orm/mongo/small] Test Ended: t_single_store_revenue in 827.567ms
[orm/mongo/small] Running Test: t_all_stores_revenue
[orm/mongo/small] Test Ended: t_all_stores_revenue in 42064.63ms
[orm/mongo/small] Running Test: t_single_customer_expenditure
[orm/mongo/small] Test Ended: t_single_customer_expenditure in 36.651ms
[orm/mongo/small] Running Test: t_single_customer_expenditure_at_store
[orm/mongo/small] Test Ended: t_single_customer_expenditure_at_store in 53.245ms
[orm/mongo/small] Running Test: t_top_spenders
[orm/mongo/small] Test Ended: t_top_spenders in 55434.563ms
[orm/mongo/small] Running Test: t_top_spenders_at_store
[orm/mongo/small] Test Ended: t_top_spenders_at_store in 56205.503ms
```