Two Advanced Optimization Approaches for SQL Queries

Fangzhou Liu
University of Rochester
Computer Science Department
fliu14@ur.rochester.edu

Jiang Shang
University of Rochester
Data Science Department
jshang5@ur.rochester.edu

ABSTRACT

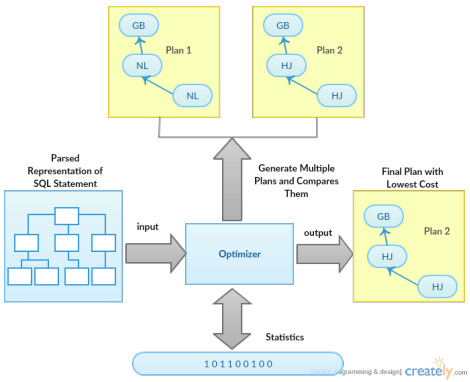
Today's complex world requires numerous data manipulations, over massive data sets stored in large database systems. It can be as simple as "finding the address of a person with SSN# 123-45-6789," or more complex like "finding the average salary of all the employed married men in California between the ages 30 to 39, that earn less than their wives". So, speeding up the query can significantly increase the efficiency and guarantee the service quality. In this paper, we mainly focused on the query optimization techniques from two perspectives, one is in heuristics level that can help speed up the SQL queries, the other are techniques that can accelerate the query during the compile or executing phase.

1. INTRODUCTION

Optimization has always been, and still is, a central topic in database research [1]. In database systems, the term Query refers to extract specific tuples or data according to some pre-set conditions. In modern database system, there embeds a specific software called Optimizer that mainly take the responsibility of Query optimization. Figure 1 shows the architecture of an Optimizer. Generally, it works with the following procedure:

1) Take a SQL Query statement as input.
2) Rewrite the SQL Query statement into a semantically equivalent statement with lower costs.
3) Generate a bunch of Execution Plans with the same semantic.
4) Selected the best plan with minimum cost calculated by an internal Estimator.
5) Executed the selected plan and output the queried data.

Fig. 1. Query Optimizer Architecture



Query Transformation is a simple level of query optimization, it firstly transfers the SQL statement into a Query Tree or Query Graph and then apply heuristic query-processing strategies to rewrite the query. The Query transformation was deployed based on the following rules: [1]

- Perform Selection/Projection operations early.
- Perform operations with smaller join first when doing consecutive join operations.
- Compute common expressions once and save result.

However, the query statement can be further optimized if we can further accelerate the Plan Selection time given that the number of candidate plans can grow exponentially as the query becomes more and more complex. Cited from [2], "*more time is spent in evaluating the query plan than actually calculating the query result*". The optimizer should strike a balance between selecting best performance, which will cost more time in plan selecting phase, and less selecting time, which will lead to less optimized plan. In addition, thus the Query statements will be finally translated to machine code, we can borrow the ideas coming from Complier Optimization Theory to further improve the performance of SQL Queries.

The next session will briefly introduce a robust algorithm that yields better performance when selecting plans for complex SQL queries with multiple JOIN operations. Then, we will introduce a novel system that combines the compiler technology with SQL optimizations, that can execute SQL queries more efficiently on large volumes of data.

2. SKYLINE DYNAMIC PROGRAMMING

One common algorithm that was applied for plan selecting is Dynamic Programming (DP). It can enumerate and identify the most optimal plan in a shorter time. However, the tradition DP Selecting algorithm only works well with moderately complex queries, when it comes to more complex queries, the performance drops, even though some refinement was added, like Iterative Dynamic Programming (IDP), which is doing DP iteratively with a significantly reduced subset of executing plans. In this session, we will introduce a more powerful DP algorithm called Skyline Dynamic Programming (SDP).

2.1 SDP Algorithm Detail

The novel metrics for this algorithm is that (a) It applies pruning on segment of the join graph, instead of the entire join graph. (b) It adopts a multi-way pruning strategy based on combination of Selectivity, Cardinality and Costs. So, comparing with the traditional DP algorithm, it saves both executing time and space.

2.1.1 Skyline Definition

Skyline is an operator that works as a filter in a SQL query. It keeps those objects that are not worse than others. For example, when buying international flight ticket from US to China, we would like to choose those flight with minimum stops, however, a direct flight without stopping is extremely expensive. In this case, the Skyline operator would only present those flight plan that are not worse than others in both price and number of stops.

2.1.2 Pruning Strategies in SDP

Next, we present the special pruning strategies in this algorithm. Before passing to this algorithm, each input, which is the *Join-Composite-Relations(JCRs)* will be tagged with a feature-vector includes the following attributes: [ROWS(R), COSTS(C), SELECTIVITY(S)]. Then, the algorithm will apply the Skyline operator mentioned before to filter objects based on their RC, CS and RS values. Then, all these three subsets were united and all JCRs that do not belong to this set will be pruned.

Table 1. Multi-way Skyline Pruning[1]

Input JCRs	Feature Vector [R, C, S]	Skyline		
		RC	CS	RS
1-2-3	[187638, 49386, 3.9E-5]	√	√	-
1-2-5	[122879, 52132, 1.0E-5]	√	√	√
1-3-5	[242620, 49386, 1.0E-5]	-	-	-
1-4-5	[241562, 55388, 6.65E-6]	-	-	√
1-5-6	[385275, 52632, 4.5E-6]	-	√	√

As Table 1 shows, the algorithm input is a lists of JCRs {1-2-3, 1-2-5, 1-3-5, 1-4-5, 1-5-6}, the survivor of this algorithm is {1-2-3, 1-2-5, 1-4-5, 1-5-6}, which was selected by Skyline Operator at least once based on three criteria. Here the JCR {1-3-5} will be pruned.

2.1.3 SDP Algorithm Running Step

Known the concept of skyline and the pruning strategy, we now present the running procedure for this SDP algorithm.

STEP 1: Apply the standard DP algorithm for the first iteration, select the best plan for each relation.

STEP 2: Enumerate each join relation pairs in standard DP and split these pairs into two groups: **PruneGroup (PG)** and **FreeGroup (FG)**, with the criteria that whether the **Join Composite Relations (JCR)** is a hub relation.

STEP 3: Apply the Skyline Pruning Strategy on PG and Apply the standard DP on FG until there are only two additional relations to be joined for each composite.

2.2 SDP Performance Evaluation

Here is the quantity evaluation of this algorithm. Table 2 and Table 3 show the overall optimization quality and overheads of this algorithm comparing with standard DP and IDP after applying that to a star-chain-15 query. In Table2, I means IDEAL solution, G means GOOD plan, A means ACCEPTABLE plan, B means BAD plan, W means WORST-CASE plan-cost increase ratio. We use the DP as the standard one.

Table 2. Plan Quality (DP, IDP, SDP)

Query Join Graph	Technique	Plan Quality (%)				
		I	G	A	B	W
Star-Chain-15	DP	100	0	0	0	1
	IDP	2	44	54	2	10.9
	SDP	80	20	0	0	1.2

In this table (Table 2), IDP can select only 2% IDEAL plans and it will select about 56% percent of the inefficient query plans, while for SDP, all plans it selects located in the Good region or beyond. This means that the SDP algorithm lead to more efficient queries.

We can see from the other table (Table 3) that SDP has better performance with less Memory and Time consumption.

Table 3. Optimization Overheads

Query Join Graph	Technique	Memory (in MB)	Time (in sec)	Costing (in plans)
Star-Chain-15	DP	32.39	1.00	8.3E5
	IDP	7.39	0.20	1.3E5
	SDP	4.33	0.10	0.5E5

Furthermore, when it comes to scaling queries with more join nodes, the performance is far better than other DP algorithms. Table 4, 5 show the optimization results when applying to star-chain-23 query. Here we let the SDP result as the standard one. From Table 4, it is obvious that the SDP works significantly better than IDP whose selected plan all located below Good level.

Table 4. Scaled Plan Quality

Query Join Graph	Technique	Plan Quality (%)				
		I	G	A	B	W
Star-Chain-23	DP	*	*	*	*	*
	IDP	0	0	12	88	25.3
	SDP	100	0	0	0	1

Table 5. Scaled Optimization Overheads

Query Join Graph	Technique	Memory (in MB)	Time (in sec)	Costing (in plans)
Star-Chain-23	DP	*	*	*
	IDP	460.37	54.7	4.5E6
	SDP	55.33	1.08	0.4E6

[1] Data for Table 1-5 comes from [3]

3. QUERY OPTIIZATION SYSTEM

Now we can go a step deeper to see whether we can apply some compiler optimization technology when transferring the SQL code into machine code. According to [4], the current query processing model doesn't fully utilize the advantage of modern system architecture with large main memory space, faster CPUs, register and caches. So, it puts forward a system that enhance the SQL performance.

3.1 Compilation-level Optimization

In modern database systems, queries will be finally interpreted into executable code by an internal interpreter, followed the iterator-model described in [2]. Basically, each SQL operator consists of a combination of open (), next()[2] and close() methods. However, as the weakness of most Dynamic Languages, which were interpreted by an interpreter, whose performance are lower than those static programming languages, which was compiled by a compiler, SQL execution speed can also be improved by utilizing compiler optimization techniques. In the meanwhile, getting the benefit of modern CPUs pipelining and SIMD (Single Instruction Multiple Data) instructions.

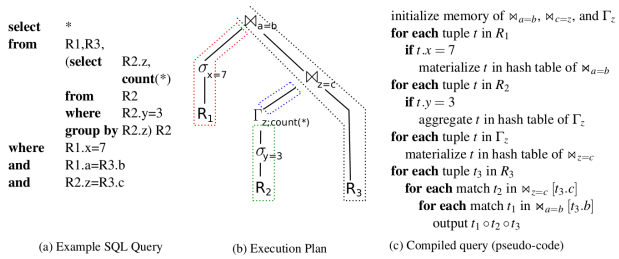
3.1.1 Vectorization

Vectorization is a process that can transfer a bunch of series executed instructions into vectorized expression, processing one or more input arrays and store the result in an output array homogeneously. In DBMS, instead of calling functions for each single tuple, we now calling the same function with a block of tuples. Thus the data is represented in single-dimension array format. So, using vectorization, we can significantly accelerate the data accessing rate. Work [2] doing the performance evaluations for vectorization processing in PROJECT, SELECT and HASH JOIN operations. The result is that the vectorization can produce the best executing performance for SQL queries combined with compilation.

3.1.2 Data Centric Query Compiling

Instead of passing data between operator to the other, this compiling technique maximize the data locality by keeping the attributes in CPU register as long as possible by introducing the pipeline. When data was loaded, it will pass through all operators that can work on it directly until meet a pipeline breaker. All operators perform their work, but do not write their result back to the memory. The goal of this approach is to access the memory as rarely as possible thus the memory access is quite expensive. That's also the reason why we introduced the index or B tree structure when processing the queries.

Fig. 2 From SQL to Executable Code



(a) Example SQL Query (b) Execution Plan (c) Compiled query (pseudo-code)

[2] next(): produces one new tuple

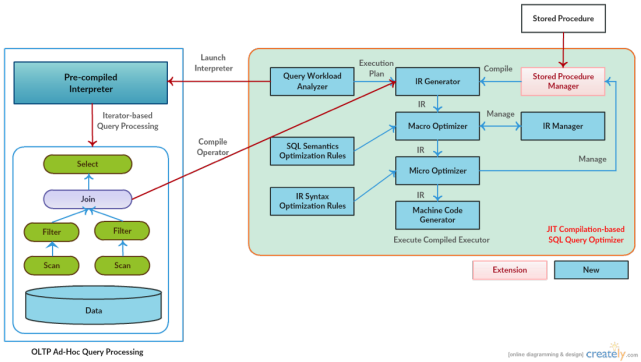
Figure 2 shows the process how the code was generated based on a SQL query in this method. The region marked by each color indicates different pipeline fragment. Tuples will flow through these fragment and kept in registers.

The performance of this compiling methods was given by [5]. The conclusion is that this Data Centric Query execution, with produce/consume model and LLVM compiler backend, works efficiently.

3.2 System Architecture

By analyzing previous researches, we here also introduce a JIT (Just-in-time) based DBMS systems. This system transforms SQL queries into machine independent IR at the query plan or operator level and then applies various optimizations on it and finally output the machine code generated by JIT compiler. Figure 3 presents this system architecture. Next, we'll briefly explain some components that were newly added in this new system.

Fig. 3. System Architecture Overview



- **Query Workload Analyzer** decides whether user's ad-hoc query is JIT applicable or not.
- **IR Syntax Optimization Rules** provides optimization rules considering only on the IR level, not the SQL semantics.
- **IR Manager** caches IRs generated from previous SQL and provides the IR if the same SQL query are executed**.**
- **Micro Optimizer** is the place where the IR Syntax Optimization Rule was applied. Some optimizations like SIMD and pipelining are performed here.

4. CONCLUSION

In this paper, we first analyze the current SQL optimization methods and then discuss two additional ways in which we can further accelerate the query speed, one is through plan selecting and the other is through query executing. For the first one, we introduce a Skyline Dynamic Programming (SDP) that can improve the performance comparing with the standard DP algorithm, especially in complex query conditions which contains numerous JOIN operations. Then, for the query executing part, we explain some modern compilation-level optimization approaches, like vectorization and pipelining, that can significantly accelerate the executing speed for SQL queries in modern hardware architectures. And in the end, we put forward a refined SQL optimization system

that combines the current SQL Optimizer with the JIT Compilers together when optimizing the query statements.

5. REFERENCES

[1] Danda Li, Lu Han; Yi Ding, "SQL Query Optimization Methods of Relational Database System," *Second International Conference on Computer Engineering and Applications*, pages 557-560, 2010

[2] J. Sompolski, M. Zukowski, and P. Boncz, "Vectorization vs. Compilation in Query Execution," Proc. of DaMoN '11, New York, NY, USA, 33-40, 2011

[3] Gopal Chandra Das, and Jayant R. Haritsa, "Robust Heuristics for Scalable Optimization of Complex SQL Queries," *IEEE 23rd Internation Conference on Data Engineering*, pages 1281-1283, 2007

[4] Myungcheol Lee; Miyoung Lee; ChangSoo Kim. A JIT Compilation-Based Unified SQL Query Optimization System. In *6th International Conference on IT Convergence and Security (ICITCS).* 2016

[5] T. Neumann and V. Leis, "Compiling Database Queries into Machine Code," IEEE Data Engineering Bulletin, vol. 37, 2014

[6] G. Graefe and W. J . McKenna, "The Volcano Optimizer Generator: Extensibility and Efficient Search," Proc. of ICDE '93, pages 209-218, 1993

[7] Query Optimizer, *Oracle Database SQL Tuning Guide*, https://docs.oracle.com/database/121/TGSQL/tgsql_optcncpt.htm#TGSQL194, Figure 4-1, Figure 4-2

[8] S.B. Navathe, R. Elmasri, *Fundamental of Database Systems 7th Edition,* Chapter 19.1, Chapter 19.5.5, pp 692, pp 725-726

[9] Borzsonyi, Stephan; Kossmann, Donald; Stocker, Konrad (2001). "The Skyline Operator". *Proceedings 17th International Conference on Data Engineering*: 421–430.