Optimizing Data Queries Over Heterogeneous Sources

Nuno Grade[1], Lúcio Ferrão[2], and João Costa Seco[1⋆]

[1] CITI - Departamento de Informática, FCT Universidade Nova de Lisboa
[2] OutSystems, SA

Abstract. The challenge of using heterogeneous data-sources is of greatest importance in the development of modern software systems. The current demand is for enterprise software systems with high interoperability that seamlessly integrate local database data with external (and sometimes remote) services (*e.g.* SalesForce, SAP) that expose their core entities through remote APIs.
To be able to express these heterogeneous queries, developers must explicitly fetch, filter and join data. Optimizations to this kind of queries are either inexistent, or ad-hoc and highly dependent on the developer's domain knowledge.
We present a language-based approach to seamlessly and efficiently mix local database tables and remote web-service APIs. Our system builds on top of the .NET LINQ framework, trapping the execution of the query and using developer hints and statistics to adaptively follow the best possible execution plan. Our optimization approach involves not only choosing the fastest and more selective APIs on each execution step, but find the more convenient combination between local and remote operations.
Preliminary usability tests showed that our optimizing query execution engine outperforms the best filtering and joining ad-hoc algorithms designed by highly-skilled developers, while requiring minimum effort to novice developers.

1 Introduction

Business drivers, such as the imperative for speed to market, the agility to change business processes and models, and pressure to reduce operational costs are forcing organizations to move from traditional on-premise data storage like relational databases to cloud-based Software as a Service (SaaS) solutions. This trend has a deep impact on the enterprise applications architecture, forcing custom application development to deal with data that is scattered across relational repositories and online repositories that are only accessible via remote web-service APIs.

Querying data that is scattered across several heterogeneous systems requires the explicit programming of filtering and joining algorithms usually using multiple query languages and remote APIs. Even the simplest of operations like

⋆ Research co-supported by FCT Pest UI527 2011

filtering and merging data sets may become cumbersome, inefficient, error prone and not automatically optimized. Certainly not accessible to a novice programmer as fetching data directly from a database. Thus, the challenges a developer must overcome to ensure good performance while manually dealing with heterogeneous data sources are the following:

- Complexity: developers need to manually choose and explicitly design the query execution plans. This is necessary for the simplest of operations like joining two data sources.
- Awareness: Developers need to be aware of the actual performance of each remote API and the overall plan performance.
- Evolution: As data and network topologies evolve, query execution plans need to be adapted.

The motivation for this work is to simplify the integration scenarios where data is scattered across relational databases and remote APIs with a solution that could be later integrated in the *OutSystem Platform*, an integrated development platform that focus on enterprise custom software development and operation. This project is inspired by *OutSystems* use cases where customers frequently have a central SQL database that is used to extend either SalesForce or SAP core entities, only available through remote web-service APIs.

This paper presents a language-based system that uses a common query language layer for database tables and remote web-service APIs, and leverages on an automatic and adaptive optimization algorithm to provide an efficient and robust way of combining heterogeneous data sources. This work was developed as a master thesis [7]. We use a SQL-like model (LINQ) to abstract the actual API used to get data from remote sites. Since APIs may be overlapping, the choice of the most adequate remote method at each step of the query plan, is also an outcome of the optimizing algorithm.

Our approach leverages on standard results on database query optimizers [1, 8, 9, 11, 12, 18] and web-service orchestration optimizers [3, 5, 10, 17], and proposes a novel adaptive algorithm that on each query execution step chooses the next operation that leads to better execution time. The algorithm is also able to identify situations where several steps can be clustered in the local database, and executes them in small batches. Technically, we proceed by using the Re-LINQ framework [14] that allows a developer access to the query abstract representation, and build a query graph that can then be executed step by step in the best possible order using a specially designed coverage algorithm.

As part of the integration process of various data-sources one needs to create a local model of the remote core entities. Our architecture enhances that underlying data model, not only with collected runtime statistics, but also allowing developers to explicitly annotate data objects code with domain dependent hints – e.g. primary keys of entities in web-services, number of records – and statistics that help the algorithm to find the best execution plan. An example of the interplay between statistics and annotation is explained in sections 3.1 and 3.2.

To validate our approach, we performed tests resorting to the expertise of several OutSystems developers. We have setup a simple scenario and asked de-

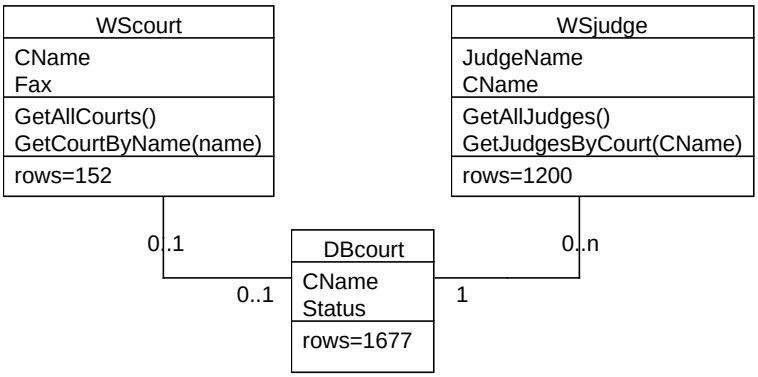

Fig. 1. UML Class diagram of the example scenario.

```
from wsCourt in WScourt
join dbCourt in DBcourt on wsCourt.CName equals dbCourt.CName
join wsJudge in WSjudge on dbCourt.CName equals wsJudge.CName
where wsCourt.CName == "T. de Contas"
select wsJudge.JudgeName, wsCourt.Fax, dbCourt.Status;
```

Fig. 2. A LINQ query on the example scenario (Figure 1).

velopers to design some queries to filter and join a combined set of local tables and remote APIs. We then compared code size and performance of the resulting queries, which demonstrated in the tested cases that all developers significantly underperformed in at least one of the queries, and the execution plans produced by our algorithm was comparable to the best results for each query.

The remainder of the paper is structured as follows: we start by explaining our general approach to optimizing heterogeneous queries in section 2. Section 3 explains how this model was implemented on top of .NET LINQ, and section 4 briefly depicts our validation procedure and obtained results. We conclude in sections 5 and 6 by presenting some related work and concluding remarks.

2 Heterogeneous queries

Consider the example scenario, whose datamodel is depicted in Figure 1, and a sample query with the LINQ code in Figure 2. Assume all attributes are mandatory, and $rows = x$ represents the estimated number of rows for each virtual entity.

In this scenario we are fetching court data from a local database table `DBcourt` and performing a join with two remote web-services. The web-services complement the information about courts (`WScourt`) and about judges working on courts (`WSjudge`). Results are filtered by a condition on attribute `CName` of web-service `WScourt`. The web-service `WScourt` data can be obtained either by

```
FROM E_0
  JOIN E_1 ON J_1
  JOIN E_2 ON J_2,
  ...,
  JOIN E_m ON J_m
  WHERE F_1 AND ... AND F_n
  SELECT A_1, ..., A_k
```

Fig. 3. Simplified syntax for generic queries

API `GetAllCourts()` that retrieves all the courts or API `GetCourtByName(CName)` that only retrieves a single court info given a court name (see Figure 1). Similarly data from web-service `WSjudge` can be obtained either by API `GetAllJudges()` or API `GetJudgesByCourt(CName)`. Intuitively, in the query of Figure 2 where there is a filter on a single court, and since the number of courts and judges in the external web-services can be high, the APIs `GetCourtByName` and `GetJudgesByCourt`, that are more selective, are more adequate than the more general ones (`GetAllCourts` and `GetAllJudges`), and will allow for a more efficient query execution.

2.1 Query Optimization

For the sake of simplicity we focused our analysis on a subset of queries with the form described in Figure 3, where names $E_0, \ldots, E_m$ denote virtual entities (database tables, or core entities available through web-service remote calls), that are joined through the corresponding inner join conditions $J_1, \ldots, J_m$, and filtered by the filtering conditions $F_1, \ldots, F_n$. Finally $A_1, \ldots, A_k$ represent the set of selected attributes.

The algorithm proceeds in two phases, the preparation phase, and the execution phase. In the preparation phase the query is transformed into a graph with the form described in Figure 4. In this graph, virtual entities $E_0, \ldots, E_m$ are represented as rectangular nodes, join conditions $J_1, \ldots, J_m$ are represented by arcs between the virtual entities involved in the join conditions, and finally the filtering conditions $F_1, \ldots, F_n$ are represented as dashed arcs that connect the virtual entity referred by the filter condition and a special node that represents the execution of the filter expression. Each virtual entity node has one of three states: *pending*, *executing* (marked to be resolved in the current cycle), and *resolved*. Each arc has one of two states: *pending*, and *resolved*.

The preparation phase involves the following steps:

1. Build the graph by iterating each virtual entity, join condition, and filter.
2. Flood the graph with extra filtering and join conditions by making use of the transitive properties for equalities in the join and filtering conditions. For each pair of conditions of the form ($E_1.A_1{=}E_2.A_1$, $E_1.A_1{=}X$) we generate and insert in the graph $E_2.A_1{=}X$. This is an optimization step that can be safely ignored.

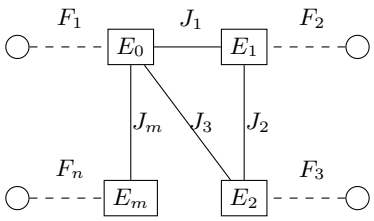

Fig. 4. Generic (sample) graph of a generic query.

3. Mark all the virtual entity nodes and all arcs as *pending*.
4. Populate all the virtual entity nodes with the estimated rows for each entity, starting with the total number of rows for each entity.
5. Populates all the arcs with the estimated resulting number of rows for each arc based on the number of rows of each connected node. In order to estimate the number of rows of a join condition in the form $E_1.A_1 = E_2.A_2$ the algorithm uses the expression:

$$\mathsf{min}\left(\frac{C(E_1.A_1) \times C(E_2.A_2)}{distinct(E_1.A_1)}, \frac{C(E_1.A_1) \times C(E_2.A_2)}{distinct(E_2.A_2)}\right)$$

   where
   - $T(E_i)$ = Number of rows for entity E_i
   - $null(E_i.A_i)$ = Number of rows with null values for attribute $E_i.A_i$
   - $C(E_i.A_i) = T(E_i) - nulls(E_i.A_i)$
   - $distinct(E_i.A_i)$ = Number of distinct values for A_i in E_i

   In order to estimate the number rows for a filter condition with the form $E_i.A_i = k$ we use:

$$\frac{C(E_i.A_i)}{distinct(E_i.A_i)}$$

   Both expressions capture the basic optimization mechanisms used by existing database systems as described in [16].

To illustrate the building of a graph, consider the query described in Fig. 2. Fig. 5 represents the same query in the end of the preparation. In this graph, the condition for F_1,F_2,F_3 is `CName="T.␣de␣Contas"`. In these conditions the virtual entity prefix in the condition is omitted as each filter applies to a single entity. The condition for J_1 is `wsCourt.CName=dbCourt.CName`, the condition for J_2 is `dbCourt.CName=wsJudge.CName`, and the condition for J_3 is `wsCourt.CName=wsJudge.CName`. In this graph, J_3, F_2, and F_3 arcs were only created in step 2 exploring the transitive properties of J_1, J_2, and F_1. Assuming we know $distinct(wsCourt.CName) = T(wsCourt)$, $distinct(dbCourt.CName) = T(dbCourt)$, $distinct(wsJudge.CName) = T(wsJudge)/6$, and that all attributes are mandatory, we have enough information to populate the estimated number of rows for each arc.

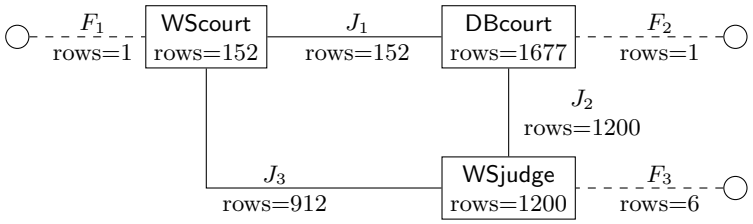

Fig. 5. Graph representation

The execution phase takes the graph as input and repeats the following steps until there are no more pending arcs in the graph:

1. Build priority queue of arcs that represent the minimum spanning tree based on the Kruskal's algorithm applied to query graphs [9]. The resulting priority queue contains pending arcs ordered by the minimum number of estimated rows.
2. Identify a number of arcs, starting from the head of the priority queue involving a single database source. If the first arc from the head of the priority queue involves more than one data source or an API based virtual entity, just pick (identify) the first arc from the queue.
   Mark as *executing* any *pending* virtual entity node that is directly connected to the identified arcs.
3. Resolve all the virtual entity nodes marked as *executing* using the procedure in section 2.2. The result of this procedure is a dataset kept in memory.
4. Apply any *pending* filters (arcs) connected to *executing* nodes by filtering the target dataset with the respective condition. Mark those arcs as *resolved*.
5. Execute any *pending* joins that connect two *executing* or *resolved* nodes by explicitly joining the two datasets in memory. Mark those arcs as resolved.
6. Update all *executing* nodes in this iteration with new estimated row count with the effective row counts in the resulting data set.
7. Update the estimated row count for the pending arcs that are connected to *executing* nodes using the estimation expressions of the building phase.
8. Mark as *resolved* all the *executing* nodes in this iteration, repeat from step 1 until there are no more *pending* nodes.

When this cycle ends, the graph contains no mode pending arcs, and all the nodes were resolved. Notice the algorithm is adaptive in the sense that it uses information from the previous cycles to find the best path for each cycle. Also notice that Kruskal's algorithm from step 1 only selects a minimum spanning tree for the given graph, but since all the arcs need to be resolved, we execute the remaining arcs in step 4 and step 5.

2.2 Fetching Data for a Virtual Entity

Given a set of *executing* nodes that can be fetched from a single database, we first pick all the associated filter conditions, and all the join conditions that can

be immediately sent to the database – join conditions that do not involve *pending* nodes. Filter and join conditions are used to retrieve the minimum amount of data from the database, using a single query. In this step, join conditions that involve nodes that were previously resolved are translated to conditions using literals from previously fetched data. Whenever filter and join conditions are sent to the database, they are marked as *resolved* in the graph.

To resolve a node that represents a remote APIs, we pick the existing API set and select the ones that can be used in the current execution step (e.g. using a `GetEntityByName(name)` can only be used if the value for parameter `name` is already known at this time). Then, for each available API we compute the number of times it needs to be called. This is based on the number of distinct values from the rows previously fetched that are being joined to the current node, i.e. an API `GetEntityById(id)` needs to be called as many times as the number of distinct *id* values. Finally, the chosen API is the one that gives the minimal total execution time, where the total time is obtained using the average execution time and the number of calls needed at the given step.

2.3 Execution Phase Example

To illustrate the execution phase, consider the graph depicted in figure 5. Executing this graph involves the following steps:

1. Build the priority queue with the following order: F_1, F_2, F_3, J_1, J_3.
2. Select a single arc F_1 and mark node *wsCourt* as *executing*.
3. Resolve node *wsCourt* using the API `GetCourtByName` to retrieve the minimum required data for *wsCourt* as invoking `GetCourtByName` once. It is faster than invoking `GetAllCourts`. Mark F_1 as *resolved* as it was ensured by the `GetCourtByName` API.
4. Update node *wsCourt* with rows=1.
5. Update the estimated rows for J_1 and J_3, resulting in "(J1) rows=1" and "(J3) rows=6".
6. Mark node *wsCourt* as *resolved*.
7. Build a new priority queue with the following order: J_1, F_2, J_3, F_3.
8. Pick a single arc J_1 and mark node *dbCourt* as *executing*.
9. Resolve node *dbCourt* using the query
   `SELECT * FROM dbCourt WHERE CName=@CName`
   to retrieve the minimum required data. Join the data obtained in the previous iteration with the data obtained in this step. Mark arc J_1 as *resolved*.
10. Resolve filter F_2 and mark it as *resolved*.
    Note: this step could be optimized as F_2 is redundant with F_1. This kind of optimization was left as future work.
11. Update node *dbCourt* with rows=1.
12. Update the estimated rows for J_2, resulting in "(J2) rows=6".
13. Mark node *dbCourt* as *resolved*.
14. Build a new priority queue with the following order: J_2, F_3.
15. Pick a single arc J_2 and mark *wsJudge* as *executing*.

16. Resolve *wsJudge* using the API `GetJudgesByCourt` once to retrieve the minimum required data. Join the data obtained in the previous iteration with the data obtained in this step. Mark arc J_2 as *resolved*.
17. Resolve filter F_3 and mark it as *resolved*.
18. Resolve join J_3 and mark it as *resolved*.
19. Mark node *wsJudge* as *resolved*.

At this point, no more *pending* arcs are present in the graph. The algorithm stops and returns the selected attributes from the current dataset in memory. When compared with ad-hoc carefully crafted implementations by experienced developers, the execution plans defined by our approach do not introduce a significant overhead in practice. A more careful evaluation is presented in section 4, and a similar, but more illustrated example can be found in [7].

3 Implementation

The most widely used language to query databases is SQL [6], and one of its most interesting integration extensions for .NET is LINQ [2], which enables .NET developers to express queries in an integrated and type safe way. The standard LINQ framework supports the writing of heterogeneous queries, however, the default execution plan is a straightforward in-memory based interpretation of the join and filtering commands, which many times results in sub-optimal network and memory consumption, with significant impact on scalability and efficiency.

In order to intercept, and change LINQ execution pipeline, we used a framework called Re-LINQ [14]. Re-LINQ parses a LINQ query and produces an abstract representation of it, thus allowing custom adapters to manipulate the query model. This allows a specially designed algorithm to decide and execute the query by invoking the query executor adaptors corresponding to each data source. Many implementation challenges we faced were drawn by the novelty of the work itself, and the by the lack of maturity of the Re-LINQ framework. Accomplishing basic tasks like checking whether a query involves more than a single data source or directly passing queries to the database, are many times filled with unanswered questions.

In the adaptation process, a developer using our approach needs to define, the so-called *Virtual Entities*. Virtual entities are represented by .NET classes that represent tabular data, stored either in a local database, or behind custom web-service APIs. On one hand, mapping database entities, virtual entities trivially map the exact same structure. On the other hand, mapping a set of custom remote APIs requires assuming a simple naming convention that informs the optimizer of the possible APIs that can be used to query a specific remote entity by a given attribute (*e.g.* customers can be obtained with a *GetAllCustomers()* or by a *GetCustomerById(id)*).

A small performance note, the implementation of the in-memory join operations needs to use hash joins instead of nested loop joins (the naïve approach) in order to constrain processing times under O(n) instead of O(n*n).

```
[EntityHint(totalRows = 150)]
public class WSjudge {
       [ColumnHint(unique=true, distinctRatio=100, nullsRatio=0)]
       public string JudgeName {get; private set}

       [ColumnHint(unique=false, distinctRatio=35, nullsRatio=0,
                            foreignKeyTo=typeof(DBcourt)]
       public string CourtId {get; private set}
}
```

Fig. 6. Hint Decorations for Virtual Entities in C Sharp

3.1 Hints

To support a minimal estimated number of rows of step 5 of the preparation phase, hints can be added to the virtual entities. Hints are given as C# attributes that developers use to decorate classes that represent virtual entities. The following meta-information can be specified:

- estimated total number of rows of the entity
- unique attributes
- rate of distinct values for each attribute
- rate of null values for each attribute
- relation between attributes of different virtual entities (foreign-key to another virtual entity's unique attribute)

This kind of hints are enough to guide the query execution algorithm allowing for a correct estimation of most join operations. Also, many developers are already familiar with these concepts since they are available in traditional relational databases query interfaces. Figure 6 depicts the way in which hints can be used to decorate a sample virtual entity in C#. Both `distinctRatio` and `nullRatio` represent an estimated ratio between 0% and 100%.

3.2 Statistics

Hints are used to help the optimizer avoid uninformed decisions. To complement that static information provided by developers, the query optimizer also keeps track of statistical information based on effective retrieved data, specially when retrieving all data from virtual entities. The optimizer always gives priority to statistics over hints when they exist.

Statistics are specially important when retrieving data from remote APIs. For each remote API, we keep track of:

- average execution time
- average of number of rows returned

These measures are used by optimizer in order to pick the most efficient remote API on each execution step. The choice of the API at each execution step is based on the estimated execution time for the same step – the execution time multiplied by necessary number of calls. To capture the average execution time for each API, a fixed sliding window of execution times is kept in the model.

4 Results and validation

The optimizer results were tested against a small set of experienced developers inside OutSystems R&D team. Each developer was presented the model from Fig. 1 containing one database entity and two remote entities as well as the number of rows for each entity. The remote entity had an API used to fetch all results for the given remote entity, and more specific APIs to retrieve filtered data. Each API had their average times annotated and visible to the developers. We simulated a loaded environment by artificially inflating the execution time of the remote APIs to 400ms for the APIs that retrieve all the results for a given remote entity and to 50ms for the more specific APIs.

Developers were asked to write the pseudo-code to execute three queries. The first two are a simple join between two entities, and the last one required to join three entities. The pseudo-code was manually converted to C# by the authors, and the resulting execution times were measured several times in order to reduce the testing infrastructure errors to less than 5%.

All the developers reached a correct solution. The average execution time for each query is presented in Table 1. We marked in **bold** the results that were significantly slower than the fastest alternative. With these tests we found that the algorithm was relatively close to the best human alternatives, and it was frequent for developers to code using naïve approaches even when they were asked to be careful with the execution time.

The two most relevant patterns we observed in the naïve approaches are: using nested loops to join two previously fetched data sets, and fetching data inside a loop even when data is repeated for multiple instances. Such naïve approaches have a significant impact in the execution time, in this case, up to 10 times slower than the results obtained by the optimizer.

Although the sample's size is small, it is significant that all developers underperformed in at least one query when compared to our algorithm.

5 Related Work

There is a quite extensive and stable literature on query optimization [1, 15, 16] that focuses on the two key components of a query evaluation: the query optimizer and the query execution engine. Optimization is heavily based on meta-information provided by the database catalog, that keeps statistics about the database structure, its contents, and data arrangement (e.g. disk sectors). At the database level, the main focus is on ordering of joins, selecting indexes to apply, and rapidly reducing the number of records. On an heterogeneous

	$Query_1$	$Query_2$	$Query_3$
Optimizer	0.62	0.55	0.70
$Developer_1$	0.57	0.51	**4.56**
$Developer_2$	**1.00**	0.50	**7.00**
$Developer_3$	0.56	0.50	**1.26**
$Developer_4$	**0.98**	0.51	0.72

Table 1. Average Execution Times in Seconds.

context, other factors arise. The response time of a web-service, the amount of data transfered, and the kind of API used, became the most critical factors.

Approaches like [9, 13] represent database queries using graphs where relations are nodes, and join operators and predicates are represented as well (as arcs). In the domain of heterogeneous queries, [17] considers query plans over databases and web services, also represented as graphs.

The uniform concept of selectivity [4, 17], which we use, appears as a measure to estimate the cost of an operation. The more selective an operation is, the less records it produces and therefore the better it is for future operations in the execution plan. The general concept can be applied to both database tables and to remote APIs. Call selectivity [17] is based on the number of retrieved rows, per given input. As for column selectivity [4] it is computed from number of distinct values in a column divided by the total number of rows in that entity.

Selectivity is used in [17] to compute the optimal order of web service invocations in a query plan, for queries concerning database and web service entities. Nonetheless, their approach groups database and webservices into two separate groups. Unlike this, we implement an adaptive mix, inspired in [9], that allows for extra optimization oportunities.

We extend all the above approaches by repeating the optimization analysis every time the algorithm switches the kind of data sources. This allows us to maximize the information available at each step. This works well if the number of different data sources is kept small, and the cost of transporting information represents the significant overhead.

6 Concluding remarks

We presented an adaptive optimizing query execution engine that deals with heterogeneous data-sources. It takes into account both developer hints and statistics to find the best execution plan. We assume that in this context, the latency and data transference time represent the bigger query overhead, and hence explore all the available information to minimize the transmission time. We also batch as much as possible database join operations. Our work enable developers to write heterogeneous queries as such, instead of being forced to design complex memory based join operations.

The ultimate goal of this work is to inspire and improve the integrated *OutSystems Platform* using a consistent query language to fetch data from both a database and remote APIs. The preliminary results give us confidence that there is a simple and effective solution that covers the most common scenarios.

References

1. Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the 17th ACM Symposium on Principles of database systems*, PODS '98, pages 34–43, New York, NY, USA, 1998. ACM.
2. Suzanne W. Dietrich and Mahesh Chaudhari. The LINQ between XML and databases: a gentle introduction. *J. Comput. Sci. Coll.*, 25(4):158–164, April 2010.
3. Weimin Du, Ming-Chien Shan, and Umeshwar Dayal. Reducing multidatabase query response time by tree balancing. *SIGMOD Rec.*, 24(2):293–303, May 1995.
4. Nigel R. Ellis and Rodger N. Kline. Automatic database statistics creation. US Patent 2002/0087518 A1, July 2002.
5. Cem Evrendilek, Asuman Dogac, Sena Nural, and Fatma Ozcan. Multidatabase query optimization. *Distrib. Parallel Databases*, 5(1):77–114, January 1997.
6. Int. Organization for Standardization, Int. Electrotechnical Commission, and Joint ISO IEC Technical Committee Information Technology. *Information Technology - Database Languages - SQL: ISO/IEC 9075*. Int. standard. ISO, IEC, 1992.
7. Nuno Grade. Data Queries Over Heterogeneous Sources. Master's thesis, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2013.
8. R. Guravannavar and S. Sudarshan. Reducing Order Enforcement Cost in Complex Query Plans. In *23rd Int. Conference on Data Engineering*, pages 856 –865, 2007.
9. P.B. Guttoski, M.S. Sunye, and F. Silva. Kruskal's algorithm for query tree optimization. In *11th Int. Database Engineering and Applications Symposium*, 2007.
10. Donald Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, December 2000.
11. Quanzhong Li, Minglong Sha, V. Markl, K. Beyer, L. Colby, and G. Lohman. Adaptively reordering joins during query execution. In *23rd Int. Conference on Data Engineering*, pages 26 –35, 2007.
12. C. Mishra and N. Koudas. Join reordering by join simulation. In *IEEE 25th International Conference on Data Engineering*, pages 493 –504, 2009.
13. Arnon Rosenthal and Cesar Galindo-Legaria. Query graphs, implementing trees, and freely-reorderable outerjoins. *SIGMOD Rec.*, 19(2):291–299, May 1990.
14. Fabian Schmied. re-linq: A general purpose linq foundation. https://www.re-motion.org/download/re-linq.pdf, 2009.
15. D.E. Shasha, P. Bonnet, and P. Bonnet. *Database Tuning: Principles, Experiments, and Troubleshooting Techniques*. Morgan Kaufmann Publishers, 2003.
16. A. Silberschatz, H.F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 2010.
17. Utkarsh Srivastava, Kamesh Munagala, Jennifer Widom, and Rajeev Motwani. Query optimization over web services. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, 2006.
18. D.D. Straube and M.T. Ozsu. Query optimization and execution plan generation in object-oriented data management systems. *Knowledge and Data Engineering, IEEE Transactions on*, 7(2):210 –227, apr 1995.