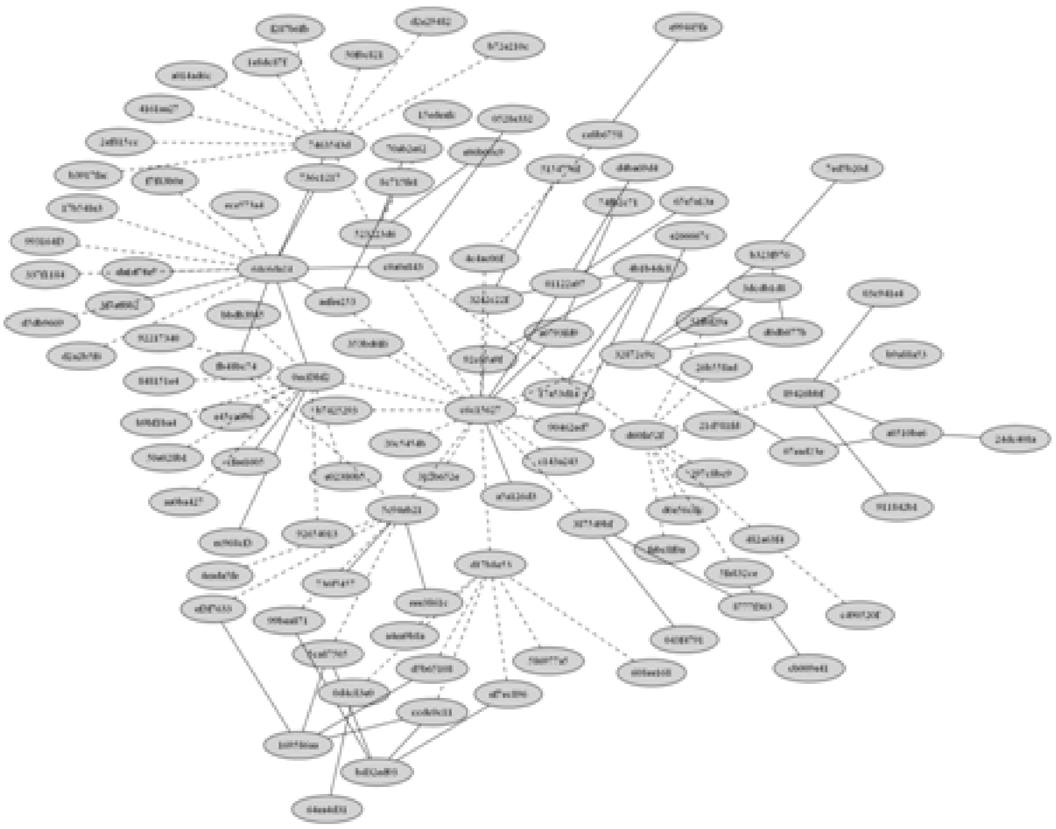
Graph-Based Querying
On top of the Entity Framework



Omar Pakker
omarpakker+uva@gmail.com

November 4, 2014, 101 pages

Supervisor: Jurgen J. Vinju
Host organisation: University of Amsterdam, http://www.uva.nl/en/home

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Contents

Abstract 4

1 Motivation 5
- 1.1 Problem . . . 5
- 1.2 Existing Solutions . . . 5
- 1.3 Problems with the Existing Solutions . . . 5
  - 1.3.1 Problems with OODBMS' . . . 5
  - 1.3.2 Problems with ORMs . . . 5
- 1.4 Solution . . . 6
- 1.5 Solution Validation . . . 7
- 1.6 Contribution . . . 7

2 Background 9
- 2.1 Relational Databases . . . 9
- 2.2 Object/Relational Impedance Mismatch . . . 9
- 2.3 Object-Oriented Databases . . . 9
- 2.4 Object-Relational Mapping Frameworks . . . 10
- 2.5 Graphs & the application in Graph-Based Querying . . . 10
- 2.6 Cognitive Dimensions . . . 10
- 2.7 Query Performance . . . 11
  - 2.7.1 Table size, length & amount . . . 11
  - 2.7.2 Amount of relations . . . 12
  - 2.7.3 Amount of queries & round-trips . . . 12
  - 2.7.4 Vendor data provider implementation . . . 13
  - 2.7.5 Object-to-Table mapping . . . 13
  - 2.7.6 Garbage Collection . . . 13
  - 2.7.7 Pre-fetching . . . 13
  - 2.7.8 Entity Framework . . . 13
  - 2.7.9 Hardware . . . 14
  - 2.7.10 Software . . . 15

3 Related Work 16
- 3.1 Graph Matching & Rewriting . . . 16
- 3.2 Other Query Languages . . . 16
  - 3.2.1 Query by Example (QBE) . . . 16
  - 3.2.2 SciQL . . . 17
  - 3.2.3 LINQ . . . 17
  - 3.2.4 XPath . . . 17
  - 3.2.5 JXPath . . . 17
  - 3.2.6 XQuery . . . 18
  - 3.2.7 Triple Graph Grammars . . . 18

4 Implementation Specifications 19
- 4.1 Features . . . 19

4.1.1 Feature Differences . . . 19
4.1.2 Supported Relations . . . 20
4.2 Implementation Analysis . . . 21
4.2.1 Graph-Based Querying . . . 21
4.2.2 Measurements . . . 21
4.2.3 Model Definitions . . . 24

5 Syntax **27**
5.1 Hypotheses . . . 27
5.2 Test Setup . . . 28
5.2.1 Cognitive Dimensions . . . 28
5.2.2 Code Snippet Selection . . . 29
5.3 Analysis . . . 32
5.3.1 Direct association . . . 33
5.3.2 Associations on subtypes . . . 35
5.3.3 Northwind (direct associations) . . . 38
5.4 Conclusion . . . 38
5.5 Threats to Validity . . . 38

6 Performance **39**
6.1 Hypotheses . . . 39
6.2 Test Setup . . . 40
6.2.1 Test Data . . . 40
6.2.2 System Specifications . . . 43
6.2.3 Timer Specifications . . . 44
6.2.4 Databases . . . 44
6.2.5 Measurement Method . . . 44
6.2.6 Test Groups . . . 45
6.3 Analysis . . . 45
6.3.1 Inheritance only . . . 45
6.3.2 Associations on Subtypes . . . 46
6.4 Conclusion . . . 47
6.5 Threats to Validity . . . 47

7 Future Work **48**
7.1 Implementation Method . . . 48
7.2 Additional Functionality . . . 48
7.3 Graph Operations . . . 48
7.4 Query Construction . . . 48
7.5 Entity Framework's Future . . . 49

8 Acknowledgements **50**

Bibliography **51**

Appendix A Measurement Results **55**
A.1 No associations on sub-types . . . 55
A.1.1 Inheritance 3 . . . 55
A.1.2 Inheritance 4 . . . 58
A.1.3 Inheritance 5 . . . 61
A.1.4 Inheritance 6 . . . 64
A.2 Associations on sub-types . . . 67
A.2.1 Inheritance 3 . . . 67
A.2.2 Inheritance 4 . . . 75
A.2.3 Inheritance 5 . . . 83
A.2.4 Inheritance 6 . . . 91

A.3 Northwind 99

Abstract

Background - Requesting complex structures of related data from relational databases that do not have a single object defined in the object model of your ORM often requires numerous data requests to the ORM (and thus database), the use of code that represents SQL, or in the worst case, actual SQL code. As such, you run into the relational model within your program code which is one of things the ORM was supposed to prevent.
Purpose - The purpose of this thesis was to investigate graphs as a possible solution to representing these complex data structures and using those to faster retrieve the data and provide developers with a way to define what to retrieve without coming into contact with the relational model.
Method - We compare the performance of it against the Entity Framework using two types of timers and across different databases and we compare the syntax using Cognitive Dimensions to evaluate which is easier to read and understand.
Results - We found that Graph-Based Querying can outperform the Entity Framework and that the syntax is easier to understand.
Conclusions - We conclude that we can still improve on the way we communicate with relational databases from an OO language and that Graph-Based Querying can be a possible solution as it both improves performance and improves the readability of the code.

Chapter 1

Motivation

1.1 Problem

The original relational model as defined by Codd is a widely used model in database systems. The biggest issue with the relational model and modern object-oriented languages is the object/relational mismatch, otherwise referred to as the impedance mismatch [Car96, IBNW09]. To solve this problem several solutions have been developed/pursued over the years but the integration and use of the relational model within general-purpose programming languages is a complex problem and existing solutions such as ORMs often incur a performance overhead.

1.2 Existing Solutions

To solve the Object/Relational mismatch, several solutions such as Object-Oriented databases and Object-Relational mappers are currently available. Object-Oriented databases were researched to great extent during the 1980's as one of the solutions to the object/relational impedance mismatch but they never surpassed the use of relational databases. The other solution to the object/relational mismatch was the use of Object-Relational mappers. An Object-Relational Mapper sits between the relational database and the program code and wraps the database types and relations into objects; it functions as a translation layer between the program code and the database.

1.3 Problems with the Existing Solutions

1.3.1 Problems with OODBMS'

With an OODBMS you will need to request each object type, after which relations can be accessed through the objects. This would require some code that represents a select (ie. db.TypeAObjects.Get(index)) after which the relations point directly to the related object. Furthermore, an OODMBS is bound to the programming language it is designed for whereas a RDBMS is not. This introduces the problem where old data can not easily be migrated to a new program if this is implemented in a different programming language nor easily shared between different applications (ie. a web interface in PHP for clients and a management application in C++ for the employees).

1.3.2 Problems with ORMs

Data retrieval with an ORM is behind the scenes a bit more complex than with an OODBMS. You are required to request the objects like you would in an OODBMS but once you want to access a relation, a new query has to be executed on the database to retrieve the data associated with this relation. This results in n queries for n relations and thus greatly impacts program performance, or the programmer is required to write code that represents a SQL query to retrieve all the relations in advance.

Listing 1.1: Entity Framework code for the retrieval of data in Figure 1.1

```
dbContext.OSet.Where(o => o.Id == SelectEntityId)
    .Include(‘‘E00’’);

dbContext.OSet.Where(o => o.Id == SelectEntityId)
    .Join(dbContext.E00Set, o => o.Id, e00 => e00.O_Id, (o, e00) => e00)
    .Join(dbContext.A00Set.OfType<A10>(), e00 => e00.Id, a10 => a10.E00_Id, (e00, a10) =>
        a10)
    .Join(dbContext.B00Set, a10 => a10.Id, b00 => b00.A10_Id, (a10, b00) => b00);
```

To illustrate what this code could look like, we try to retrieve the red objects/relations of the dataset shown in Figure 1.1.

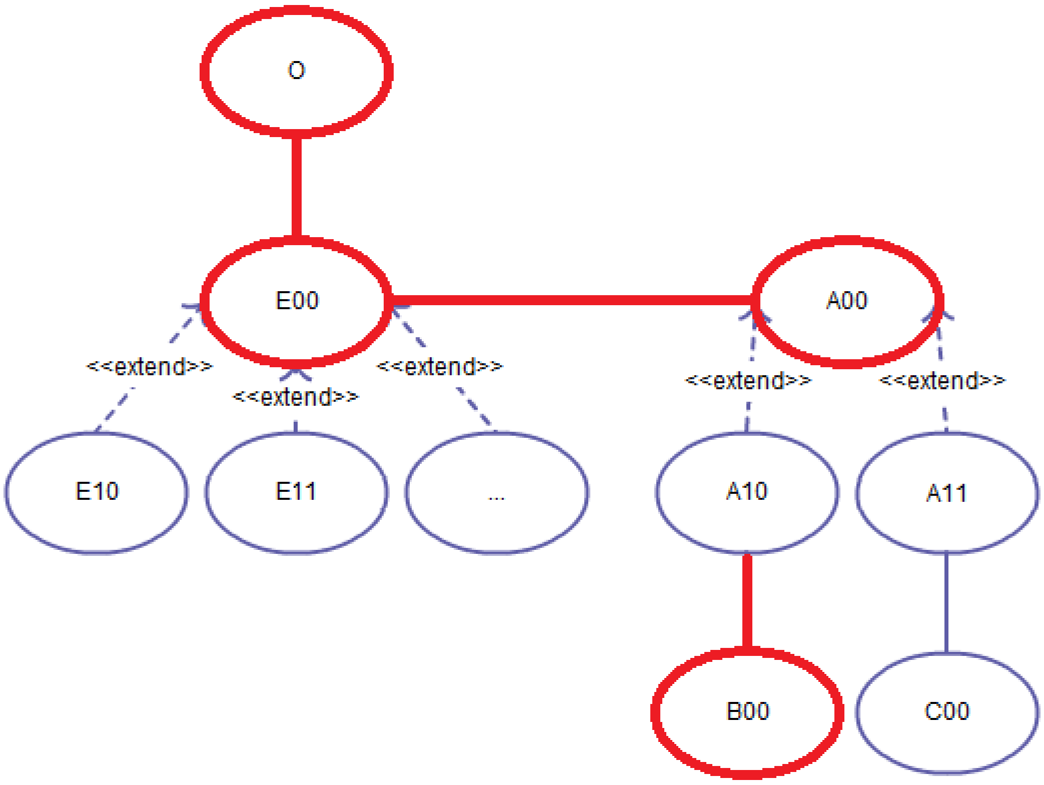


Figure 1.1: Retrieval of objects O, related object E00, its related A00 object, and for A10 (A00 sub-type) its related B00 object

As seen in [Mer11], the C# Entity Framework code used to retrieve this data represents SQL join and where statements (see Listing 1.1).

In Chapter 2 we go into more detail of both Object-Oriented databases and Object-Relational mappers.

1.4 Solution

This is where Graph-Based Querying (GBQ) comes in. Graph-Based Querying is the use of graphs to query for data from a database. Graph-Based Querying attempts to solve the problem of writing code that represents a SQL query and decreasing the amount of queries that need to be executed on

Listing 1.2: GBQ code for the retrieval of data in Figure 1.1

```
new SqlGraphShape(dbContext)
    .Edge<O>(x => x.E00Set)
    .Edge<E00>(x => x.A00Set)
    .Edge<A10>(x => x.B00Set)
    .Load<O>(o => o.Id == SelectEntityId);
```

the database to increase execution performance as well as improving code readability. We do this by using graphs to define the relations between the objects you want to retrieve from the database. The programmer defines a graph query and GBQ builds and executes a query to retrieve the data from the database.

To illustrate this, we try to retrieve the same data as before (see Figure 1.1) but now we write the code that you would need with GBQ as opposed to the Entity Framework. This code can bee seen in Listing 1.2. As we only need to know the objects and relations that are involved in the data we want to retrieve, we can greatly simplify the code required. As relations in a database are always made using a primary and foreign key, we can infer this information thus simplifying the code. Furthermore, as we no longer require the programmer to supply database fields for the data retrieval statement, we also eliminate the need for code that represents a SQL query.

Since this graph query also defines all the objects and relations that are involved in the data request at once, we can construct a query and execute this in a single round-trip to the database. This allows us to greatly increase the performance of the data retrieval when compared to the Entity Framework snippet (see Listing 1.1).

1.5 Solution Validation

To determine the performance benefits of Graph-Based Querying and the advantages and disadvantages of the syntax when compared to the Entity Framework, we created two experiments to answer the following questions:

1. Can Graph-Based Querying provide us with faster data retrieval?
2. Can the Graph-Based Querying syntax simplify code creation and maintainability?

To answer the first question we measure the performance of the Entity Framework against Graph-Based Querying using two timers; one for wall clock time and one for CPU time. The wall clock timer supplies us with the total duration of building, executing and constructing the objects for the database query. The CPU timer provides us with information on how long the Entity Framework and Graph-Based Querying take to construct the query and process the results.
The second question requires a way to compare different syntaxes against each other and what the impact of the syntax is on how a developer writes and reads code using that syntax. For this we use the Cognitive Dimensions framework as described by Green [BBC+01, T. 96]. This framework allows for the comparison of notational systems and information artefacts using several dimensions. These dimensions provide a way to discuss the differences and cognitive impact using broad terms. As such, this framework has been used, among other things, to compare diagrams [KBB02] and interfaces [Gol09]. We argue about the advantages and disadvantages of each syntax using several code snippets written with Graph-Based Querying and the Entity Framework.

1.6 Contribution

- *Verification of the results found in the article by M. de Jonge [Mer11].*
  We replicate the tests of the article by recreating the query structures, datasets and populations

used in the article.

- *Demonstration of the impact of Graph-Based Querying on different databases.*
  The article only tests on Sql Server. We expand on this by demonstrating the performance gains of Graph-Based Querying over the standard Entity Framework code on different databases in addition to Sql Server.

- *Syntax comparison using the Cognitive Dimensions framework.*
  With use of the cognitive dimensions framework we argue about the advantages and disadvantages of Graph-Based Querying code versus Entity Framework code.

- *Demonstration of the feasibility of Graph-Based Querying for the retrieval of complex data structures.*
  With the different tests we demonstrate when Graph-Based Querying performs better than the Entity Framework and when Graph-Based Querying becomes a feasible addition to the Entity Framework.

- *A code base for further development and testing.*
  The base for this project was created by M. de Jonge (see [Mer11]). It has been extended with support for a large amount of the features that can be found in the models that can be created with the Entity Framework, as well as the addition of support for other databases. We describe this in more detail in Chapter 4.
  The code will be made available for further testing and development. The following projects can be expected to be part of the code:

  – Performance measurement framework project.
  – General graph shape project (base for different implementations).
  – Graph-Based Querying implementation for the Entity Framework.
  – Project that defines the models and tests used for the measurements in this thesis.
  – A project for each database that configures the connection to the different databases.
  – Unit and functional tests for the general graph project and Entity Framework graph project.

Chapter 2

Background

2.1 Relational Databases

The relational model was first created by E.F. Codd [E.F69, E.F70]. Relational databases have a general programming interface; SQL [SQL]. Any programming language that can make connections to databases can execute the same SQL command and the database will be able to execute that command. However, as relational databases are not bound to a specific programming language and designed around the relational model, it suffers from the object/relational impedance mismatch problem. This problem is the set of problems encountered when mapping tables to objects and vice versa. To solve this, programmers utilize Object-Relational Mappers to deal with the object/relational mismatch.

2.2 Object/Relational Impedance Mismatch

The object/relational (impedance) mismatch is the set of problems encountered when mapping tables to objects and vice versa. The different data types in the relational database and the OO language is one of these problems. One example is the lack of by-reference (pointers) types in the relational model, while OO languages rely on by-reference types. Another example is the string collation: in the relational model string collation is defined with the column type beforehand, whereas in OO languages collation is often only used as an argument when comparing or sorting strings. In addition, relational databases do not support OO concepts such as encapsulation, accessibility (access modifiers), polymorphism, etc (see [IBNW09]).
Object-Relational mapping frameworks attempt to solve these mismatch problems by functioning as a translation layer between objects and the relational model.

2.3 Object-Oriented Databases

Object-Oriented databases have been considered as a possible alternative to relational databases in OO environments. In the 1980's OODBMS's were researched but they never managed to replace relational databases. Several factors for this are described by Carey [Car96].
In an Object-Oriented database, data is represented as objects as opposed to tables. The advantages of this is that the object-relational mismatch does not exist with this type of database. Furthermore, accessing data can be faster when compared to relational databases as objects can be retrieved directly; the objects directly reference the related object. In relational databases this requires a lookup in the target table. One of the disadvantages of when compared to a relational database is that an OODBMS is bound to a specific programming language and that data can not easily be shared between or migrated to applications written in a different programming language.

2.4 Object-Relational Mapping Frameworks

Object-Relational mapping frameworks are used as a translation layer between relational databases and the objects in a programming language. ORMs are another solution to the object/relational mismatch problem; the ORM hides the relational model and supplies the programmer with an object model instead. This simplifies development for the programmers by allowing the ORM to instantiate the corresponding objects and persisting the changes to the corresponding fields in the database. However, as seen in the Entity Framework snippet shown before (1.1), the programmer may still be required to write code that represents a SQL statement (the 'Join' statement used in the snippet is a prime example) and thus has to understand the relational model and SQL and often use it or write code that looks a lot like SQL instead of programming in idiomatic OO. Furthermore, the efficiency of the translation layer is unpredictable and can greatly differ for each ORM implementation.
As measured by P. van Zyl [Zyl06] we can see that a rather large overhead can be added by an ORM in several operations when compared to an OODBMS. However, he also concludes that in a few cases, the ORM did not perform slower. We describe several factors that impact performance in Section 2.7. We show that Graph-Based Querying can further increase performance.

Entity Framework The Entity Framework [Mic13] is an object-relational mapper for the .NET Framework. The Entity Framework was developed by Microsoft but the source-code has been publicly available since July 2012. The Entity Framework provides tools to create an abstract model on top of your relational model which maps objects to your database tables and cells and vice versa. It provides two APIs to make data request against the model; Entity SQL and LINQ-to-Entities [Atu07, Vij08]. The model can be created using code or a visual designer.
In Graph-Based Querying, we extract the mapping information from this model to construct the required queries for the defined graph.

2.5 Graphs & the application in Graph-Based Querying

A graph consists of nodes (vertices) that are connected by lines (edges). In computer science graphs serve several purposes, one of which is the representation of related data. In this case the objects are the vertices and the relations to other objects are the edges. For more in-depth information about graphs, we suggest the graph theory book by Diestel [Die12].
Graph Based Querying utilizes graphs for the representation of data to simplify how the programmer defines what data he wants to retrieve. In short, Graph-Based Querying constructs queries from graph shapes that consist of objects that represent database data.
Figure 2.1 shows the graph that defines the data for the code in Listing 1.1. To make graph creation as easy as possible, inheritance relations are inferred by Graph-Based Querying. In Graph-Based Querying the developer only needs to define a shape with edges from O to E00, E00 to A00 and A10 to B00.
Figure 2.2 shows how our GBQ implementation is positioned in relation to the database.

2.6 Cognitive Dimensions

As mentioned before in Section 1, we utilize the Cognitive Dimensions framework to argue about the code written with Graph-Based Querying and the Entity Framework to determine whether one has advantages or disadvantages over the other.
The Cognitive Dimensions framework [BBC+01] was created to assist with the qualitative evaluation of the design of notational systems and information artefacts. It focusses on design decisions and the trade-offs that occur between decisions.
To evaluate whether the code written with Graph-Based Querying is an improvement over the code written with the Entity Framework or not, we use these cognitive dimensions to weigh code snippets of both against each other. With the dimensions we discuss why the snippets are, or are not, easier to understand and maintain.

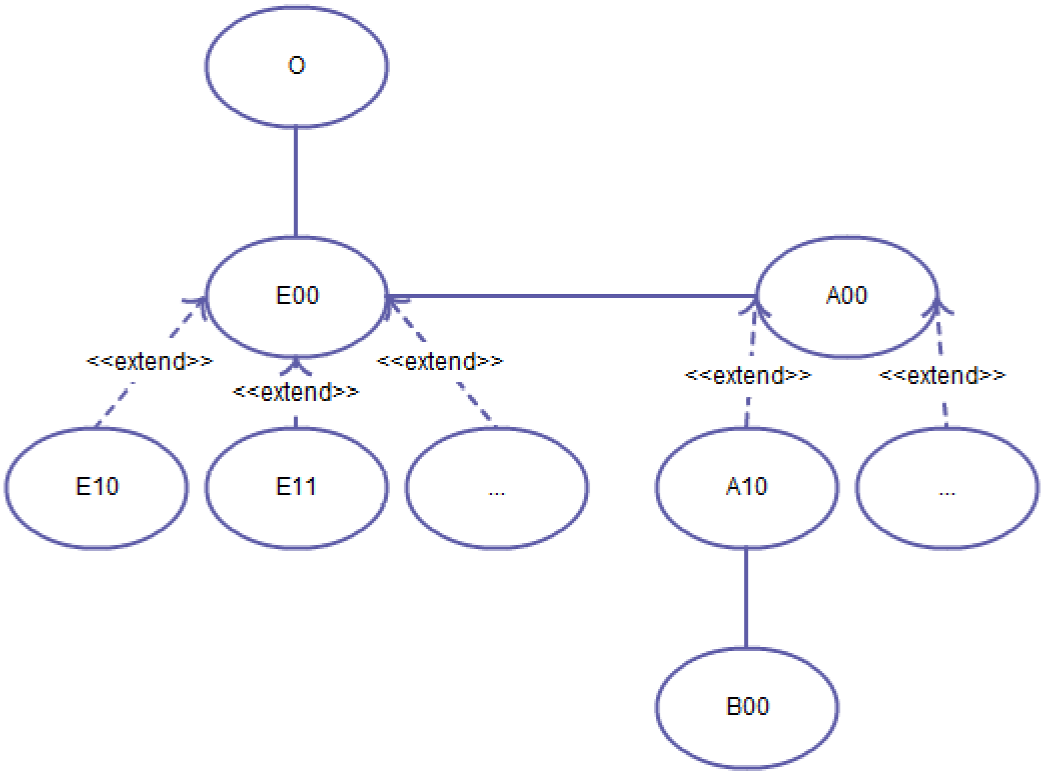


Figure 2.1: Graph representation of data. The circles are nodes/vertices and the lines are the edges.

2.7 Query Performance

The performance of queries and program execution can be influenced by several different factors. In this section we describe numerous factors that can influence the measured time and how this impacts the tests we perform to measure the performance difference between the Entity Framework and Graph-Based Querying. A few of these can be found in [Jer10].

2.7.1 Table size, length & amount

Larger tables or multiple tables require more time to be read from disk as well as more time to be transformed into objects by the ORM when compared to smaller or less tables.
As such, different table sizes can impact the measured time. To prevent this we use the same basic field types for each table:

- Id : int
- DataField : string (NVARCHAR (MAX))

Note that foreign key fields may be present if there is a relation.

The table length also impacts the measured time and therefore we define several database populations to measure the impact this has on Graph-Based Querying and the Entity Framework. In the comparative analysis only results from the same population are compared.

The amount of tables can also impact the measured time. To eliminate this, we test Graph-Based Querying and the Entity Framework against each other only on the same dataset. We created several datasets with an increasing amount of tables to evaluate the impact on each.

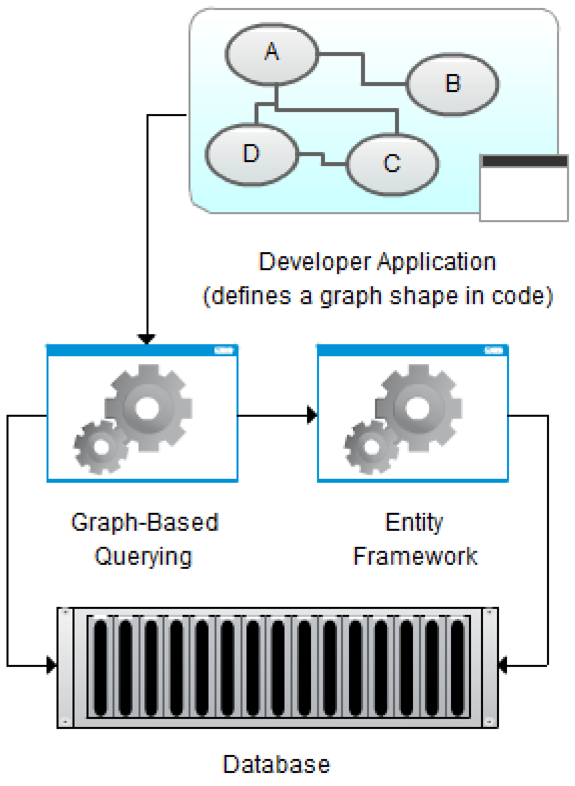


Figure 2.2: Graph-Based Querying in relation the developers' program and the database.

2.7.2 Amount of relations

Queries that have a greater amount of relations require the database to not only load a larger amount of tables, it also requires the database to perform a search to find the related entry. Both of these impact the measured time. The measured time increase caused by the searches grows with the length of the table and the amount of relations involved.
As we mentioned before, we ensure that our tables are the same length. This leaves the amount of relations as the only impact on the search time. To prevent the searches from impacting the comparison between Graph-Based Querying and the Entity Framework, we only compare datasets that have the same relations.
To measure the impact of more relations on the measured time we do test datasets with more relations. However, the smaller set is only compared to the bigger set queried with the same method; either Graph-Based Querying or the Entity Framework.

2.7.3 Amount of queries & round-trips

The amount of queries you need to execute to retrieve the requested data impacts the time the database needs to return the results. In case of the Entity Framework, each statement can be regarded as a query; each statement results in a query to the database. This not only increases the amount of queries to the database but also the amount of round-trips; each statement has to generate the SQL code, open a connection and materialize the objects from the returned data.
By creating a graph that defines all the associations the developer wants to retrieve, we attempt to decrease the amount of queries we need to execute as well as decrease the amount of round-trips. This is one of the main areas in which Graph-Based Querying improves performance.

2.7.4 Vendor data provider implementation

Vendor specific implementations of the data providers can influence the times we measure for Graph-Based Querying and the Entity Framework. It may well be possible for a specific vendor to implement the data provider in such a way that the queries generated for the Entity Framework are more efficient than the ones we generate with Graph-based Querying. As the data provider implementation is made for a specific database, it would allow for the use of advanced functionality in the database (ie. PL/SQL for Oracle or Transact-SQL for Sql Server). With Graph-Based Querying we generate queries that conform to the SQL standard and as such do not use database specific optimizations.
To investigate whether a data provider influences the Entity Framework performance when compared to Graph-Based Querying, we test both Graph-Based Querying and the Entity Framework against several different databases using their specific data providers.

2.7.5 Object-to-Table mapping

Mappings in an ORM describe how relational data is exposed to objects and how the objects are stored in tables. To make sure the data the user updates and object and retrieves the object from the database, the mapping must return the updated information. To prevent the wrong results an ORM must verify that the mappings are valid and that the mappings round-trip the data; the retrieval of data after it has been updated should return the correct data. As we can see in paper [BJP+13] this is a complex problem and in one of their cases the measured time dropped from 8 hours for full mapping compilation to 50 seconds with incremental compilation.
As Graph-Based Querying uses the mapping information from the Entity Framework, it suffers the same performance impact when the mappings are validated and therefore this does not impact the performance measurement comparison.

2.7.6 Garbage Collection

Garbage collection releases memory that has been used by objects that are no longer referenced. To do so, the collector can halt the running program to clean up these unused objects. This halt in program execution can impact the total time we measure during our tests. For this purpose we disable garbage collection while the tests are running and instead run the garbage collector before and after each test to prevent it from influencing our results.

2.7.7 Pre-fetching

Pre-fetching is the process of loading data from disk in advance. While pre-fetching is also a functionality you can find within Windows, this is disabled on our system and as such we only deal with pre-fetching functionality implemented in the database software.
Sql Server implements a pre-fetching system [Cra08, Fab12] which can impact our measurements as the population we use grows. As the queries we create differ from the queries the Entity Framework creates, the results may change once the pre-fetching is enabled. Pre-fetching is enabled when Sql Server's query plan assumes that the amount of rows that need to be analysed exceeds a certain threshold.

2.7.8 Entity Framework

As we use the Entity Framework, we also have to consider the overhead that the Entity Framework can add. An article by Microsoft ([Mic14e]) mentions several factors that influence the performance of the Entity Framework:

- *Cold vs. Warm Query Execution*
  As we can see in the article, the first query execution, or cold query, the Entity Framework has to load and validate the model. This increases the measured time. To prevent this from influencing our measurements, we execute the queries a few times before we start measuring.

- *Caching*
  The Entity Framework caches data on several levels; the metadata cache which is build with the first query, the query plan cache which stores the generated database commands if a query is executed more than once and the object cache which keeps track of objects that have been retrieved with a DbContext instance (also known as the first-level cache).
  As we mentioned before, we execute the queries several times before we start measuring. As such, the metadata and query plan caches are constructed and used thus do not impact the measured times.
  Each test uses a new DbContext instance and as such, the object cache is always clear for each new test.

- *Auto-compiled Queries*
  Before a query can be executed against the database it must go through a few steps. Query compilation is one of these. Subsequent calls with the same query allow the Entity Framework to use the cached plan and as such it can skip the plan compiler. The article mentions several conditions that may cause the plan to be recompiled. Our tests do not trigger any of these conditions and as such this does not impact our measurements.

- *NoTracking Queries*
  NoTracking basically disables the object cache and as such it may give a performance increase in read-only scenarios that do not request the same entity several times. When requesting the same entity several times, NoTracking makes it impossible to skip object materialization by using the object cache. In our tests we do not use the NoTracking functionality so all tests are impacted equally.

- *Query Execution Options*
  The Entity Framework support different way to construct queries. Of the options we could use we dropped the ones that do not also materialize the objects (i.e. EntityCommand queries; can be seen as a SQL query over the objects as opposed to over the tables) as this would skew the measurements in favour of the Entity Framework since Graph-Based Querying materializes objects. We also dropped the query methods that require SQL code (i.e. Store and SQL Queries). As we mentioned before, all our models utilize the DbContext and as such we also drop the queries that utilize the ObjectContext. The last two methods are Entity SQL and LINQ. As Entity SQL is similar to actual SQL but instead over entities, we also drop this as we want to use the Entity Framework in the most Object-Oriented way. This leaves us with the LINQ implementation.

- *Loading Related Entities*
  The Entity Framework can lazily load related objects or eagerly. If we where to use lazy loading for the Entity Framework we would give Graph-Based Querying an unfair advantage as the Entity Framework would need to make many more round-trips to the database when we use lazy loading as opposed to eager loading. As such we use eager loading. This creates a fair timing comparison as Graph-Based Querying also eagerly loads the requested data.

2.7.9 Hardware

The hardware can also impact the measured results. To minimize the differences in measured times we run the tests on the same system. However, the parts within the system can still impact the measured wall clock times:

- Disk I/O; the speed in which data is read from disk can change.
- CPU; cache clearing, buffer resets or throttling can influence the measurements.
- North- & Southbridge; the rate at which the bridges transfer data from disk to memory to the CPU can fluctuate.

To decrease the impact of the disk, we use an SSD to prevent seek time from impacting our measurements. While the SSD does not read in constant speed for everything, it does not need to move a reader head to the other side of the drive to read data thus the I/O impact is decreased.
CPU throttling has been disable to prevent this from impacting our wall clock measurements and the program is locked to a single core to prevent cache clearing and buffer resets caused by core switching to lessen the impact on our wall clock time.
We can not lessen the impact of the North- and Southbridge in our measurements.

2.7.10 Software

Other software running on the same system can also influence the measured wall clock time. While shutting down the majority of applications already decreases the load on the system, operating system crucial processes can not be terminated and can interrupt the thread.
To decrease the occurrences of this and thus the impact on the measurements, we give our process a higher priority than the other processes. The database processes are run normally.

Chapter 3

Related Work

In this chapter we describe several pieces of work that relate to data querying, as well as the use of graphs for this purpose. We first look at previous work in the area of graph matching and rewriting, followed by several different query languages that have been developed over the years.

We provide a short summary of each piece of work and how Graph-Based Querying makes use of the ideas of some of these approaches and how it is positioned in relation to this work.

3.1 Graph Matching & Rewriting

Previous research has been done in the usage of graphs for database programming. P.J. Rodgers designed an experimental visual database language (Spider) aimed at programmers [P.J97]. While this visual language makes the creation of complex data requests easy, the problem with this implementation is that there is no way to utilize this from within a programming language.
Another approach to work with graph transformations on relational databases was proposed by Varró [Var05]. This approach relies on the use of views in the database. The database views are used to define the matching patterns for the graph transformation. Such a view contains all the successful matchings for the rule. Inner joins are then used to handle the graph matching. The problem with this approach is that it requires the developer to define all the graphs as database views in advance. This requires the developer to access the database to create a new graph.
Graph-Based Querying sits between these approaches. Instead of defining a new language, it is implemented in an existing programming language. The graphs are defined through code, thereby allowing programmers to write the graph within their application as opposed to in the database. It also creates the queries for these graphs during runtime, as ORMs do for objects. This allows for programmers to create any type of graph they need without requiring access to the database to add new views.

3.2 Other Query Languages

3.2.1 Query by Example (QBE)

QBE is, like SQL, a language for querying relational data. It differs from SQL in that it is a graphical query language as opposed to a text-based query language. It was developed around the same time as SQL, during the 1970s, at IBM's Laboratory Research Center [Zlo77]. As a visual language it allows relatively inexperienced users to create simple queries without prior knowledge of query languages. However, QBE becomes less useful and has problems as the complexity of queries increase and it is less complete; it does not support universal or existential quantification [OW93].
With Graph-Based Querying we attempt to allow inexperienced users to create queries for complex data structures without prior knowledge of query languages. And as we mentioned before, by implementing it as a library the developer does not need to learn a visual language and can work within their application to define the data they want to retrieve.

3.2.2 SciQL

SciQL is a query language based on SQL, originally designed for scientific systems [Ker11]. It extends SQL with arrays as a first class type. A key innovation of this is the extension of SQL:2003 with structural grouping in addition to value based grouping. I.e., fixed sized and unbounded groups based on explicit relationships between their dimension attributes.
The main drawbacks of the approach SciQL uses is that it requires special implementation into the database and that there is no database that supports this by default. Furthermore, it focusses specifically on how to handle and work with array data, leaving the problem of retrieving large amounts of related information. SciQL provides no mechanics to more easily retrieve complex structures of related data. If we were to create a SQL extension such as SciQL, we would also need to expand existing ORMs to work on it. However, this would mean we still need to write code that represents this SQL extension, which brings us back where we started.
Because extensions to the SQL language such as SciQL are still bound to the relational model, these are not able to solve this problem.

3.2.3 LINQ

LINQ, which stands for Language INtegrated Query, is a set of features that brings powerful query capabilities to C# and Visual Basic. It can be extended to potentially support any type of data by the way of so called Data Providers. The default assemblies contain support for operating on .NET Framework collections, SQL Server databases, ADO.NET Datasets and XML documents.
Syntactically LINQ in a way represents SQL with its SELECT, WHERE, ORDERBY, etc. Even more so when you write the LINQ queries in the comprehension syntax, but in contrast to SQL, it will not even compile when the LINQ query is invalid thus saving you from run-time execution problems as you would have with SQL. Also, because LINQ is integrated into C# and Visual Basic, the programmer can work with a language he is familiar with so there is no need to learn a separate language. However, because it still represents SQL statements, it does not simplify the way in which you construct large complex data structures as you have to deal with the same keywords and same way of looking at data as with SQL; in a relational manner.

3.2.4 XPath

XPath is a language created to define (parts of) an XML document and utilizes path expressions to navigate the XML document. XPath became a W3C recommendation at November the 16th, 1999.
XPath works in a hierarchical manner and path expressions can be written in forms that match specific sub-paths in the hierarchy. It however does not support extensive query-like options such as joins and as such, data can not be filtered nor joined where needed. XPath is therefore more of a hierarchical selection language than a query language. While this language does allow you to easily define paths to retrieve data from, it does not retrieve the data of objects along the path. You can retrieve all the data by using several paths but this does not decrease the amount of statements you would need to write to retrieve the data and can not optimize the calls as one.
XPath forms the basis for query languages such as XQuery.

3.2.5 JXPath

JXPath is an interpreter of XPath written in Java [Apa]. It applies XPath expressions to graphs of objects. Just as with XPath, JXPath has no problem selecting paths or matching sub-paths in a data structure. It also supports the creation of objects within that data structure. However, just as XPath suffers from this problem, it does not support the retrieval of object along the path expression. It only retrieves the object(s) at the end of the path expression. Because of this, while it is possible to define paths relatively easy, it is not possible to retrieve all the data in the path using a single expression.

3.2.6 XQuery

XQuery is the language to query XML files and is build on XPath expressions [W3C]. It shares the same data model as XPath and supports the same functions and operations as XPath. XQuery became a W3C recommendation at January the 23rd, 2007.
XQuery is to XML as SQL is to databases. Because of this, XQuery forms no improvement over SQL in the construction of queries for complex data structures. While the path expressions it inherits from XPath are flexible and therefore allow for matching on sub-paths as well, the queries over these path expressions (also known as FLWOR-expressions) are analogous to SQLs SELECT, FROM and WHERE.

3.2.7 Triple Graph Grammars

Triple Graph Grammars are a technique to define the relation between two different models in a declarative way. TGGs have been around since the mid 1990's and has been used with a main focus on model-to-model transformations. TGGs are compiled into a forward and backward graph translation (bi-directional) that take a source or target graph as input and create the corresponding target or source graph as output. Because the relation between two models can not only be defined but also made operational, this bi-directional conversion is possible. This could even be used to synchronize and maintain correspondence between two models.

While TGGs have been around for a while, they still suffer from several fundamental problems that are still unsolved. 1) Most published approaches either use inefficient graph grammar parsing or backtracking algorithms or rely on not very well-defined constraints of processed TGGs. 2) Negative application conditions are either excluded or in such a way that it destroys the fundamental TGG properties. 3) No appropriate means for modularization, refinement and re-use of TGGs. These problems are further described by Schürr [Sch08] and Klar [Kla07].

Graph Based Querying represents the forward transformation part of TGGs. It transforms a defined graph (the source model) into a query (the target model) to be executed on the database. The implementation we created here does not support the backward (query to graph) transformation.
Because of this relation to TGGs, problems 1 and 3 play a role in Graph-Based Querying as well.
Problem 1 can result in the generation of inefficient queries, leading to degraded performance. For instance, if paths A->B->Z and A->B->C->D->Z are defined within Graph-Based Querying and the Zs of the latter path are a subset of the first, it would be inefficient to build and execute a query for the second path.
Problem 3 can prevent the re-use and extension of graphs. With Graph-Based Querying the graphs can only be re-used if the object model is the same. If either association fields or classes differ, are renamed or removed, Graph-Based Querying graphs can not be re-used if the graph uses any of these fields or classes.

Chapter 4

Implementation Specifications

In this chapter we describe the changes to Graph-Based Querying as implemented in [Mer11]. We describe how the solution is set up and how we attempt to improve query performance, as well as what additional relations and model functionality we support and how we match up to the relations supported by the Entity Framework.
We also describe how the models for our tests are defined and how we take the measurements for our tests.

4.1 Features

This implementation of Graph-Based Querying is based on the version by Merijn de Jonge [Mer11] and relies on the Entity Framework (also see Figure 2.2) for the mapping information. The original implementation does not support all the different features you can use in a model defined with the Entity Framework. This can result in run-time problems if you use a model that is not created for use with GBQ in mind. As such, our implementation attempts to support more of the features you can find in an Entity Framework model. While this may degrade the overall performance of Graph-Based Querying as more processing needs to be done, it will allow for better compatibility with the models that can be created with the Entity Framework.

4.1.1 Feature Differences

In this section we take a look at the different mapping features supported by the Entity Framework models, Graph-Based Querying as implemented in [Mer11] and Graph-Based Querying as implemented for this thesis. In table 4.1 we compare the different mapping features that are supported by the different GBQ versions and the Entity Framework.

We compare database functionality that can be represented in Entity Framework models and the ability for each to retrieve the data for such a model; we check the support for composite keys, entity splitting, the association and inheritance types, as well as the ability to map fields of the database to differently named entity fields (ie. mapping the database column ‘dbo.MyObjects.MyObjectsId’ to the property ‘MyObject.Id’ instead of ‘MyObjects.MyObjectsId’).
We also check whether each correctly links the loaded objects together using the correct object properties (so called ‘navigation’ properties in the Entity Framework). These properties should hold the correct object(s); the objects that are associated with the current object. This allows the developer easy access to the related objects without the need to somehow link them together manually using keys. More information about mapping can be found on MSDN [Mic14c].
Lastly, we check what filtering options are available to specify what to retrieve from the database.

As mentioned before, we compare the support for different association types. The Entity Framework supports two forms of associations:

1. Foreign Key Exposed; the foreign key is present in the entity and mapped to the entity
2. Independent; the foreign key is not present in the entity and mapped to the association itself hence the only relational information present in the entities is the primary key.

Foreign Key Exposed associations were introduced in Entity Framework 4. It is up to the developer to weigh the pros and cons of the associations to decide which one to use for a given situation.

We also look at the inheritance types. The Entity Framework supports three forms of inheritance:

1. Table per Type; each type has its own table. Subtypes only contain newly introduced fields thus inner joins to base type tables are required to construct the whole object
2. Table per Hierarchy; all types are store in a single table and a special 'discriminator' column is used to get the type
3. Table per Concrete Class; each non-abstract class gets its own table with all fields from its base types included in the table.

As with the association types, it is up to the developer to weigh the pros and cons and decide on the best type to use for a given situation.

Complications of the old implementation

When we look at table 4.1 we can see that the original version of Graph-Based Querying on which this research builds, supports a minimum of features. It does not support entities with composite keys, mapping several objects to the same table (entity splitting) nor the ability to map database fields to entity fields with a different name. The measured performance improvements when compared to standard Entity Framework queries may well be caused by the omission of these features.
For example, the Entity Framework has to look up the database fields that correspond to an object property to construct a proper SQL query, whereas the Graph-Based Querying implementation builds a query from the object properties directly. While this is faster, it can yield invalid results and cause run-time problems; any database field name that does not exactly match the object property name will cause the query to fail.
Furthermore, it relies on foreign key associations, which means that your objects will always include additional relational information; the foreign keys.

New implementation

The new implementation of Graph-Based Querying attempts to support as much of the Entity Framework as possible to allow for the full range of Entity Framework usage scenarios, with the added benefit of Graph-Based Querying. This is so that the developer is not limited in the functionality of the Entity Framework he can use by Graph-Based Querying. The inclusion of most Entity Framework functionality will also show whether or not Graph-Based Querying still performs better than the Entity Framework for the retrieval of complex data structures when it supports most of the models that can be created with the Entity Framework. A possible trade-off introduced here is full-range support vs. performance.
We support the use of composite keys, entity splitting, independent associations and differing field names. As such, this implementation does not suffer from the run-time problems we see with the old implementation.

4.1.2 Supported Relations

In addition to the mapping features we also take a look at the different relational types that may be present in a database and the ability of both Graph-Based Querying versions and the Entity Framework to process these relations. The relations and support for each can be seen in table 4.2.

As stated earlier, the Entity Framework supports two forms of associations. While most of the relation types can be represented in either a Foreign Key Exposed association or an Independent association, some can not. The relations that can not be represented as a Foreign Key Exposed association, do not work with the first version of Graph-Based Querying due to the lack of support for Independent associations. As a result, models with either Optional-to-Optional or Many-to-Many relations would not work.
The new version supports loading of all the relations currently present in the Entity Framework using either Independent associations or Foreign Key Exposed associations.

4.2 Implementation Analysis

In this section we shortly describe the two main parts of our project; the Graph-Based Querying implementation and how we implemented the code to take our measurements.

4.2.1 Graph-Based Querying

This project is the library that developers reference in their project, together with the Entity Framework, to use Graph-Based Querying. It exposes a single class that can be used define graph queries. The graph is defined with calls to the Edge function of this class (see Listing 4.1). Internally it uses the mapping information from the Entity Framework to retrieve the appropriate database tables and fields. When the developer calls Load() it constructs the required database request on its own, bypassing the Entity Framework query generation, and materializes the returned data into objects. These objects are then attached back to the Entity Framework so the Entity Framework can then be used for further operations and change tracking.

Constructing the queries outside of the Entity Framework supplies us with the possibility to create batch requests allowing us to decrease the amount of round-trips to the database, which improves the performance of data retrieval.
We also construct different queries for the retrieval of inheritance structures. The Entity Framework makes use of UNION to include the different sub-types. However, the use of UNION requires a definition for each column present on the different sub-types, setting them to NULL for types that don't actually have that column, or the UNION drops the fields that are not defined on the first select query of the UNION. Afterwards it joins the resulting set with the base class. In Graph-Based Querying we directly use a JOIN on the sub-types instead so we do not have to check NULL values and define empty columns.

4.2.2 Measurements

To perform the measurements to compare Graph-Based Querying and the Entity Framework, we created a performance testing framework project that defines the basis for tests and aggregating the results and a project that references this project and implements the actual tests for the different data sets. We did not use the test setup as created for [Mer11] as this did not use the high precision timer, nor did it time the test process itself, resulting in the inability to exclude the test process itself from the measured time.

A basic test implementation can be seen in Listing 4.2. It is possible to exclude the Setup() and Cleanup() functions in the test implementation but in our case, we respectively initialize and dispose of the DbContext in these two functions.
As we mentioned in 2, to decrease the impact of different processes and core switches on our measurements, we lock our process to a single core and increase its priority. How we do this can be seen in Listing 4.3.
We also mentioned that the Garbage Collector can interrupt the process. As such we disable the collector and run it manually before and after a test. Listing 4.4 shows how we run the garbage collector in our test framework. We call GC.Collect() twice in RunGC() as the collector may bring

Listing 4.1: Edge functions used to define the graph shape

```
public GraphShape<TEntity> Edge<TFrom>(Expression<Func<TFrom, TEntity>> edge)
    where TFrom : TEntity
{
    return Edge<TFrom, Func<TFrom, TEntity>>(edge, edge.Body);
}

public GraphShape<TEntity> Edge<TFrom>(Expression<Func<TFrom, IEnumerable<TEntity>>> edge)
    where TFrom : TEntity
{
    var expression = edge.Body as UnaryExpression;
    if (expression != null)
    {
        if (edge.Body.NodeType != ExpressionType.Convert)
        {
                var msg = String.Format(
                    "Edge expression '{0}' is invalid: the lambda expression has an
                         unsupported format.", edge);
                throw new Exception(msg);
        }
        return Edge<TFrom, Func<TFrom, IEnumerable<TEntity>>>(edge, expression.Operand);
    }
    return Edge<TFrom, Func<TFrom, IEnumerable<TEntity>>>(edge, edge.Body);
}

private GraphShape<TEntity> Edge<TFrom, T>(Expression<T> edge, Expression body)
    where TFrom : TEntity
{
    var mExpr = body as MemberExpression;
    if (mExpr == null || !(mExpr.Expression is ParameterExpression))
    {
        var msg = String.Format("Edge expression '{0}' is invalid: it should have the form
             'A => A.B'", edge);
        throw new Exception(msg);
    }

    var propInfo = mExpr.Member as PropertyInfo;
    if (propInfo != null)
    {
        _edges.Add(new Edge<TFrom>(propInfo));
    }

    return this;
}
```

Listing 4.2: Minimalistic implementation of a test used for our measurements

```
protected override void Setup()
{
    //Initialize the DbContext. The context is the Unit of Work and the Repository of the
        Entity Framework model.
    DbContext = new ModelContext();
}

protected override int DoTest()
{
    //Define the query/statement to execute and measure here (EF and GBQ snippets we
        describe later go here).
    var allMyObjects = DbContext.MyObjects.ToList();
    return allMyObjects.Count;
}

protected override void Cleanup()
{
    //Dispose of the DbContext
    DbContext.Dispose();
    DbContext = null;
}
```

Listing 4.3: Changing the applications' affinity and priority

```
//Prevent process from changing cores
var currentProcess = Process.GetCurrentProcess();
currentProcess.ProcessorAffinity = new IntPtr(Settings.ProgramSettings.Affinity);
currentProcess.PriorityClass = ProcessPriorityClass.High;
//Prevent normal threads from blocking this thread
Thread.CurrentThread.Priority = ThreadPriority.Highest;
```

Listing 4.4: We run the collector before and after the test run

```
private void RunGC()
{
    GC.Collect(GC.MaxGeneration, GCCollectionMode.Forced);
    GC.WaitForPendingFinalizers();
    GC.Collect(GC.MaxGeneration, GCCollectionMode.Forced);
}

public void Run(TimeSpan minimumRunTime)
{
    var oldLatency = GCSettings.LatencyMode;
    GCSettings.LatencyMode = GCLatencyMode.SustainedLowLatency;

    foreach (var test in Tests)
    {
        RunGC();
        var result = test.Run(IgnoreRunResults, minimumRunTime);
        RunGC();
    }

    GCSettings.LatencyMode = oldLatency;
}
```

objects back to life if the object has a finalizer (destructor) that still needed to be run. The second call to collect cleans these objects.

With the optimizations taken care of, our test framework runs and measures the tests using two different timers. One for the CPU time (the time the thread itself is running) and one for the real time that the tests need to execute and receive the results form the database. For the latter it uses the system's high resolution timer if one is available. The implementation of the run function that takes the measurements and stores the results can be seen in Listing 4.5. By decreasing the impact of the process on the measured real time and separately measuring the time the process is running, we can more precisely see the time that is actually spend on the database (real time - cpu/thread time).

It is possible for the framework to ignore the results of a test and as mentioned in chapter 2, we use this to run the tests a few times to allow the Entity Framework to build the metadata cache etc. Any exceptions that occur are always collected and stored.

We use the results of these measurements in our result graphs A.

4.2.3 Model Definitions

The models that we use for the tests are based on the models as seen in [Mer11]. This allows us to do a relative comparison on the performance difference we measure between the Entity Framework and Graph-Based Querying and the relative performance difference as measured in [Mer11]. As we also test on different databases, we instead used the Code-First [Mic14a] approach as opposed to the Model-First [Mic14b] approach. The reason for this is that the EDMX model stores database specific information, preventing a model created for Sql Server to work on, for instance, PostgreSql. Code-First generates this information at run-time, allowing the model to execute on different databases.
The impact on performance for this change is only present for the first execution. As we already run the tests several times without measuring, this does not influence our results.

Listing 4.5: Test run implementation. Shows how the timers are used to measure the time and how the results of a run are calculated

```
internal int Run(bool ignoreResults, TimeSpan minimumRunTime)
{
   TotalRunsCount++; //How many times this test has run

   int result = 0;

   try
   {
      int subSteps = 0;
      var sw = new Stopwatch(); //Real time stopwatch. High precision.
      var esw = new ExecutionStopwatch(); //CPU/Thread time stopwatch. 15ms minimum.
      esw.Start();

      while (esw.Elapsed < minimumRunTime)
      {
         subSteps++;

         Setup();

         sw.Start();
         result ^= DoTest();
         sw.Stop();

         Cleanup();
      }

      esw.Stop();

      if (!ignoreResults)
      {
         var duration = new TimeSpan(sw.Elapsed.Ticks / subSteps);
         var threadDuration = new TimeSpan(esw.Elapsed.Ticks / subSteps);
         _testResults.Add(new TestResults(TotalRunsCount, duration, threadDuration));
      }
   }
   catch (Exception e)
   {
      _failedRuns.Add(new TestFailure(TotalRunsCount, e.Message,
          e.AggregateInnerExceptionMessages()));
      result = -1;
   }

   return result;
}
```

Table 4.1: Comparison of supported features in EF, GBQ 1 and GBQ 2

Application / Feature	**Composite Keys**	**Associations**	**Inheritance**	**Field name change**	**Navigation Properties**	**Filtering Options**	**Entity Splitting**
Entity Framework	Yes	Independent & Foreign Key Exposed	Table per Hierarchy, Table per Type & Table per Concrete Class	Yes	Yes	LINQ	Yes
GBQ 1	No	Foreign Key Exposed	Table-per-Type	No	Yes	Primary Key only	No
GBQ 2	Yes	Independent & Foreign Key Exposed	Table-per-Type	Yes	Yes	'Where' only	Yes

Table 4.2: Comparison of supported relations in EF, GBQ 1 and GBQ 2

Application / Relation	**Many-to-Many***	**Many-to-Many, Link Entity****	**One-to-One**	**One-to-Optional**	**Optional-to-Optional***	**One-to-Many**	**Optional-to-Many**
Entity Framework	Yes	Yes	Yes	Yes	Yes	Yes	Yes
GBQ 1	No	Foreign Key Association only	Foreign Key Association only	Foreign Key Association only	No	Foreign Key Association only	Foreign Key Association only
GBQ 2	Yes	Yes	Yes	Yes	Yes	Yes	Yes

*: Independent Association only. This relation does not support exposing foreign keys.

**: Can only be used in an Independent Association when the link entity has a surrogate primary key.

Chapter 5

Syntax

Even if a solution performs great, if the syntax is hard to use or extremely confusing, the solution may eventually be unusable because it can not be understood or used without a lot of effort.
For this reason we asked the question mentioned in chapter 1:

- Can the Graph-Based Querying syntax simplify code creation and maintainability?

In this chapter we try to answer this question. As mentioned in chapter 1, we use the cognitive dimensions framework to argue about different code snippets written with Graph-Based Querying and the Entity Framework.

5.1 Hypotheses

When we observed the statements we write to retrieve related objects with the Entity Framework we noticed that direct relations and relations on sub-types required different statements to retrieve the data. The main difference we observed was that directly related objects can be retrieved with a single statement.

We also observed that the Entity Framework code used in [Mer11] uses the 'Join' function and the 'Include' function that relies on string based paths. The Entity Framework also supplies other functions to perform the same functionality. For the 'Include', there is a strongly-typed alternative and instead of the 'Join' we can use 'SelectMany'. As we are joining related objects on primary and foreign key and not unrelated objects on some random fields, 'SelectMany' can be used. The resulting SQL is the same for this situation.
In this thesis we use these different functions.

As a result of these observations we created the following hypotheses to answer the question above:

1. Entity Framework code as used in [Mer11] is harder to understand than Entity Framework code as used in this thesis.

2. Entity Framework code as used in [Mer11] is more error-prone than Entity Framework code as used in this thesis.

3. Graph-Based Querying code is not easier nor harder to understand and maintain than Entity Framework code when retrieving directly related objects.

4. Graph-Based Querying code is easier to understand and maintain Entity Framework code when inheritance is part of the objects to be retrieved.

With this analysis we attempt to verify these hypotheses.

Entity Framework code as used in [Mer11] is harder to understand than Entity Framework code as used in this thesis

The 'Join' statement that is used in [Mer11] represents the relational JOIN and as such, requires knowledge of the relational model. The developer is required to define the set that needs to be joined as well as the keys to join on (as we are joining related objects, this always is the primary key on one end and the foreign key on the other). We expect that the 'SelectMany' statement we use makes it easier for the developer to understand the code as the information required to construct the join is inferred by this statement.

Entity Framework code as used in [Mer11] is more error-prone than Entity Framework code as used in this thesis

The 'Include' statement that is used in [Mer11] relies on string based paths. These are not checked at compile time and mistakes are only discovered at run-time. As such we expect that the strongly typed 'Include' statement we use decreases the chances of making a mistake.

Graph-Based Querying code is not harder to understand and maintain than Entity Framework code when retrieving directly related objects

As directly related objects can be included in a single statement in the Entity Framework we expect that Graph-Based Querying will not provide much in terms of improvements to nor any decline in the readability and maintainability of the code.

Graph-Based Querying code is easier to understand and maintain Entity Framework code when inheritance is part of the objects to be retrieved

Using the 'Join' statement requires knowledge of the relational model and requires the developer to know the whole join path to retrieve data from relations on subtypes. The 'SelectMany' statement does not require this knowledge but still needs extra information to properly retrieve sub-type relations; the developer has to use the 'OfType' statement to select the proper sub-type. In Graph-Based Querying the developer only needs to define the start and end of a relation on object level. To do this the developer does not need to know the relational model and can define all relations at once. We expect that this makes the code written with GBQ easier to understand and maintain.

5.2 Test Setup

In this section we describe the different Cognitive Dimensions and provide a short description as well as the snippets we use in our comparison.

5.2.1 Cognitive Dimensions

To be able to say something about the code written with the Entity Framework and Graph-Based Querying, we utilize the Cognitive Dimensions Framework [BBC+01]. From this framework we select a few dimensions and argue about how those apply to certain code snippets. The framework currently consists of 14 dimensions but is gradually expanding.
We use the following subset:

- *Viscosity: Resistance to Change* Many user actions are required to accomplish one goal. This can be repetition of the same action or an action that requires follow-up actions.
- *Visibility: Ability to View Components Easily* The system reduces visibility by hiding information using encapsulation.
- *Hidden Dependencies: Important Links between Entities Are Not Visible* If one entity sites another, which sites a third, changing the value of the third entity may have unexpected results; important links are not visible.
- *Role-Expressiveness: The Purpose of an Entity Is Readily Inferred* The notation makes it easy to discover why the programmer built the structure in a particular way.

- *Error-Proneness: The Notation Invites Mistakes and the System Gives Little Protection* Certain notations invite errors. Preventing those errors can solve this problem.
- *Abstraction: Types and Availability of Abstraction Mechanisms* Systems that allow many abstractions are potentially difficult to learn.
- *Closeness of Mapping: Closeness of Representation to Domain* How close the notation relates to the entities it describes. The Entity Framework is an ORM framework that maps Object-Oriented language to the relational model of a database. As such, we look at how close it is able to map to the Object-Oriented domain.
- *Consistency: Similar Semantics Are Expressed in Similar Syntactic Forms* Similar information is not obscured by different representations to prevent compromising usability.
- *Diffuseness: Verbosity of Language* A notation can be to long-winded or occupy a large piece of working space.
- *Hard Mental Operations: High Demand on Cognitive Resources* A notation can make things complex or difficult to work out in your head.
- *Provisionality: Degree of Commitment to Actions or Marks* The degree of commitment to actions or marks. Is there a hard constraint on the order of doing things or not?
- *Progressive Evaluation: Work-to-Date Can Be Checked at Any Time* The user can stop in the middle to check work so far, find out how much progress has been made or what stage the work is in.

We excluded the following dimensions:

- *Premature Commitment: Constraints on the Order of Doing Things* Ie. being forced to declare identifiers too soon.
  We do not include this dimension as both the Entity Framework and Graph-Based Querying require the existence of a database and a model that maps to the database to function. The commitment to this model needs to be made in both cases so there is no difference between the two.
- *Secondary Notation: Extra Information in Means Other Than Formal Syntax* Support for secondary (non-formal) notations that can be used however the user likes (ie. comments).
  As both the Entity Framework and GBQ use C#, the developer can use the same type of secondary (non-formal) notation (ie. comments, indentation, etc.). There is no difference between the two for this dimension thus this dimension is not included. The Entity Framework does support different (formal) notations but these are different abstractions of the same functionality and are covered by the dimension 'Abstraction: Types and Availability of Abstraction Mechanisms'.

5.2.2 Code Snippet Selection

As we mentioned with the hypotheses, we observed that directly related objects and relations on sub-types require different statements to retrieve the data. For this reason we select a snippet that retrieves directly related objects and a snippet that retrieves relations on sub-types.
For each we include a snippet for the Entity Framework and Graph-Based Querying. As our Entity Framework code differs from the Entity Framework code used in [Mer11] (in notation only! Functionality is the same) we also include those. As such we end up with 3 snippets that we compare against each other; the Entity Framework code as in [Mer11], Entity Framework code as used in this thesis and Graph-Based Querying code.
We also include snippets for data retrieval on the Northwind database to demonstrate the usage on a more realistic dataset as well.

Direct association

The following snippets retrieve an object ‘O’ and its related ‘E00’ objects.
These snippets are taken from the performance tests that we used to compare the Entity Framework and Graph-Based Querying.

Entity Framework as in [Mer11]

```
DbContext
    .OSet
    .Include("E00s")
    .Where(o => o.Id == SomeIndex)
    .ToList();
```

Entity Framework

```
DbContext
    .OSet
    .Where(o => o.Id == SomeIndex)
    .Include(o => o.E00s)
    .ToList();
```

Graph-Based Querying

```
new SqlGraphShape(DbContext)
    .Edge<O>(o => o.E00s)
    .Load<O>(o => o.Id == SomeIndex);
```

Associations on subtypes

The following snippets retrieve an ‘O’ object and its related ‘E00’ objects with its related ‘A00’ objects. If the ‘A00’ object is the ‘A10’ sub-type, it retrieves its related ‘B00’ objects. If the ‘A00’ object is the ‘A11’ sub-type, it retrieves its related ‘C00’ objects. If the ‘A00’ object is the ‘A12’ sub-type, it retrieves its related ‘D00’ objects.
These snippets are taken from the performance tests with the most associations on sub-types.

Entity Framework as in [Mer11]

```
DbContext
    .OSet
    .Include("E00s.A00s")
    .Where(x => x.Id == SomeIndex)
    .ToList();

DbContext
    .OSet
    .Where(o => o.Id == SomeIndex)
    .Join(DbContext.E00Set, o => o.Id, e00 => e00.O.Id, (o, e00) => e00)
    .Join(DbContext.A00Set.OfType<A10>(), e00 => e00.Id, a10 => a10.E00.Id, (e00, a10) =>
        a10)
    .Join(DbContext.B00Set, a10 => a10.Id, b00 => b00.A10.Id, (a10, b00) => b00)
    .ToList();

DbContext
    .OSet
    .Where(o => o.Id == SomeIndex)
    .Join(DbContext.E00Set, o => o.Id, e00 => e00.O.Id, (o, e00) => e00)
    .Join(DbContext.A00Set.OfType<A11>(), e00 => e00.Id, a11 => a11.E00.Id, (e00, a11) =>
        a11)
    .Join(DbContext.C00Set, a11 => a11.Id, c00 => c00.A11.Id, (a11, c00) => c00)
    .ToList();

DbContext
    .OSet
    .Where(o => o.Id == SomeIndex)
    .Join(DbContext.E00Set, o => o.Id, e00 => e00.O.Id, (o, e00) => e00)
    .Join(DbContext.A00Set.OfType<A12>(), e00 => e00.Id, a12 => a12.E00.Id, (e00, a12) =>
        a12)
    .Join(DbContext.D00Set, a12 => a12.Id, d00 => d00.A12.Id, (a12, d00) => d00)
    .ToList();
```

Entity Framework

```
DbContext
    .OSet
    .Where(o => o.Id == SomeIndex)
    .Include(o => o.E00s.Select(e => e.A00s))
    .ToList();

DbContext
    .OSet
    .Where(o => o.Id == SomeIndex)
    .SelectMany(o => o.E00s)
    .SelectMany(e => e.A00s).OfType<Inh6_Assoc3_A10>()
    .SelectMany(a => a.B00s)
    .ToList();

DbContext
    .OSet
    .Where(o => o.Id == SomeIndex)
    .SelectMany(o => o.E00s)
    .SelectMany(e => e.A00s).OfType<Inh6_Assoc3_A11>()
    .SelectMany(a => a.C00s)
    .ToList();

DbContext
    .OSet
    .Where(o => o.Id == SomeIndex)
    .SelectMany(o => o.E00s)
    .SelectMany(e => e.A00s).OfType<Inh6_Assoc3_A12>()
    .SelectMany(a => a.D00s)
    .ToList();
```

Graph-Based Querying

```
new SqlGraphShape(DbContext)
```

```
    .Edge<O>(o => o.E00s)
    .Edge<E00>(e => e.A00s)
    .Edge<A10>(a => a.B00s)
    .Edge<A11>(a => a.C00s)
    .Edge<A12>(a => a.D00s)
    .Load<O>(o => o.Id == SomeIndex);
```

Northwind (direct associations)

The following snippets retrieve a ‘Customer’ object and its related ‘Demographics’ and ‘Orders’ objects. From the ‘Orders’ also retrieve the ‘Order_Details’ and ‘Shippers’ objects. Lastly, we also retrieve the related ‘Products’ objects for each ‘Order_Details’ object.

Entity Framework as in [Mer11]

```
Context
    .Customers
    .Include("CustomerDemographics")
    .Include("Orders.Order_Details.Products")
    .Include("Orders.Shippers")
    .Where(c => c.CustomerID == "FRANK")
    .ToList();
```

Entity Framework

```
Context
    .Customers
    .Where(c => c.CustomerID == "FRANK")
    .Include(c => c.CustomerDemographics)
    .Include(c => c.Orders.Select(o => o.Order_Details.Select(od => od.Products)))
    .Include(c => c.Orders.Select(o => o.Shippers))
    .ToList();
```

Graph-Based Querying

```
new SqlGraphShape(Context)
    .Edge<Customers>(c => c.Orders)
    .Edge<Customers>(c => c.CustomerDemographics)
    .Edge<Orders>(o => o.Order_Details)
    .Edge<Orders>(o => o.Shippers)
    .Edge<Order_Details>(od => od.Products);
    .Load<Customers>(c => c.CustomerID == "FRANK");
```

5.3 Analysis

In this section we analyse the snippets we selected previously using the cognitive dimensions framework.

5.3.1 Direct association

In this sub-section we analyse the snippets selected in section 5.2.2 using the selected cognitive dimensions.

Viscosity: Resistance to Change The 'Include' statement of the Entity Framework used in [Mer11] requires a string as the parameter to define which directly related entity to retrieve. When we refactor the related 'E00' entity name, this string will not be updated by the refactoring tools and instead requires manual editing by the developer. The amount of actions required increases for each use of 'Include'.
The 'Include' statement in the second Entity Framework snippet does not rely on string based paths and instead is strongly typed. It uses lambda expressions that use the fields as declared in the class. As such, the include paths defined here are changed during refactoring and do not require manual labour.
In Graph-Based Querying, related entities are defined by edges using lambda expressions. These expressions use the fields as declared in the class and when this field is refactored, the lambda expressions are refactored automatically as well. Refactoring would no longer require manual actions afterwards.
We can conclude that the Entity Framework snippet as used in [Mer11] is not easily changed and that the second Entity Framework snippet as well as the Graph-Based Querying snippet provide benefits. We can also see that the second Entity Framework snippet and Graph-Based Querying both use strongly typed code to define the relations and that neither has a clear advantage over the other for this dimension.

Visibility: Ability to View Components Easily For directly associated entities, the code required for either, shows which entities are involved and the visibility for all snippets is about equal. The only benefit the second Entity Framework snippet and Graph-Based Querying have is that the appropriate field can easily be navigated to within the IDE using 'Go To Definition' which helps with the ability to view components further down in the system.
As such, the second Entity Framework snippet and the Graph-Based Querying snippet improve the ability to view components when compared to the first Entity Framework snippet but neither provides a benefit over the other for this dimension.

Hidden Dependencies: Important Links between Entities Are Not Visible In the first Entity Framework snippet you can see an include to 'E00s'. You can however not directly see what this references to; you need to know the type of 'OSet'. Furthermore, if the property was changed, the code would only crash during run-time. You can not see the dependecy between the include and the property.
The second Entity Framework snippet already provides some improvement as the includes are strongly typed. You still can not see directly what the type is of the object with property 'E00s' but because it is strongly typed, you are able to see the dependency between the include and the property. If the property changes, the code can not compile.
In the Graph-Based Querying snippet you can see that in the edge declaration, a property 'E00s' on objects of type 'O' is meant. Both the type and the property can be seen at a glance. As with the second Entity Framework snippet, if the property was changed, the Graph-Based Querying snippet would fail to compile thus showing you a dependency between the edge declaration and the property.
We conclude that the first Entity Framework hides important dependencies that can result in errors during program execution and that the second Entity Framework snippet as well as the Graph-Based Querying snippet are improvements by exposing this dependency.

Role-Expressiveness: The Purpose of an Entity Is Readily Inferred In all snippets we can easily see what the purpose is of the statements. In both Entity Framework snippets we can see that we attempt to retrieve something from the database from 'OSet' and that we must include the 'E00s' and only retrieve when the 'Id' of the objects in OSet is equal to a certain value.
In the Graph-Based Querying snippet we can see that we are defining a graph with an edge from type

‘O’ on property ‘E00s’ and that we only want to load when the ‘Id’ of the ‘O’ objects is equal to a certain value.
In all cases we can not readily infer the type of the ‘E00s’ include but with the second Entity Framework snippet and the Graph-Based Querying snippet we could easily use the IDE tools to find out. We can not do this with the string used in the first Entity Framework snippet.

Error-Proneness: The Notation Invites Mistakes and the System Gives Little Protection The ‘Include’ statement from the first Entity Framework snippet is sensitive to errors because it utilizes a string value to define the property path to load. This value has to be typed in by the developer and is prone to spelling errors or other naming mistakes. Furthermore, the developer does not know that something is wrong until the program is running; only when the program is running and the statement is executed the Entity Framework will figure out the string is invalid and throw and exception.
Because the second Entity Framework and the Graph-Based Querying snippet use lambda expressions, non-existent fields can never be entered, auto-completion can help the developer filling in the field name and syntax errors are detected at compile-time as opposed to runtime.
As such we conclude that the second Entity Framework snippet and the Graph-Based Querying snippet are improvements but that neither improves over the other for this dimension.

Abstraction: Types and Availability of Abstraction Mechanisms The Entity Framework supports different methods you can use to load data. EntitySql, which is string based and therefore not type-safe and Linq-to-Sql as used in the above snippet. The latter supports a different notation as well. This gives the developer three different forms to retrieve data with. Without knowledge of any, the developer needs to investigate three possible ways to write code. In addition within a single notational form there are different statements that can perform the same functionality (as seen in the two Entity Framework snippets). This further increases the difficulty of learning the system.
With Graph-Based Querying there is only a single syntax in which you can write code to retrieve data thus allowing the developer to skip the step in which he has to decide what syntax to use.
We conclude that while Graph-Based Querying provides less abstractions to retrieve data, it is easier to understand because the developer does not need to invest time and effort in figuring out the different abstractions.

Closeness of Mapping: Closeness of Representation to Domain When comparing the Entity Framework code snippets against that of Graph-Based Querying, we can see that the snippets still represent the relational model. You can see the Context as the database itself, OSet as the table, Include as an INNER JOIN and the Where as a WHERE.
Instead of representing a relational model, Graph-Based Querying represents a graph and is therefore not closely mapped to the Object-Oriented model either.
Is this a problem? Part of an ORM is the separation of the relational world and the object-oriented world. Graph-Based Querying makes the separation of the relational model more prominent but in the process represents a graph. However, we think this should make it easier for developers without knowledge of the relational model to write code to retrieve data.

Consistency: Similar Semantics Are Expressed in Similar Syntactic Forms For these snippets the semantics for both Entity Framework snippets and the Graph-Based Querying snippet do not change when you want to retrieve another directly related item. For the Entity Framework you add another Include, for Graph-Based Querying you add another Edge.
Things change for the Entity Framework when you want to retrieve another directly related entity when that entity is instead related to E00. In Graph-Based Querying the code stays the same; you just add an Edge for E00. For the Entity Framework, you do not add another include. In the first snippet you would need to change the E00 include to ‘Include(“E00.Related”)’. For the second snippet you would need to append ‘.Select(e => e.Related)’ to the include of E00.
As such we conclude that Graph-Based Querying makes it easier to write code as the semantics do not change depending on the situation.

Diffuseness: Verbosity of Language The first Entity Framework snippet takes up the least amount of space (in character count), followed by the second Entity Framework snippet (+5 characters), followed by the Graph-Based Querying snippet (+7 more characters).
The second Entity Framework snippet and the Graph-Based Querying snippet are more diffuse but in our opinion none of the snippets are more or less verbose than the other. Each snippet contains the least amount of code necessary to function. We conclude that if as little code as possible is a must, Graph-Based Querying would not be an option. However, we also conclude that this small of a difference most likely will not impact the time needed to read the code.

Hard Mental Operations: High Demand on Cognitive Resources Neither of these snippets require the developer to thing about nested structures, only about directly related entities. Graph-Based Querying has no benefit over the Entity Framework snippets when we look at how difficult it is to work out the required code. We conclude that for directly related objects, this dimension does not change between the snippets.

Provisionality: Degree of Commitment to Actions or Marks All snippets are written as a single statement and all require the developer to define what to include before loading the data. The only difference is that the Entity Framework can define filters separately from loading, whereas GBQ takes a filter in the load statement. This means that the developer is unable to set up a filter before loading the graph. It can not be changed or expanded later on. This makes Graph-Based Querying a little less flexible in the order in which you can do things.
Because Graph-Based Querying requires you to have a filter at the moment you load, we conclude that Graph-Based Querying requires more commitment.

Progressive Evaluation: Work-to-Date Can Be Checked at Any Time In the Entity Framework snippets the developer is able to check the result of each part of the statement. The set itself, the set with include, and the set with include and filter. In Graph-Based Querying the developer needs to define at least one edge before he can test the result and this edge needs to be connected to the type requested when calling 'Load'.
We conclude that this can make it harder to check your work with Graph-Based Querying. While this does not directly impact the ease in which you understand the code itself, it can impact the ease in which you test each step to detect any possible problems.

5.3.2 Associations on subtypes

In this sub-section we analyse the snippets selected in section 5.2.2 using the selected cognitive dimensions.

Viscosity: Resistance to Change The problems as described in the previous section still apply for the snippets shown here. In addition, if we want to adapt the Where expression for the Entity Framework, we would have to do so in four places in those snippets. Graph-Based Querying requires only one change.
We conclude that because the filter is contained in one spot, Graph-Based Querying is easier to maintain. The developer doe not need to check for duplications in other locations.

Visibility: Ability to View Components Easily In the first Entity Framework snippet we have to analyse each Join sequence to be able to tell what is or is not included in the result. The Join sequences do tell us more about how the objects are included; what key we join on and what entity set the objects are pulled from. However, it is not easy to see what base set the next join uses unless you are familiar with the function.
With the second Entity Framework it is easier to see what is included from where and what the execution chain is. Ie. we can see that from O we select the E00s, of which we select the A00s, of which we only take the ones of type A10, of which we select the B00s.
With Graph-Based Querying we can immediately spot which object have and edge in our graph and

thus which objects are loaded. We can not see what the complete path would be as we can with the second Entity Framework snippet. For that we need to look at all the edge definitions and mentally make the connections between them. In the design for Graph-Based Querying we regard the extra information we can see in the Joins as irrelevant as a join on related objects is always the same; foreign key on one side, primary key on the other.
While Graph-Based Querying sacrifices some openness and customizability, we conclude that this makes it more clear and easier to use by hiding information that is not important in the Object-Oriented world (such as primary- and foreign keys and datasets).

Hidden Dependencies: Important Links between Entities Are Not Visible In addition to what was mentioned before, Graph-Based Querying internally creates paths for the different edges to join all the types together. While the developer can infer this information from the edges he defines, this information is not exposed. The Entity Framework does not hide this information. On the contrary, in the Entity Framework snippets the developer is required to define how the types link together. While this does make the dependencies visible, it also implies that the developer needs to define code that can be inferred and would not necessarily require developer input.
One could argue that the inheritance dependency is an important link between types but this link can be seen in the class hierarchy, hence we regard this not important in this context.
We conclude that Graph-Based Querying hides links between entities but that these links are not important to the developer in this context. As such we conclude that the Graph-Based Querying code is easier to maintain as the developer does not need to concern himself with information that can be inferred.

Role-Expressiveness: The Purpose of an Entity Is Readily Inferred The join statements in the first Entity Framework snippet do not instantly convey how and what they are joining. The developer needs properly look at the parameters to see what properties are used for the join and which object is used to continue with after the join.
In the second snippet it is easy to see the purpose of the select statements and it is easy to see that the next continues with the objects selected in the preceding select.
In the Graph-Based Querying snippet the developer can instantly see what types are included and what properties are used to retrieve related objects.
In none of the snippets we can infer what the types the properties are. But because Graph-Based Querying clearly shows the types used for the 'from' end of an edge and what properties the relations are on, we argue that Graph-Based Querying code makes it easier to discover which types certain properties are on and makes it easier to understand.

Error-Proneness: The Notation Invites Mistakes and the System Gives Little Protection With the first Entity Framework snippet the developer has to define on which keys to join and has to make sure these are valid. Especially when the underlying data model changes and key relations change, the developer may have to correct the join statements. Failure to do so may result in unexpected behaviour later on.
The second Entity Framework snippet creates the required join connections using the appropriate fields thus preventing the developer from joining on the wrong fields. However, to load all the relations on sub-types it still requires multiple statements in which the filter is duplicated for each statement. Failing to change the filter at all used locations can result in unexpected behaviour.
Graph-Based Querying automatically creates the required join connections and can retrieve all the relations on sub-types within a single statement thus no code duplication is present.
We conclude that while the second Entity Framework snippet is already an improvement, Graph-Based Querying further decreases the chances of errors in the code, making it easier to maintain.

Abstraction: Types and Availability of Abstraction Mechanisms These snippets do not affect this dimension in a different way. The Entity Framework still has three different ways in which a developer can write code and Graph-Based Querying still maintains the single syntax.

Closeness of Mapping: Closeness of Representation to Domain As stated in the previous section, the Entity Framework still represents the relational model. In the snippets for sub-type associations this is even more visible as each of the four statements can be seen as a query to the database.
While an Edge in Graph-Based Querying is eventually some join at database level, on code level it represents a connection between node A and node B as seen in graphs. We argue that hiding the relational model from the Object-Oriented world makes it easier to use and understand.
While we use a graph representation in return, relations between objects in the Object-Oriented world can already be looked at as a graph and as such we conclude that this more closely represents the way in which a developer looks at object relations in an Object-Oriented world than the relational model.

Consistency: Similar Semantics Are Expressed in Similar Syntactic Forms In the first Entity Framework snippet we use two functions to join and load related data; 'Include' and 'Join'. In the second snippet these are the 'Include' and 'SelectMany'. This means that for the retrieval of directly related objects we use a different construct than we do for the retrieval of relations on sub-types. To accomplish similar things we are required to do different things.
In Graph-Based Querying each of these relations is defined as an Edge and it does not matter if this is on a base-type or on a subtype. You define the from type and define what field the edge is on.
We conclude that this makes it easier to read and to maintain the code written with Graph-Based Querying. The developer can do the similar things in the same way.

Diffuseness: Verbosity of Language The joins in the first Entity Framework snippet require a lot of information to function. While the second snippet already improves on this, it still requires complete paths for the retrieval of relations on sub-types.
With Graph-Based Querying we infer as much information as possible thus allowing for a shorter and more concise notation. While we do sacrifice customizability in terms of what you can join on, when retrieving related objects this is always the foreign key on one side and the primary key on the other so we see this as a non-issue.
As such, we conclude that the shorter and more concise notation we have with Graph-Based Querying makes it easier to read and write code.

Hard Mental Operations: High Demand on Cognitive Resources The first Entity Framework snippet uses joins and these require knowledge of the underlying relational model to create the proper join sequence. To be able to do so the developer needs to ask himself several questions such as; what is my primary key?, what is the foreign key?, are my foreign keys exposed to the object or do I need to access the key through a relation?).
The second snippet already decrease the amount of mental operations by eliminating the need to ask those questions. The 'SelectMany' function that is used in this snippet infers the information it needs to construct a proper join. To retrieve relations on sub-types the developer still needs to keep the whole path in mind to construct the correct statement.
As with the second Entity Framework snippet, Graph-Based Querying requires no knowledge of the underlying relational model and therefore the developer does not need to worry about any questions that relate to information from the relational model. Graph-Based Querying further decreases the mental load as the developer defines edges on a in- to out-type basis; the path in between is inferred.
We conclude that Graph-Based Querying requires less hard mental operations by inferring as much as possible, making it easy to read and write complex graph queries.

Provisionality: Degree of Commitment to Actions or Marks The Entity Framework statements can be executed in an arbitrary order but the statement itself has to form the complete path for the objects. The developer can not change the order of the functions in the statements.
In Graph-Based Querying there is still only a single statement for which the same applies as mentioned in the previous section. As opposed to the Entity Framework snippets, with Graph-Based Querying it does not matter in which order you declare the edges. As long as all the nodes are somehow connected to the type requested in 'Load', they will be loaded. This makes it easy to expand or change

the defined graphs at a later moment. The developer can add, remove and change edge declarations without invalidating the next edge declarations.
We conclude that this dynamic structure makes it easy to maintain the code as the developer is not constrained to a specific order.

Progressive Evaluation: Work-to-Date Can Be Checked at Any Time As described earlier, in the Entity Framework snippet the developer is able to check the result of each part of the statement. In addition, the Entity Framework snippets use several statements to retrieve all the data. Each of these can also be tested in parts.
For Graph-Based Querying the same still applies and as such the conclusion for this dimension does not differ.

5.3.3 Northwind (direct associations)

All entities in the Northwind dataset are directly related. There is no inheritance present in this dataset and therefore all the observations and conclusions mentioned in section 5.3.1 apply here as well. The snippets we showed in section 5.2.2 are only included to demonstrate the usage of Graph-Based Querying and the Entity Framework on a more realistic dataset as opposed to the test datasets.

5.4 Conclusion

From our analysis we can conclude that the Entity Framework code as used in [Mer11] has its problems. When compared to the Entity Framework code as we wrote it we conclude that it is easier to make mistakes in this code and that the code requires more effort to understand. As such, we conclude that hypotheses 1 and 2 can be regarded as valid.

From our analysis we can also conclude that the Entity Framework snippet we use and the Graph-Based Querying snippet do not differ to much from each other when querying directly related objects. Only when a developer attempts to learn to write code in either the Entity Framework or Graph-Based Querying we concluded that Graph-Based Querying makes it easier as it only supports a single notational form. As such, we regard hypothesis 3 as valid.
When we look at the analysis for relations on sub-types, there are a few things in which Graph-Based Querying is lacking when compared to the Entity Framework (such as progressive evaluation). In our opinion these are of lesser importance than the benefits Graph-Based Querying brings and that these do not negatively impact the readability and the ease in which the developer understands the Graph-Based Querying code. We therefore conclude that hypothesis 4 can be regarded as valid as well.

5.5 Threats to Validity

While the cognitive dimensions framework provides a way to compare different notations and information artefacts, it is possible that in real world usage opinions differ. As seen in [KBB02], the empirical study does not conform to the cognitive dimensions analysis.
In our case it is possible that a developers' familiarity with the Entity Framework may influence the ease in which he analyses Entity Framework code. This could make it easier for him to understand Entity Framework code as opposed to Graph-Based Querying code which he may never have seen before.

Chapter 6

Performance

In this chapter we try to answer the question:

- Can the Graph-Based Querying syntax simplify code creation and maintainability?

We set up several tests to investigate what the difference in performance is between Graph-Based Querying and the Entity Framework.

6.1 Hypotheses

We observed that directly related objects can be retrieved with the Entity Framework in a single statement. We also observed that with relations on sub-types, a new statement is required to retrieve those object. As each sub-type relation requires a new statement, the resulting code will require n-statements for n-(sub-type) relations.

As mentioned in chapter 4, we attempted to support more of the variants of models that can be created with the Entity Framework. As such, more processing is performed in this version when compared to the version created in [Mer11]. We expect that this impacts the overall performance of Graph-Based Querying.

We also observed that in [Mer11], the tests specifically retrieve the concrete types and that this outperforms selecting the set itself. We expect that with the developments of the Entity Framework, this is no longer the case.
Based on these observations we created the following hypotheses:

1. Graph-Based Querying queries perform as good as the standard Entity Framework queries when retrieving directly related objects.

2. Graph-Based Querying queries perform better than the Entity Framework when inheritance is part of the object structure to be retrieved.

3. The performance gap between Entity Framework and Graph-Based Querying is less than measured in [Mer11].

4. Specifically selecting the concrete types from a set for retrieval in the Entity Framework performs worse than selecting the set itself.

With the tests as defined in section 6.2 we attempt to verify these hypotheses.

Graph-Based Querying queries perform as good as the standard Entity Framework queries when retrieving directly related objects
Directly related objects can be retrieved with the Entity Framework with a single statement using the 'Include' function. For this reason we expect that Graph-Based Querying will be able to execute

the statement as fast as the Entity Framework. To test this hypothesis we created datasets that do not have relations on sub-types.

Graph-Based Querying queries perform better than the Entity Framework when inheritance is part of the object structure to be retrieved
If related objects are not defined on the base type but on sub-types, the Entity Framework 'Include' statement can not be used and the developer needs to write 'SelectMany' statements instead (as seen in the snippets in 5). In addition, for every subtype relation a new sequence of those statements is needed. In Graph-Based Querying the developer defines all the relations to load and the shape is transformed as a whole instead of separately for each sub-type relation. We expect this to result in an increase in performance over the Entity Framework as it decreases the amount of round-trips and allows for the database to perform optimizations on the whole query structure.

The performance gap between Entity Framework and Graph-Based Querying is less than measured in [Mer11]
This implementation of Graph-Based Querying attempts to support most of the functionality present in the Entity Framework. Due to this, we expect that the relative performance increase of Graph-Based Querying as seen in [Mer11] is decreased. This hypothesis indicates the trade-off between support for all database models and worse performance or higher performance and limited support for different models.

Specifically selecting the concrete types from a set for retrieval in the Entity Framework performs worse than selecting the set itself
In [Mer11], specifically selecting the concrete types in a set (using 'OfType') resulted in faster data retrieval than just selecting the set itself. While doing this improves performance, it also results in an 'OfType' statement for each concrete type and the query would no longer be valid when new concrete types are added. Improvements to the Entity Framework have made it so that this should no longer the case and that selecting the set itself results in better performance.

6.2 Test Setup

In this section we describe the datasets, system and timers used to measure the performance of Graph-Based Querying and the Entity Framework.

6.2.1 Test Data

Datasets

The data models used to test the performance of Graph-Based Querying is the same as used in [Mer11] to replicate the findings of the article.
We also add the Northwind [Mic11] dataset to include a more realistic dataset as opposed to the datasets used in [Mer11] to demonstrate the code with a real-world based setting. However, due to the lack of inheritance relations in this dataset, we expect that Graph-Based Querying does not perform at its full potential.

The datasets consist of an 'O' type with a One-to-Many relation to several 'E' types. These 'E' types form an inheritance tree where all basetypes are abstract and have 2 subtypes. Only the leaf 'E' types can be instantiated 6.1. The tests with this structure are called Inh3, Inh4, Inh5 and Inh6. These tests focus on hypothesis 1.

In addition to the above, we have tests where the 'E00' type has a relation with the 'A00' type 6.2. This 'A00' type is abstract and has 3 concrete subtypes. each of these subtypes can have an additional relation 6.3, 6.4, 6.5. These tests are named 'InhX, Assoc 1' , 'InhX, Assoc 2' and 'InhX,

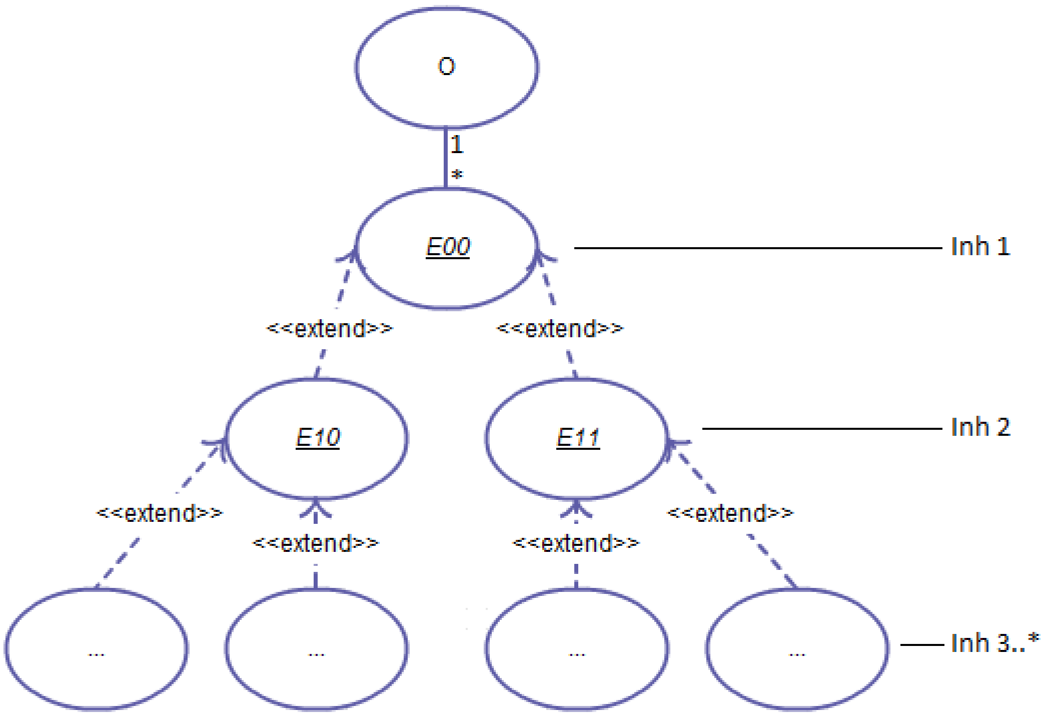


Figure 6.1: O to E association and E inheritance structure

Assoc 3' respectively. These tests focus on hypothesis 2.

The measured results for these datasets for Graph-Based Querying and the Entity Framework are compared with the results in [Mer11] and the relative difference between the two is used to verify hypothesis 3.

Population

The populations used for these datasets is the same as used in [Mer11] as well. Table 6.1 describes the populations that were used for the datasets. The relations between the type populations is described in 6.2.1.

- Each O has 1 E, where each next E is one of the different concrete subtypes.

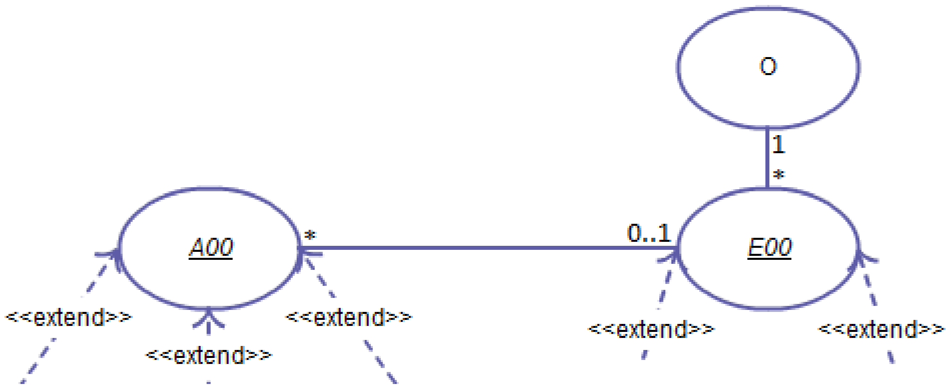


Figure 6.2: E00 to A00 association

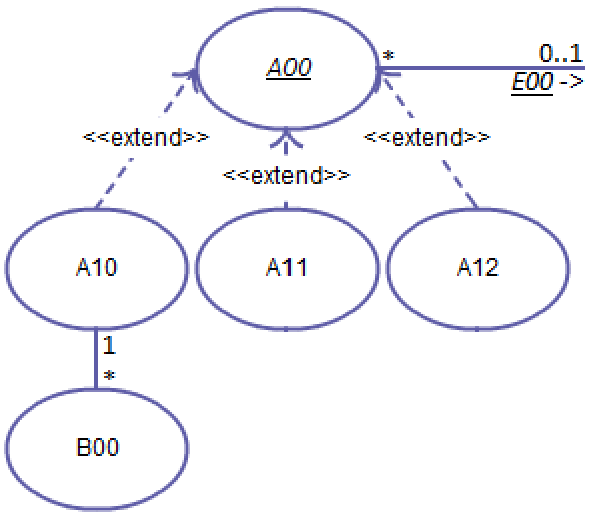


Figure 6.3: A00 structure for Assoc1 tests

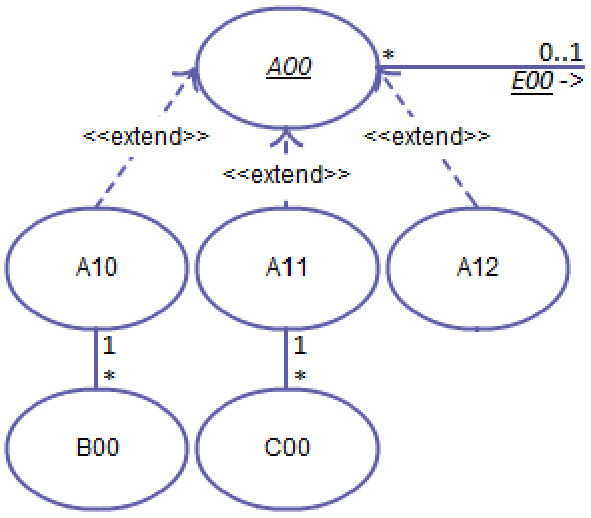


Figure 6.4: A00 structure for Assoc2 tests

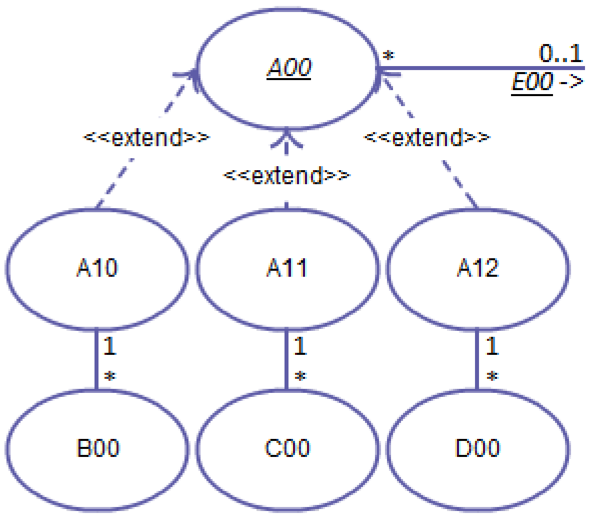


Figure 6.5: A00 structure for Assoc3 tests

Table 6.1: Populations

Population	**O**	**E**	**A**	**B**	**C**	**D**
1	100	100	75	5	5	5
2	200	200	75	5	5	5
3	300	300	75	5	5	5
4	400	400	75	5	5	5
5	500	500	75	5	5	5
6	600	600	75	5	5	5
7	700	700	75	5	5	5
8	800	800	75	5	5	5
9	900	900	75	5	5	5
10	1000	1000	75	5	5	5

- Each E has x As, where each next A is one of the 3 concrete subtypes.
- Each A10 (A, subtype 1) has x Bs.
- Each A11 (A, subtype 2) has x Cs.
- Each A12 (A, subtype 3) has x Ds.

6.2.2 System Specifications

All tests have been executed on the same system so measurements are not influenced by the specifications of different systems. The system specifications are as follows:

- nVidia GeForce GTX670
- Intel i7 3770K @3.5GHz
- 16GB DDR3 RAM
- Crucial M4 SSD
- Windows 8.1 Update 1

To prevent other processes from interrupting the tests and thus increasing the measured time by including time spend executing other processes, the process is run in high priority. Furthermore, it is locked to a specific core to prevent slow-downs caused by core switching and CPU buffer resets.
Garbage Collection is set to SustainedLowLatency to prevent blocking garbage collections. Instead, garbage collection is run before and after each test to prevent the garbage collector from occasionally increasing the measured time and skewing the results.

6.2.3 Timer Specifications

As mentioned in 2, we measure the time it takes from executing the retrieval statement until the database returns the data and the objects are returned. To differentiate between time spend executing code and waiting for the database, the tests have been measured with 2 different timers.

The first timer uses the high resolution timer of the system to measure the real time taken to execute the code, run the queries and retrieve the results. It includes the time the program is waiting for the database to return the results. The high resolution timer is accurate within 292 nanoseconds.

A second timer is used to measure the time the CPU spends running the program. This timer does not measure the time it takes for the queries to be executed, nor the time for the database to return the data. In other words, it only measures the time the program/thread is actually running and not waiting for an external process or thread. This timer is accurate within 15 milliseconds.

6.2.4 Databases

The Entity Framework requires a data provider to be able to communicate with the database. These data providers are responsible for connecting and interacting with the database. While there are more generic data providers for ODBC and OLE DB, specialized providers are assumed to perform better and allow for optimizations for their specific database. For this reason, we test Graph-Based Querying on several different databases for which there are Entity Framework data providers. The tests were run on several databases to investigate whether or not the data providers for the databases influence the performance of the Entity Framework in relation to Graph-Based Querying.

Table 6.2: Database selection overview

Database	**Storage Type**	**EF Data Provider**
SQL Server (LocalDb V12)	Row & Column with limitations [Mic14d]	Microsoft
MySQL 5.6	Row	MySql Connector/Net
PostgreSQL 9.3.4	Row	Npgsql

6.2.5 Measurement Method

To prevent measurement errors caused by the low precision of the thread timer, tests are executed several times and the measured duration overall is then divided by the amount of runs (each test is run as many times as needed to create a thread run-time of at least 500 milliseconds). The resulting value is then used as the result for this single benchmark. In turn, each benchmark is run several times and all the benchmark results together are used to create the graphs in appendix A.
We use the median value instead of the average, as extreme values can greatly impact the average. The median instead represents a value that occurs most often.
While you can see the absolute time on the graphs, we are less interested in the absolute time. Instead, we are interested in the relative time difference between the Entity Framework and Graph-Based Querying.

6.2.6 Test Groups

The test group set-up we use differs slightly from the one used in [Mer11]. In the article it defines 'EFSingleQuery', 'EFMultiQuery' and 'GraphBasedQuerying'. As the 'EFSingleQuery' is unable to work for the models with associations on sub-types (it is impossible to retrieve this data with the Entity Framework in a single statement), measuring the performance for this has no real use. Instead we define the categories 'EF' and 'GBQ' for the 'EFMultiQuery' and 'GraphBasedQuerying' respectively. The article also uses several statements that specify the retrieval of concrete types instead of a single statement for the data set. As we said before, for that Entity Framework version that resulted in the best performance. To see if this performs better than just specifying the database set, we introduced the 'EF Concretes' category in which we specifically retrieve the concrete types instead. This is to validate hypothesis 4.

6.3 Analysis

In this section we discuss the measurements we took for the same datasets as defined in [Mer11]. The result graphs can be found in appendix A.

6.3.1 Inheritance only

When we look at the result graphs for Sql Server (A.1.1, A.1.2, A.1.3, A.1.4), we can see that specifically retrieving concrete types is the slowest way to retrieve these object for most of our datasets. Only at Inheritance 6 we can see that the measured real time for both the standard and concrete types query become equal. This means that for even larger inheritance trees it may become feasible to specifically request the concrete types, however this can potentially result in a very large amount of code as a retrieval statement is needed for each concrete type. In our case, the Inheritance 6 dataset has 32 concrete types and as such has 32 statements to retrieve them all.
While specifically retrieving the concrete types and the standard query begin to perform equally with the larger trees, we can see that Graph-Based Querying is continuously able to query the data in the fastest way possible. Even for the Inheritance 6 tree we can see that it still outperforms the Entity Framework.
When we look at the CPU timing graphs for the same tests (A.1.1, A.1.2, A.1.3, A.1.4), we can see that Graph-Based Querying and the Entity Framework spend about the same time executing their code. The test that specifically requests the concrete types spends more time executing because each concrete type has its own statement to execute.

When we take a look at the graphs for PostgreSql, we can see that it deviates from the Sql Server results. While the CPU time graphs show us approximately equal times for Graph-Based Querying and the Entity Framework as well (A.1.1, A.1.2), we can see that the Entity Framework statement becomes slower in relation to Graph-Based Querying when the inheritance tree grows (A.1.3, A.1.4). The PostgreSql data provider spends more and more time constructing the queries for the larger the inheritance trees and becomes slower than Graph-Based Querying in the process.
The total time measurements for PostgreSql show the most difference when compared to the results from Sql Server. While the concrete tests still perform the worst, in PostgreSql the concrete and standard Entity Framework statement do not eventually perform as fast as the other (A.1.1, A.1.2, A.1.3, A.1.4). In addition, the Entity Framework statement actually performs faster than Graph-Based Querying on PostgreSql for the Inheritance 3 tree (A.1.1). Anything larger and Graph-Based Querying performs faster (A.1.2, A.1.3, A.1.4). This means that for small inheritance trees Graph-Based Querying has no performance benefit over the Entity Framework when used with PostgreSql.

With MySql we can see that, as with Sql Server, Graph-Based Querying is the fastest for the data retrieval (A.1.1, A.1.2, A.1.3, A.1.4). We can also see that specifically requesting the concrete types when using the Entity Framework is actually faster on MySql than the standard Entity Framework statement and by a rather large amount. This is the opposite of what we observed with Sql Server and PostgreSql.

The CPU times for MySql (A.1.1, A.1.2, A.1.3, A.1.4) show that the Entity Framework runs for a much longer time when compared to the CPU times for Sql Server and PostgreSql. This shows that the MySql data provider has a much larger impact on the total time for the Entity Framework than the data providers for Sql Server and PostgreSql. As Graph-Based Querying does not use the data providers to construct the queries, the measured time gap between Graph-Based Querying and the Entity Framework becomes greater the slower the data provider is. This indicates that a more optimized data provider may be able to decrease the time of the Entity Framework and make the Entity Framework more suitable for smaller inheritance trees, as we can already in part see with PostgreSql.
What we also see with MySql is gaps in the results; the Inheritance 6 tree could not be retrieved with Graph-Based Querying. The problem we encountered with Graph-Based Querying on MySql is that MySql employs a JOIN statement limit (61 JOIN statements maximum). We reach this limit with the query we generate with our implementation for the Inheritance 6 tree and thus the query does not execute. As the Entity Framework generates queries with UNION statements it does not reach this limit for our datasets and still returns valid results. This shows that the current implementation method of Graph-Based Querying, which constructs queries with JOIN statements instead, can fail if the database employs a limit. As such, the effectiveness is currently dependent on the database that is being used.

6.3.2 Associations on Subtypes

Datasets that have associations on sub-types are what Graph-Based Querying was designed for. If we look at PostgreSql we can see that the larger the inheritance tree, the faster GBQ is compared to the Entity Framework (A.2.2, A.2.2, A.2.2, A.2.3, A.2.3, A.2.3, A.2.4, A.2.4, A.2.4) with the exception of the Inheritance 3 tree (A.2.1, A.2.1, A.2.1). As we also saw in the inheritance only test results, PostgreSql is able to perform faster than Graph-Based Querying for this inheritance tree. However, since we now deal with associations on subtypes, this only applies to the smaller populations. Because Graph-Based Querying sends the request for all the data at once, it allows the database to work with the queries as a whole whereas the Entity Framework sends the requests for each sub-type association separately. The database does not process all the queries at the same time and can not optimize the queries or loaded data, resulting in a longer execution time. It is also important to note that once pre-fetching is enabled in Sql Server, it does so later for Graph-Based Querying. This shows that the queries we create, while retrieving the same rows, are not analysed by Sql Server as exceeding the threshold at the same time as for the Entity Framework. We can also see that when pre-fetching is enabled, Graph-Based Querying can perform slower than the Entity Framework.
The CPU time for all these tests show similar results both for SqlServer (A.2.1, A.2.1, A.2.1, A.2.2, A.2.2, A.2.2, A.2.3, A.2.3, A.2.3, A.2.4, A.2.4, A.2.4) and PostgreSql (A.2.1, A.2.1, A.2.1, A.2.2, A.2.2, A.2.2, A.2.3, A.2.3, A.2.3, A.2.4, A.2.4, A.2.4). Graph-Based Querying spends more time constructing a query but because it construct everything at once, it can send and retrieve the data faster. This is why it performs better once associations on subtypes are present in the dataset. With the Entity Framework, each of these associations require a separate statement which results in several round-trips to the database which increases the total time.

While we can already see some improvements over Sql Server and PostgreSql, with MySql we can see that Graph-Based Querying has an enormous performance increase over the Entity Framework (A.2.1, A.2.1, A.2.1, A.2.2, A.2.2, A.2.2, A.2.3, A.2.3, A.2.3, A.2.4, A.2.4, A.2.4). For the smallest population for the Inh3Assoc1 dataset the time the Entity Framework needs to retrieve all the data almost reaches half a second where this is barely twelve milliseconds on Sql Server and PostgreSql. This shows the impact of the data provider on the overall performance of the Entity Framework even better. The pattern MySql shows in these graphs is the same as PostgreSql; Graph-Based Querying is able to retrieve the data for all the populations in a constant time across populations whereas the Entity Framework becomes slower with each larger population.
When we look at the CPU time for MySql (A.2.1, A.2.1, A.2.1, A.2.2, A.2.2, A.2.2, A.2.3, A.2.3, A.2.3, A.2.4, A.2.4, A.2.4) we can see that Graph-Based Querying is able to execute in a constant time for each of the populations. This pattern can be seen on Sql Server and PostgreSql as well. The

CPU time for the Entity Framework on MySql does not show the same pattern. For MySql we can see that the Entity Framework is slower than Graph-Based Querying but for Sql Server and PostgreSql it is faster. This again shows that the data provider implementation impacts the performance of the Entity Framework.
The gaps we see for the Inheritance 6 datasets have the same cause as mentioned before; the MySql join limit.

6.4 Conclusion

We investigated the potential of Graph-Based Querying and created an implementation of Graph-Based Querying for the Entity Framework and compared its performance against the Entity Framework itself and the results found by M. de Jonge [Mer11]. As seen in the results discussed in the previous section, the limitations of a database can influence the value of Graph-Based Querying as it may prevent it from functioning. We also see that it still performs better than the Entity Framework for the data structures it was designed for; inheritance trees with relations on sub-types.

When we check our first hypothesis against the results 6.3.1, we can see that it mostly validated. Graph-Based Querying is only slower for the retrieval of directly related entities when used on PostgreSql and the Inheritance 3 dataset. For the other sets it performs as well as the Entity Framework or better. Important to note is that when we look at the results for MySql, we can see that MySql is unable to execute the queries we construct for the Inheritance 6 dataset and that limitations of the database affect the use of the current implementation of Graph-Based Querying.
The second hypothesis is mostly valid as well. The results show 6.3.2 again that only PostgreSql with the Inheritance 3 dataset is able to outperform Graph-Based Querying but now only on the smallest populations. For MySql we can see that Graph-Based Querying greatly increases the performance but as noted before, it does not function on MySql once the constructed query exceeds 61 joins. This limit currently prevents Graph-Based Querying from functioning on MySql with datasets such as the Inheritance 6 dataset.

When we look at the measurement graphs, we can see that the relative performance increase over the Entity Framework is less then measured in the original article [Mer11] when we use Sql Server, which confirms hypothesis 3, as those tests were also run on Sql Server.
If we look at the results for PostgreSql as well, we can still see that the performance increase is less and that sometimes the Entity Framework performs better instead. When we look at the results for MySql we can see that the relative performance increase of Graph-Based Querying over MySql is much greater than what was measured in [Mer11].
Lastly, the graphs show that even now, specifically selecting the concrete types for retrieval can still outperform selecting the set itself. On Sql Server we can see that as the population grows or the inheritance tree grows, specifically selecting the concrete types for retrieval eventually outperforms selecting the set itself. We can also see that the database itself can impact this; with MySql we can see immediately that specifically selecting the concrete types outperforms selecting the set itself. As such, hypothesis 4 turns out to be invalid. This also indicates that there is still room for improvement in the query construction for the retrieval of Table-per-Type hierarchies.

6.5 Threats to Validity

As we mentioned before, the data providers for the Entity Framework impact the overall time of the Entity Framework. As such, the measurements of the Entity Framework can change when the vendor of a specific data provider creates a new version.
We also mentioned that Graph-Based Querying improves performance by decreasing the amount of round-trips. As such, manually constructed queries may still be able to outperform Graph-Based Querying but this would require knowledge of and working with the relational model and SQL code, which is what we already attempted to avoid with an ORM in the first place.

Chapter 7

Future Work

7.1 Implementation Method

In this project Graph-Based Querying was created as an extension on the Entity Framework and available as a separate library. Further improvements in performance may be possible by utilizing internal information and functions of the Entity Framework to speed up SQL building and attaching to the database- or object context (ie. the ShaperFactory). However, this would mean that we need to either use reflection to access these internal functions, or embedding Graph-Based Querying into the Entity Framework.
It would also be an option to make Graph-Based Querying independent of the Entity Framework so the Entity Framework is no longer required if you just want to use Graph-Based Querying. Since Graph-Based Querying uses ORM information from the Entity Framework, this separation would mean that it needs to implement some ORM functionality itself.

7.2 Additional Functionality

The current implementation has no support for self-referencing entities. Graph-Based Querying could make loading self-referencing entities easier as well by implementing support for this. Right now, with the Entity Framework, a join is required for each self-referencing entity level you want to retrieve.
It also lacks support for Table per Hierarchy and Table per Concrete Class inheritance structures. To further expand the support of Entity Framework features this could be implemented.

7.3 Graph Operations

Since Graph-Based Querying builds on the idea of utilizing graphs as a way to query data, supporting graph operations such as transitive closure would be another way to further increase the value of Graph-Based Querying.

7.4 Query Construction

The current implementation of Graph-Based Querying constructs ANSI compliant SQL queries. As databases are not necessarily compatible with ANSI queries (be it completely incompatible or that it requires configuration changes to allow ANSI queries), Graph-Based Querying may be unable to execute the query. Out-of-the-box compatibility with different databases can further be improved by using the DbExpression API present in the Entity Framework and passing the resulting expression tree to the data provider to create the SQL query.

7.5 Entity Framework's Future

Entity Framework 7 is in development and is completely redesigned from the ground up. This may increase the performance of the Entity Framework greatly when compared to the older versions which all have been build on the same code base. When Entity Framework 7 is released it should be checked whether or not it performs better than Graph-Based Querying. If it does, Graph-Based Querying could be recreated on Entity Framework 7 (as it stands now, chances are that the current implementation method is not compatible with EF7).
Entity Framework 7 should also support non-relational databases so testing Graph-Based Querying over different database types is another interesting aspect that could be tested with Entity Framework 7.

Chapter 8

Acknowledgements

I would like to thank the teachers of the University of Amsterdam for the informative past year and interesting courses.

I'd also like to thank Jurgen Vinju for the time he spend guiding me with this thesis and for helping me improve my writing skills little by little.

Lastly I'd like to thank Merijn de Jonge for supplying me with his code and for answering some of my questions during the process of understanding said code.

Bibliography

[Apa] Apache Commons. JXPath. URL: http://commons.apache.org/proper/commons-jxpath/users-guide.html.

[Atu07] Atul Adya and José A. Blakeley and Sergey Melnik and S. Muralidhar. Anatomy of the ADO.NET Entity Framework. In *In SIGMOD'07*, 2007.

[BBC+01] A. F. Blackwell, C. Britton, A. Cox, T. R. G. Green, C. Gurr, G. Kadoda, M. S. Kutar, M. Loomes, C. L. Nehaniv, M. Petre, C. Roast, C. Roe, A. Wong, and R. M. Young. *Cognitive Dimensions of Notations: Design Tools for Cognitive Technology.* 2001. URL: http://www.springerlink.com/content/hrj8lgkhaq96v904.

[BJP+13] Philip A. Bernstein, Marie Jacob, Jorge Perez, Guillem Rull, and James F. Terwilliger. Incremental mapping compilation in an object-to-relational mapping system (extended version). Technical Report MSR-TR-2013-45, June 2013. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=191107.

[Car96] Carey, Michael J. and DeWitt, David J. Of Objects and Databases: A Decade of Turmoil. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, pages 3–14, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc. URL: http://dl.acm.org/citation.cfm?id=645922.673465.

[Cra08] Craig Freedman. Random Prefetching. MSDN, 10 2008. URL: http://blogs.msdn.com/b/craigfr/archive/2008/10/07/random-prefetching.aspx.

[Die12] Diestel, Reinhard. *Graph Theory, 4th Edition.*, volume 173 of *Graduate texts in mathematics*. Springer, 2012.

[E.F69] E.F. Codd. Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks. 1969.

[E.F70] E.F. Codd. A Relational Model of Data for Large Shared Data Banks. In *Communications of the ACM*, volume 13, pages 377–387. ACM, June 1970. doi:10.1145/362384.362685.

[Fab12] Fabiano Amorim. SQL Server Prefetch and Query Performance. Simple Talk, 5 2012. URL: https://www.simple-talk.com/sql/performance/sql-server-prefetch-and-query-performance/.

[Gol09] Gene Golovchinsky. Cognitive dimensions analysis of interfaces for information seeking. *CoRR*, abs/0908.3523, 2009.

[IBNW09] C. Ireland, D. Bowers, M. Newton, and K. Waugh. A classification of object-relational impedance mismatch. In *Advances in Databases, Knowledge, and Data Applications, 2009. DBKDA '09. First International Conference on*, pages 36–43, March 2009. doi:10.1109/DBKDA.2009.11.

[Jer10] Jeroen Back. Theory and experimental evaluation of object-relational mapping optimization techniques. Master's thesis, University of Amsterdam, January 2010.

[KBB02] Maria Kutar, Carol Britton, and Trevor Barker. A comparison of empirical study and cognitive dimensions analysis in the evaluation of UML diagrams. In J. Kuljis, L. Baldwin, and R. Scoble, editors, *Proceedings of the 14th Workshop of the Psychology of Programming Interest Group*, pages 1–14, June 2002. URL: http://www.ppig.org/papers/14th-kutar.pdf.

[Ker11] Kersten, M. and Zhang, Y. and Ivanova, M. and Nes, N. SciQL, a Query Language for Science Applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, AD '11, pages 1–12, New York, NY, USA, 2011. ACM. doi:{10.1145/1966895.1966896}.

[Kla07] Klar, Felix and Königs, Alexander and Schürr, Andy. Model Transformation in the Large. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 285–294, New York, NY, USA, 2007. ACM. doi:{10.1145/1287624.1287664}.

[Mer11] Merijn de Jonge. Improving Entity Framework Query Performance Using Graph-Based Querying. 2011. URL: http://www.codeproject.com/Articles/247254/Improving-Entity-Framework-Query-Performance-Using.

[Mic11] Microsoft. Northwind Db. CodePlex, 2011. URL: https://northwinddatabase.codeplex.com/.

[Mic13] Microsoft. Entity Framework. CodePlex, 2013. URL: https://entityframework.codeplex.com/.

[Mic14a] Microsoft. Entity Framework Code First. MSDN, 2014. URL: http://msdn.microsoft.com/en-us/data/jj193542.

[Mic14b] Microsoft. Entity Framework Model First. MSDN, 2014. URL: http://msdn.microsoft.com/en-us/data/jj205424.

[Mic14c] Microsoft. Modeling and Mapping. MSDN, 2014. URL: http://msdn.microsoft.com/en-us/library/vstudio/bb896343%28v=vs.100%29.aspx.

[Mic14d] Microsoft. MSDN - Sql Server Column Store. MSDN, 2014. URL: http://msdn.microsoft.com/en-us/library/gg492088.aspx.

[Mic14e] Microsoft. Performance Considerations for Entity Framework 4, 5, and 6. May 2014. URL: http://msdn.microsoft.com/en-US/data/hh949853.

[OW93] G. Ozsoyoglu and H. Wang. Example-based graphical database query languages. *Computer*, 26(5):25–38, May 1993. doi:10.1109/2.211893.

[P.J97] P.J. Rodgers and P.J.H. King. A Graph-Rewriting Visual Language for Database Programming. *Journal of Visual Languages & Computing*, 8(5–6):641–674, 1997. doi:{10.1006/jvlc.1997.0033}.

[Sch08] Schürr, Andy and Klar, Felix. 15 Years of Triple Graph Grammars. In *Proceedings of the 4th International Conference on Graph Transformations*, ICGT '08, pages 411–425, Berlin, Heidelberg, 2008. Springer-Verlag. doi:{10.1007/978-3-540-87405-8_28}.

[SQL] ISO/IEC 9075. URL: http://www.iso.org/iso/home/search.htm?qt=9075&sort=rel&type=simple&published=on.

[T. 96] T. R. G. Green and M. Petre. Usability Analysis of Visual Programming Environments: a 'cognitive dimensions' framework. *JOURNAL OF VISUAL LANGUAGES AND COMPUTING*, 7:131–174, 1996.

[Var05] Varró, Gergely and Friedl, Katalin and Varró, Dániel. Graph Transformation in Relational Databases. *Electron. Notes Theor. Comput. Sci.*, 127(1):167–180, March 2005. doi:{10.1016/j.entcs.2004.12.034}.

[Vij08] Vijay P. Mehta. *Pro LINQ Object Relational Mapping with C#*. Apress, July 2008.

[W3C] W3C. XQuery. URL: http://www.w3.org/standards/xml/query.

[Zlo77] M. M. Zloof. Query-by-example: A data base language. *IBM Syst. J.*, 16(4):324–343, December 1977. URL: http://dx.doi.org/10.1147/sj.164.0324, doi:10.1147/sj.164.0324.

[Zyl06] Zyl, Pieter van and Kourie, Derrick G. and Boake, Andrew. Comparing the Performance of Object Databases and ORM Tools. In *Proceedings of the 2006 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries*, SAICSIT '06, pages 1–11, Republic of South Africa, 2006. South African Institute for Computer Scientists and Information Technologists. doi:{10.1145/1216262.1216263}.

List of Figures

1.1 Retrieval of objects O, related object E00, its related A00 object, and for A10 (A00 sub-type) its related B00 object 6

2.1 Graph representation of data. The circles are nodes/vertices and the lines are the edges. 11
2.2 Graph-Based Querying in relation the developers' program and the database. 12

6.1 O to E association and E inheritance structure 41
6.2 E00 to A00 association 41
6.3 A00 structure for Assoc1 tests 42
6.4 A00 structure for Assoc2 tests 42
6.5 A00 structure for Assoc3 tests 43

List of Tables

4.1 Comparison of supported features in EF, GBQ 1 and GBQ 2 26
4.2 Comparison of supported relations in EF, GBQ 1 and GBQ 2 26

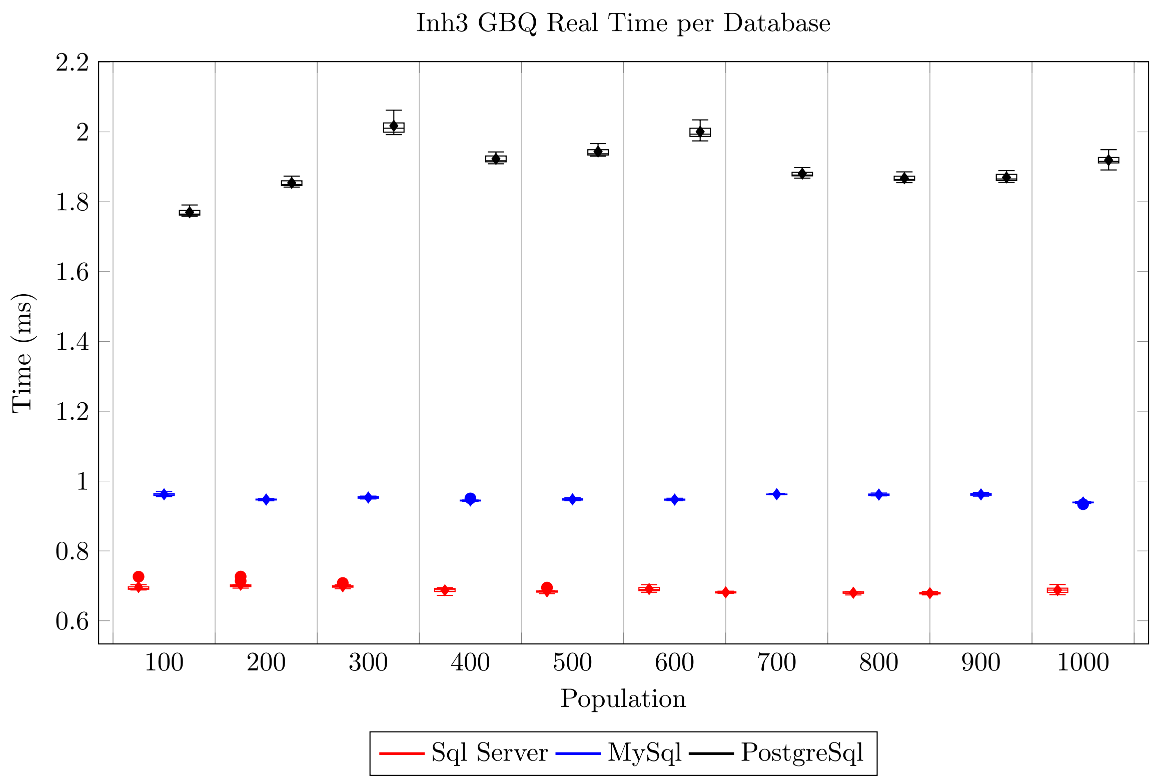
6.1 Populations 43
6.2 Database selection overview 44

Appendix A

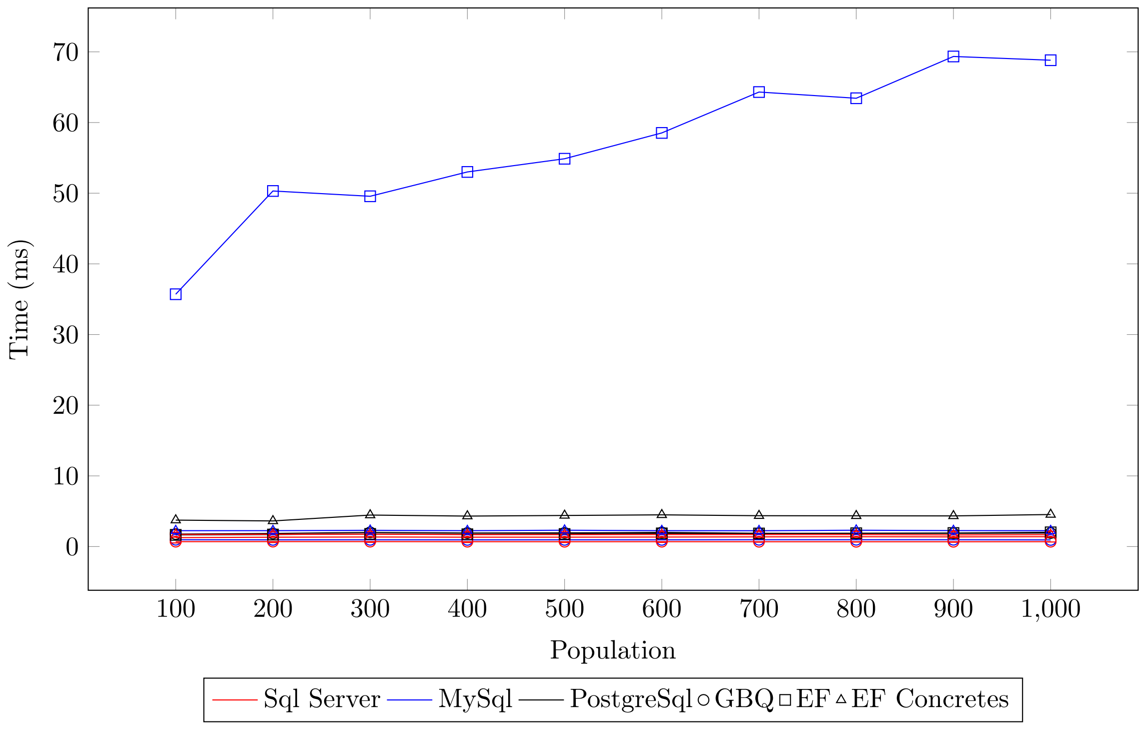
Measurement Results

A.1 No associations on sub-types

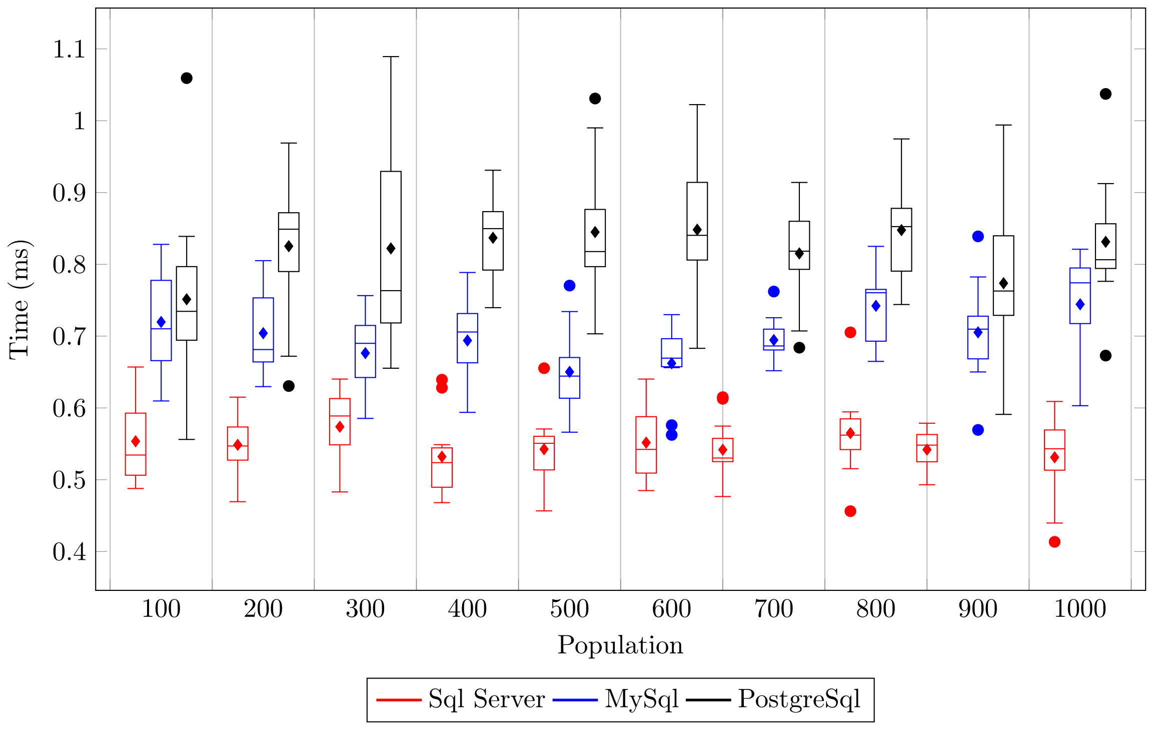
A.1.1 Inheritance 3

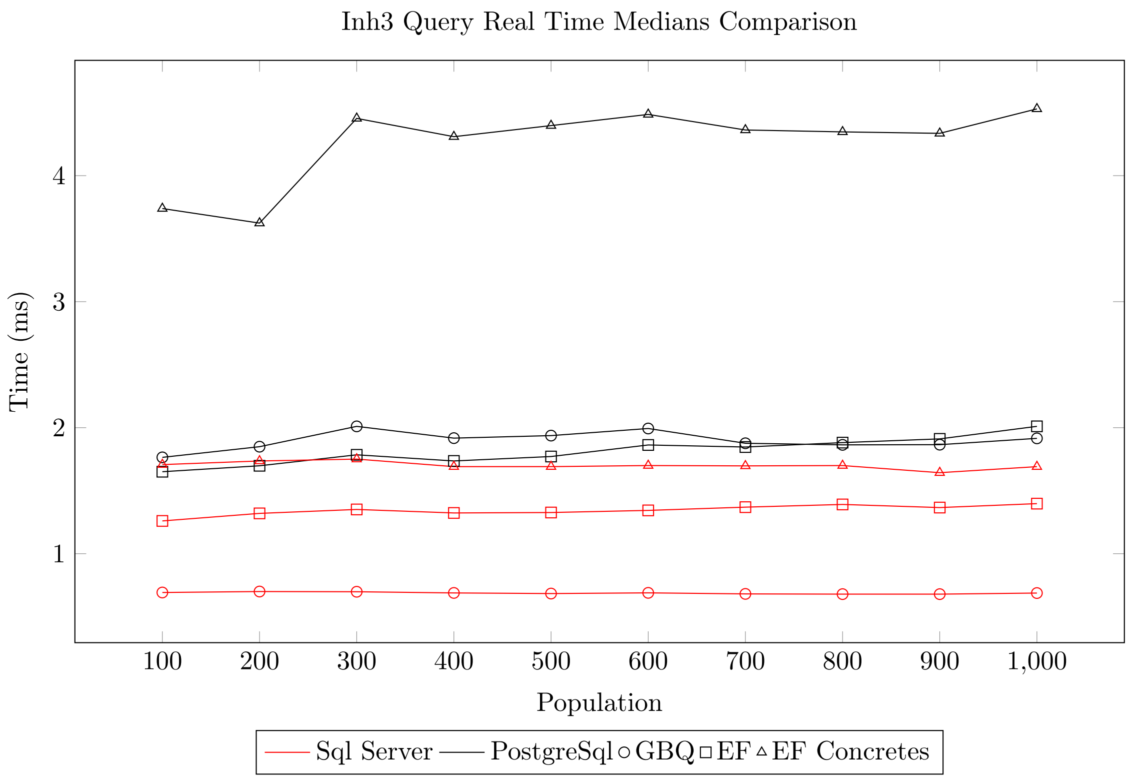


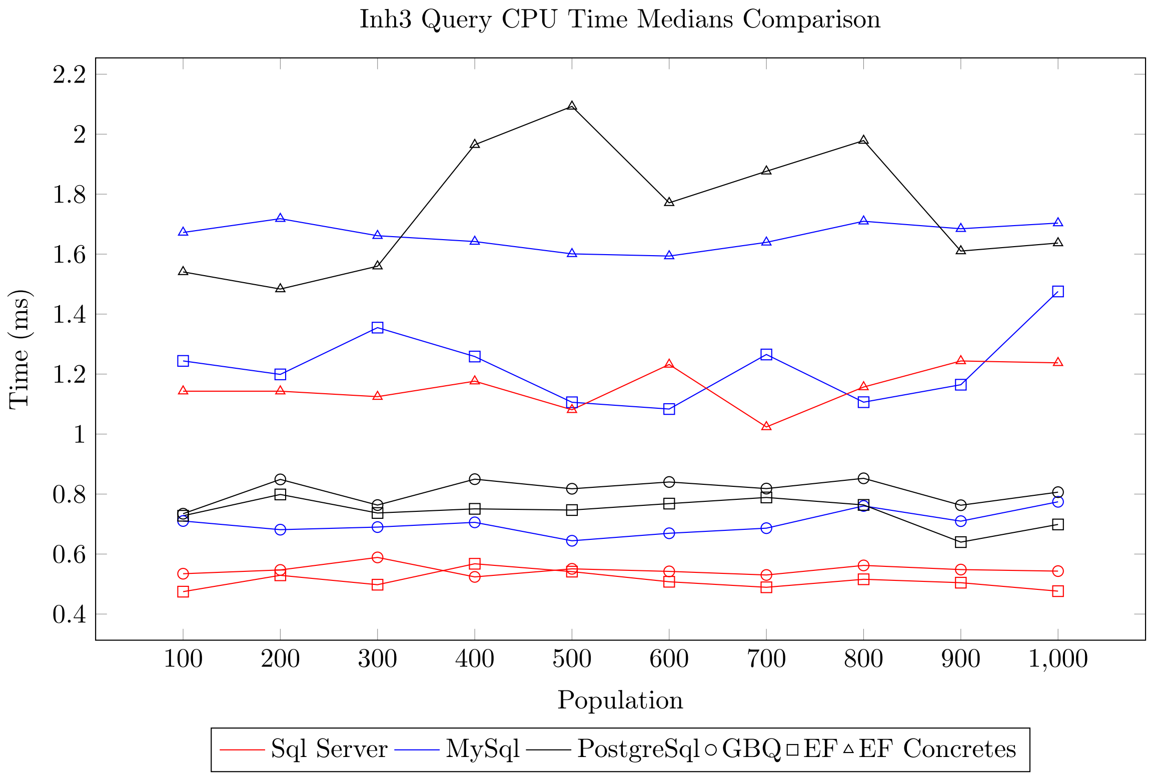
Inh3 Query Real Time Medians Comparison



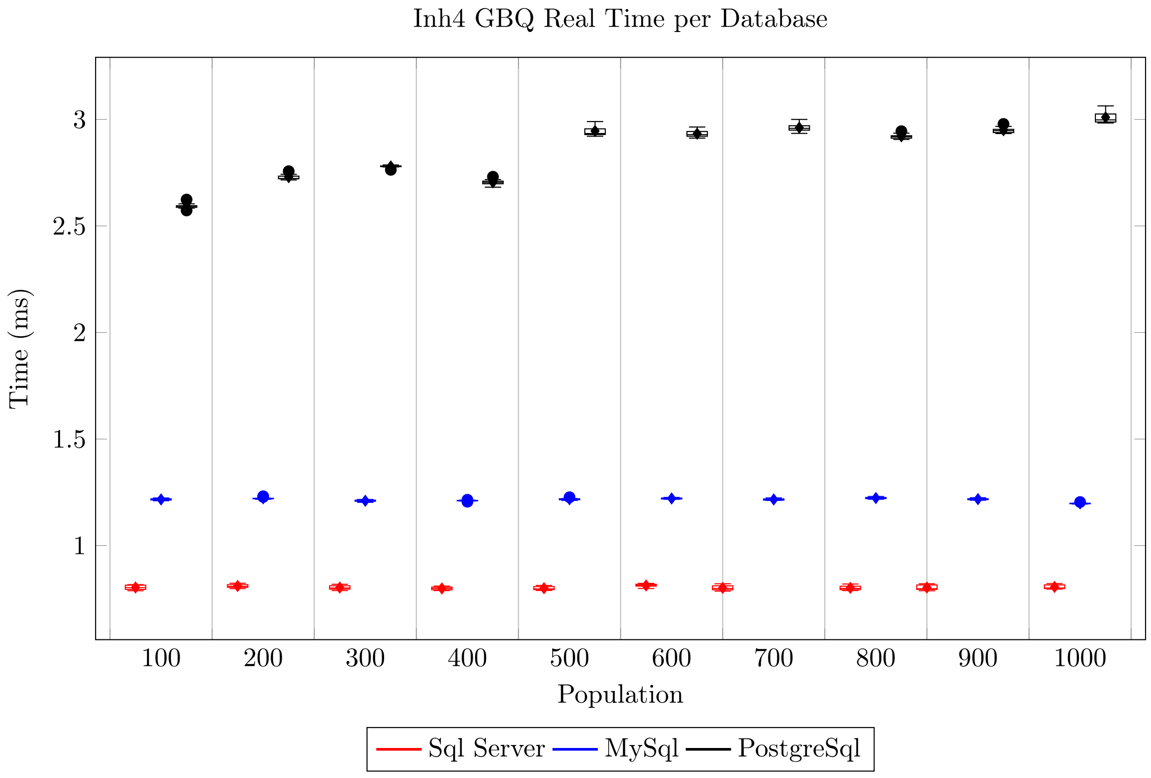
Inh3 GBQ CPU Time per Database

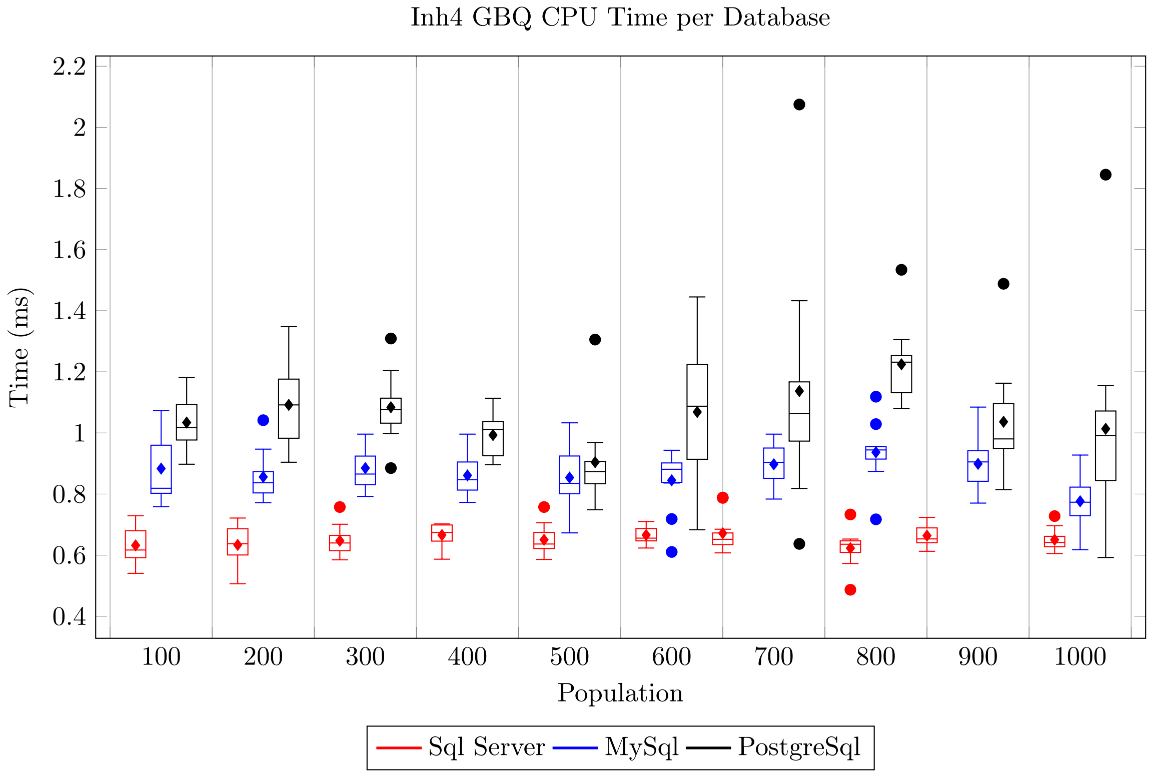


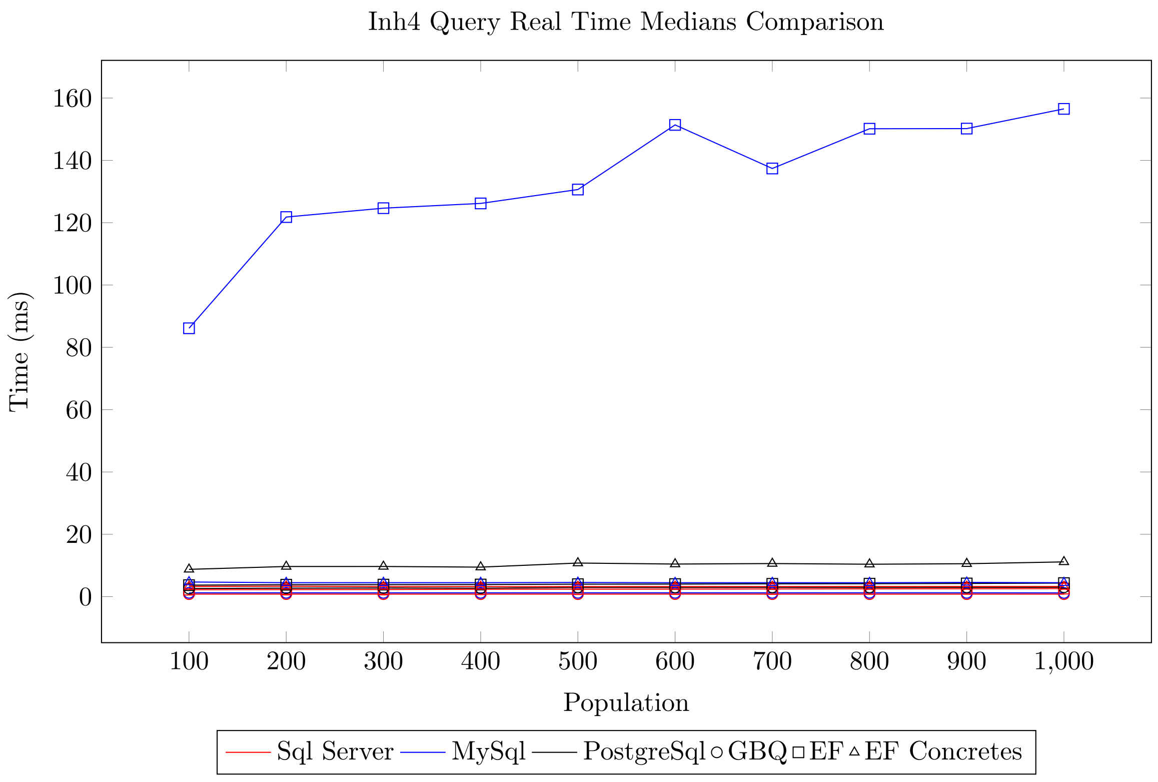

Inh3 Query Real Time Medians Comparison
Time (ms)
1
2
3
4
100
200
300
400
500
600
700
800
900
1,000
Population
Sql Server
PostgreSql
GBQ
EF
EF Concretes

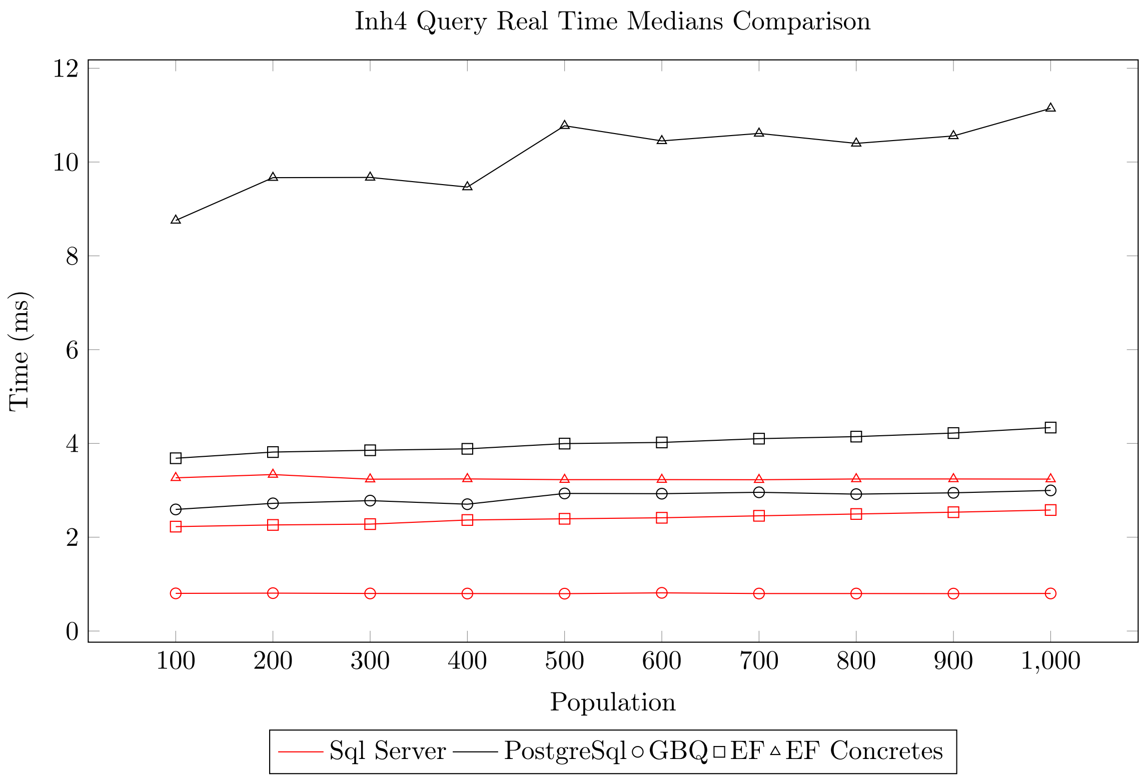

Inh3 Query CPU Time Medians Comparison
Time (ms)
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
2.2
100
200
300
400
500
600
700
800
900
1,000
Population
Sql Server
MySql
PostgreSql
GBQ
EF
EF Concretes

A.1.2 Inheritance 4

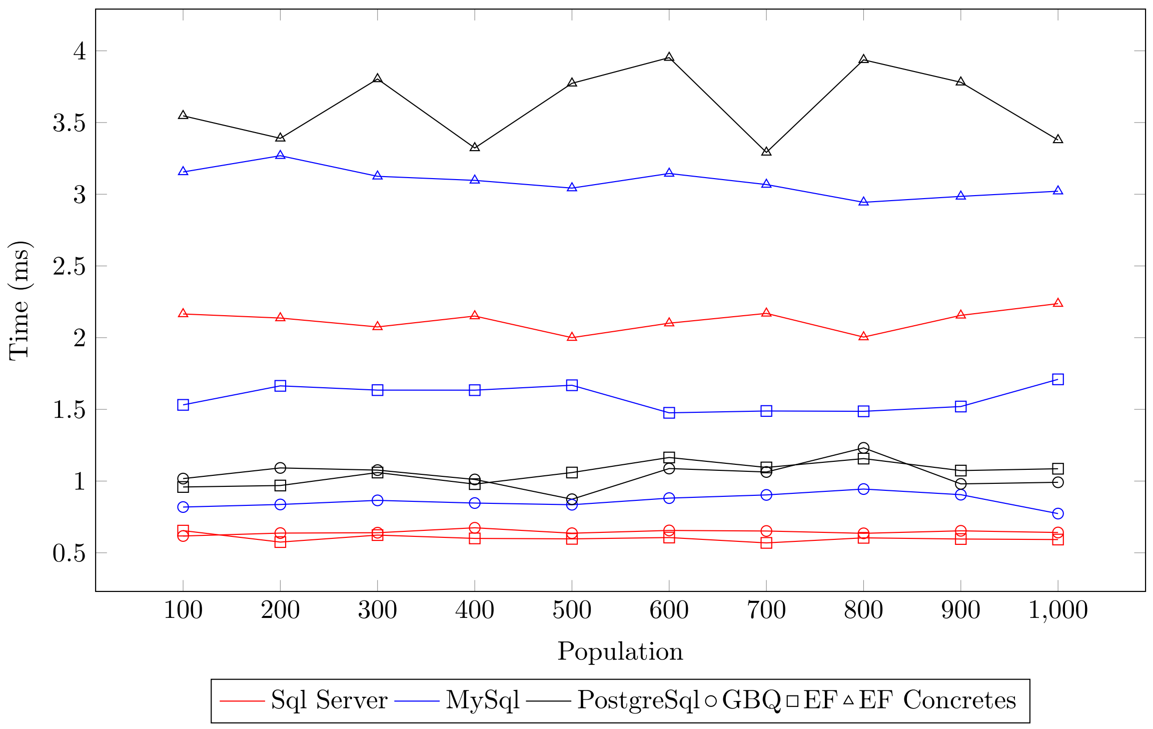





Inh4 Query Real Time Medians Comparison
Time (ms)
Population
160
140
120
100
80
60
40
20
0
100
200
300
400
500
600
700
800
900
1,000
Sql Server
MySql
PostgreSql
GBQ
EF
EF Concretes

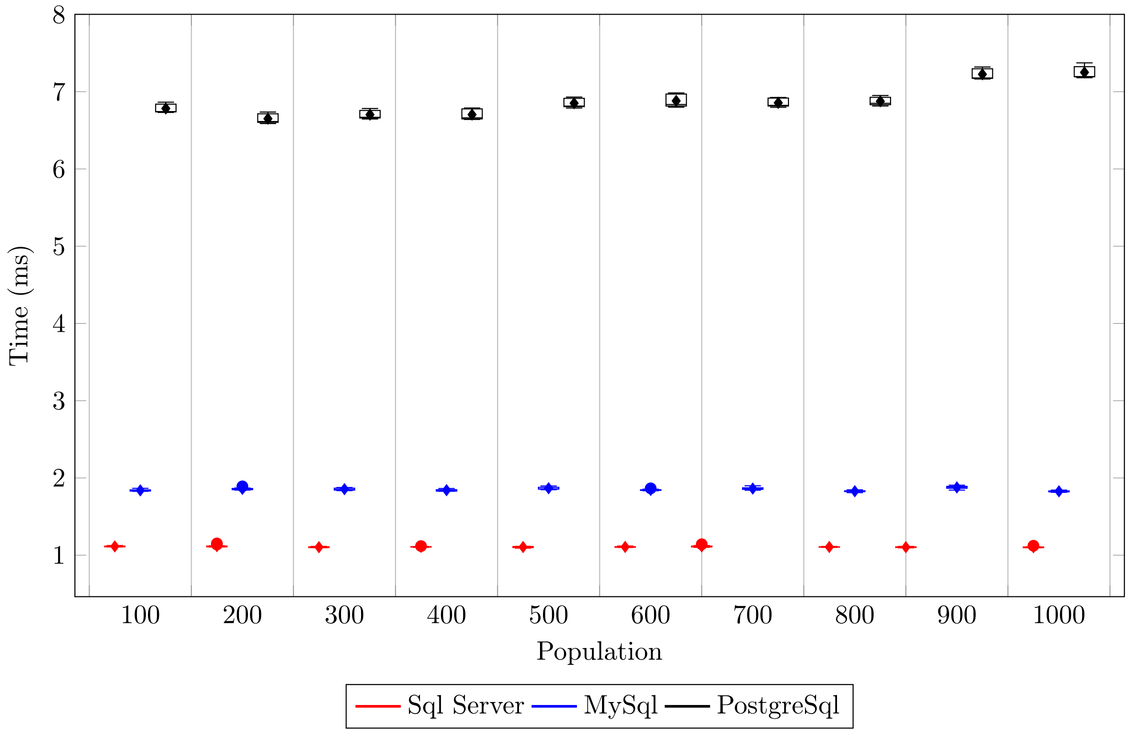

Inh4 Query Real Time Medians Comparison
Time (ms)
Population
12
10
8
6
4
2
0
100
200
300
400
500
600
700
800
900
1,000
Sql Server
PostgreSql
GBQ
EF
EF Concretes

Inh4 Query CPU Time Medians Comparison

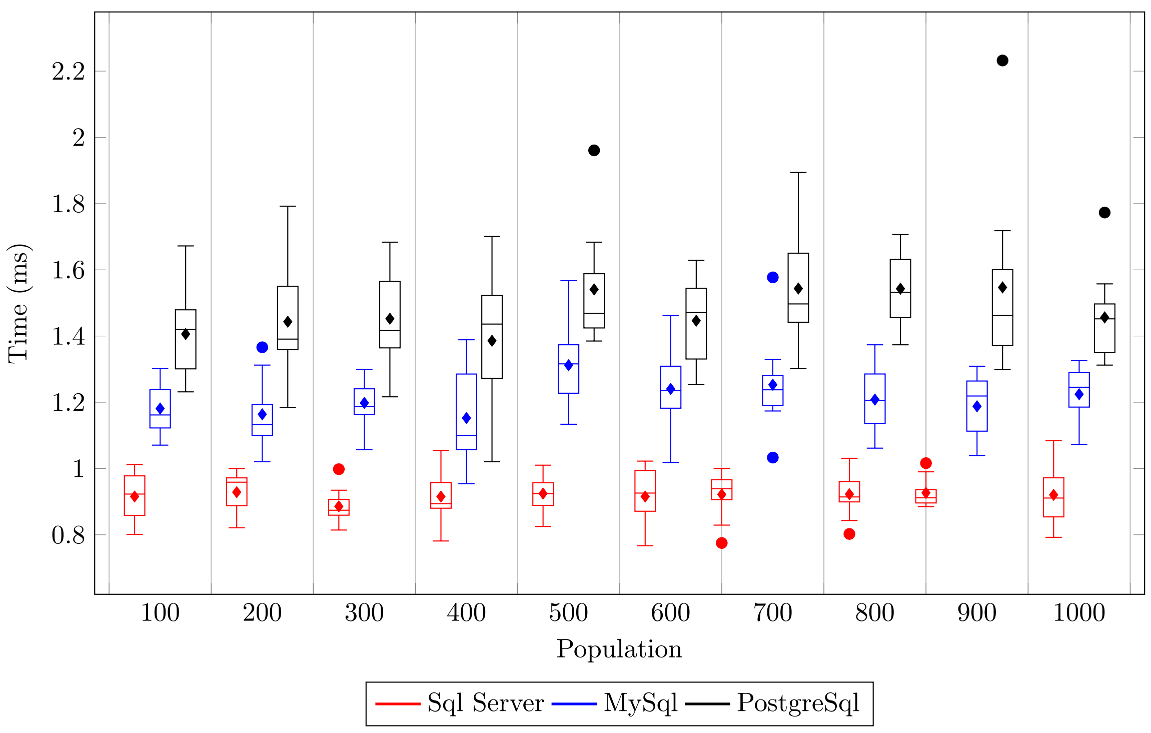


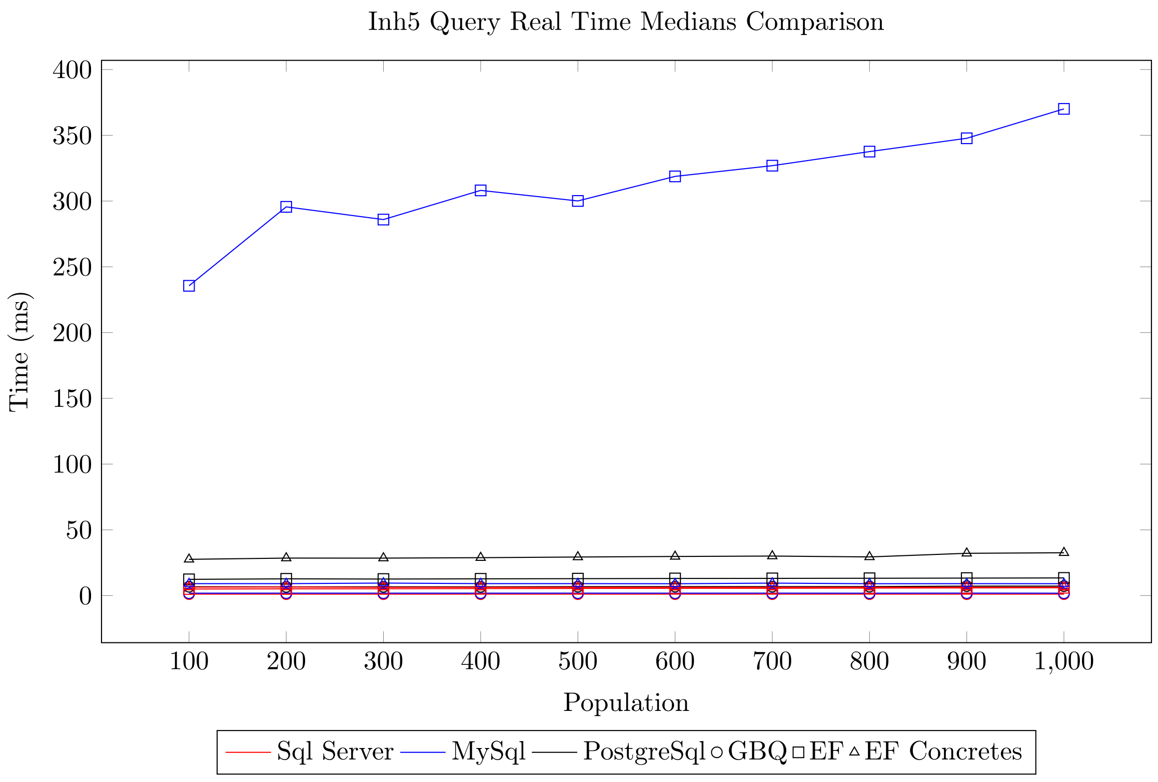
A.1.3 Inheritance 5

Inh5 GBQ Real Time per Database

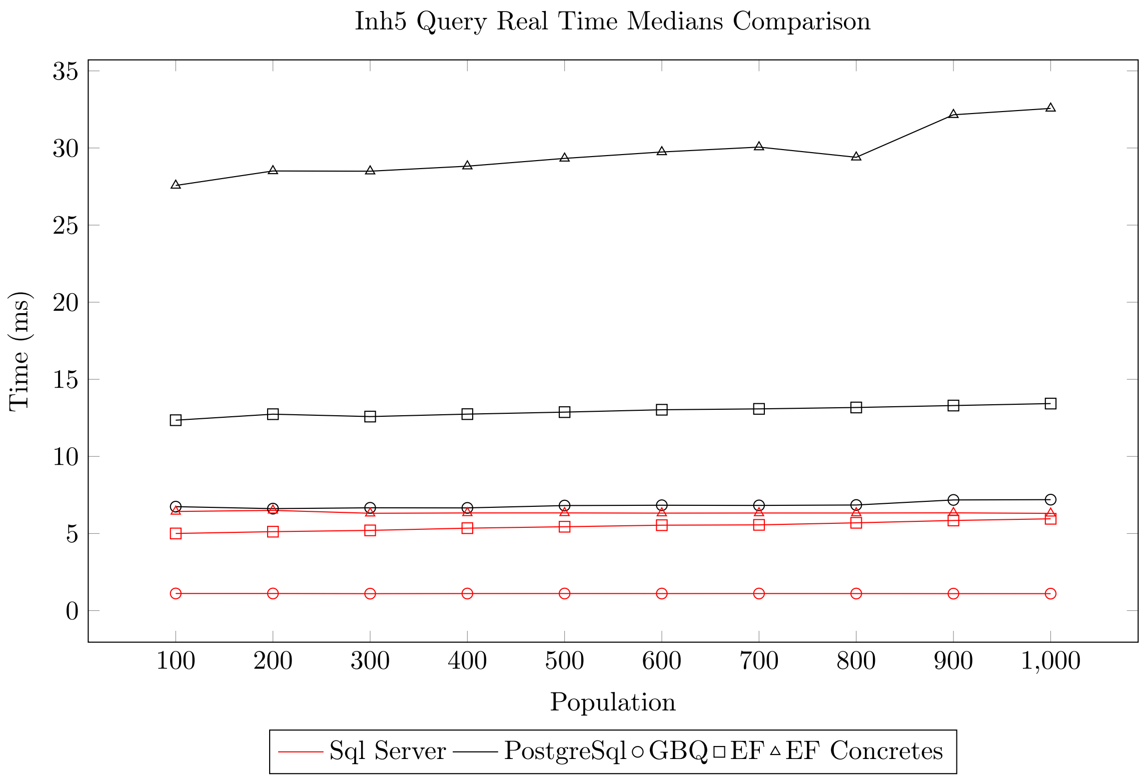


Inh5 GBQ CPU Time per Database



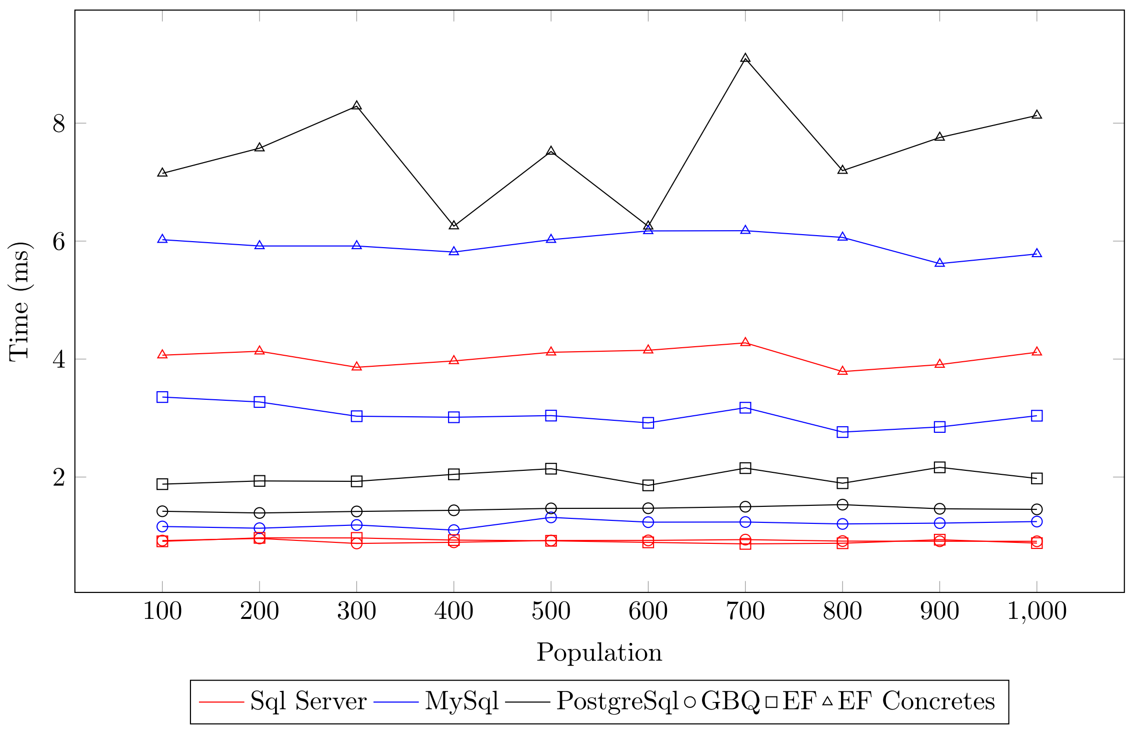


Inh5 Query Real Time Medians Comparison
Time (ms)
0
50
100
150
200
250
300
350
400
100
200
300
400
500
600
700
800
900
1,000
Population
Sql Server
MySql
PostgreSql
GBQ
EF
EF Concretes



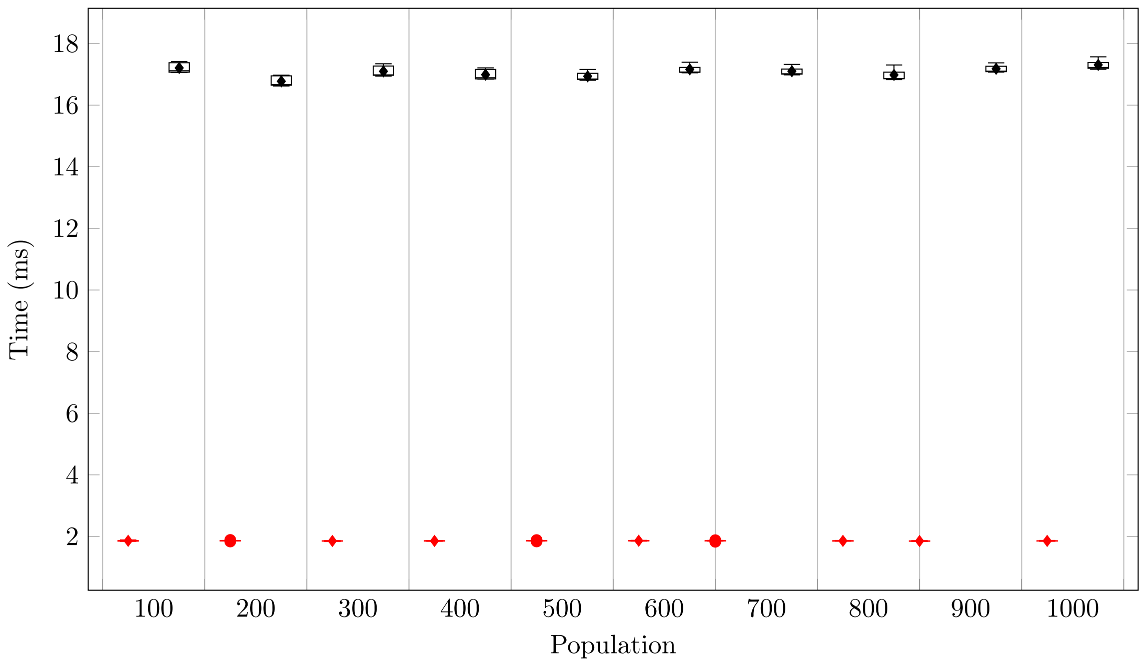
Inh5 Query Real Time Medians Comparison
Time (ms)
0
5
10
15
20
25
30
35
100
200
300
400
500
600
700
800
900
1,000
Population
Sql Server
PostgreSql
GBQ
EF
EF Concretes

Inh5 Query CPU Time Medians Comparison

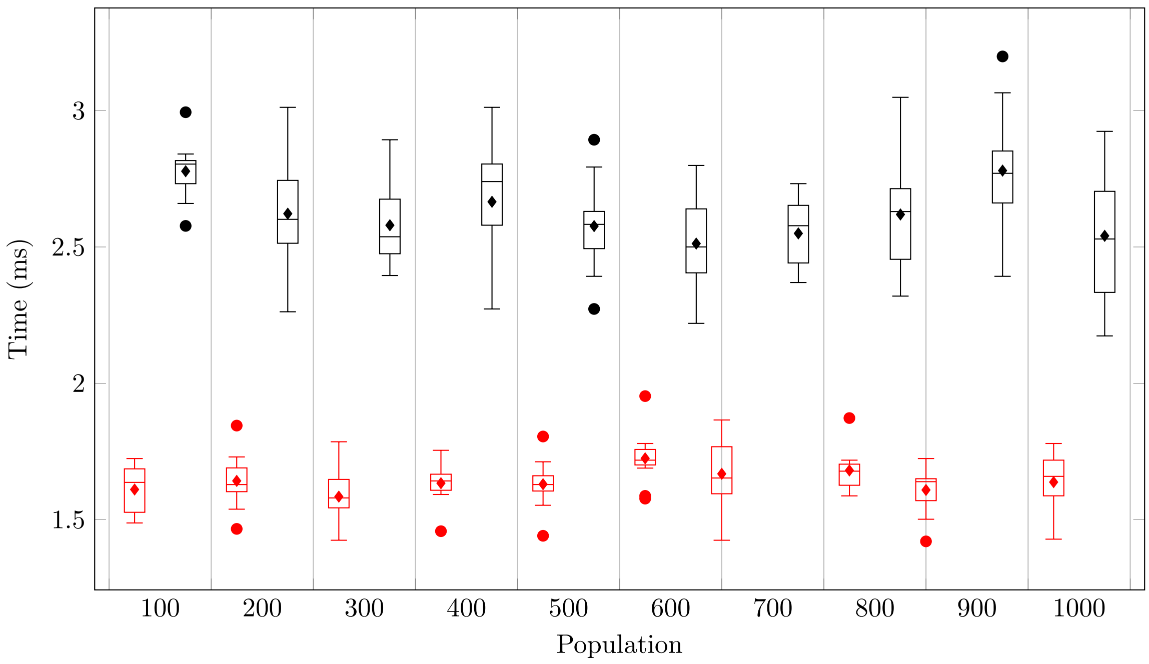


A.1.4 Inheritance 6

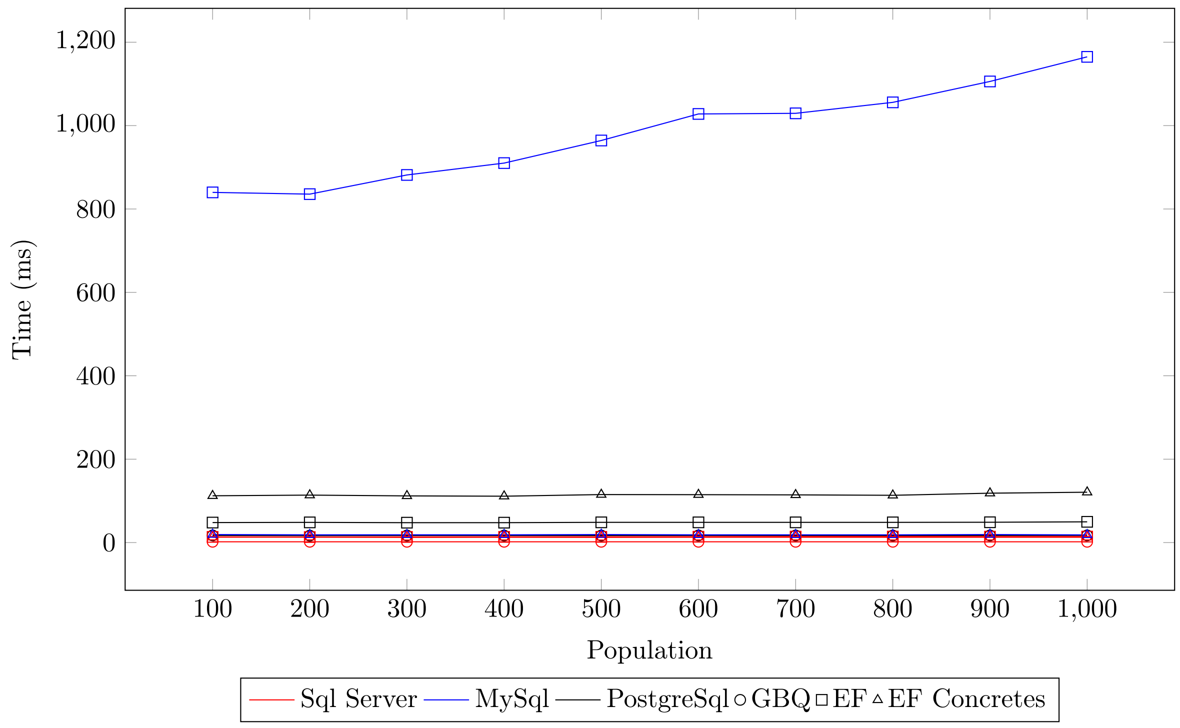
Inh6 GBQ Real Time per Database



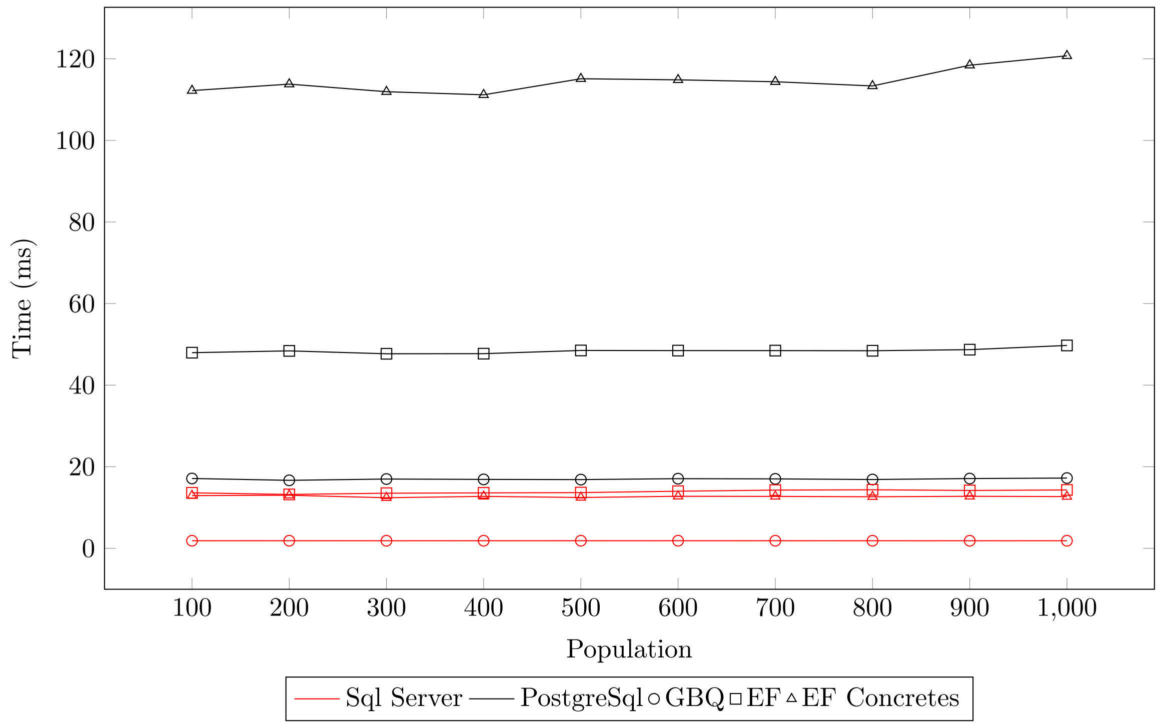
Inh6 GBQ CPU Time per Database

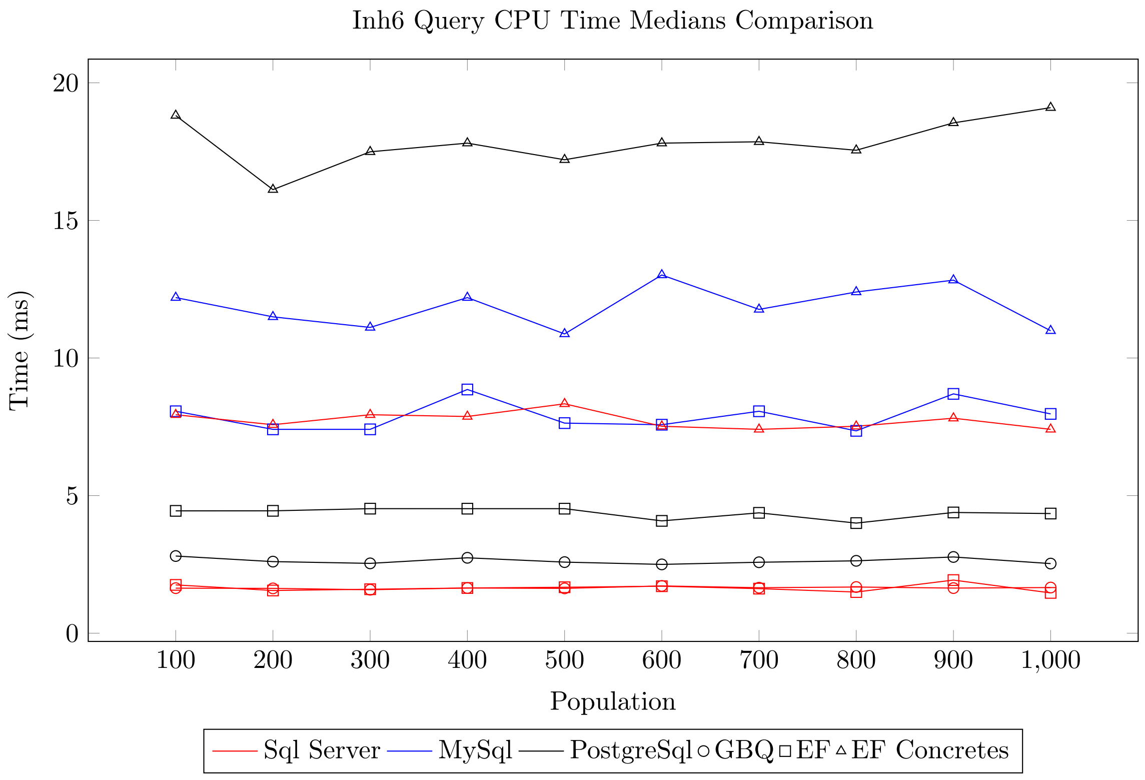


Inh6 Query Real Time Medians Comparison



Inh6 Query Real Time Medians Comparison

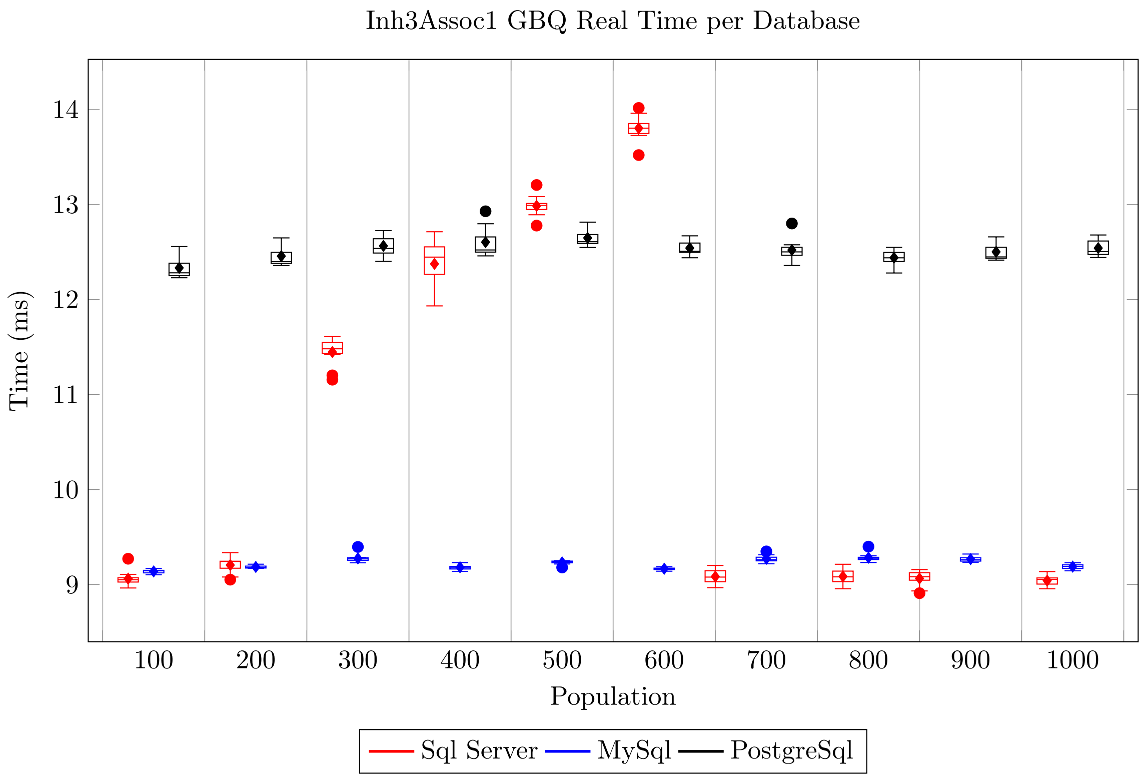


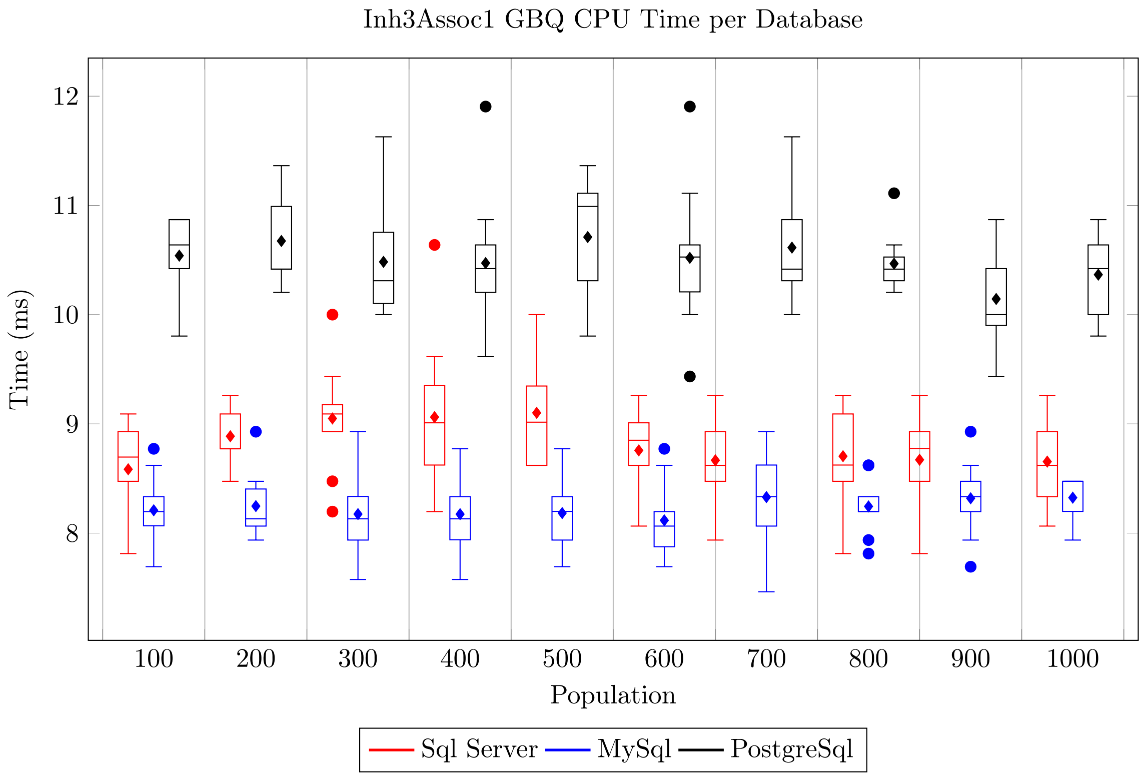

Inh6 Query CPU Time Medians Comparison
Time (ms)
0
5
10
15
20
Population
100
200
300
400
500
600
700
800
900
1,000
Sql Server
MySql
PostgreSql
GBQ
EF
EF Concretes

A.2 Associations on sub-types

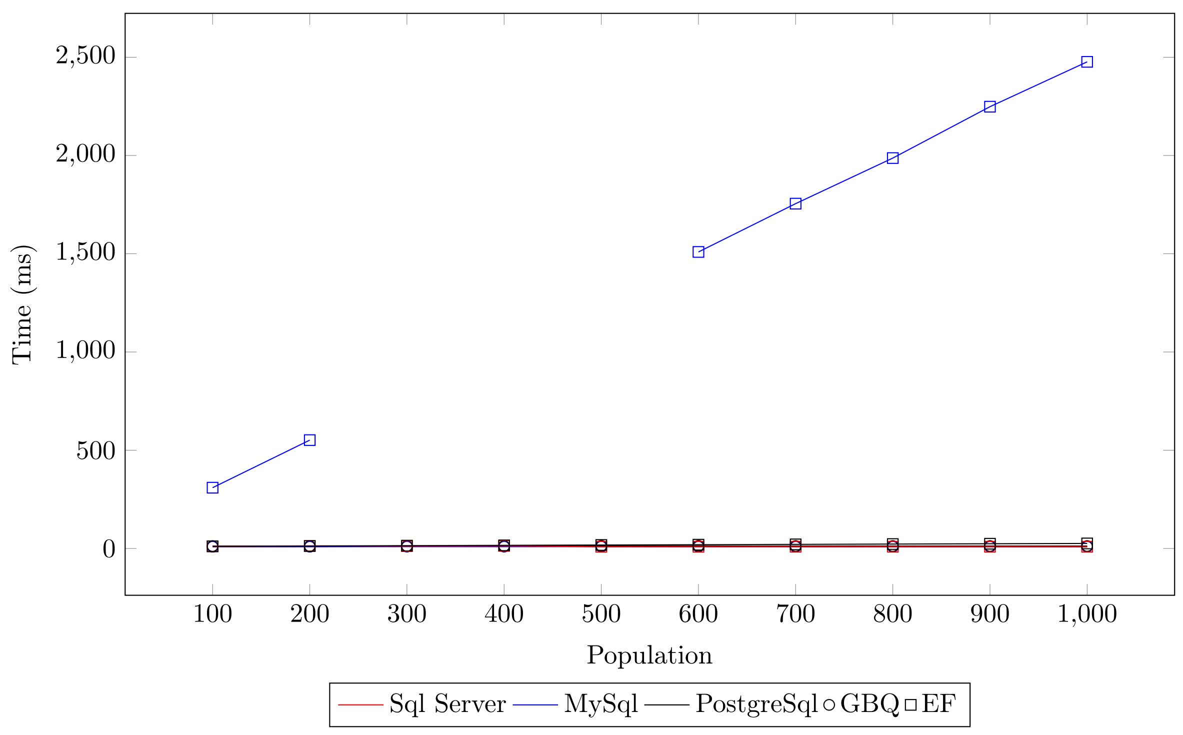
A.2.1 Inheritance 3

Assoc 1

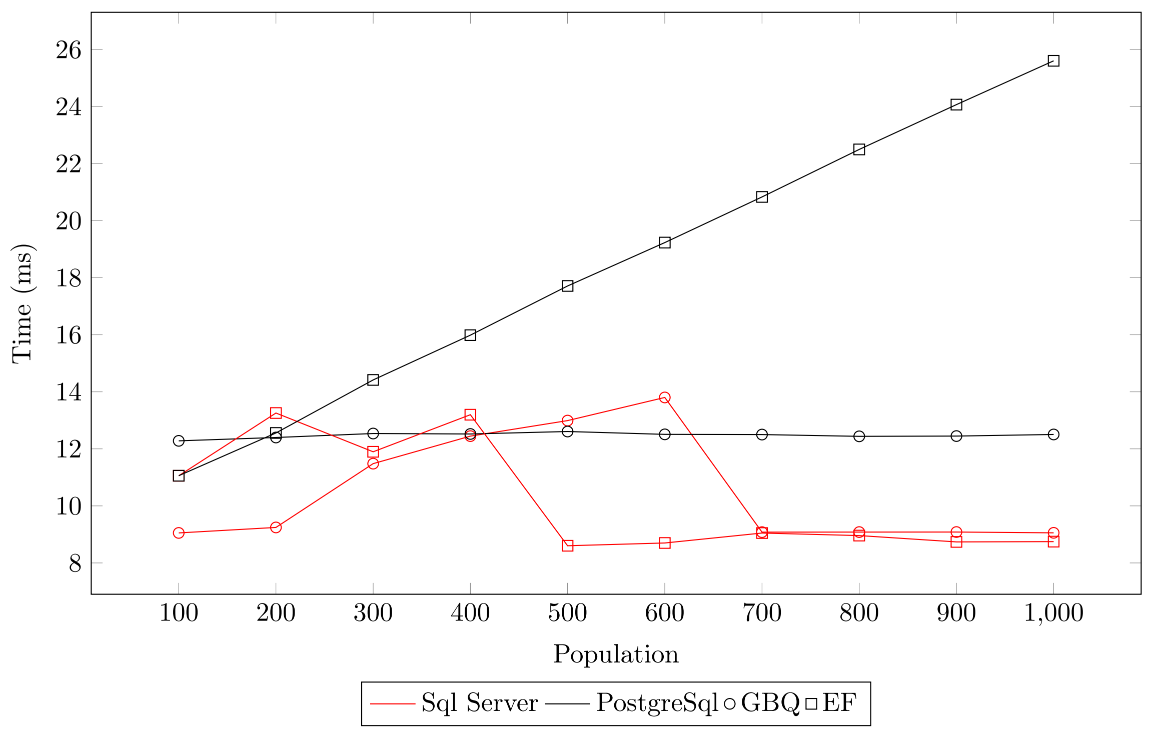




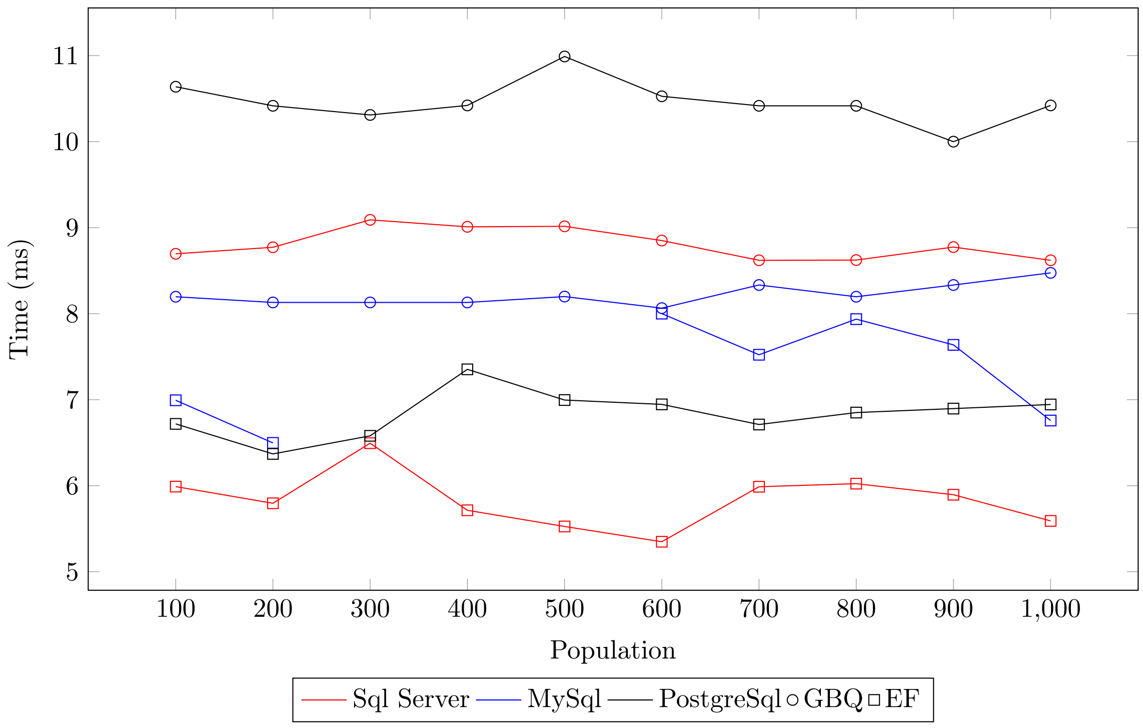
Inh3Assoc1 Query Real Time Medians Comparison



Inh3Assoc1 Query Real Time Medians Comparison

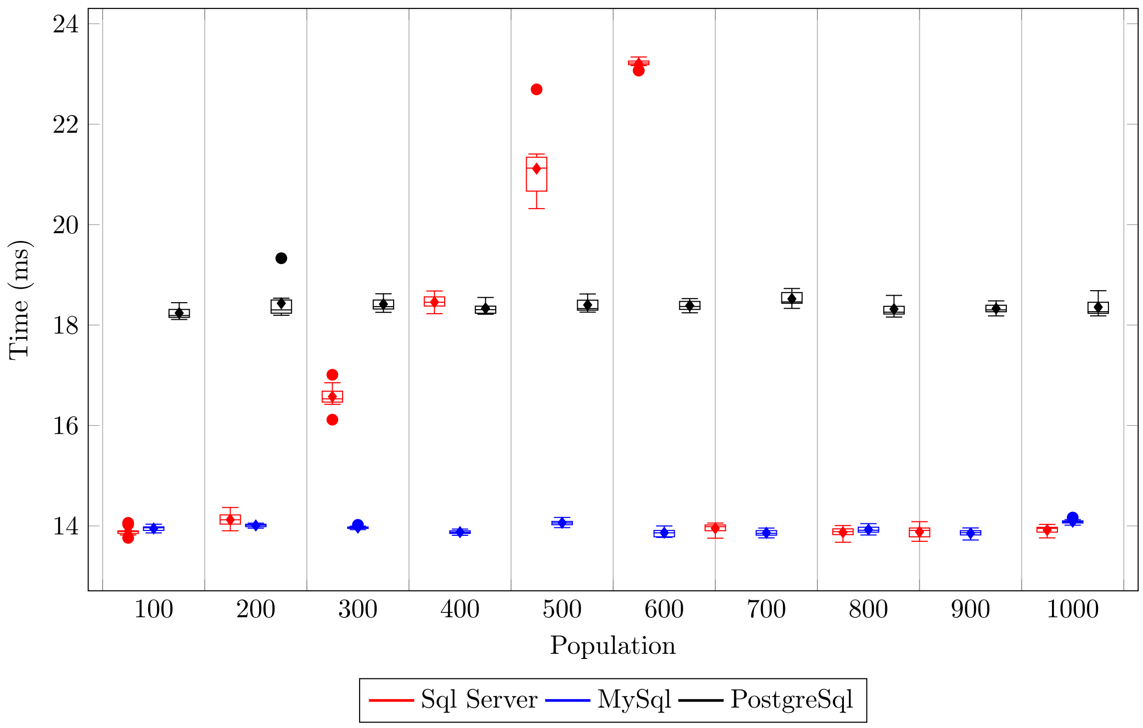


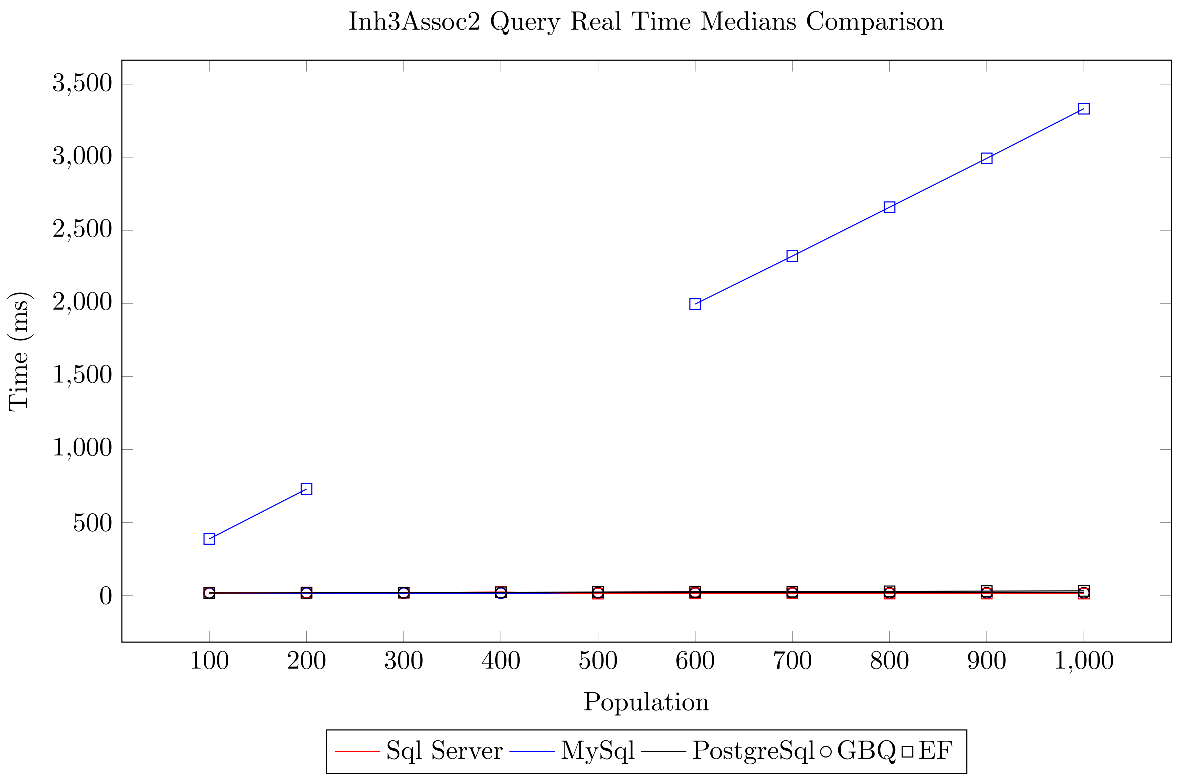
Inh3Assoc1 Query CPU Time Medians Comparison

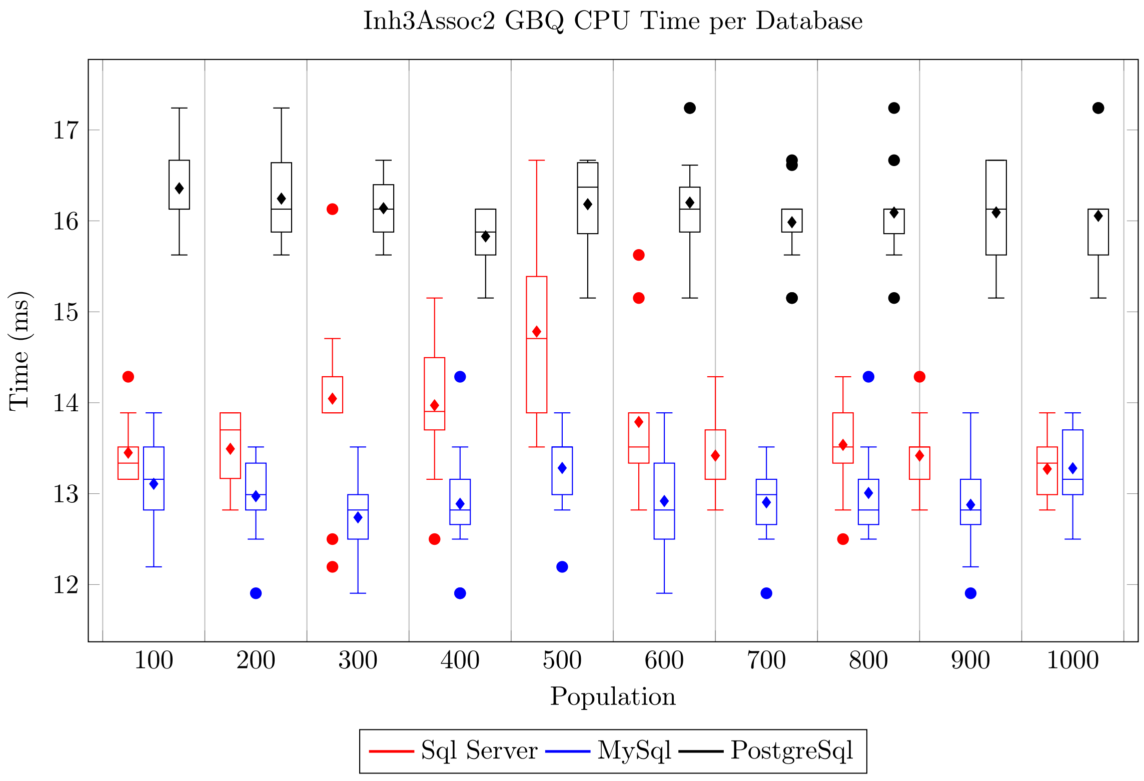


Assoc 2

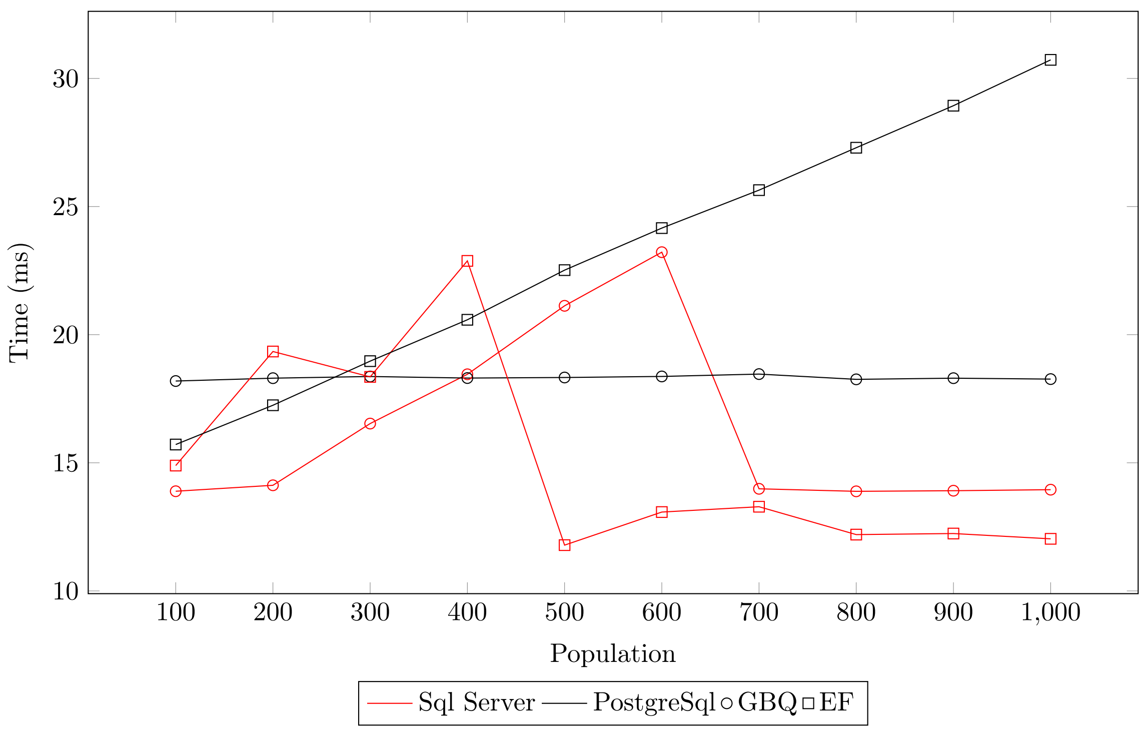
Inh3Assoc2 GBQ Real Time per Database



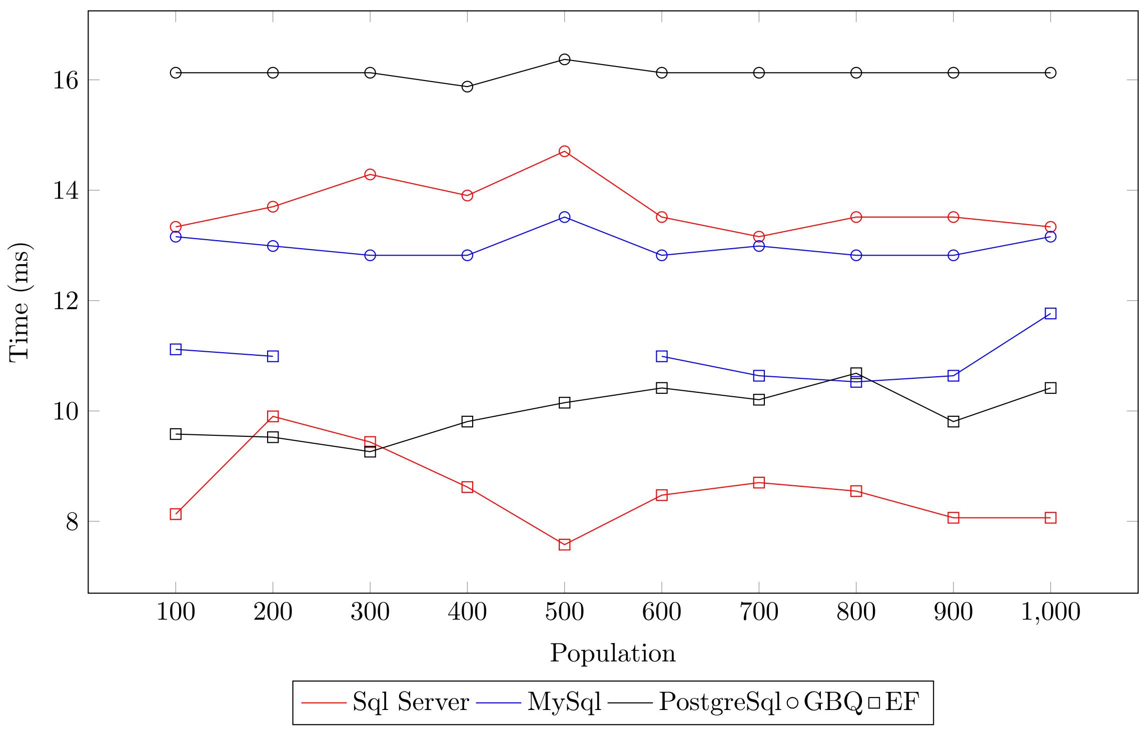

Inh3Assoc2 Query Real Time Medians Comparison
Time (ms)
3,500
3,000
2,500
2,000
1,500
1,000
500
0
100
200
300
400
500
600
700
800
900
1,000
Population
Sql Server
MySql
PostgreSql
GBQ
EF


Inh3Assoc2 GBQ CPU Time per Database
Time (ms)
17
16
15
14
13
12
100
200
300
400
500
600
700
800
900
1000
Population
Sql Server
MySql
PostgreSql

Inh3Assoc2 Query Real Time Medians Comparison

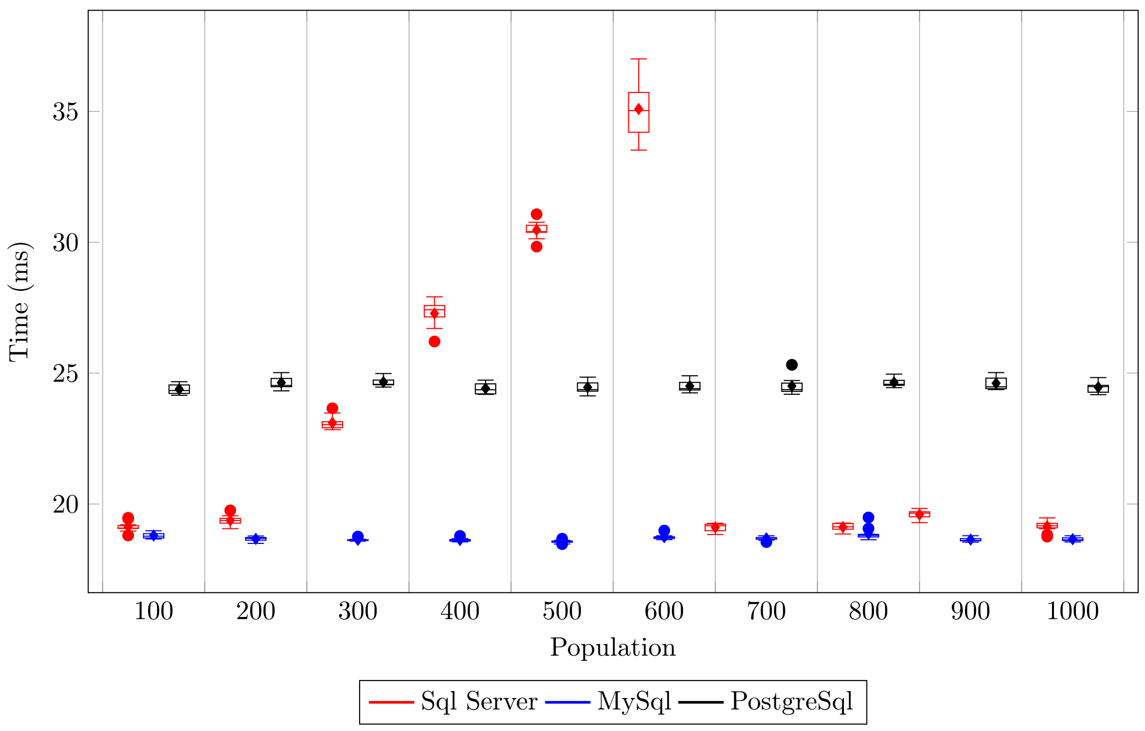


Inh3Assoc2 Query CPU Time Medians Comparison

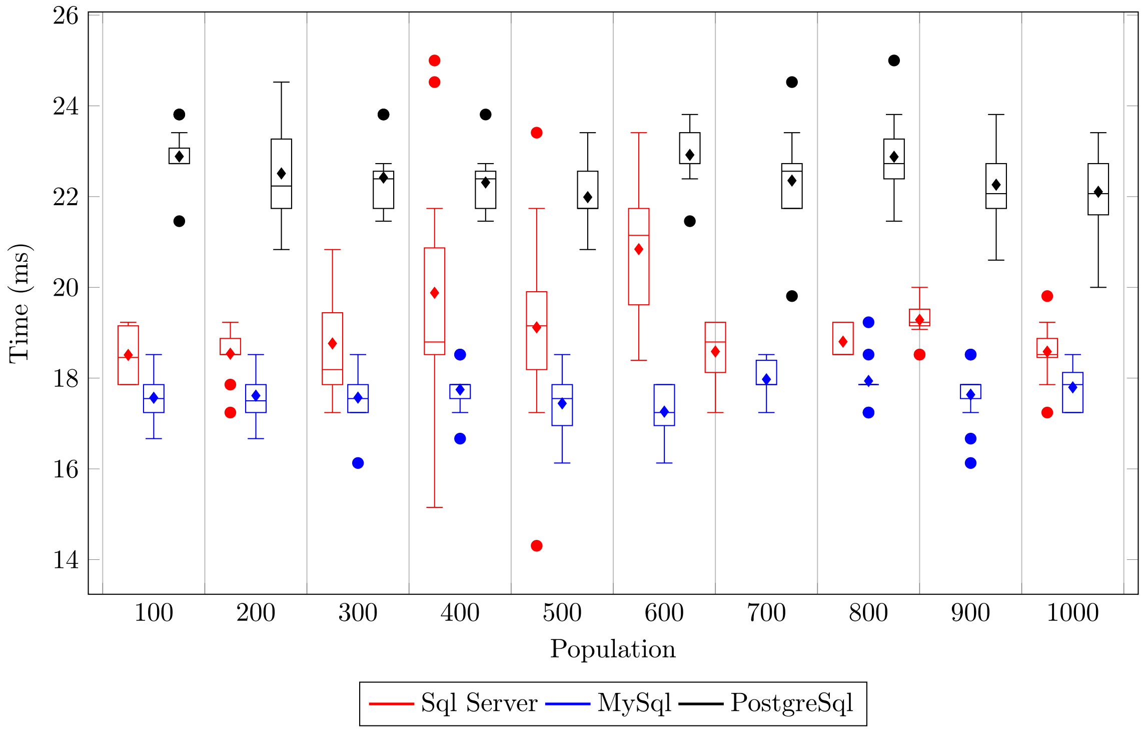


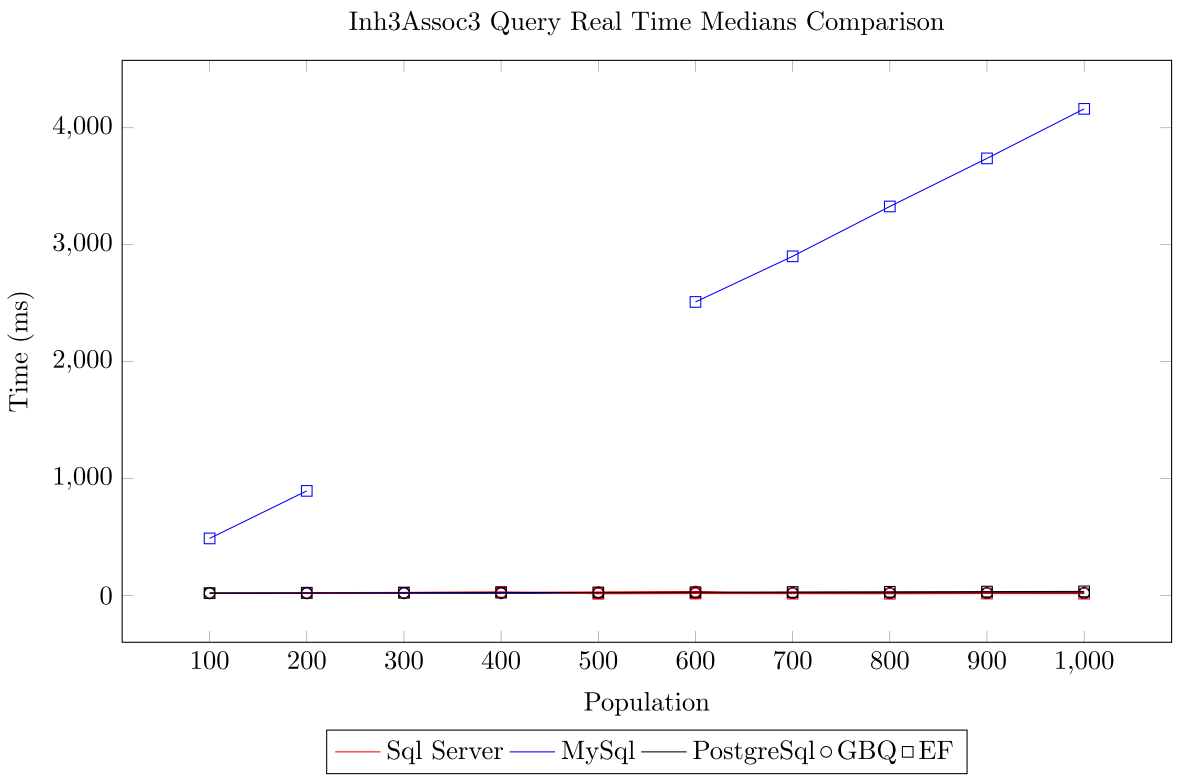
Assoc 3

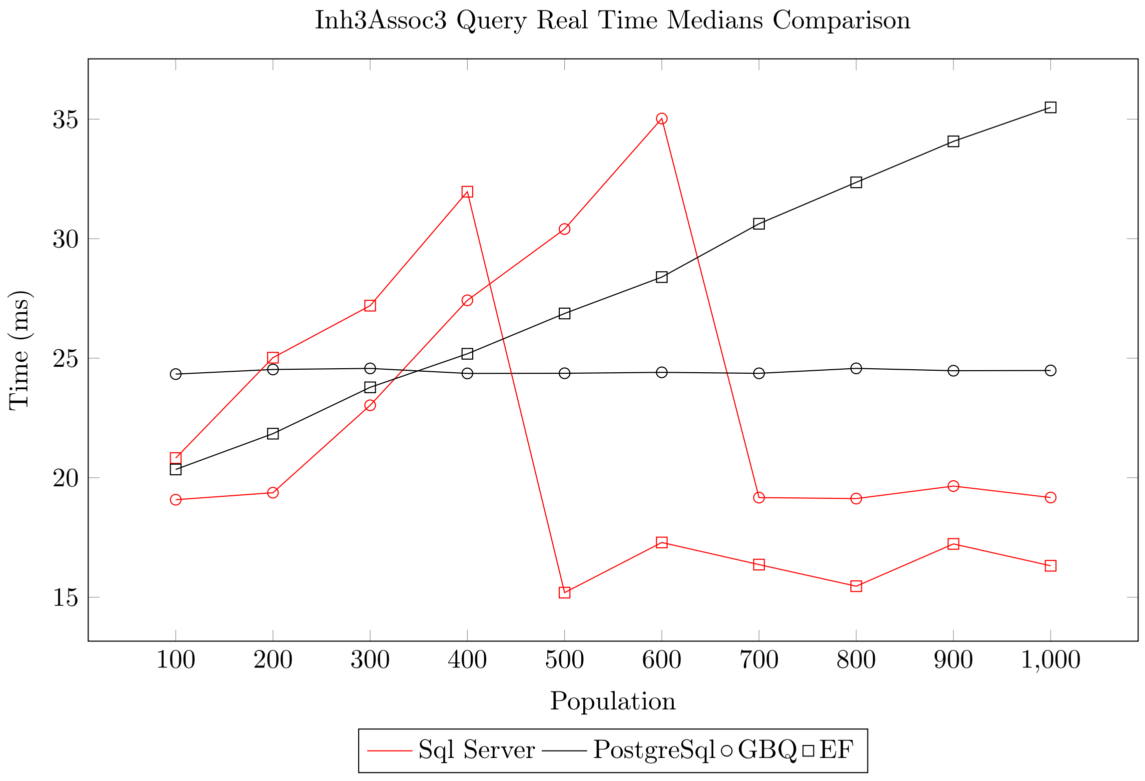
Inh3Assoc3 GBQ Real Time per Database

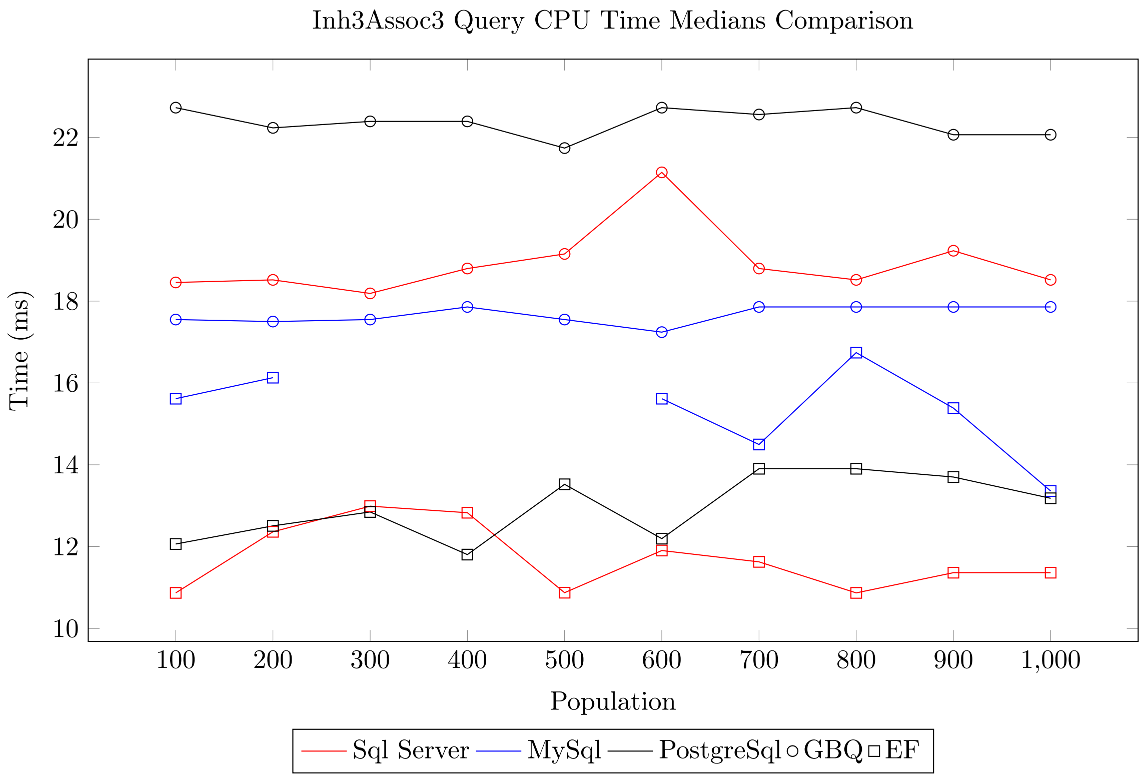


Inh3Assoc3 GBQ CPU Time per Database




Inh3Assoc3 Query Real Time Medians Comparison
Time (ms)
4,000
3,000
2,000
1,000
0
100
200
300
400
500
600
700
800
900
1,000
Population
Sql Server
MySql
PostgreSql
○ GBQ
□ EF

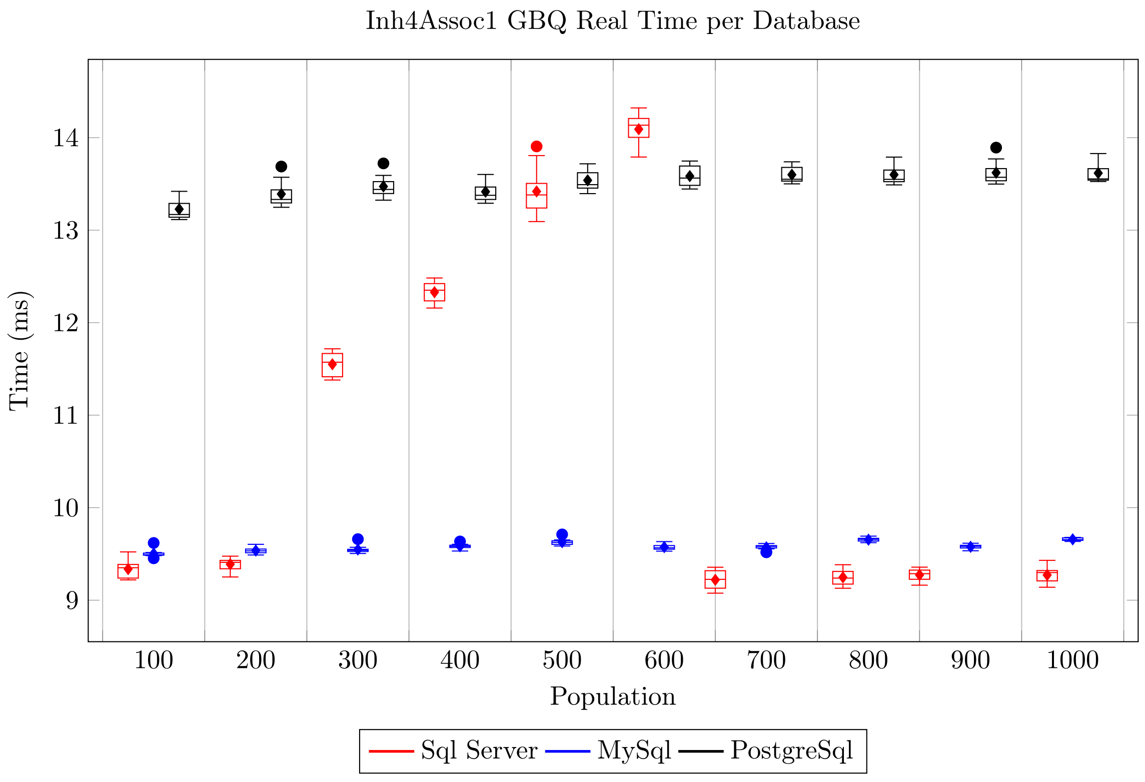

Inh3Assoc3 Query Real Time Medians Comparison
Time (ms)
35
30
25
20
15
100
200
300
400
500
600
700
800
900
1,000
Population
Sql Server
PostgreSql
○ GBQ
□ EF

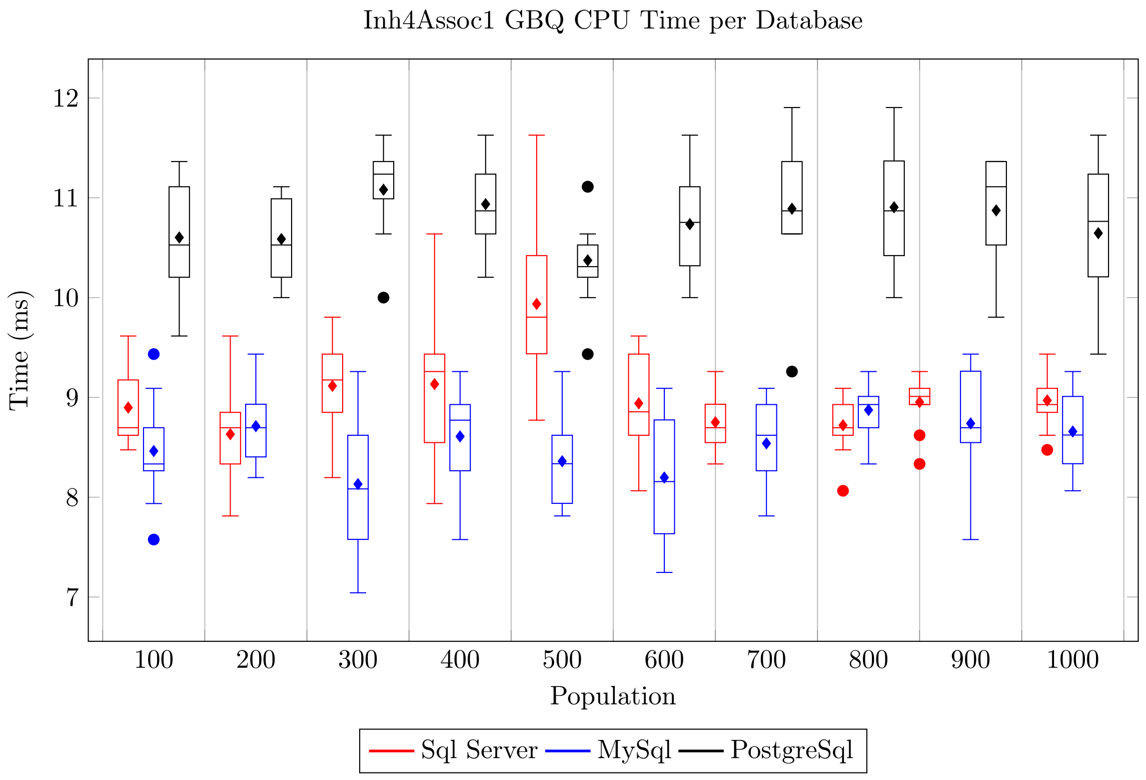


Inh3Assoc3 Query CPU Time Medians Comparison
Time (ms)
22
20
18
16
14
12
10
100
200
300
400
500
600
700
800
900
1,000
Population
Sql Server
MySql
PostgreSql
○ GBQ
□ EF

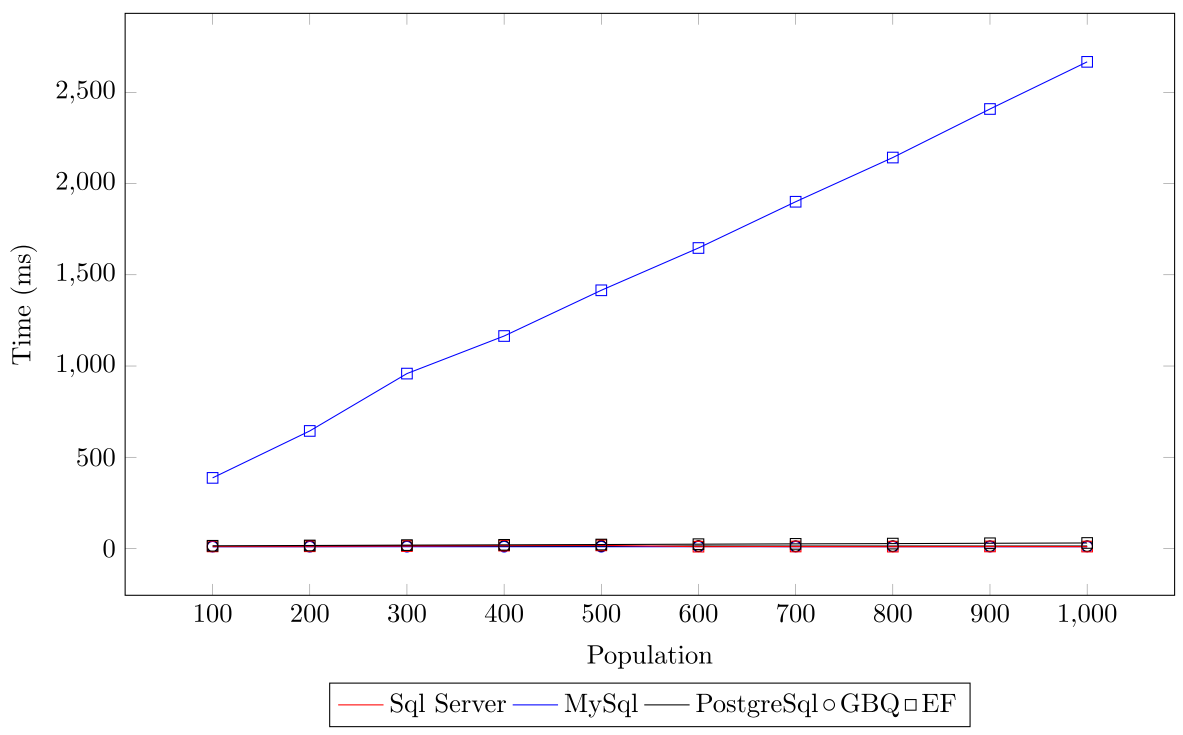
A.2.2 Inheritance 4

Assoc 1

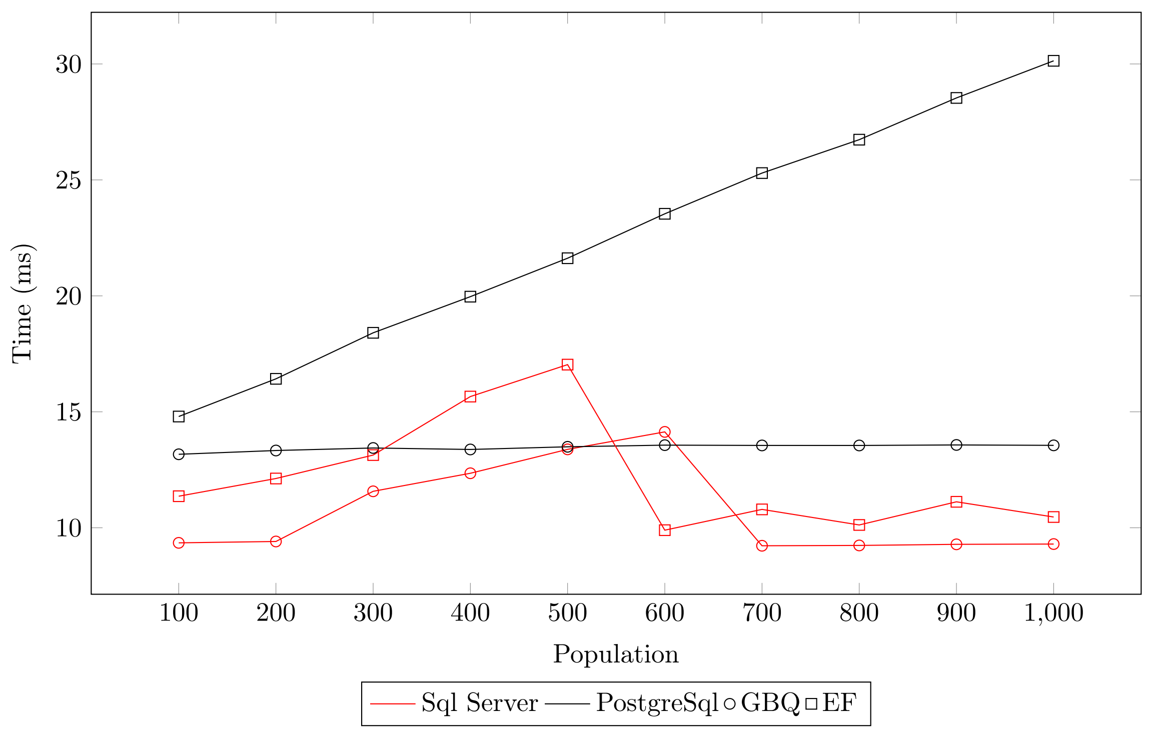




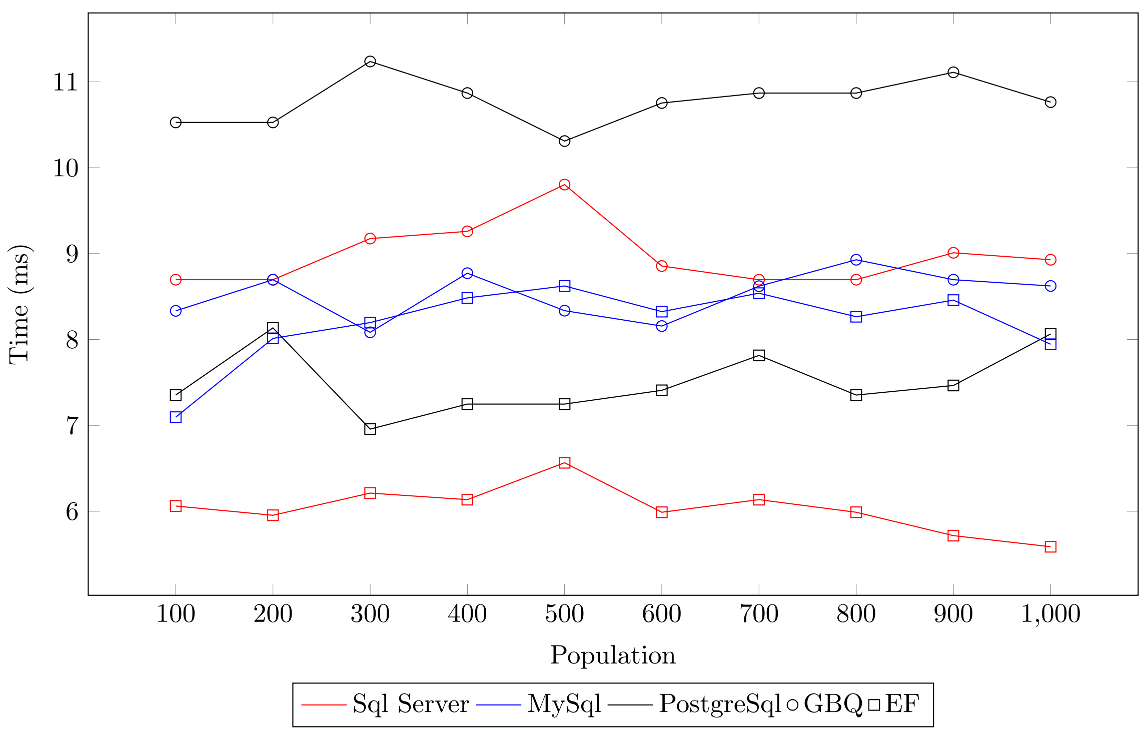
Inh4Assoc1 Query Real Time Medians Comparison



Inh4Assoc1 Query Real Time Medians Comparison

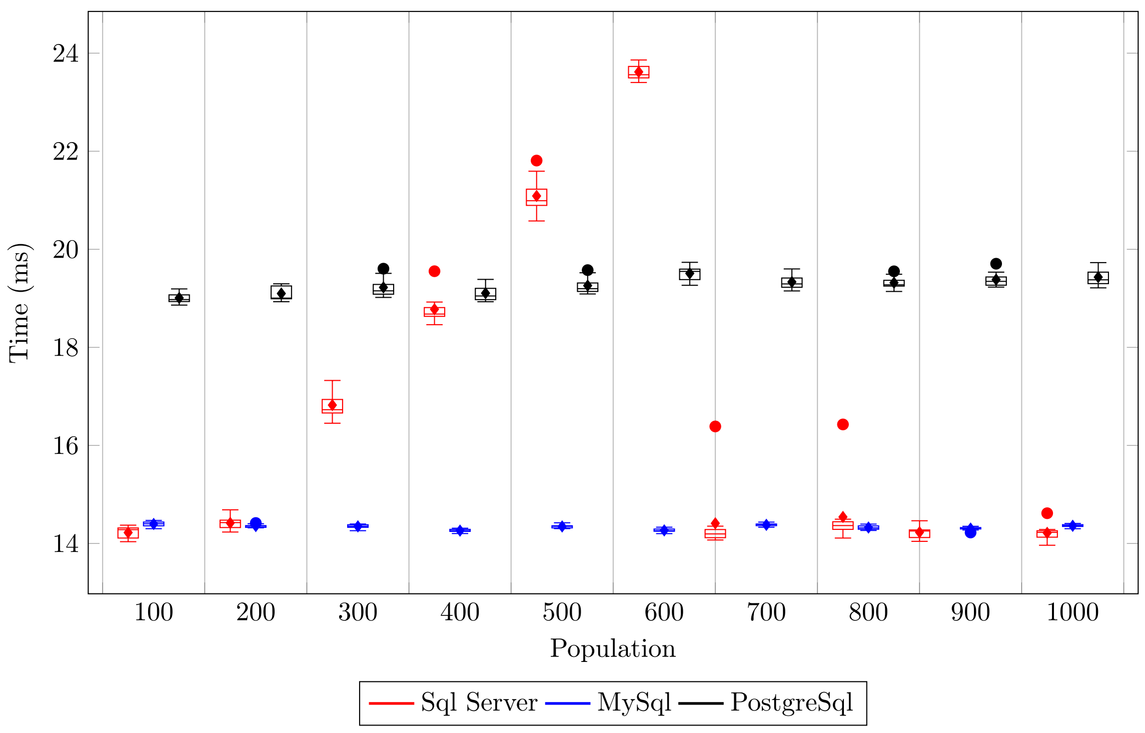


Inh4Assoc1 Query CPU Time Medians Comparison

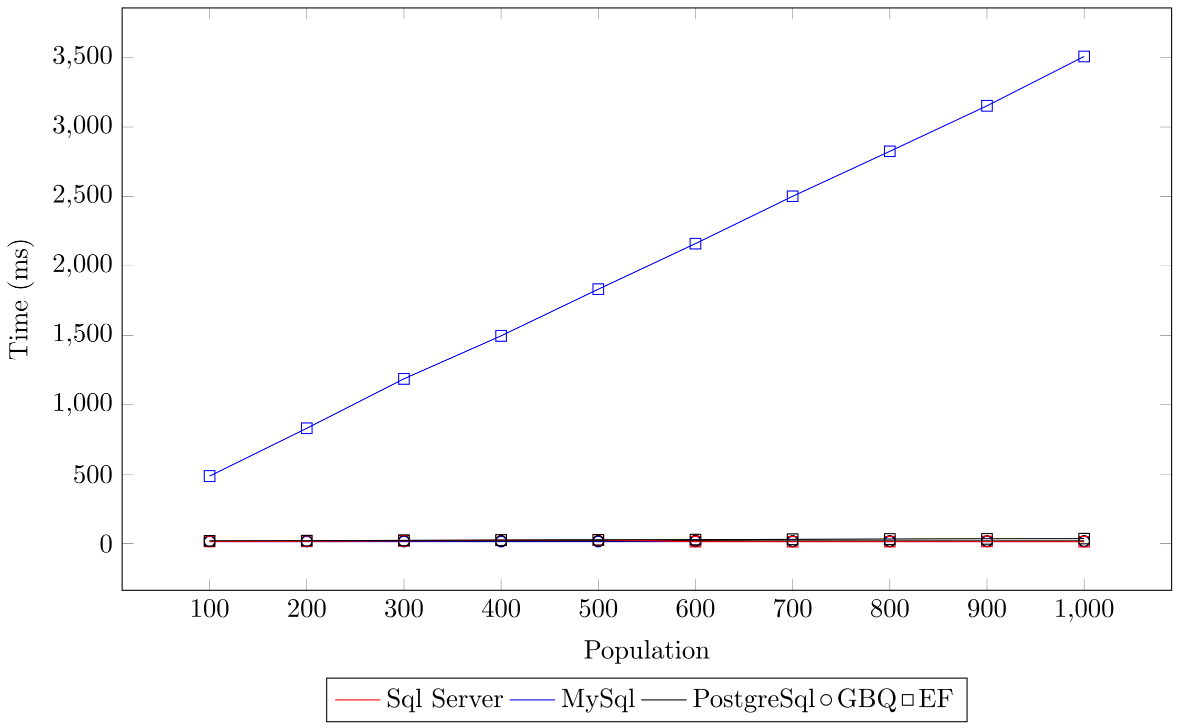


Assoc 2

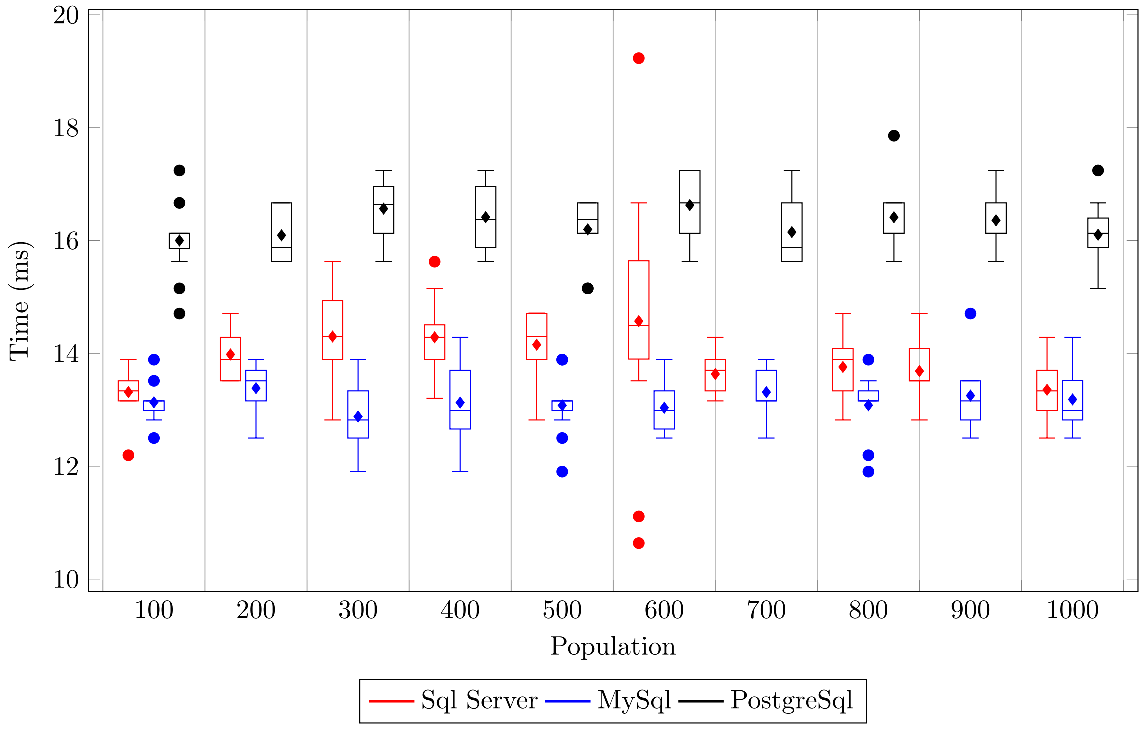
Inh4Assoc2 GBQ Real Time per Database

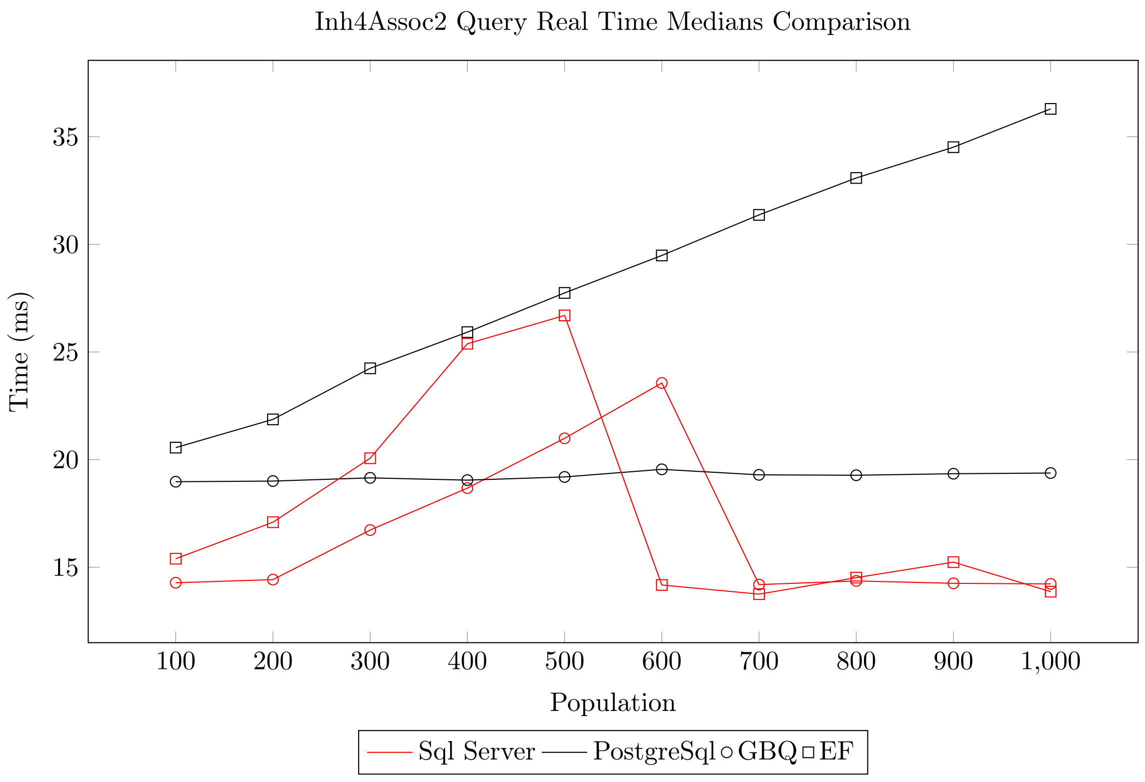


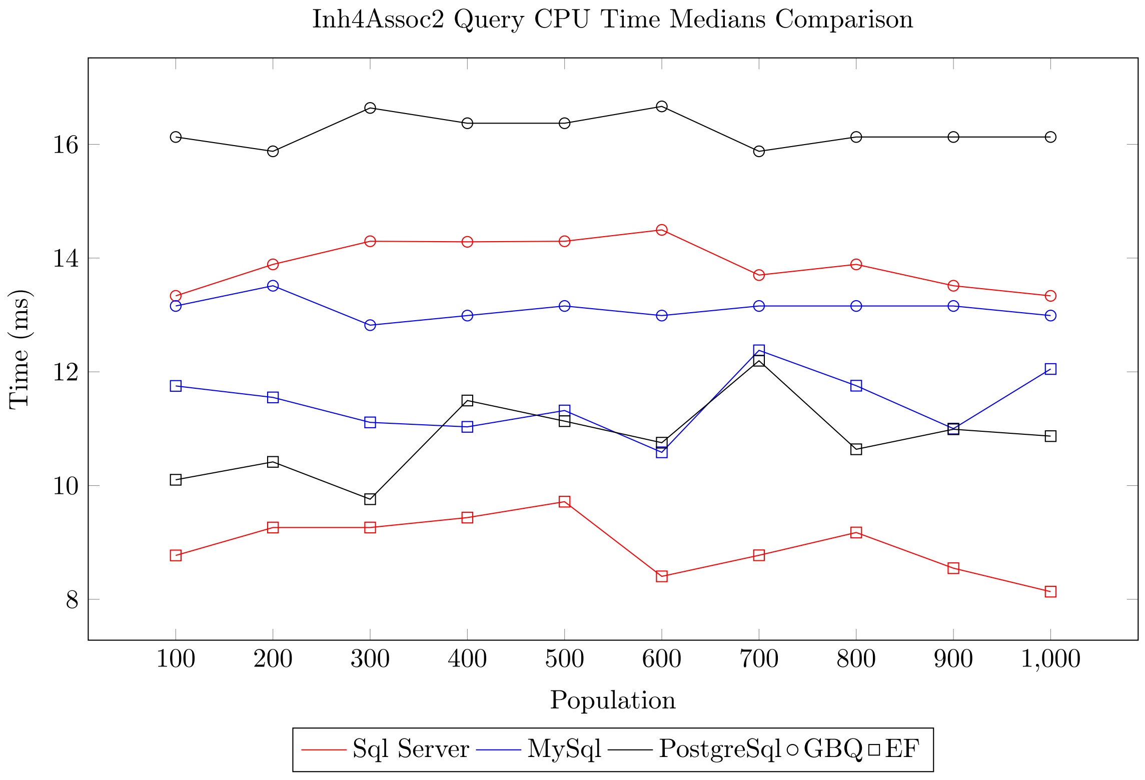
Inh4Assoc2 Query Real Time Medians Comparison



Inh4Assoc2 GBQ CPU Time per Database

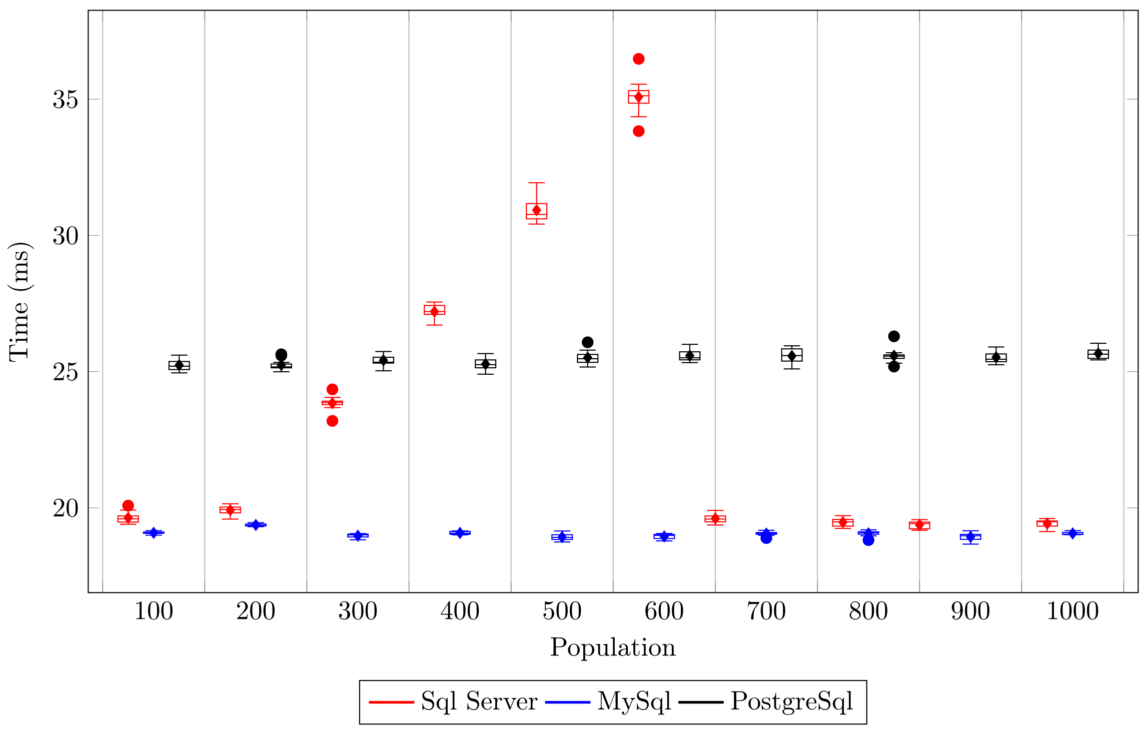



Inh4Assoc2 Query Real Time Medians Comparison
Time (ms)
35
30
25
20
15
100
200
300
400
500
600
700
800
900
1,000
Population
Sql Server
PostgreSql
GBQ
EF

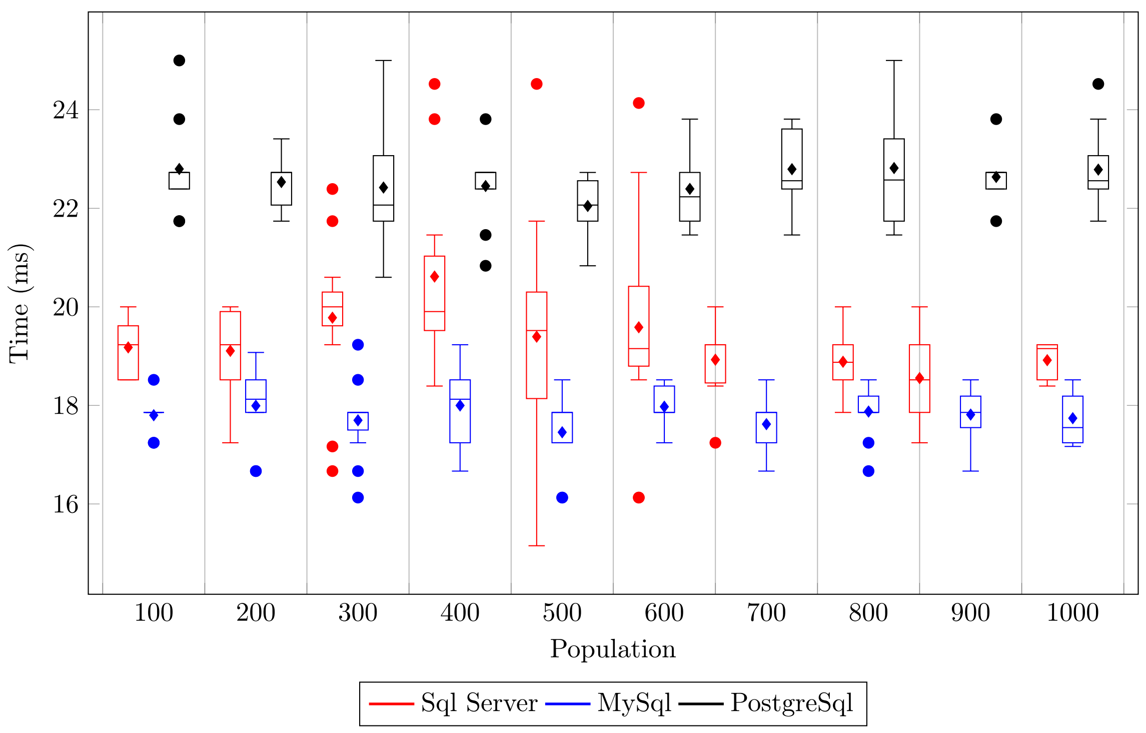

Inh4Assoc2 Query CPU Time Medians Comparison
Time (ms)
16
14
12
10
8
100
200
300
400
500
600
700
800
900
1,000
Population
Sql Server
MySql
PostgreSql
GBQ
EF

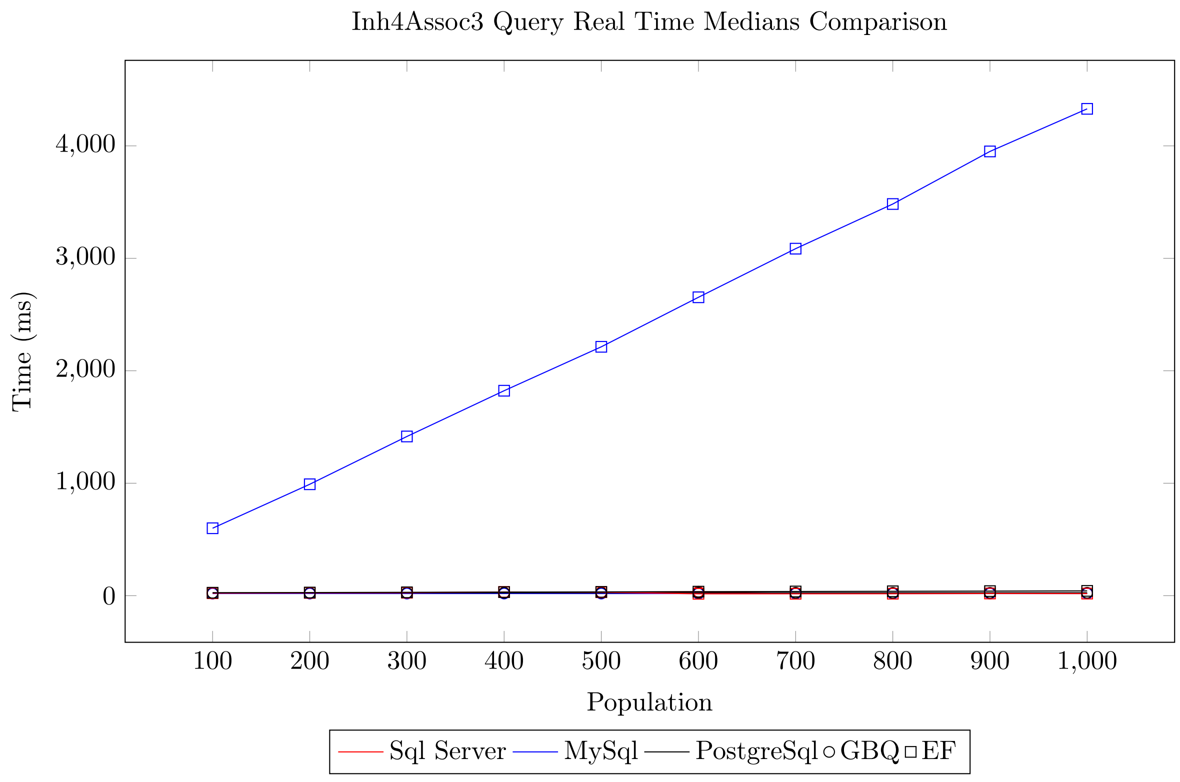
Assoc 3

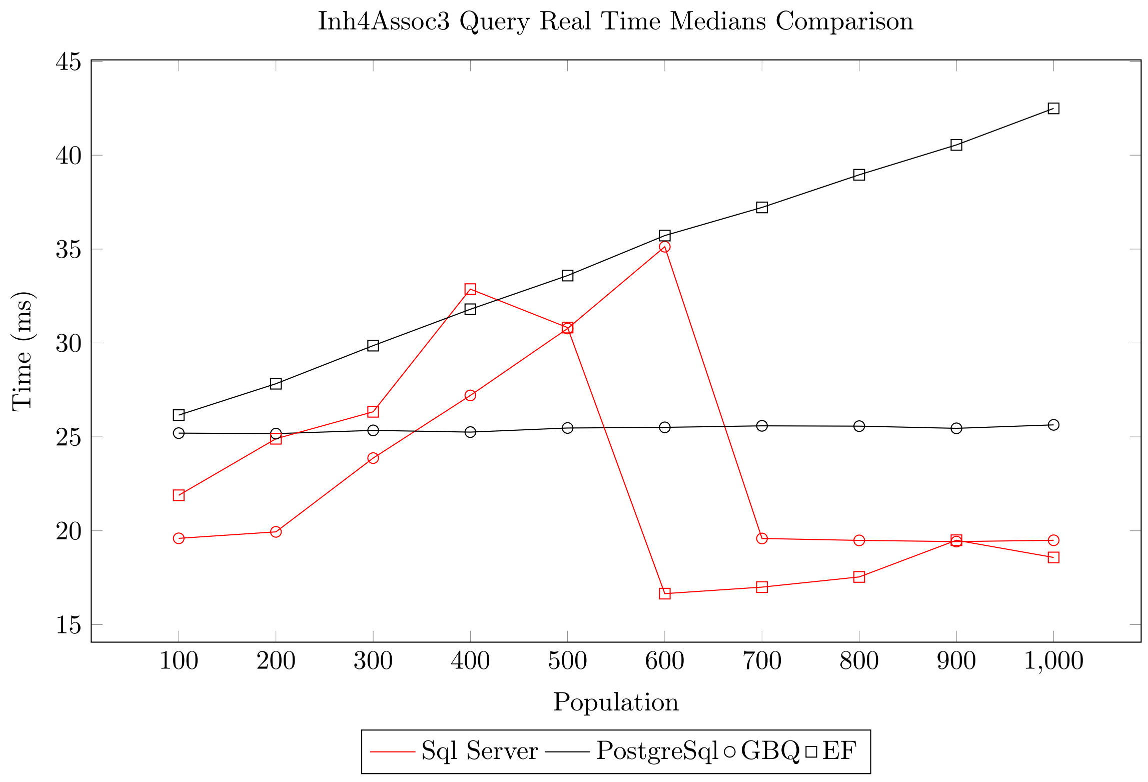
Inh4Assoc3 GBQ Real Time per Database



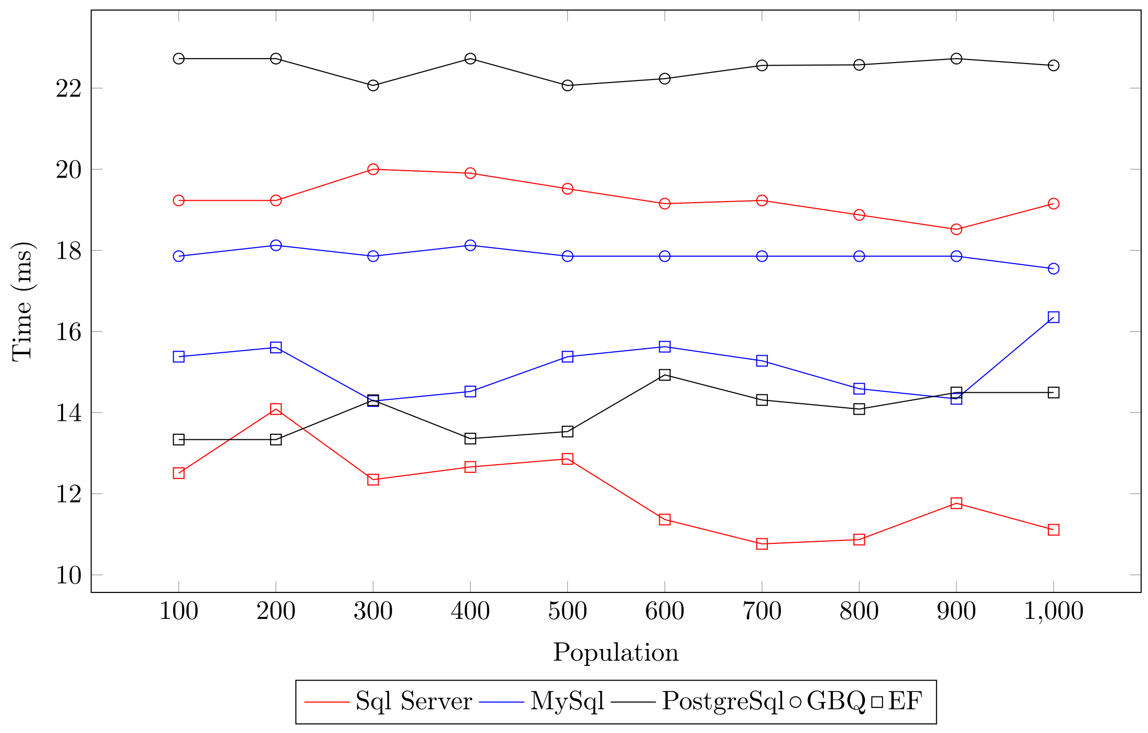
Inh4Assoc3 GBQ CPU Time per Database




Inh4Assoc3 Query Real Time Medians Comparison
Time (ms)
4,000
3,000
2,000
1,000
0
100
200
300
400
500
600
700
800
900
1,000
Population
Sql Server
MySql
PostgreSql
○ GBQ
□ EF

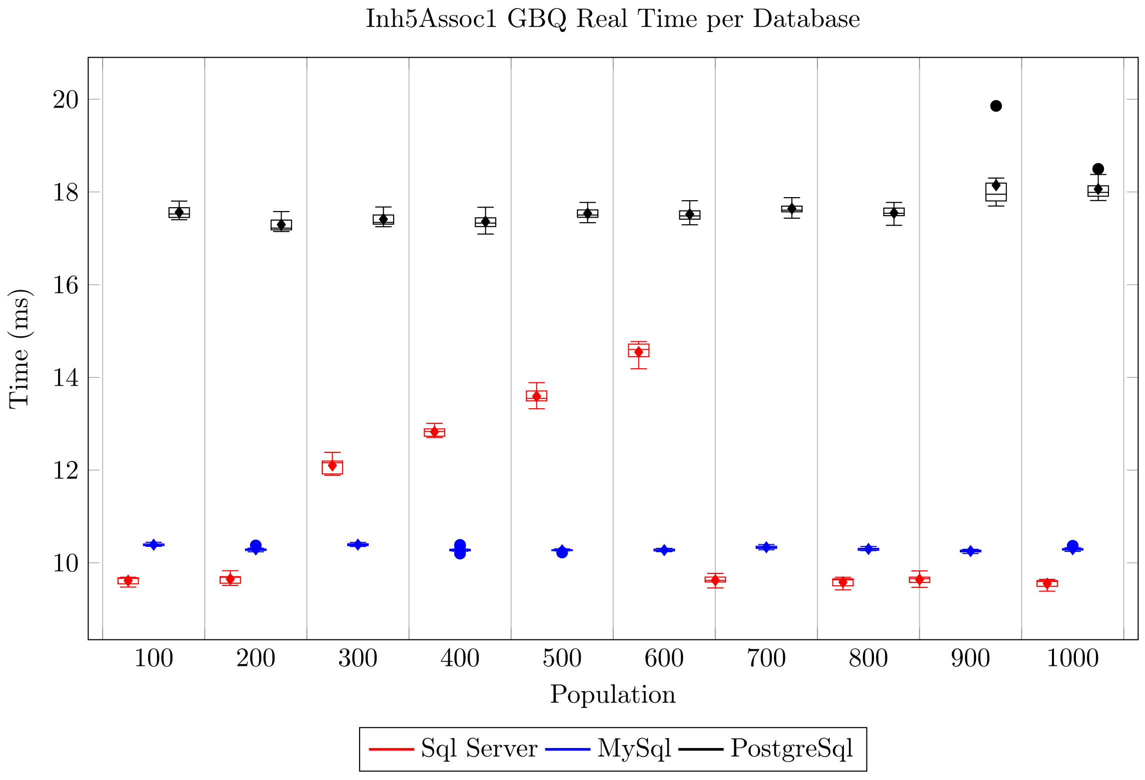

Inh4Assoc3 Query Real Time Medians Comparison
Time (ms)
45
40
35
30
25
20
15
100
200
300
400
500
600
700
800
900
1,000
Population
Sql Server
PostgreSql
○ GBQ
□ EF

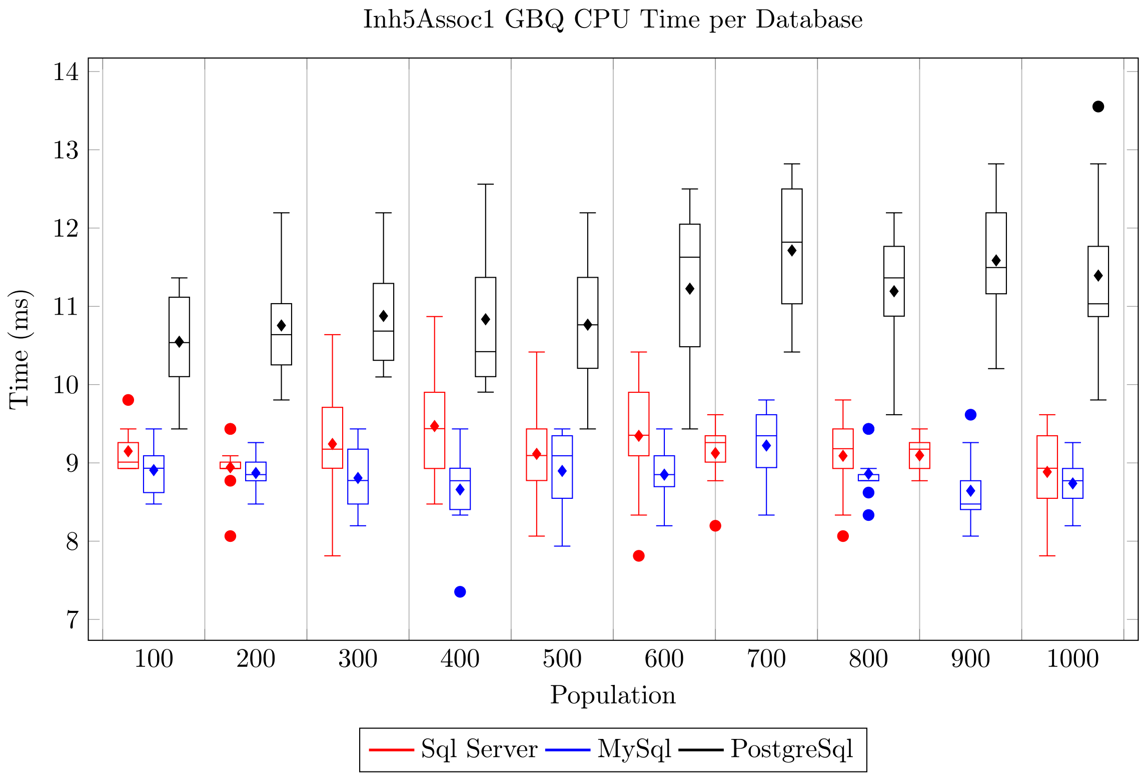
Inh4Assoc3 Query CPU Time Medians Comparison



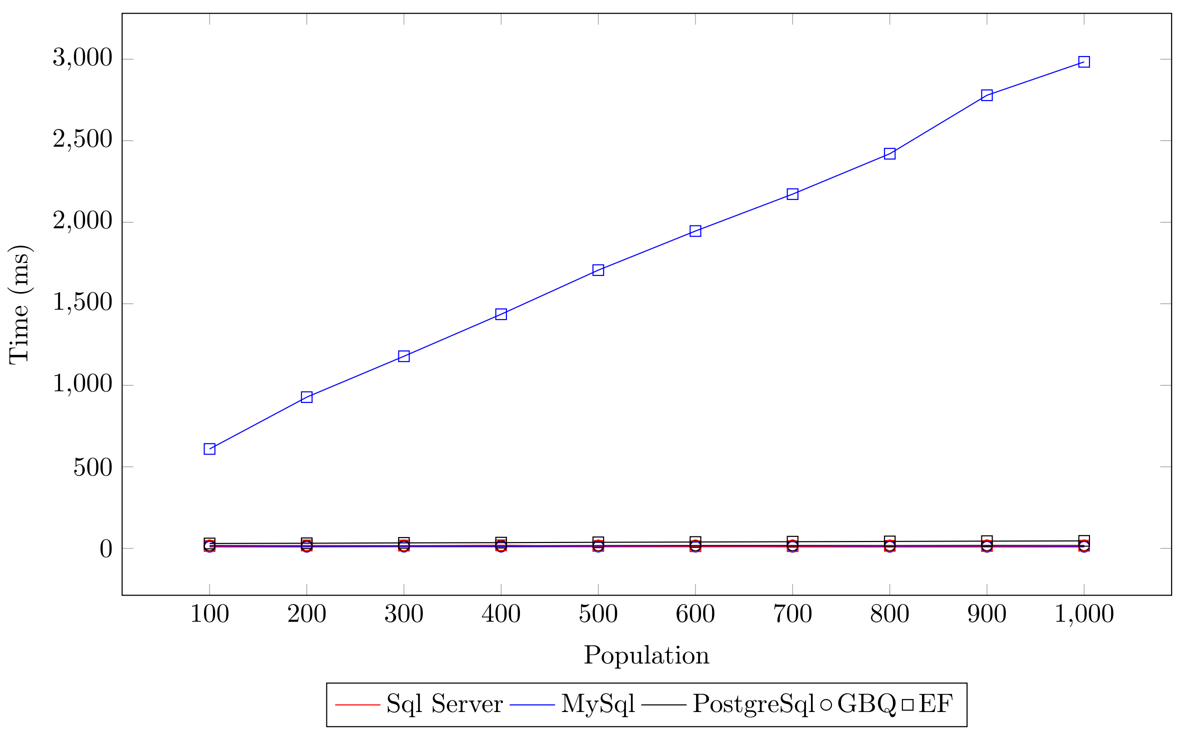
A.2.3 Inheritance 5

Assoc 1

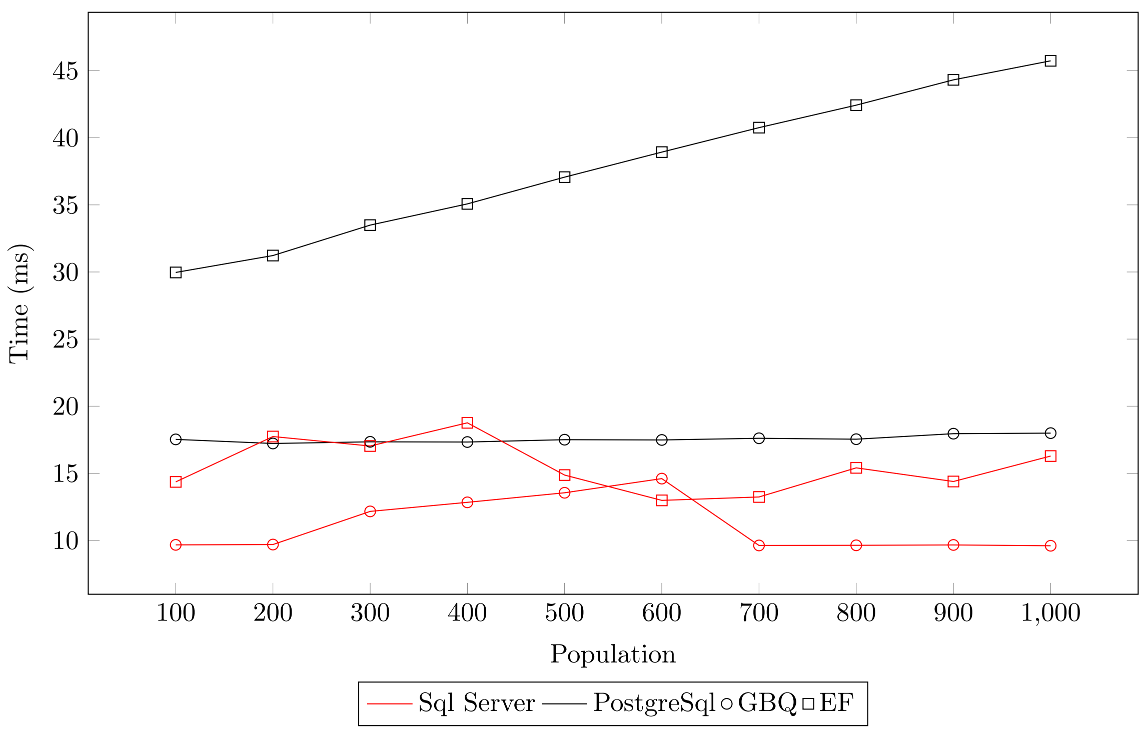




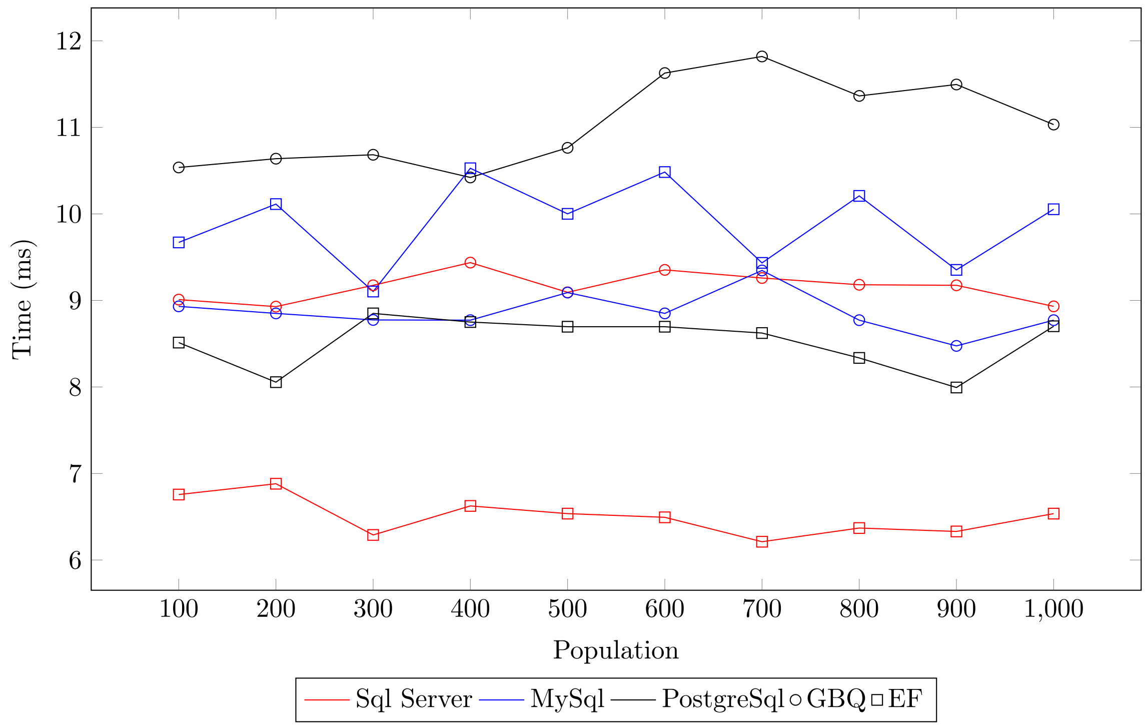
Inh5Assoc1 Query Real Time Medians Comparison



Inh5Assoc1 Query Real Time Medians Comparison

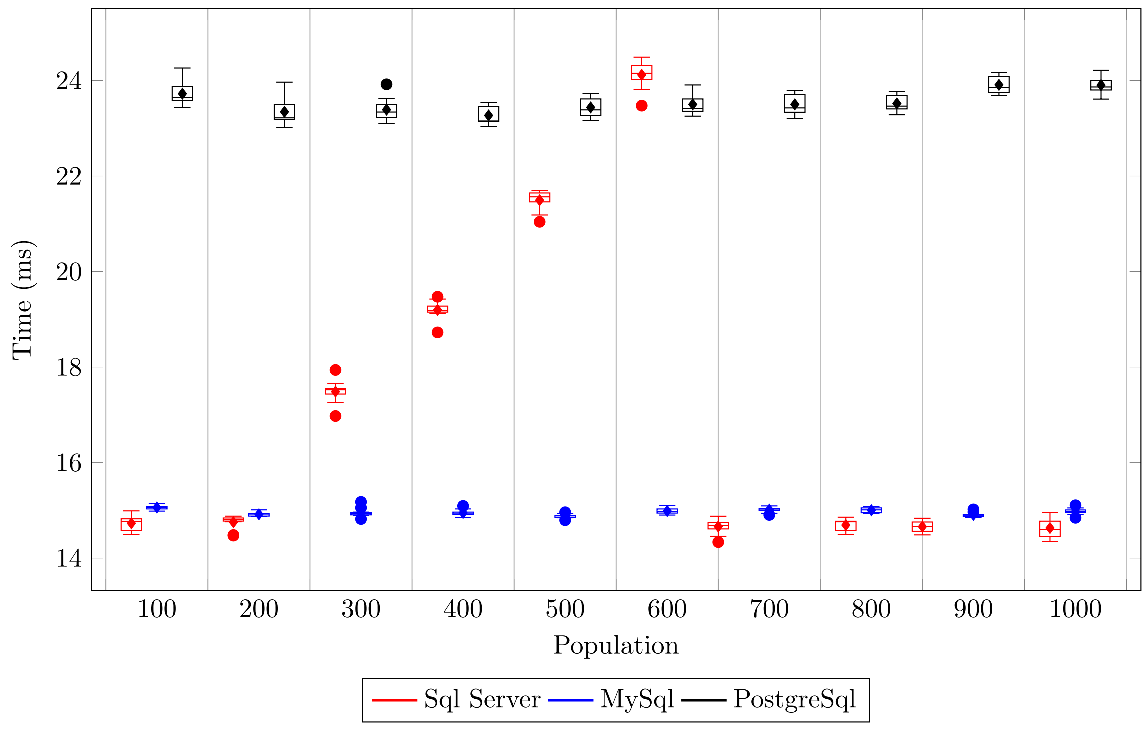


Inh5Assoc1 Query CPU Time Medians Comparison

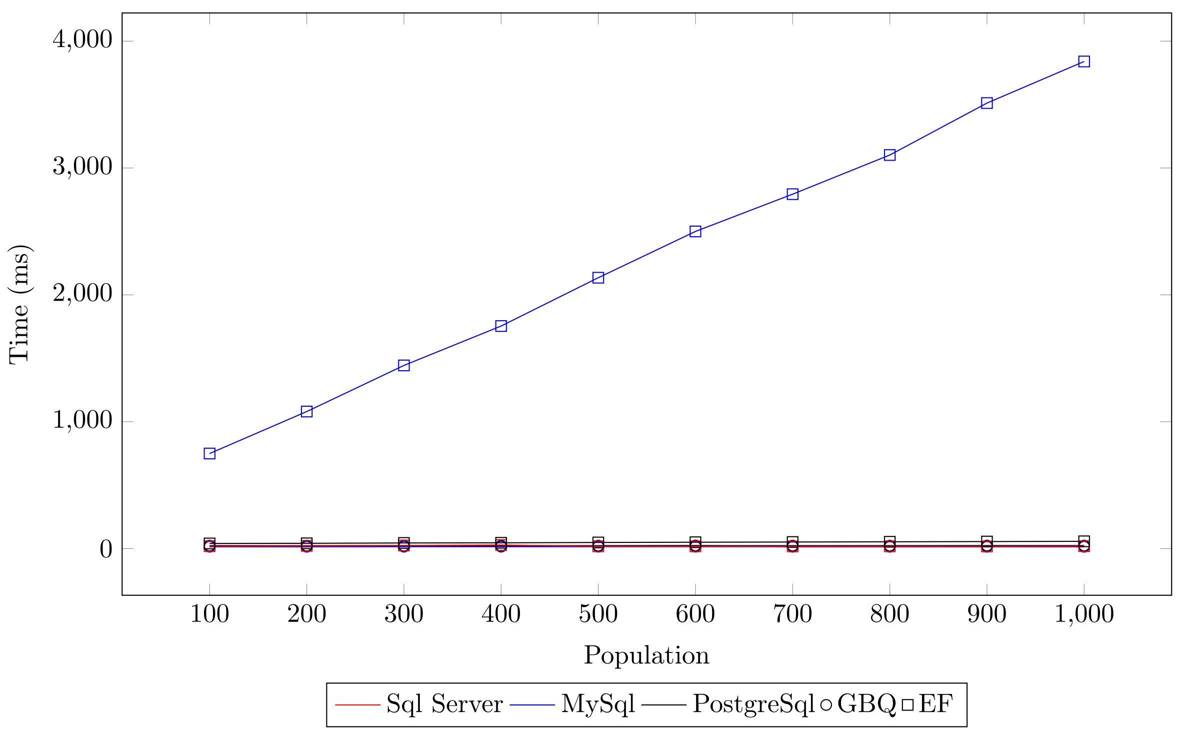


Assoc 2

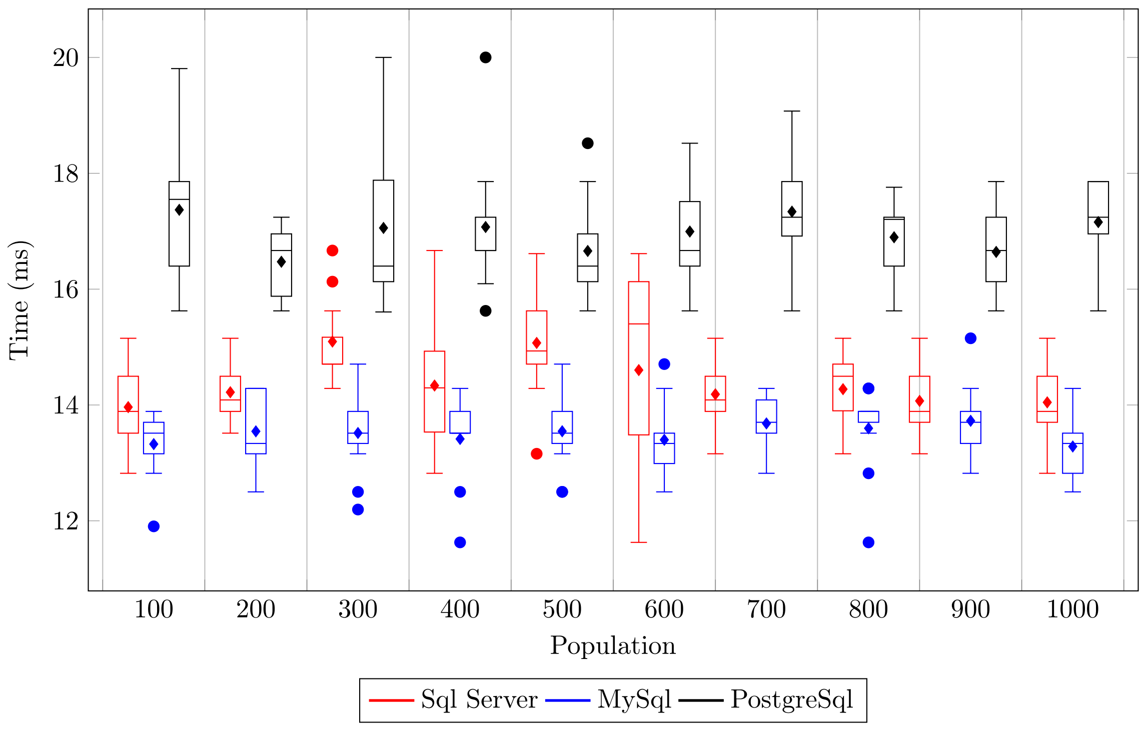
Inh5Assoc2 GBQ Real Time per Database



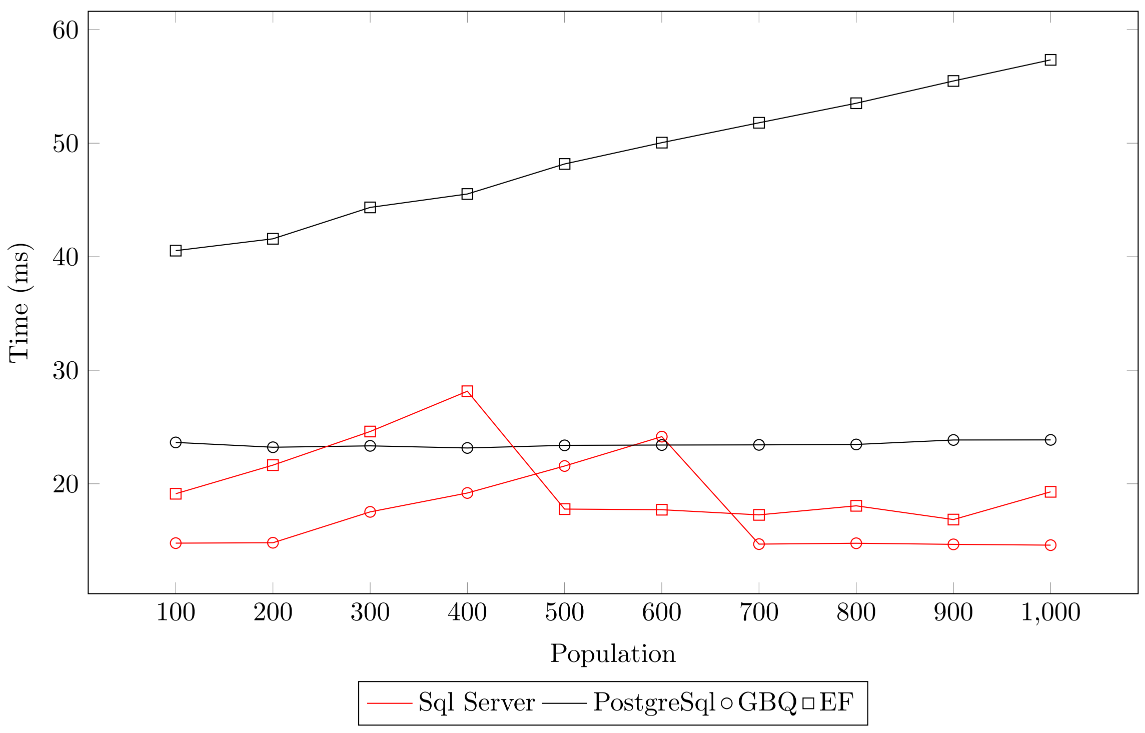
Inh5Assoc2 Query Real Time Medians Comparison



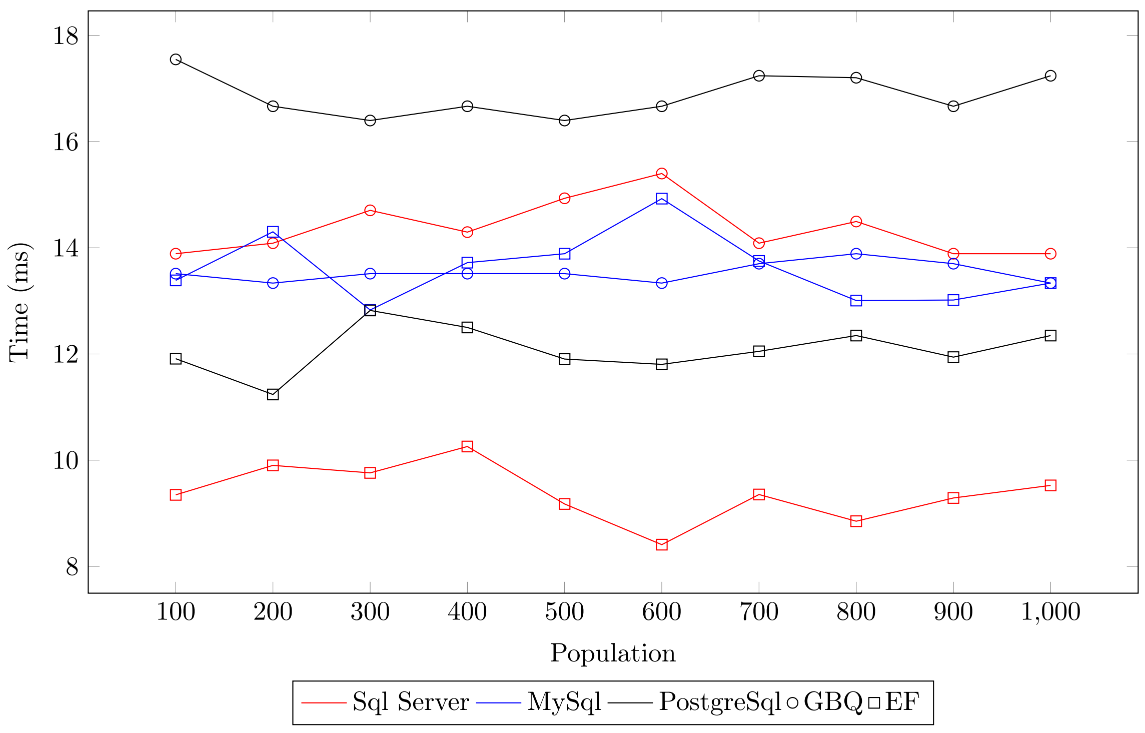
Inh5Assoc2 GBQ CPU Time per Database



Inh5Assoc2 Query Real Time Medians Comparison

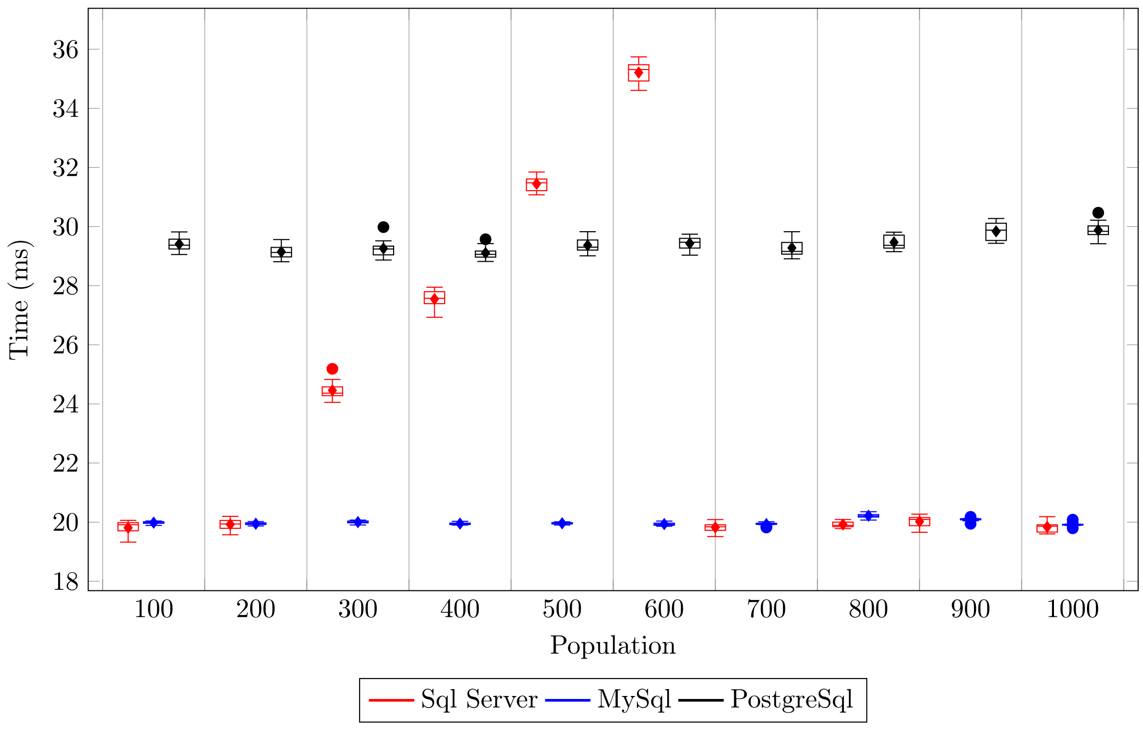


Inh5Assoc2 Query CPU Time Medians Comparison

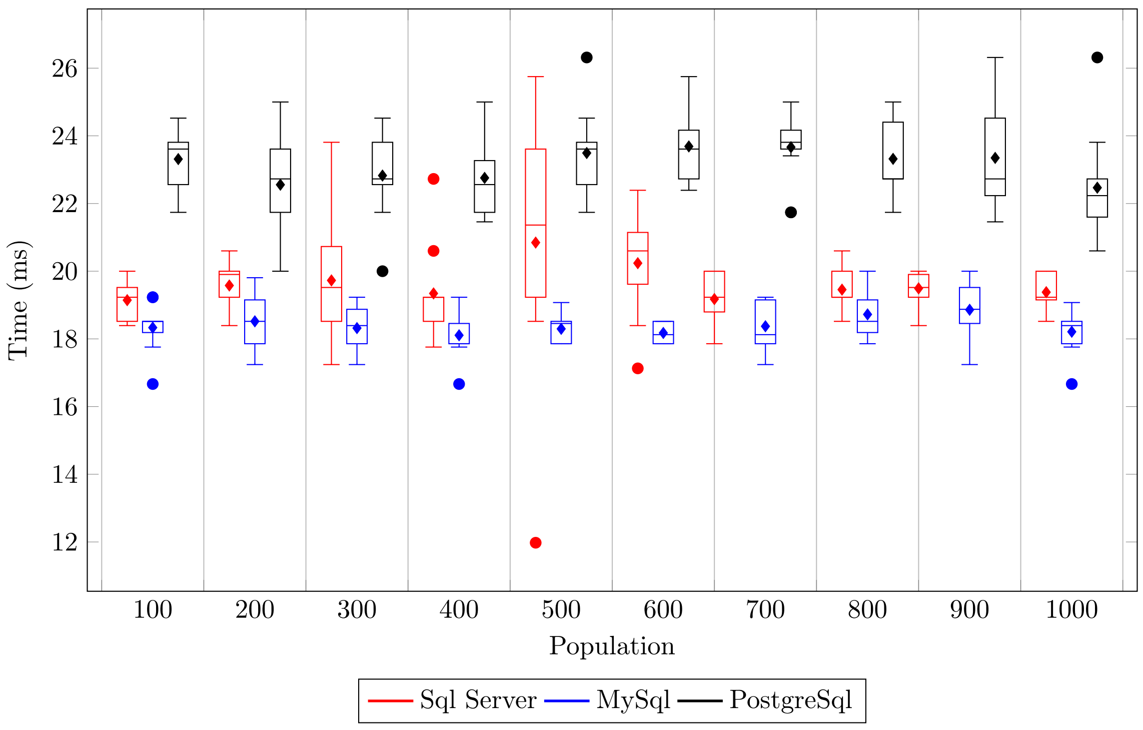


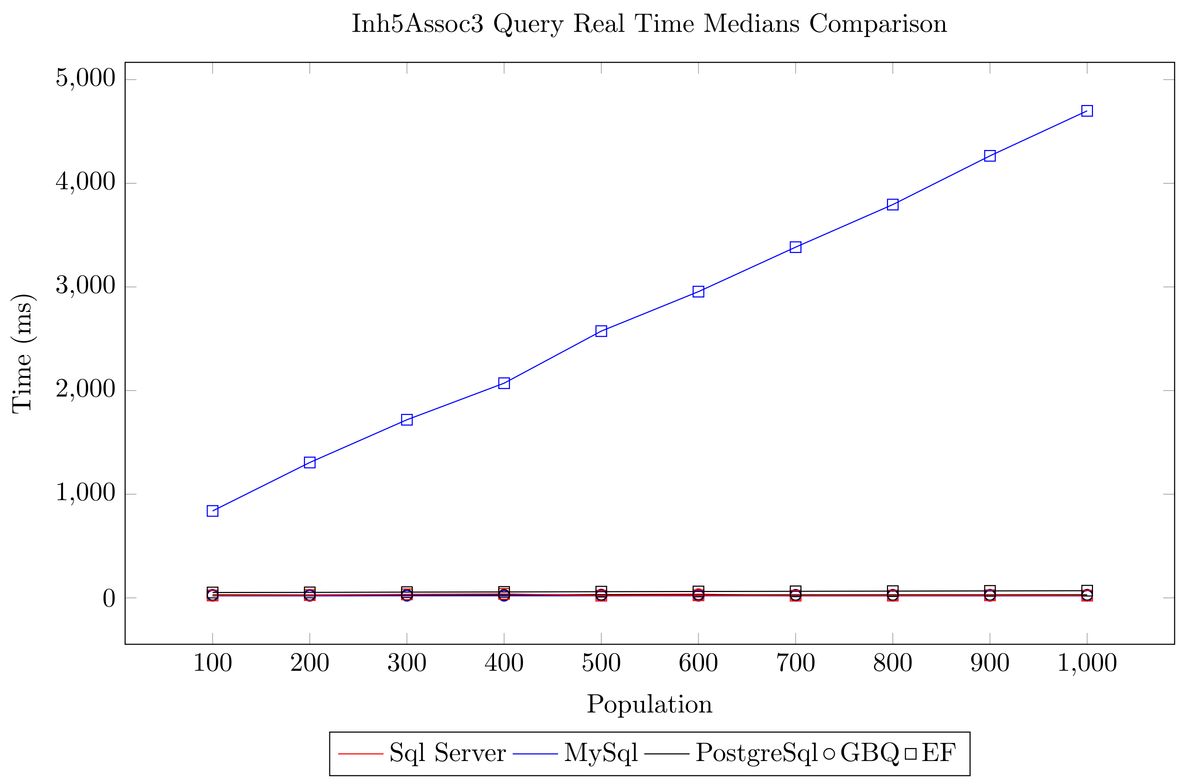
Assoc 3

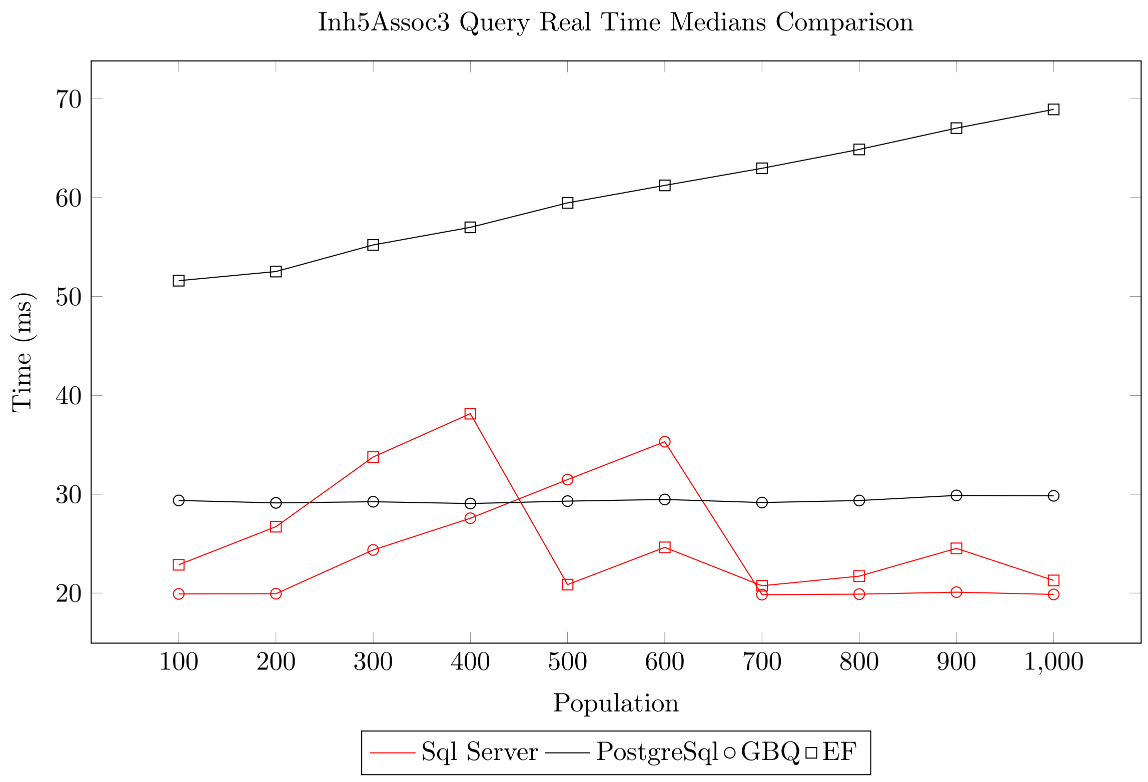
Inh5Assoc3 GBQ Real Time per Database

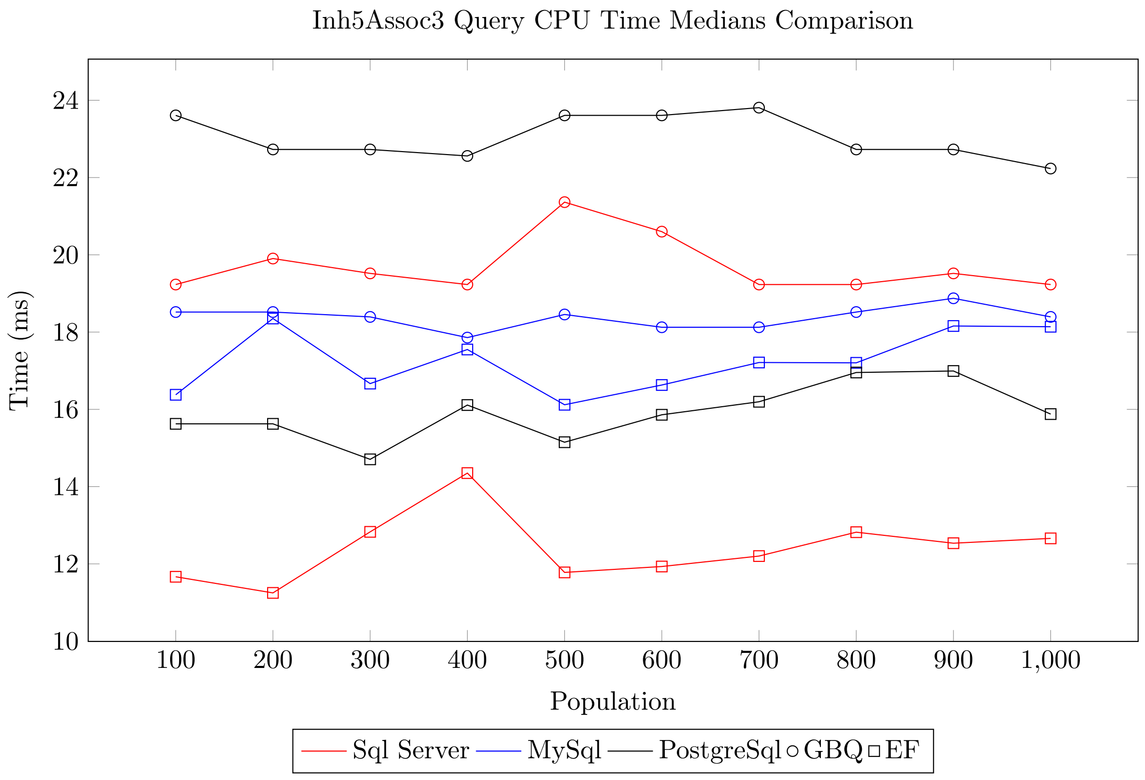


Inh5Assoc3 GBQ CPU Time per Database



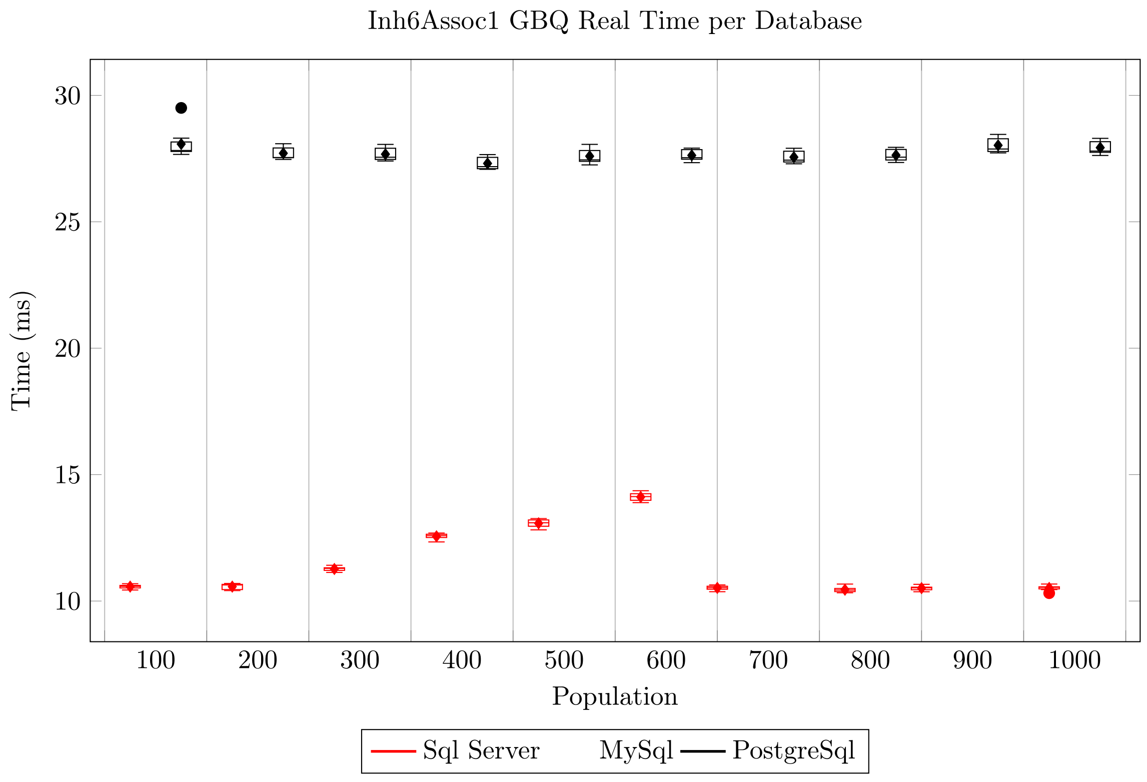

Inh5Assoc3 Query Real Time Medians Comparison
Time (ms)
5,000
4,000
3,000
2,000
1,000
0
100
200
300
400
500
600
700
800
900
1,000
Population
Sql Server
MySql
PostgreSql
GBQ
EF

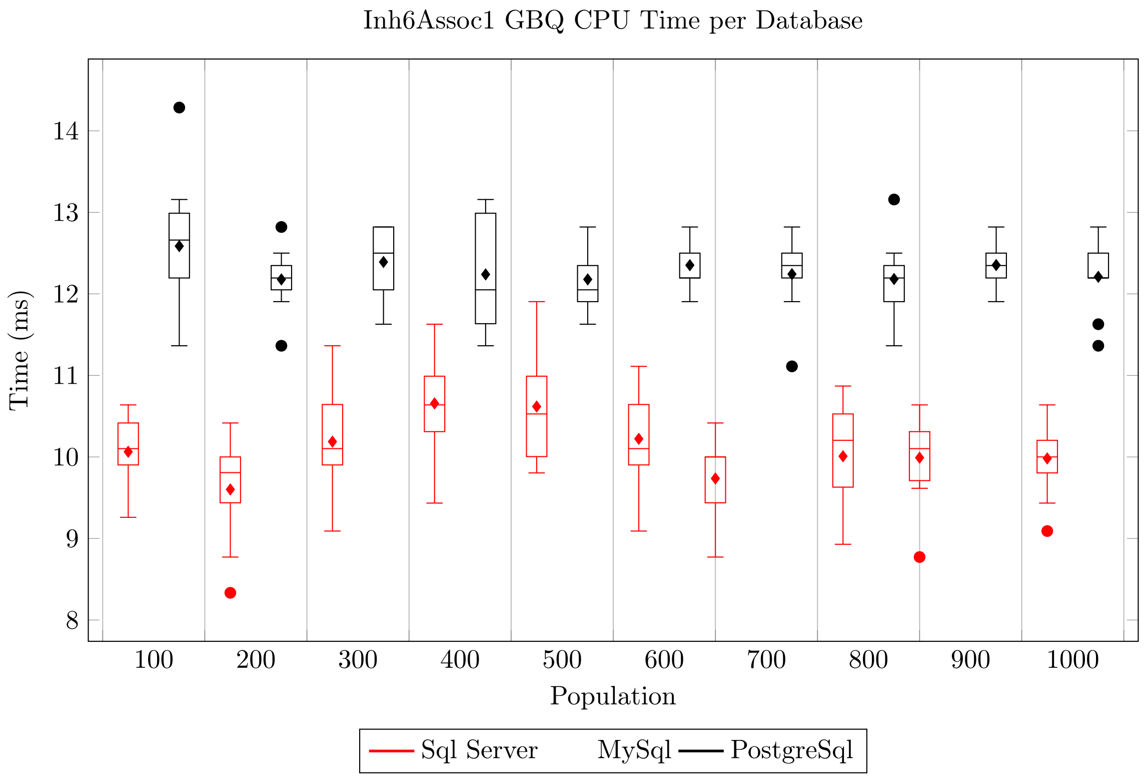

Inh5Assoc3 Query Real Time Medians Comparison
Time (ms)
70
60
50
40
30
20
100
200
300
400
500
600
700
800
900
1,000
Population
Sql Server
PostgreSql
GBQ
EF

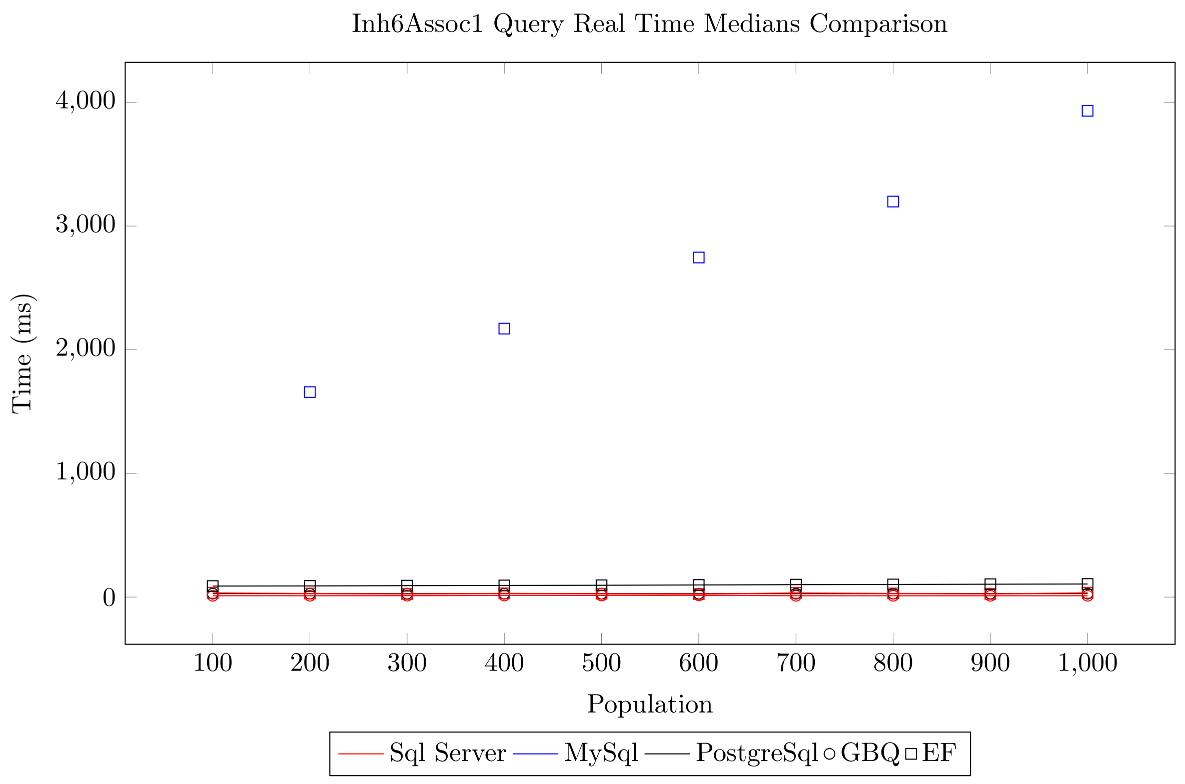

Inh5Assoc3 Query CPU Time Medians Comparison
Time (ms)
10
12
14
16
18
20
22
24
100
200
300
400
500
600
700
800
900
1,000
Population
Sql Server
MySql
PostgreSql
○ GBQ
□ EF

A.2.4 Inheritance 6

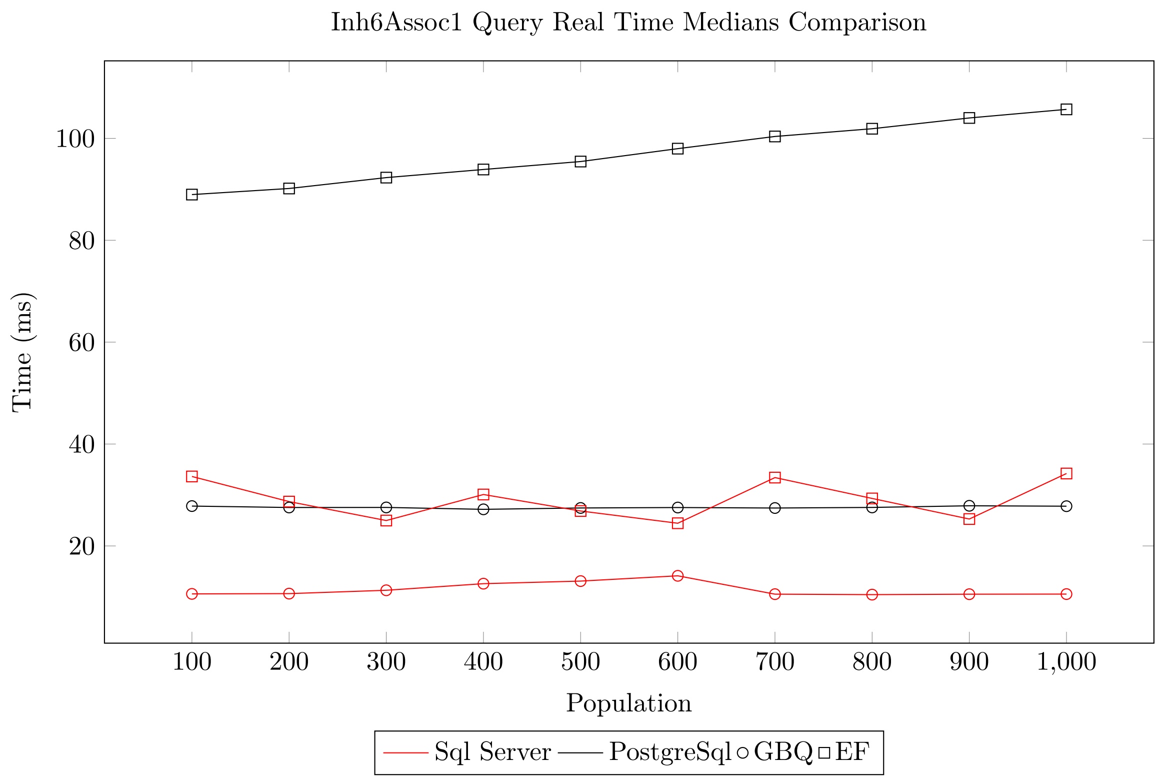
Assoc 1





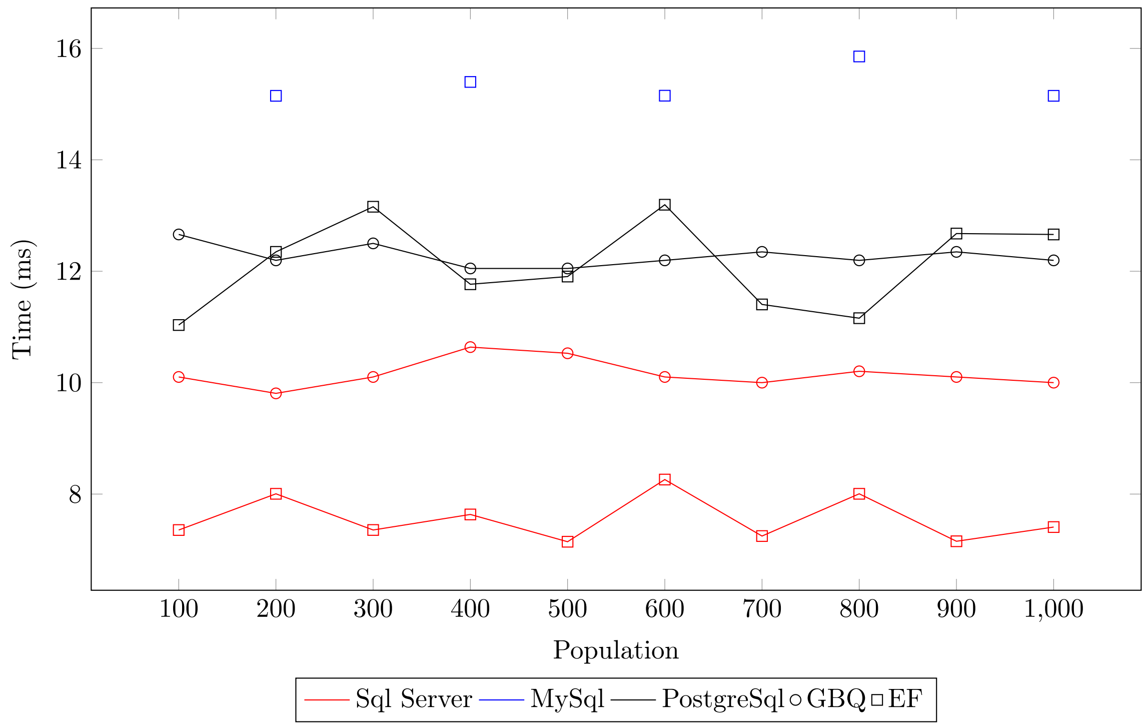


Inh6Assoc1 Query Real Time Medians Comparison
Time (ms)
4,000
3,000
2,000
1,000
0
100
200
300
400
500
600
700
800
900
1,000
Population
Sql Server
MySql
PostgreSql
○ GBQ
□ EF



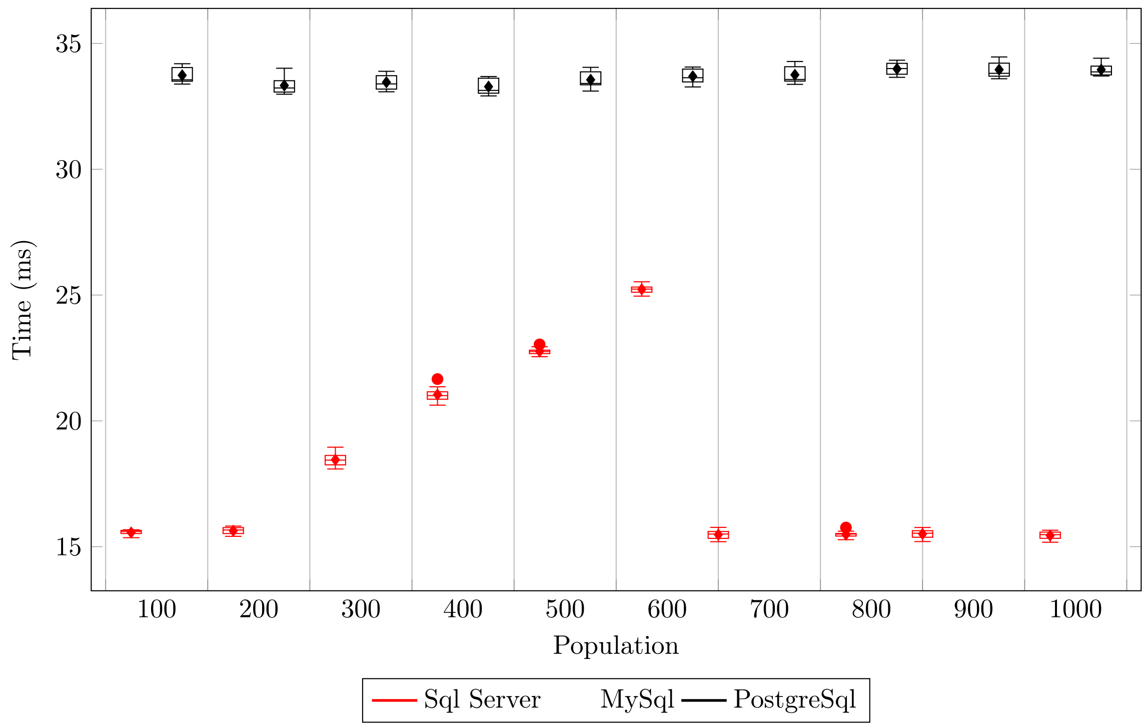
Inh6Assoc1 Query Real Time Medians Comparison
Time (ms)
100
80
60
40
20
100
200
300
400
500
600
700
800
900
1,000
Population
Sql Server
PostgreSql
○ GBQ
□ EF

Inh6Assoc1 Query CPU Time Medians Comparison

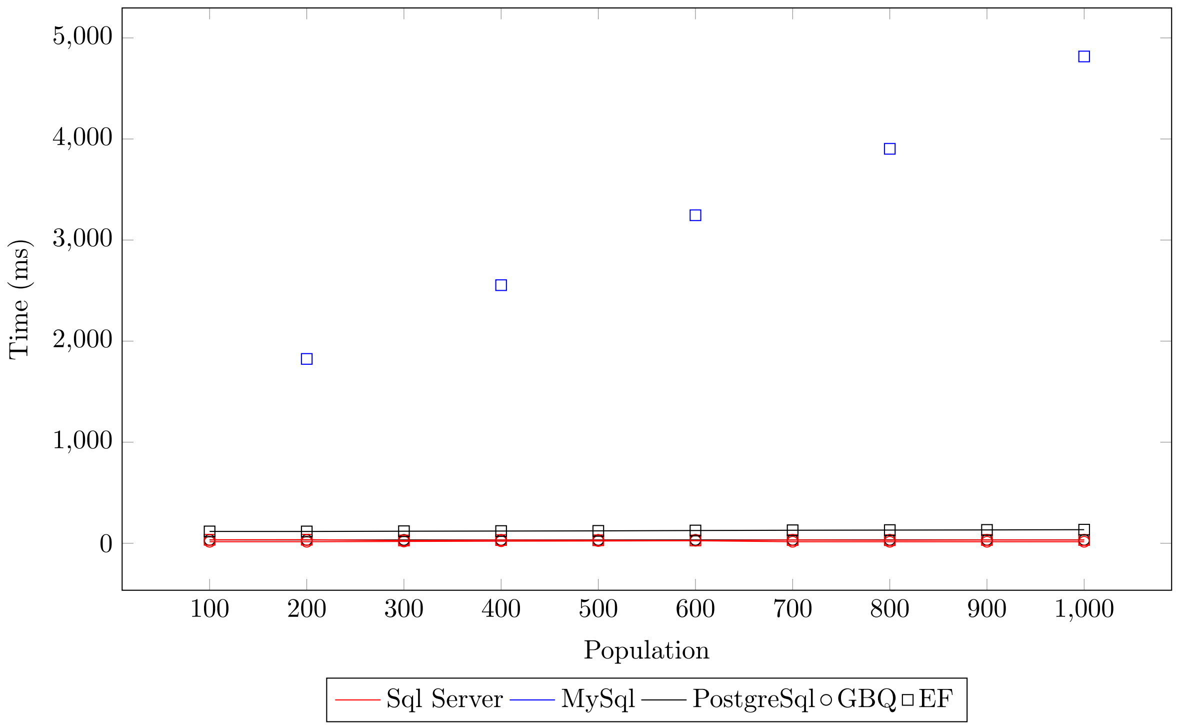


Assoc 2

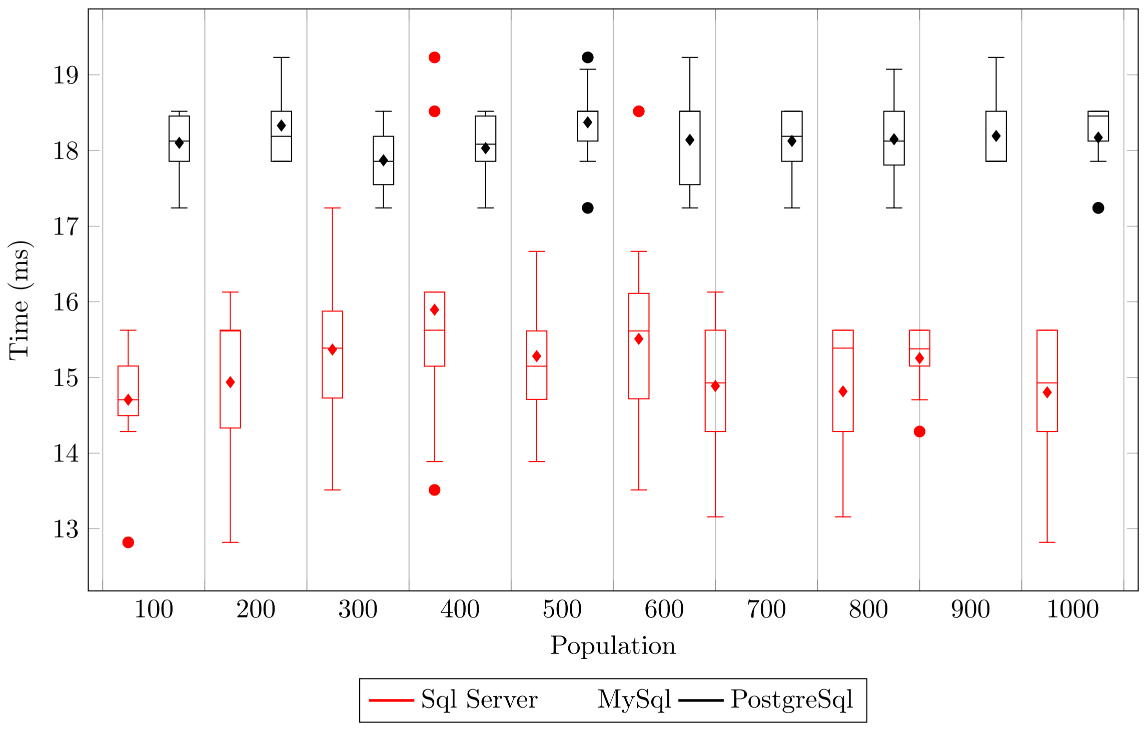
Inh6Assoc2 GBQ Real Time per Database



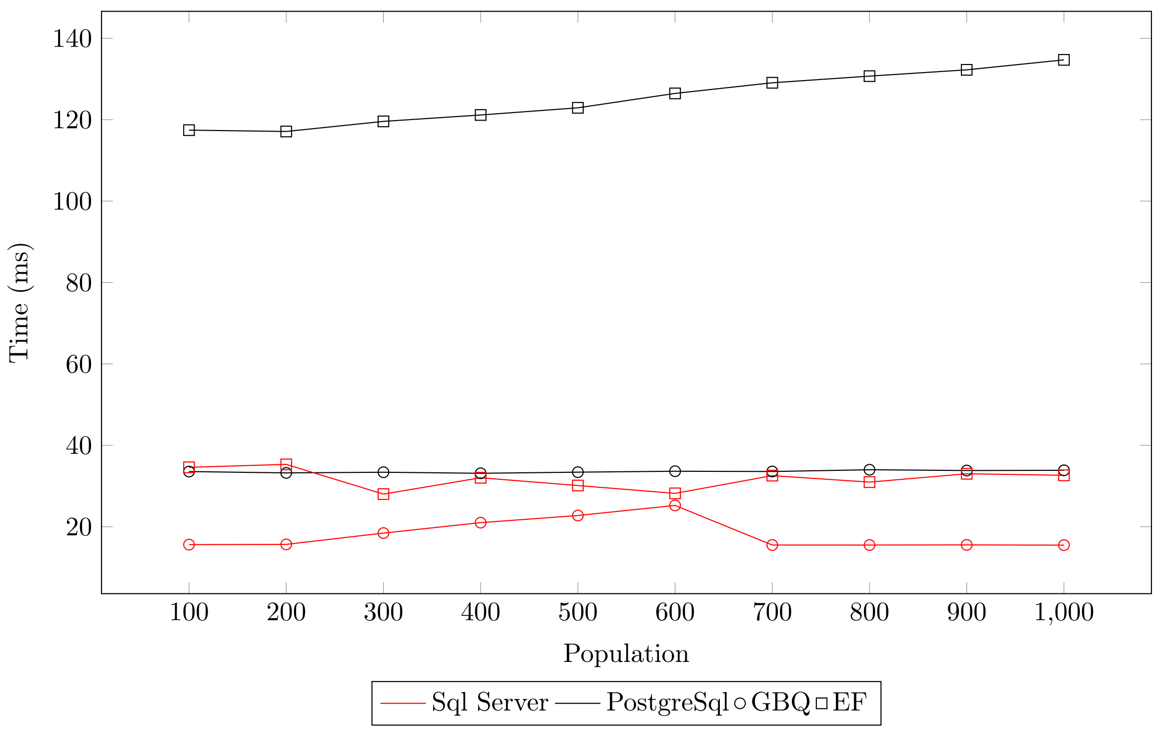
Inh6Assoc2 Query Real Time Medians Comparison



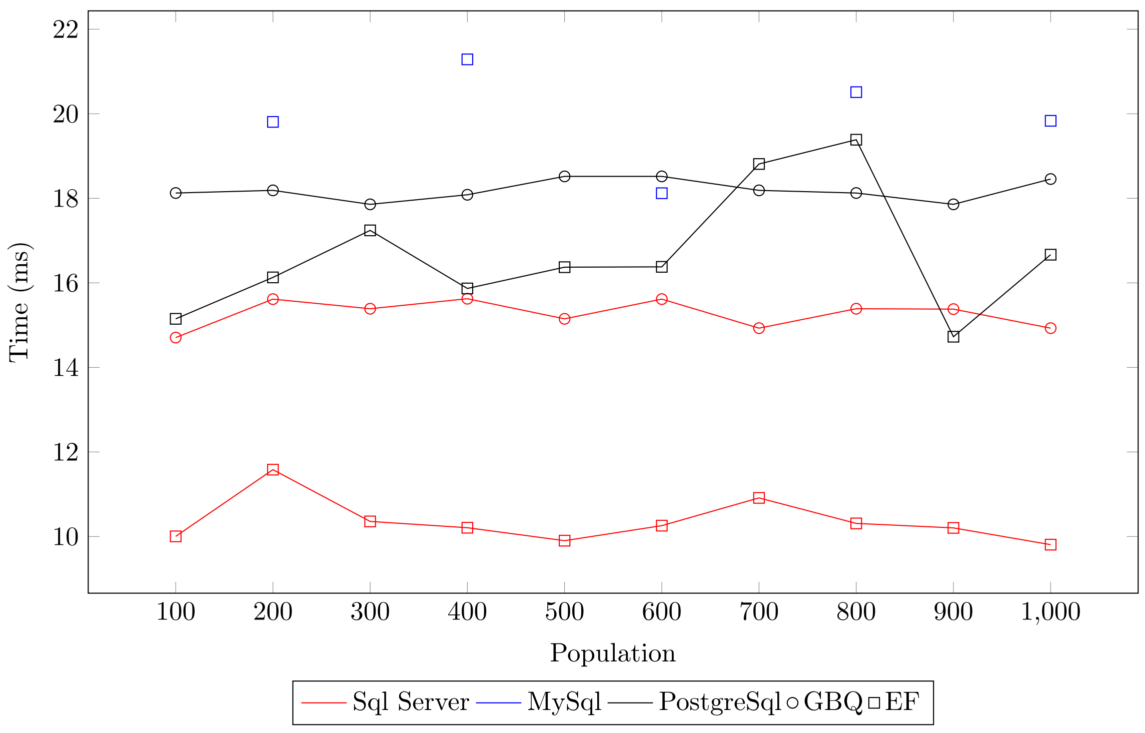
Inh6Assoc2 GBQ CPU Time per Database



Inh6Assoc2 Query Real Time Medians Comparison

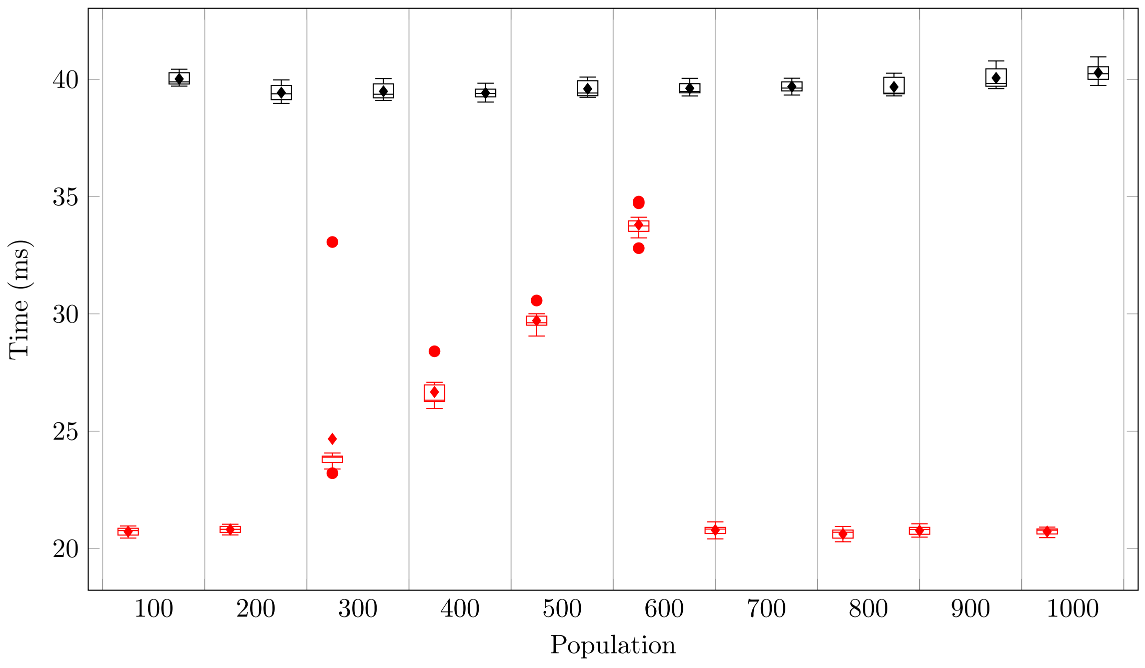


Inh6Assoc2 Query CPU Time Medians Comparison



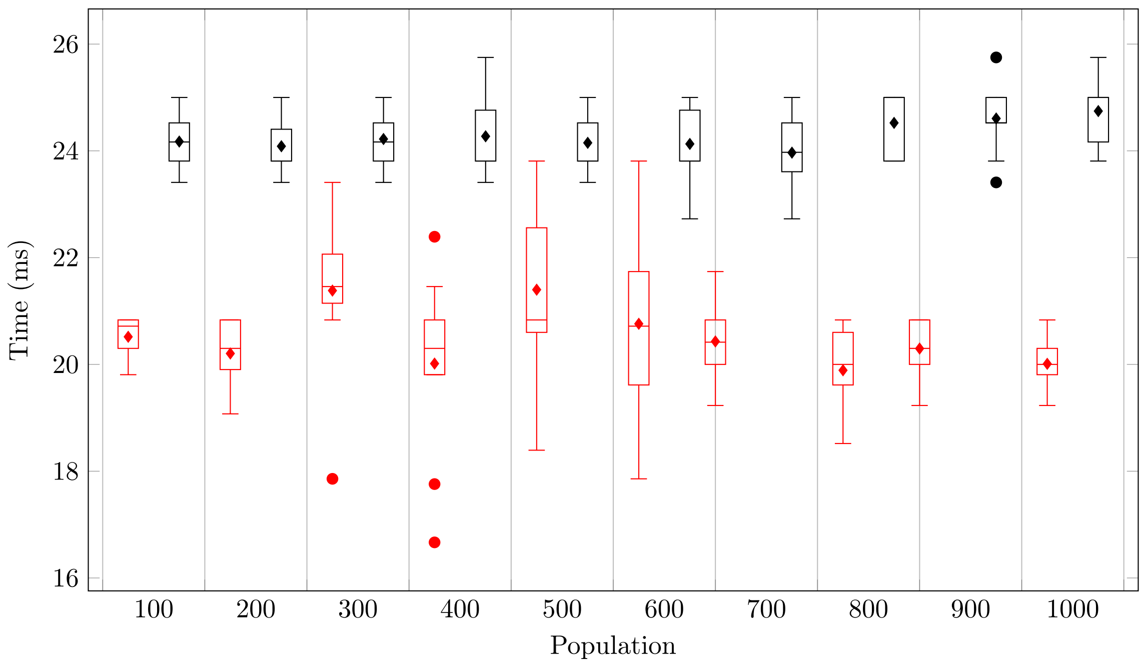
Assoc 3

Inh6Assoc3 GBQ Real Time per Database



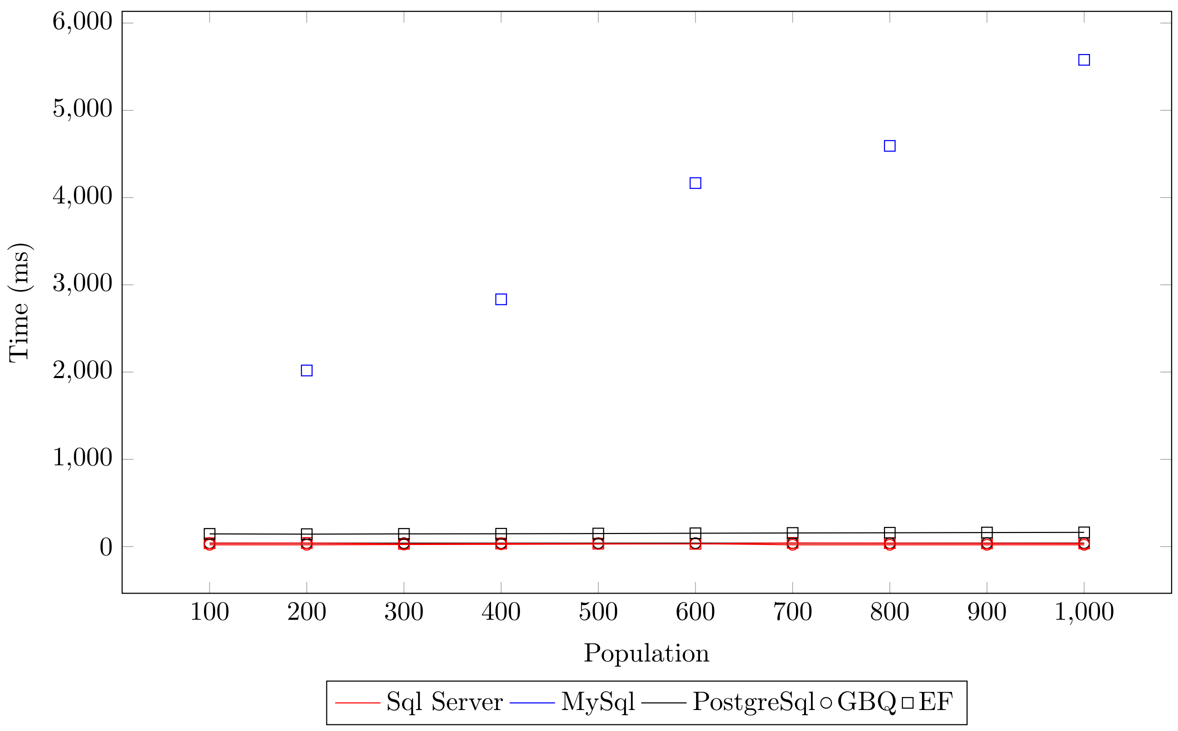
Sql Server	MySql	PostgreSql

Inh6Assoc3 GBQ CPU Time per Database

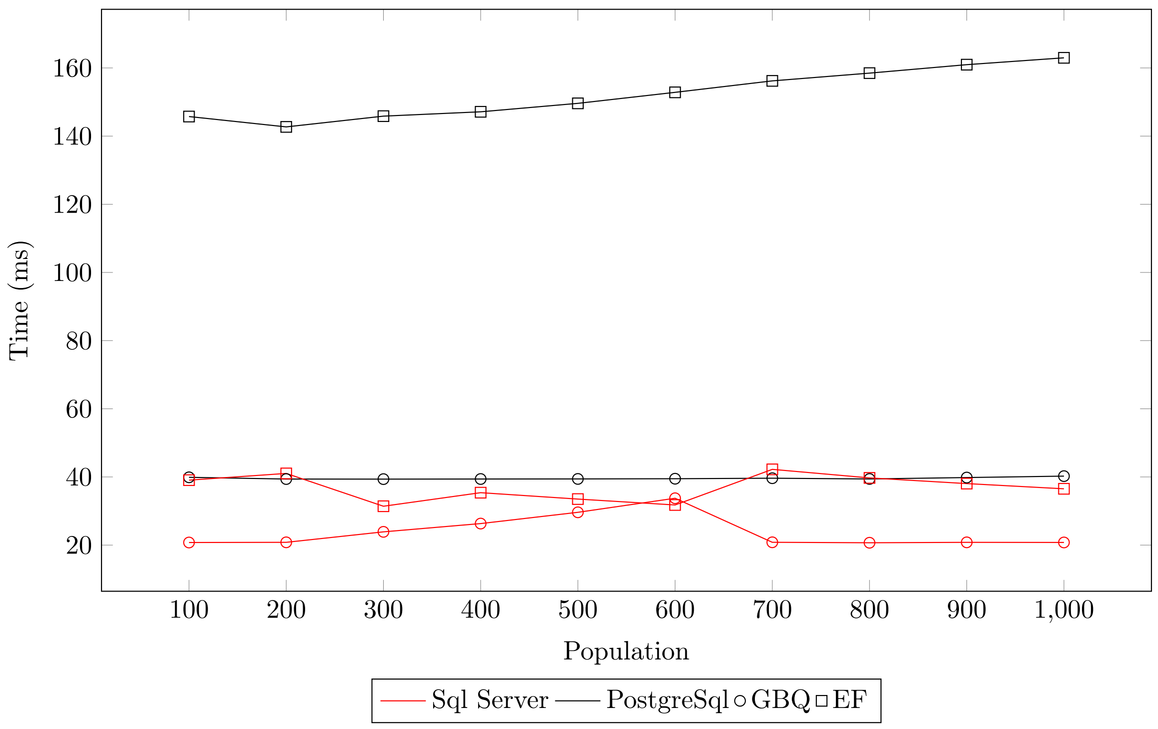


Sql Server	MySql	PostgreSql

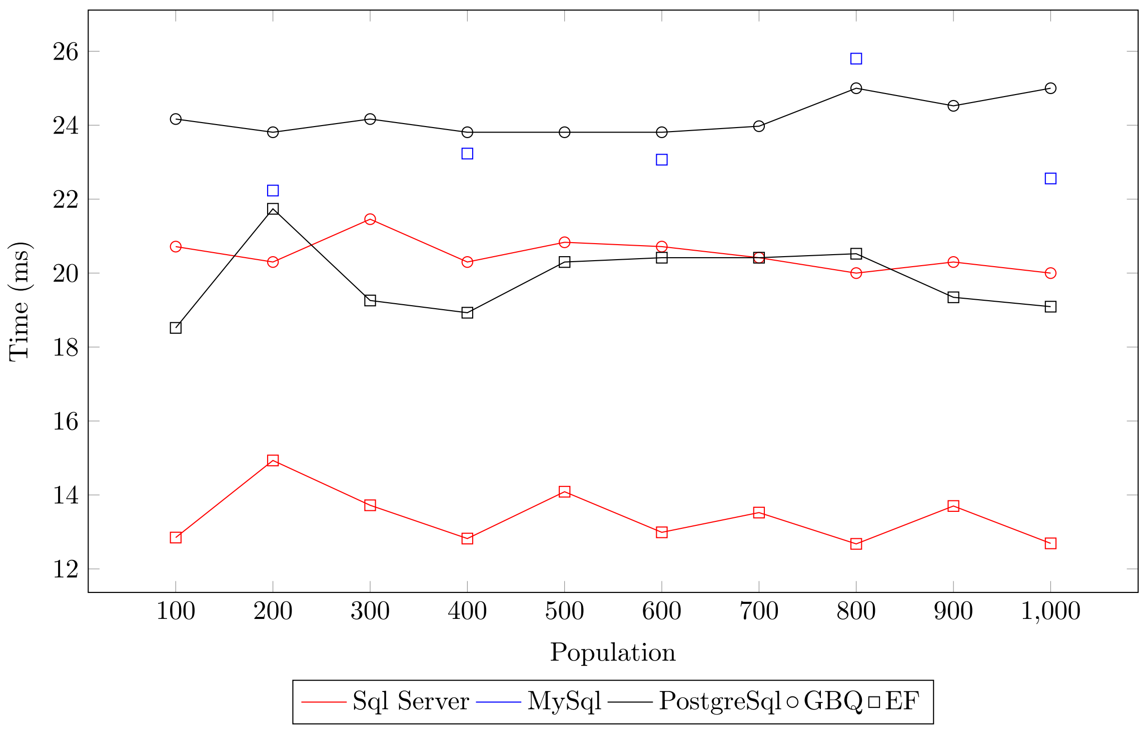
Inh6Assoc3 Query Real Time Medians Comparison



Inh6Assoc3 Query Real Time Medians Comparison

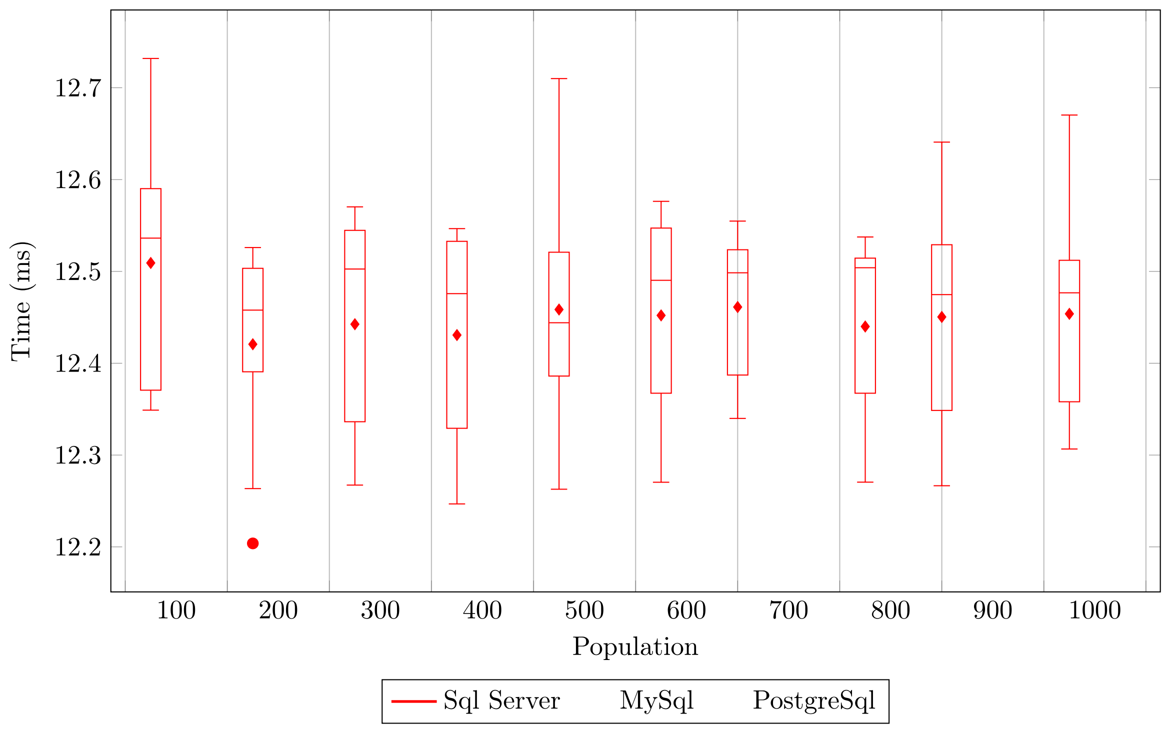


Inh6Assoc3 Query CPU Time Medians Comparison

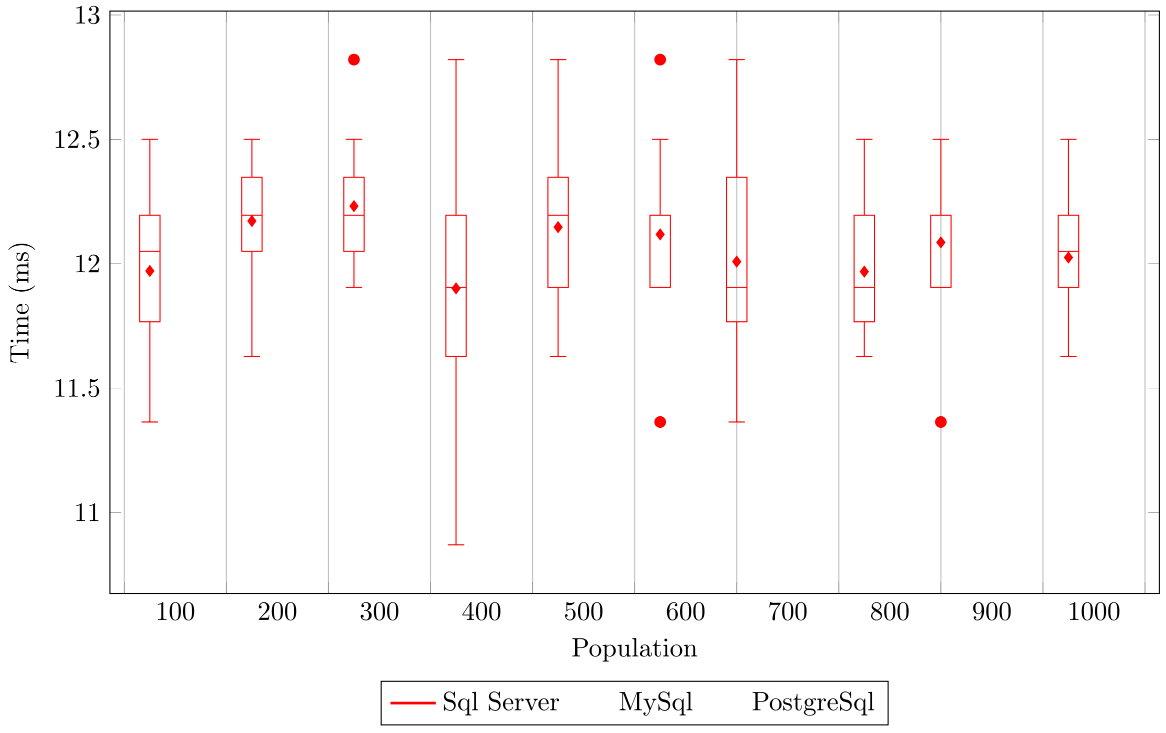


A.3 Northwind

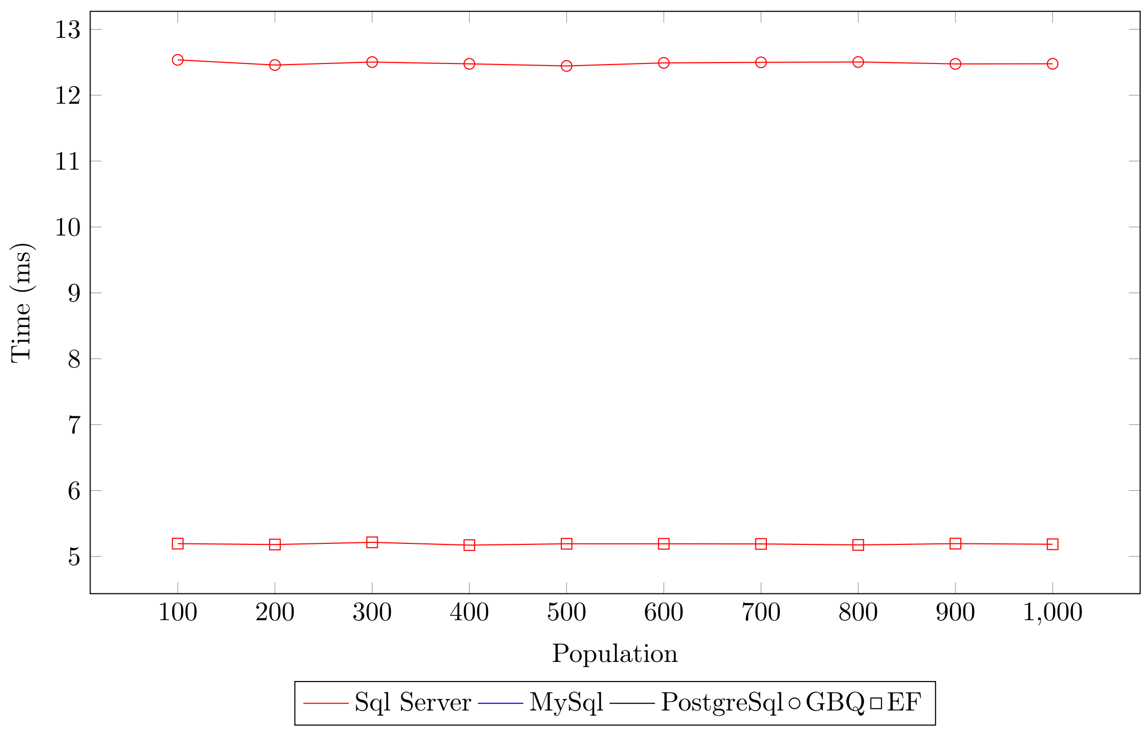
Northwind GBQ Real Time per Database



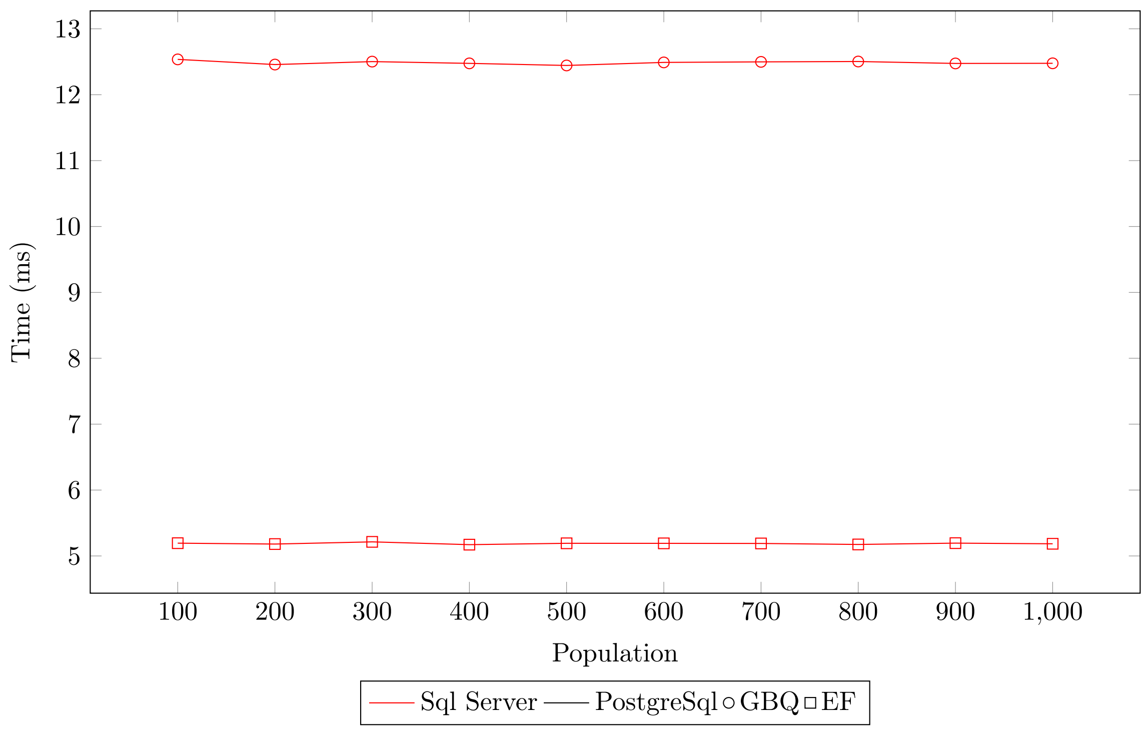
Northwind GBQ CPU Time per Database



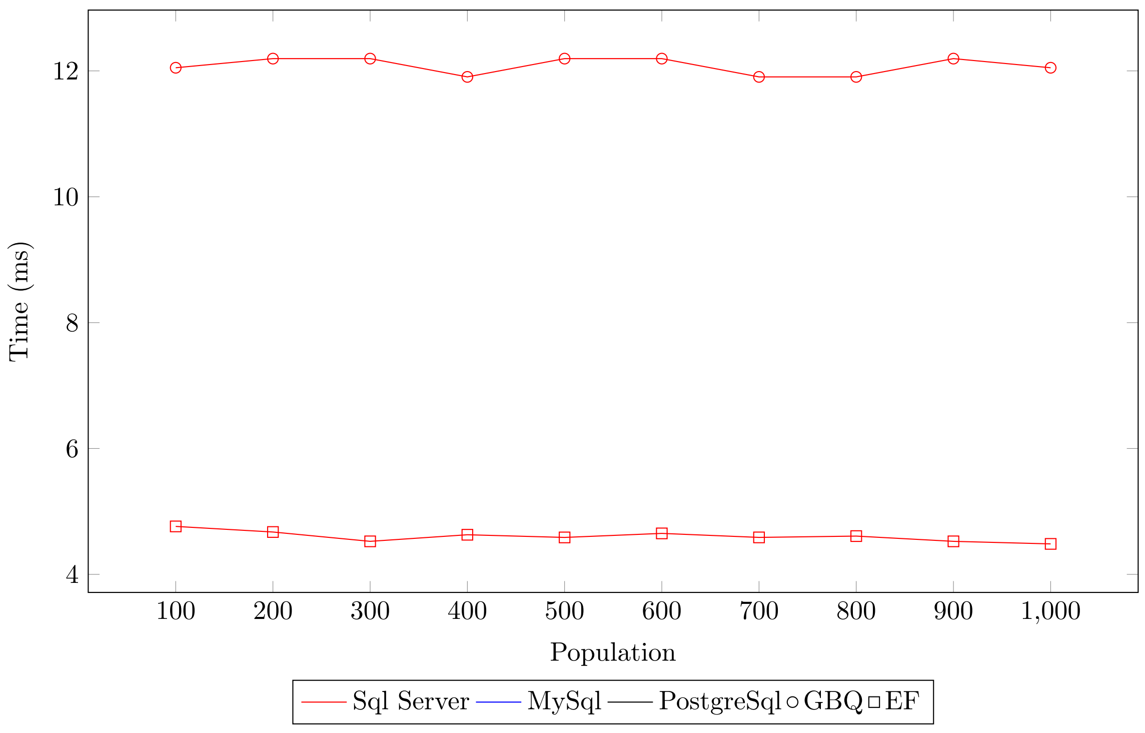
Northwind Query Real Time Medians Comparison



Northwind Query Real Time Medians Comparison



Northwind Query CPU Time Medians Comparison


Time (ms)
Population
Sql Server
MySql
PostgreSql
○ GBQ
□ EF