OptPLAN: Improving the Optimal Plan Calculation on Relational Databases

Luís Fernando Orleans*, Miguel Mendes de Brito†, Egberto Caetano Araújo da Silva‡
*Departamento de Ciência da Computação
Instituto Multidisciplinar - UFRRJ, Rio de Janeiro, Brazil
†Email: lforleans@ufrrj.br, mmdebrito@ufrrj.br, egbertocaetano@ufrrj.br

***Abstract*—In order to find the best execution plan for an SQL query, a DBMS uses information regarding the cost of disk operations, notably the cost of a sequential page reading (seq_page_cost or just *spc*) and the cost of random pages reading (random_page_cost or just *rpc*). Such information is predefined by DBMS vendors and are rarely changed - although it can cause inaccuracies in the optimization phase. This paper lists some typical scenarios where disk access costs miscalculations can lead to sub-optimal query plans and presents the OptPLAN, a tool for the PostgreSQL DBMS that calculates the correct relationship between *spc* and *rpc* and automatically set their values at the configuration properties. In our experiments, we obtained up to 69% of query speed up after *spc* and *rpc* adjustment.**

***Keywords*—OptPlan; Query Optimization; Autonomous RDBMS.**

I. Introduction

Because Relational Database Management Systems (RDBMS) provide a convenient and secure way to store and retrieve data, they became crucial components for many modern systems [1]. Also, RDBMS are a result of decades of joint efforts from both scientific and industry research teams [2], which provides a solid and well-formed theory behind them.

The Structured Query Language (SQL) is the most used tool for manipulating data stored by RDBMS [3]. Commands written in SQL are first parsed and validated. If no errors are found, the commands are then translated to an internal logical tree representation of the query, denoted as *execution plan* or just *plan*.

Several plans can be derived from a single SQL statement [3], each using different strategies and algorithms for performing relational operations, e.g., a joint operation could be represented using a Nested Loop Join algorithm in one tree and using a Hash Join in another tree. While the algorithm choosing process does not impact on query's semantics, it does directly impact on query's *cost*, i.e., the amount of disk blocks the RDBMS will transfer from/to memory during query execution.

Finding the cheapest tree is not a trivial task – in fact, [4] claims this problem is NP-complete. Each RDBMS has one or more optimization algorithms implemented for this specific problem and they vary greatly from one vendor to another. The open-source RDBMS PostgreSQL, for instance, uses a genetic query optimization approach. In addition, PostgreSQL provides a classic optimization engine based on Dynamic Programming [5]. Regardless the optimization strategy, the RDBMS server hardware configuration influences the search for the optimal plan. For instance, the same SQL statement might result in different plans depending where data is stored: in a fast SSD or in a slow HDD. Ignoring this hardware dependency can result in sub-optimal execution plans due to inefficient optimizations phases which, in turn, can result in poor query performance.

This paper presents OptPLAN, a tool that calculates both sequential and random disk read costs for a PostgreSQL RDBMS host and set those values on the configurations file, providing the optimization algorithm correct costs which results in better optimal plan search. Our results show that all example queries presented reduced costs after OptPLAN adjust the configurations file when compared to default values provided by RDBMS vendor.

The remainder of this paper is as following: Section II describes a few related works we found most relevant, while Section III reviews the steps taken by a query executor engine from a typical RDBMS. Our tool OptPLAN is defined in Section IV, the experiments we conducted along with their results are described in Section V. Finally, Section VI lists our conclusions and possible future works.

II. Related Work

The research presented in [6] discusses self-adaptable database systems based on Microsoft's AutoAdmin project. That work focuses primarily on the design of automatized databases. Other approaches for query optimization are described in [7], where the authors describe a log inspection tool, which provides several statistics for the database administrator (DBA). Those statistics are collected through Machine Learning techniques. Also using changes in the query optimizer, the work on [8] explains ORACLE 10g's capability of optimize SQL queries automatically. Such optimization is possible due to longer than usual SQL statement analysis, validating statistics used on the search for the optimal plan. Finally, Chaudhuri et al. [9] describe the advances made on self-adjustment on Microsoft's SQL Server RDBMS, which is achieved through automatic indexes creation and more efficient memory management and dynamic resource allocation approaches.

All those works do not pursuit finding the correct values for sequential and random page disks reads, neither do they have an open implementation that can be used on more than one DBMS. Our work exploits hardware-related factors from the DBMS host, enhancing the searching for the optimal plan for the SQL queries without the need of any modifications at the DBMS core.

III. Query Processing

Data is typically manipulated on a RDBMS through a series of SQL statements issued by a user or a system. Upon its arrival, each SQL statement is checked for syntax correctness and, whether no errors are found, an internal representation (using a tree format) of the statement is created. Each node of the tree represents a Logic Relational Operator (LGO) that can have several implementations, one for each specific scenario. For instance, a Join LGO can be implemented physically as Nested Loop Join (NLJ), Hash Join (HJ) or Sort-Merge Join (SMJ). The decision of which implementation should be picked is responsibility of the Query Optimizer, that searches for the cheapest execution plan that physically represents the tree. After the optimal plan is found, it gets executed by the Query Executor Engine (Figure 1).

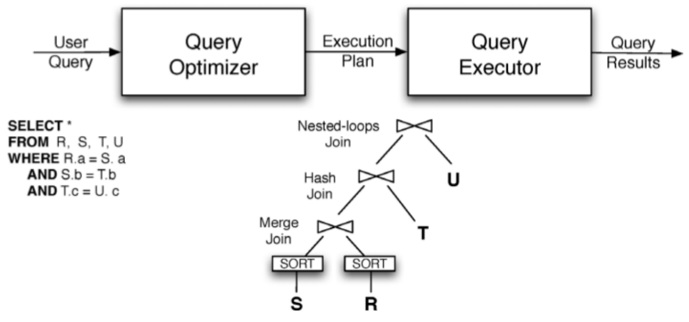


Fig. 1: Query Execution [3]

Internally, the most widespread implementation of an execution plan is through the Iterator pattern, described in [10]. Also, each LGO includes an annotation regarding which physical implementation shall be used.

A. Query Optimization

A typical query optimizer relies on host hardware information and relations statistics to find the cheapest execution plan for an SQL statement. RDBMSs often store those metadata within an internal structure called *catalog*. It is a common practice to store information about *column values*, such as histograms, ordering, among others. When combined with some hardware information, such as sequential page read cost (SPC), random page read cost (RPC), data transfer rate, etc. The Query Optimizer can correctly determine which physical relational operator should be used in order to find the cheapest plan. It is important to notice that hardware information are often represented as normalized cost values instead of absolute values, i.e., the amount of time for the hardware to process a task.

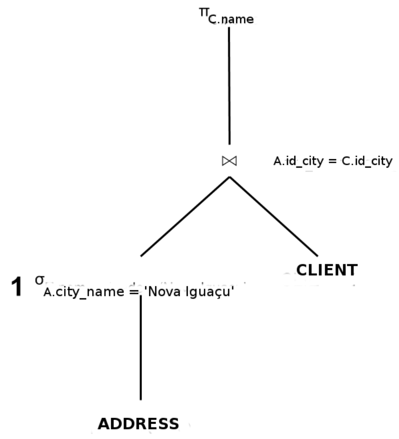
As an example, consider the following database schema:

```
CLIENT(id_client, name, email, gender,
age, id_address);
ADDRESS(id_address, street_name, number,
borough, city, state, zip);
```

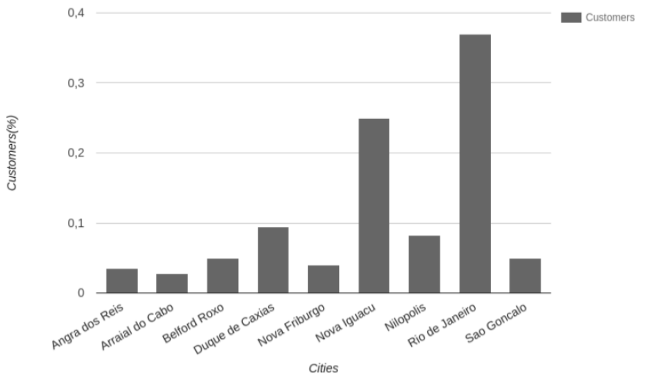
Consider there is an B-Tree index on column 'Address.city'. The following SQL statement is used to retrieve the names of all clients that live in the city of 'Nova Iguaçu':

```
SELECT C.name
FROM CLIENT C, ADDRESS A
WHERE C.id_address = A.id_address
AND A.city = "Nova Iguacu";
```

One of the possible plans that represents the SQL statement in Section III-A is shown in Figure 2(a). Now consider the Figure 2(b) is the histogram that counts how many clients live on each city.



(a) Execution Query Plan.



(b) Clients distribution among cities.

Fig. 2: Information used by the Query Optimizer.

Finally, consider that there are 1000 tuples in each table.

The cost for reading a whole table – sequentially or randomly – can be expressed by:

$$C = (blc * cl) + (t * cc), \tag{1}$$

where C is the total cost; blc is the total number of disk blocks where tuples are stored; cl is the cost for a block read. The read can be performed sequentially (cl_s) or randomly (cl_r); t is total tuples of a relation; and cc is the CPU cost for processing each tuple [11]. It is worth noting cl_s, cl_r and cc are hardware related parameters.

For the first example, let us assume the following values for the variables (the variable blc has the same value for both tables):

$$cl_s = 1; cl_r = 4; blc = 10. \tag{2}$$

Worth noting that those are default values for clean PostgreSQL installations.

When the process of selecting all Address tuples that have city values equal to 'Nova Iguaçu' (point 1 at 2(a)) is evaluated, both sequential ($C_{Sequential}$) and random (C_{Random}) reads are considered. The cost for $C_{Sequential}$ is computed as follows:

$$C_{Sequential} = (blc * cl_s) + (t * cc) \tag{3}$$
$$C_{Sequential} = (10 * 1) + (1000 * 0.01) = 20 \tag{4}$$

On the other hand, to compute the cost for C_{Random}, the Query Optimizer can use the histogram depicted in Figure 2(b) to retrieve an estimate of how many tuples will be read, as random reads use indexes to perform direct access to the data. As the frequency of Address tuples having city equals to 'Nova Iguaçu' is 0.25, hence $f_{NovaIgua\text{ç}u} = 0,25$. So, $t = 1000 * 0,25 = 250$ and the expression is as follows:

$$C_{Random} = (blc * cl_r) + (t * cc) \tag{5}$$
$$C_{Random} = (10 * 4) + (250 * 0.01) = 42.5 \tag{6}$$

which is 4 times greater than $C_{Sequential}$. Hence, the Optimizer will prefer to perform a table scan instead of random reads using the index.

However, let us consider that cl_r variable value is 1.5, i.e. a random read is one and a half times slower than a sequential read. Hence:

$$C_{Random} = (10 * 1.5) + (250 * 0.01) = 17.5 \tag{7}$$

and the Optimizer now should prefer using the index instead of a full table scan to perform the query! It becomes clear that inaccuracies on both SPC and RPC values can cause the Query Optimizer to choose a suboptimal execution plan for any SQL statement.

B. *PostgreSQL Hardware Specific Variables*

Among PostgreSQL configuration files, *postgresql.conf* is the one that holds default values for SPC (seq_page_cost) and RPC (random_page_cost) [5]. The first parameter refers to reading a single page from disk sequentially and its value is normalized to 1. On the other hand, the second parameter refers to randomly reading a single page from disk and its value defaults to 4 – which means that RPC is 4 times greater than SPC by default. As those are normalized values, increasing SPC and RPC values proportionally will not change the execution plan.

However, as PostgreSQL can be used on a plethora of different platforms, those variables should be adjusted to reflect the host's hardware. Whether the PostgreSQL server is attached to a Storage Area Network or using a conventional HDD, those relative values will vary from one configuration to another. In the next section, we present OptPLAN, a tool that computes the correct values for both SPC and RPC and sets them in the configuration file.

IV. OptPLAN

We developed a tool named OptPLAN that executes a series of tests on the hardware where the PostgreSQL is being installed and determines the up-to-date relative values for SPC and RPC, setting those values on *postgresql.conf* file. The Query Optimizer will then use correct values when searching for optimal plans, avoiding query slow down. In addition, OptPLAN is invoked periodically to detect hardware changes.

A. *Implementation*

OptPLAN was written in C, taking advantage of low-level capabilities provided by the Operating System for that language. As the Linux Kernel is also written in C, its API opens room for using disk read functions with very little overhead [12]. Hence, OptPLAN creates two test files (one for sequential reads and the other for random reads) with the same number of pages but with different contents, a strategy planned to avoid the Operating System caching mechanism. Those files are split in contiguous blocks with 512 bytes each – which are the same size of a standard disk page on Ext4, the file system used in our experiments. Nevertheless sequential reads or random reads are in use, the system reads one page at a time.

The second step in OptPLAN is to read those files. The sequential scan is performed first, by positioning the read pointer to the file beginning, i.e., to the first byte of the file. Sequential read operations are performed within a loop and the system halts when all pages were read.

Random read tests are more complex, because the pointer should not read contiguous pages. To simulate this behaviour, OptPLAN uses a array with the same number of positions that the number of pages kept in the file. That array is then shuffled and is seen as holding the positions to where the pointer should be moved. After the shuffle process is over, the array is scanned sequentially and, for each iteration, the page corresponding to the value kept on the current position is read. This process successfully simulates a table scan using a complete index transverse.

V. Experiments

This section explains how the experiments were conducted, their setup and the obtained results.

A. *Setup*

We conducted the experiments on 3 different hardware configurations, denoted here as Maq1, Maq2 and Maq 3 (All computers were running the Ubuntu Linux OS, version 14.04 LTS):

TABLE I: TEST RESULTS OF OPTPLAN READS.

File \Server	Maq1	Maq2	Maq3
102.4MB	1.359	1.332	1.334
204.8MB	1.419	1.417	1.551
307.2MB	1.423	1.429	1.556
409.4MB	1.461	1.467	1.650
512MB	1.491	1.483	1.605
1024MB	1.540	1.522	1.707
Mean	1.449	1.441	1.567

- Maq 1: a notebook with an Intel Core i7 Quad Core Mobile Processor i7-4700MQ, clocked at 2.40GHz 6MB of cache, 8GB RAM DDR3 (1600MHz) and with a 120GB Samsung SSD 120GB;
- Maq 2: a desktop with AMD Phenom(tm) II X4 955 Quad Core with 512KB cache, 4GB RAM and a 1TB Seagate HDD 7200 RPM;
- Maq 3: a desktop with an Intel Core i5 Dual-Core Processor 4210U clocked at 1.7GHz, 3MB cache, 4GB RAM DDR (1600MHz) and a 500GB HD Seagte 5400 RPM.

Each server was running a stable version of PostgreSQL 9.4 and populated with the sample database DVD Rental [13].

We used three SQL statements on our experiments, each one with different complexity. Query Q1 returns all clients that rented any movies starring the actor with name Nick Stallone (a simple query example). On the other hand, Query Q2 retrieves the tuples containing clients that never rented any movies where the actor Nick Stallone starred (A subquery example). Finally, Query Q3 returns the names of those clients that rented all movies where actor Nick Stallone starred (a complex DIVIDE example). All SQL statements were analyzed using the PostgreSQL's EXPLAIN command, which prints the chosen execution plan calculated by the RDBMS along with the best and worst costs.

In order to detect the correct values for SPC and RPC, we conducted a series of experiments on each hardware configuration. The size of files used on our experiments varied from 102.4MB to 1024MB. Those values correspond to the number of pages in each file: the 102.4MB file has a total of 200,000 512 bytes pages. Each file was read 5 times, using both sequential and random reads strategies. At the end, the we calculated the median for each round to avoid anomalies on the final result. Finally, the mean of the values obtained using all files were used on *postgresql.conf* to replace the default values of seq_page_cost and random_page_cost.

B. Results

The results of our experiments comparing the execution plans generated using PostgreSQL default values for SPC and RPC against those values corrected by OptPLAN appear on the next graphics and table below. Table I describes the ratio between random and sequential reads median values in each server. We used 6 different files to simulate accessing tables of different sizes. The mean is calculated and the updated *random_page_cost* value is set at *postgresql.conf*.

Figure 3 shows the results we obtained for the best case for Q1. Note that the costs for executing the query after updating RPC value is lower than the default in all hardware configurations. Also, note that for different hardware configurations, OptPLAN calculated different RPC values, which leads to different costs. That is not true whether the default value of RPC is used, where all hardware configurations calculated the same execution plan.

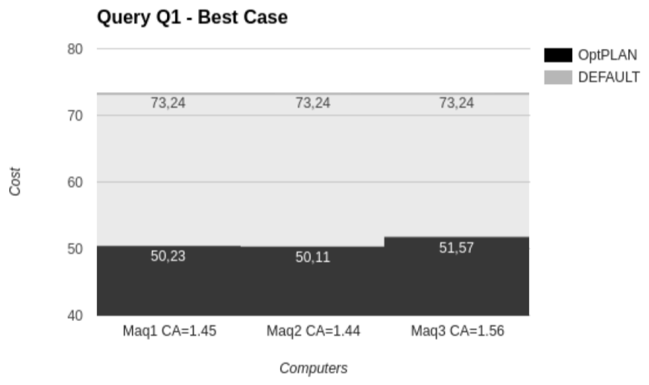


Fig. 3: Q1 - Best case

Figure 4 shows the cost values for the worst cases, where updated configurations have up to 15.3% of cost drop.

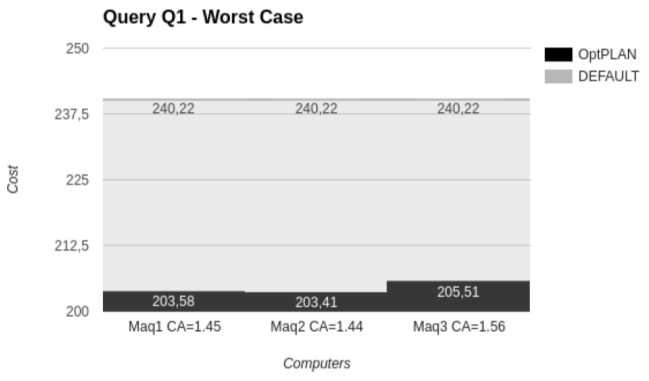


Fig. 4: Q1 - Worst case

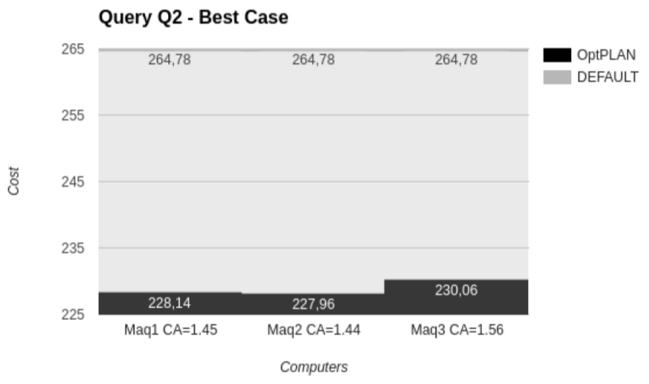


Fig. 5: Q2 - Best case

Figure 5 compares the costs for executing Q2 in the best case. Again, the scenarios where OptPLAN were used obtained better results – up to 13.9% of cost drop. Figure 6 compares

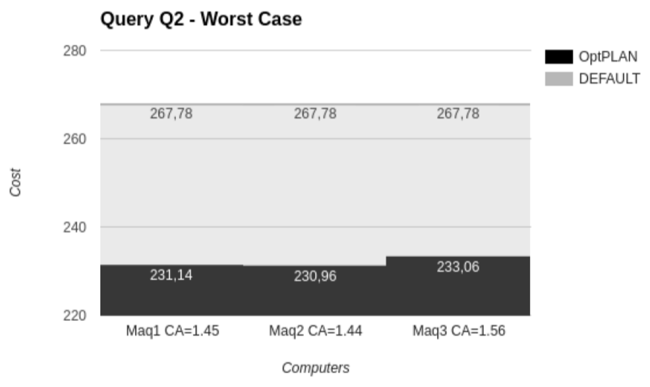


Fig. 6: Q2 - Worst case

the costs for the worst case, where OptPLAN caused a drop of up to 13.75% on the costs.

Finally, Figure 7 compares the results for the best cases on each server when executing Q3. Once again, OptPLAN utitlization resulted on better, (up to 33.4%) cheaper plans. The worst cases of Q3 plans are compared in Figure 8 and the modified RPC values resulted in up to 69% of cost drop.

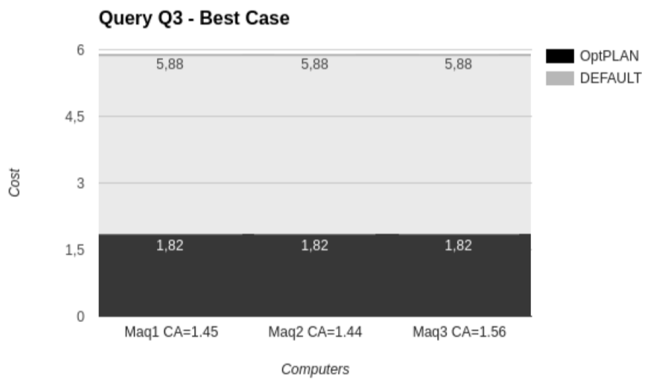


Fig. 7: Q3 - Best case

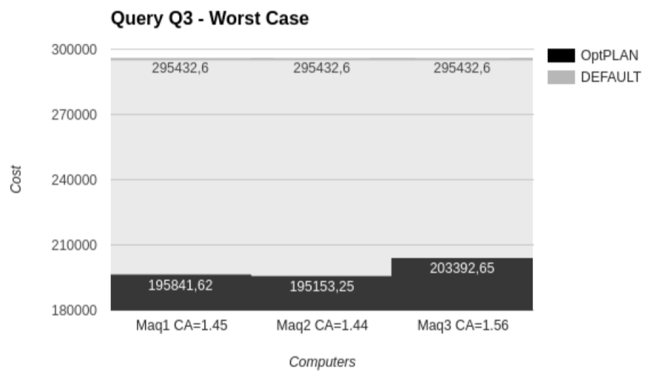


Fig. 8: Q3 - Worst case

VI. Conclusions and Future Works

In this paper, we discussed how the search for the best execution plan, i.e., the optimization of an SQL statement is affected by the hardware configurations where the RDBMS is deployed, as cost-based Query Optimizers highly depends on the capacity of that hardware to transfer data from disk to memory sequentially and randomly, execute in-memory comparisons, etc.. We focused our discussion on the PostgreSQL RDBMS due to its open source nature.

In that direction, we developed a tool called OptPLAN, that computes the speed that data is read from disk both sequentially and randomly – permitting to compute their normalized values that are used by PostgreSQL, seq_page_cost (SPC) and random_page_cost (RPC). Afterwards, we compared the costs of three SQL queries using both default and rectified SPC and RPC values in three RBDMS servers using different hardware configurations. Our results showed a performance gain of up to 69% and confirmed that is possible to obtain cheaper costs for SQL queries only by adjusting hardware related parameters.

As future works, we intend to implement a hybrid read approach, where sequential reads and random reads are performed in the same OptPLAN run. The main advantage of this combined method would be representing a more real behaviour of disk access. Also, we plan to better examine the effect of cache mechanisms on file access patterns and their impacts on OptPLAN. Finally, as our tool is tightly coupled to some functions provided by the Kernel of the Linux Operating System, we are currently evaluating the possibility to provide alternate OptPLAN implementations that work with MacOS and Windows.

References

[1] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*, 7th ed. Pearson, 6 2015.

[2] J. M. Hellerstein, M. Stonebraker, and J. Hamilton, *Architecture of a database system*. Now Publishers Inc, 2007.

[3] A. Deshpande, Z. Ives, and V. Raman, “Adaptive query processing,” *Found. Trends databases*, vol. 1, no. 1, pp. 1–140, Jan. 2007. [Online]. Available: http://dx.doi.org/10.1561/1900000001

[4] Y. E. Ioannidis, “Query optimization,” *ACM Comput. Surv.*, vol. 28, no. 1, pp. 121–123, Mar. 1996. [Online]. Available: http://doi.acm.org/10.1145/234313.234367

[5] PostgreSQL, “Query planning - planner cost constants,” http://www.postgresql.org/docs/9.3/static/runtime-config-query.html, 2016, [Online; acessado em 22 de Maio de 2016].

[6] S. Chaudhuri and V. Narasayya, “Self-tuning database systems: a decade of progress,” in *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007, pp. 3–14.

[7] B. Mozafari, C. Curino, and S. Madden, “Dbseer: Resource and performance prediction for building a next generation database cloud.” in *CIDR*, 2013.

[8] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin, “Automatic sql tuning in oracle 10g,” in *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. VLDB Endowment, 2004, pp. 1098–1109.

[9] S. Chaudhuri, E. Christensen, G. Graefe, V. R. Narasayya, and M. J. Zwilling, “Self-tuning technology in microsoft sql server,” *IEEE Data Eng. Bull.*, vol. 22, no. 2, pp. 20–26, 1999.

[10] G. Graefe, “Query evaluation techniques for large databases,” *ACM Computing Surveys (CSUR)*, vol. 25, no. 2, pp. 73–169, 1993.

[11] PostgreSQL, “Using explain,” http://www.postgresql.org/docs/9.3/static/using-explain.html, 2016, [Online; Accessed: 2016-05-22].

[12] R. Love, *Linux System Programming: Talking Directly to the Kernel and C Library*, 2nd ed. O’Reilly Media, 6 2013.

[13] “Dvd rental database,” http://www.postgresqltutorial.com/postgresql-sample-database/, 2016, [Online; Accessed: 2016-05-22].