Database Resource Allocation Based on Resilient Intermediates

Martin Kersten, Ying Zhang, Pavlos Katsogridakis,
Panagiotis Koutsourakis and Joeri van Ruth
MonetDB Solutions
Amsterdam, The Netherlands
<*lastname*>@monetdbsolutions.com

***Abstract*—Scale-out of big data analytics applications often does not pay off due to the poor performance in response time and the increasing bill due to a longer execution time on a resource limited machine. To enable a stable DBMS workload environment it helps to maintain several virtual machines hosting part of the database, so that users can send their tasks to those machines that have the best price/performance characteristics. This, however, requires a method to decide which VM should be used for a given query, and the technology needed to do this is the focus of this paper.**

I. Introduction

Since its inception, database designers have keenly looked at the opportunities to use large, distributed processing platforms. Cluster-based products are readily available, such as in appliance products from Oracle Exadata, SQL Parallel Data Warehouse, IBM Blu and Teradata, but they are often limited to a few tens of compute nodes. A plethora of research activities has shown that in all but the simplest cases achieving a good performance is at least hard, especially when a query involves joins spread over multiple compute nodes and thus requires expensive data exchange.

The predominant way out nowadays, taken by NoSQL systems such as Casandra and Impala, is to address part of the problem space by focusing on select-aggregate queries. This choice has proven to be pivotal to support big data analytics in many real-world circumstances, as shown by the widespread use of Apache Spark. The basic abstraction in Spark is a Resilient Distributed Dataset (RDD), which represents an immutable and partitioned collection of elements that can be operated on in parallel using operators, such as map, filter, persist and aggregates.

Although in many cases it is easy to scale-up for improved response time, partitioning a database to benefit from a low cloud service price tag and to overcome resource limitations of smaller machines is still a much sought after skill. This product space is addressed by Snowflake and AWS Redshift. Snowflake has been designed from a cloud perspective, taking resource management as its key driving factor. It conceptually provides every user with a complete copy of the database and relies on multi-level caching. AWS Redshift is an improved version of PostgreSQL, which has been further tuned towards better I/O bandwidth use.

In this paper we take a fresh look at resource allocation for query processing in the context where intermediates in a query plan are fully materialised before passed on towards the next operator. This model fits not only the Apache Spark programming model, but also the query execution model of our database system MonetDB(www.monetdb.org). Resilient intermediates provide new avenues for query optimisation and scheduling as its underlying computation model is based on materialisation of all intermediate steps.

The main contributions of this paper are

- we develop a simulator, called MALCOM to predict the memory footprint for queries based on resilient intermediates in MonetDB.
- We demonstrate that the approach is robust against varying data distributions.
- We demonstrate the opportunities using an extensive evaluation against TPC-H and a real-world data set.

The approach taken here differs from traditional cost-based query optimisers deployed in distributed database systems by learning about actual resource claims over time, i.e. after each query execution we have precise knowledge of the resources used. This information can be harvested and used to predict future operations of a similar nature. The rationale stems from the common knowledge that any database application environment has a limited number of 'business transactions' or 'business intelligence templates' where only some parameters are changed with each call.

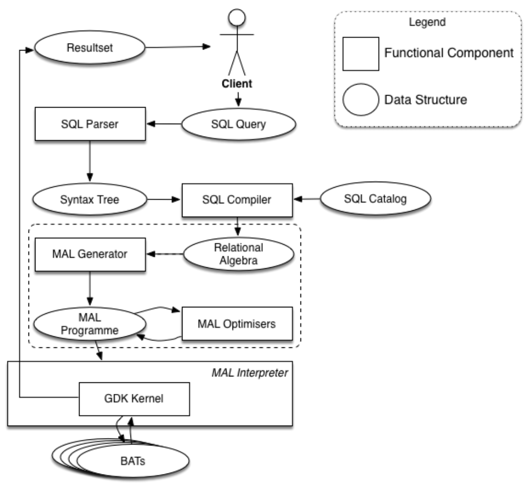

Fig. 1. MonetDB query execution architecture.

Paper Outline. Section II provides a short overview of the MonetDB architecture.Section III introduces the components and algorithms for our resource estimator MALCOM. Section IV illustrates the effectiveness of our approach in two use cases: TPC-H and air traffic.

II. Background

In this section we introduce the query execution engine of MonetDB and the performance information available for our task.

A. MonetDB Architecture

MonetDB is a widely used columnar DBMS that internally uses resilient intermediates to break up query processing in well identifying steps. A query plan is broken up into independent steps, glued together into a dataflow dependency graph. The dataflow graph is greedily consumed by the database kernel assigning a dedicated core to each operation. The resource pressure is kept at a minimum by trimming down the degree of parallel processing when the main memory resource is heavily used.

Figure 1 illustrates the components of MonetDB to execute an SQL query. MAL (MonetDB Assembly Language) is the MonetDB internal language into which SQL queries are compiled and executed. The SQL Parser and MAL Optimiser deploy well-known rewriting rules (e.g. parallelisation, and dead code/common expression/constant elimination) to reduce the intermediate sizes and processing time. They do not rely on any cost-model or pre-computed statistics.

The middle layer (in the dashed box) is a sequence of specialised optimisers that morph a logical plan produced by the SQL compiler into a physical execution plan containing relational operators expressed in MAL statements. The bottom layer (under the dashed box) contains the implementation of the MAL statements. Each operator takes as input the resilient intermediates produced by operators executed before or the persistent data on disk.

As an example, consider this simple SQL query: `SELECT COUNT(*) FROM _tables`, which is eventually translated into a physical execution plan in MAL statements to be executed by the MonetDB kernel. The box below show an excerpt of the plan:

```
C_5=<tmp_1524>[92]:bat[:oid] := sql.tid(
     "sys":str, "_tables":str);
(X_13=<sql_empty_oid_bat>[0]:bat[:oid],
X_8=<tmp_147>[99]:bat[:int] := sql.bind (
     "sys":str, "_tables":str, "id":str);
X_17=<tmp_1477>[92]:bat[:int]:=algebra.projection(
     C_5=<tmp_1524>[92]:bat[:oid],
     X_8=<tmp_147>[99]:bat[:int]);
X_18=92:lng:=aggr.count(X_17=<tmp_1477>[92]:bat[:int]);
barrier X_72=false:bit := language.dataflow();
sql.resultSet("sys.L3":str, "L3":str, "bigint":str,
     64:int, 0:int, 7:int, X_18=92:lng);
```

The MAL language is purely designed as intermediate language to express the operations. Generally, a MAL statement is an assignment of the form:

```
VAR=<FILENAME>[COUNT]:VAR_TYPE :=
     MOD.FUNC(PARAM1=<FILENAME>[COUNT]:PARAM1_TYPE, ...);
```

Every function belongs to a module. The arguments are either typed scalar values (`:type`) or a reference to a column (`:bat[:type]`). If a variable (`VAR`) refers to a column, which can be memory mapped, it is also tagged with its base `FILENAME` on disk and the number of values in this column (`COUNT`).

The above MAL program is straightforward. It first loads (`sql.bind`) and projects (`algebra.projection`) the data of one column (`_tables.id`) from the disk. Then the data is passed to `aggr.count` to compute the `COUNT`. Finally, `sql.resultSet` emits the query result. MAL statements are important data for MALCOLM: before query execution, it gives memory footprint estimation per MAL statement; after query execution, it collects execution statistics per MAL statement.

B. MonetDB Profiling Information

The MonetDB kernel can be instructed to emit profiling events for the execution of MAL statements, e.g. by establishing a connection using MonetDB's profiling tool. Figure 2 shows the before/after profiling events produced for the `algebra.projection` operation when executing the

JSON object at “start” time	JSON object at “done” time
{“source”:“trace”,	{“source”:“trace”,
“ctime”:1528314717302449,	“ctime”:1528314717302680,
“module”:“algebra”,	“module”:“algebra”,
“instruction”:“projection”,	“instruction”:“projection”,
“state”:“start”,	“state”:“done”,
“usec”:0,	“usec”:230,
“rss”:87,	“rss”:87,
“size”:0,	“size”:0,
“nvcsw”:1,	
“stmt”:”X_17. . . := **algebra.projection**(. . .);”,	“stmt”:”X_17. . . := **algebra.projection**(. . .);”,
“ret”:[{	“ret”:[{
“index”:“0”,	“index”:“0”,
“name”:”X_17”,	“name”:”X_17”,
“alias”:“sys._tables.id”,	“alias”:“sys._tables.id”,
“type”:“bat[:int]”,	“type”:“bat[:int]”,
	“kind”:“transient”,
“count”:“0”,	“count”:“92”,
“size”:0,	**“size”:368,**
“eol”:0}],	**“eol”:0**}],
“arg”:[{	**“arg”**:[{
“index”:“1”,	“index”:“1”,
“name”:”C_5”,	“name”:”C_5”,
“kind”:“transient”,	“kind”:“transient”,
“bid”:“863”,	“bid”:“863”,
“count”:“92”,	“count”:“92”,
“size”:736,	“size”:736,
“eol”:1},	**“eol”:1**},
{“index”:“2”,	{“index”:“2”,
“name”:”X_8”,	“name”:”X_8”,
...	...
“eol”:1}]}	**“eol”:1**}]}

Fig. 2. JSON profiling objects produced for an `algebra.projection` operation.

MAL program above. The left column shows the event at the `"start"` and the right column the event when the operation is `"done"`. Differences between the two objects are marked in red. Of most interest to our estimation are the properties shown for the arguments (`"arg"`) and return variables (`"ret"`). For instance, in a `"ret"` object, the field `"size"` is a good estimation of how much memory the result set of this function consumes; while in an `"arg"` object, the field `"eol":1` indicates this argument has reached its end-of-life. This information together with the `"size"` allows us to estimate how much memory is freed after this operation.

III. MALCOM Micro Models

With an abundance of profiling events we can derive a micro-model for each MAL instruction to estimate their footprint. The goal of MALCOLM is, given a MAL execution plan of an SQL query, to estimate the resource needs by *only using information from our memory footprint estimation model.*

The algorithm to estimate an upper bound of the memory needed to execute a MAL plan, is shown in pseudo code below. When a MAL plan is received, MALCOLM first annotates each MAL statement with an estimation of how much memory it will consume (`i.mem_fprint`) and release (`i.free_size`). Then the algorithm iterates over the MAL plan (i.e. the `mal_statements` below). At each iteration, it adds the memory footprint of this MAL statement (`i.mem_fprint`) to the current total memory consumption (`curr_mem`) and updates `max_mem` if necessary. After that, it adjusts `curr_mem` with the amount of memory that will be freed by this statement (`i.free_size`).

```
max_mem  = 0, curr_mem = 0
for i in mal_statements:
  curr_mem += i.mem_fprint
  max_mem = max(max_mem, curr_mem)
  curr_mem -= i.free_size
```

After the execution of the MAL plan, we update our memory footprint estimation model with the observed execution information. We initialise the estimation model with basic column statistics (`min`, `max`, `count`, etc.) that can be gathered using MonetDB’s `ANALYZE` command .

The estimation model is built by dividing MAL instructions with similar functionality (most of them represent a relational operator each) into several groups and abstracting away their specific signatures. Currently, the model includes ~10 groups. We briefly analyse each of them below. Note that we only consider bulk operators here (i.e. taking columns as operands), which are the default ones in MonetDB.

A. Load instructions

A `bind` instruction loads (or memory maps) a column into memory, thus the resturn size is the size of the column. This is a worst case assumption, because in practice not all of the column needs to be loaded.

B. Arithmetic Operators

These operators always return the same number of values as their operands (MonetDB requires both operands to have equal size). However, the data type of the output can be a larger-sized data type than both operands to capture possible overflow. Hence, their result size is computed as:

```
arith.rsize = sizeOf(arg1) * sizeof(ret.datatype)
```

C. Aggregate Operators

This category includes operations such as `sum`, `avg`, `min`, `max`, `count`, `single` and `dec_round`. The number of values returned by these operators equals the number of groups in which the input data column is divided (by earlier `GROUP BY` statements, or 1 if there is no `GROUP BY`). The most general signature of these operators takes two operands: `arg1` is a column containing the actual values to work on; `arg2` is a column containing the group IDs, one for each value in `arg1`. The output size of an aggregate operator is computed by multiplying the number of unique values in `arg2` with the size of the return data type.

```
aggr.rsize = COUNT(DISTINCT arg2) * sizeof(ret.datatype)
```

D. Limit Operators

This group includes `firstn` and `sample`. They return at most *N* values from its input column `arg` as specified by the limit. Hence, their output size is computed as:

```
limit.rsize = MIN(COUNT(arg), N) * sizeof(arg.datatype)
```

E. Grouping Operators

MonetDB currently has 24 grouping operators for different situations. For instance, the position of the input data column (`arg1`) in a `GROUP BY` SQL clause determines the use of a `GROUP` operator or a `SUBGROUP` operator in MAL. More variations of `GROUP` or `SUBGROUP` operators are used depending on the availability of auxiliary information (e.g. some statistics of the input column). However, all grouping operators generally return three columns of results: (i) a `groups` column containing the group IDs, one for each value in `arg1`; (ii) an `extents` column containing the `OID` (MonetDB internal type for Object Identifiers, denoting positions of data values in a column) of a representative of each group; and (iii) a `histo` column containing the number of values in each group corresponding the values in `extents`. The data type of `groups` and `extents` are both `OID`, and the data type of `histo` is `LNG` (MonetDB internal type for Long integers). The number of values in `extends` and `histo` is the same, and is estimated using a simple kNN algorithm based on the statistics of previous queries or basic statistics of the involved columns. Putting everything together, the total output size of a grouping operator is estimated as:

```
group.rsize = COUNT(arg1) * sizeof(OID) +
  estimate_nr_groups(arg1) * (sizeof(OID) + sizeof(LNG))
```

F. Set Operators

For the set operators we mostly compute an upper bound of the result size using heuristics:

```
unionall.rsize = sizeof(arg1.datatype) *
  (COUNT(arg1) + COUNT(arg2))
union.rsize = sizeof(arg1.datatype) *
  (COUNT(DISTINCT arg1) + COUNT(DISTINCT arg2))
ntsct.rsize = sizeof(arg1.datatype) *
  MIN(COUNT(DISTINCT arg1) + COUNT(DISTINCT arg2))
xcpt_all.size = sizeof(arg1.datatype) * COUNT(arg1)
xcpt.size = sizeof(arg1.datatype) * COUNT(DISTINCT arg1)
```

Both `UNION` and `UNION ALL` return a concatenation of their two input columns `arg1` and `arg2`, except that `UNION` eliminates the duplicates in its result. Hence, `unionall.rsize` is the precise result size of a `UNION ALL`, while `union.rsize` is an upper bound of the result size of a `UNION`, because its computation does not exclude unique values that exist in both `arg1` and `arg2`.

`INTERSECT` returns values that exist in both its input columns `arg1` and `arg2` with duplicates eliminated. Hence, `ntsct.rsize` is an upper bound of the result size, as its formula does not exclude unique values that are only in `arg1` or only in `arg2`.

Both `EXCEPT` and `EXCEPT ALL` return all values that are in its first input column `arg1` but not in the second column `arg2`. In addition, `EXCEPT` eliminates duplicates. Therefore, both `xcpt_all.size` and `xcpt.size` are upper bounds of their respective result sizes, since their computations does not exclude (unique) values that also exist in `arg2`.

G. Projection Operators

Projection operators extract a small part of a column. The arguments are a candidate list `cand` containing the OIDs of the to-be-projected values and a reference to the (persistent) column `col`. The number of elements in the output equals the number of elements of the candidate list. Hence, their exact output size is computed as:

```
proj.rsize = COUNT(cand) * sizeof(col.datatype)
```

H. Selection Operators

This operator group includes the filter operations `theta-select` and `select`. For these operators, we know that the output is always smaller than or equal to the candidate tuples considered. To estimate the result size, traditional cost-based models assume a uniform distribution of the data and calculate the fraction of the domain, i.e. the selectivity factor. In practice, however, the uniform distribution of the data assumption does not always hold, so these models have limited accuracy. In our model, we keep the results of a series of actual filter operations, so as to use them to find a "historical nearest-neighbor" for any filter operation in a MAL plan whose cost we need to estimate.

The select operators are abstracted into a single template with three operands `sel(col, range, op)`, where `col` is a reference to the (persistent) column, `range` is the selection range (low, high), and `op` is the comparison operator ($<$, $>$, $<=$, $>=$, etc). In our model, we keep a dictionary of all the selections executed so far in the format of this signature, with an extra attribute `cnt` to denote the number of values selected.

The estimation for a selection operator `sel(col, range, op)` works as follows. First, we find in the `dictionary` records of all previous selections on the same `col` with the same `op`. Then, we use a *k* nearest neighbour (kNN) procedure to find the 5 nearest records based on the selection `range`. Next, for each of the 5 records, we extrapolate the number of selected values based on the selectivity and input column size. Finally, we compute the estimated memory footprint of this selection as the average of the 5 extrapolations, multiplied by the data size of the input column. This estimation procedure is shown in the pseudo code below:

```
extrap = 0
for dict in kNN(dictionary, sel, 5)
  extrap += dict.cnt * (COUNT(sel.col) / COUNT(dict)) *
             (sel.range / dict.range)
sel.rsize = extrap/5 * sizeof(sel.col.datatype)
```

I. *Join Operators*

For a cross product of two columns (`col1`, `col2`) we know it will return `COUNT(col1) * COUNT(col2)` number of values. The `cross product` operator of MonetDB takes two data columns as its inputs, and returns two columns where each column contains OIDs referring to data values in an input column. The two OID columns together denote how the values from the input columns are aligned in the cross product result. So the exact output size of a cross product is computed as:

```
cp.rsize = COUNT(col1) * COUNT(col2) * sizeof(OID) * 2
```

The estimation model for the other join operators is similar to that of selection operators. Again, the signatures of all join operators can be abstracted into a single one with three operands `join(col1, col2, op)`, where `col1` and `col2` are the input column, and `op` the join operator (eq, left, outer, etc). We also keep a dictionary of all the previous joins in the format of this signature, annotated with an extra value `cnt` to denote the number of values returned by that particular join operation. To estimate the result size for a join operator `join(col1, col2, op)`, we first find in the dictionary records of all previous selections on the same columns with the same `op`. Then, we run a kNN to find the 5 nearest records based on the sizes of the input columns. Finally, we extrapolate the result count based on the input column sizes, and compute the result size (like cross product, two OID columns are returned). The pseudo code is shown below:

```
extrap = 0
for dict in kNN(dictionary, join, 5)
  extrap += dict.cnt * (COUNT(join.col1) / COUNT(dict.col1)) *
                       (COUNT(join.col2) / COUNT(dict.col2))
sel.rsize = extrap/5 * sizeof(OID) * 2
```

Optimizer Simulator Architecture. With the micro-models in place, we can use a simple MAL-simulator to obtain a fairly accurate indication of the memory footprint. In its basic form it simulates a sequential execution of the query. The full-blown version uses the same scheduling method as within the MonetDB code base to approximate the parallel behaviour.

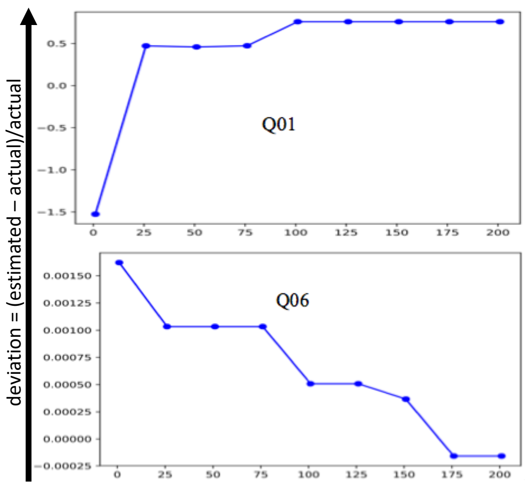

Fig. 3. TPC-H queries Q1 and Q6 prediction deviations

IV. Evaluation

In this section we discuss the results obtained with the MALCOLM prototype. For our experiments we used both TPC-H and the air traffic benchmarks. The former is the baseline against which most database systems are evaluated. Its major weakness is the uniform data distribution. Although TPC-H simplifies the analysis of the quality of a space predictor, it is not representative of real-life workloads. Therefore, we also use the air traffic benchmark[1], which consists of a single table with $>$120M rows and 100 columns of flight information. The data in this benchmark is skewed.

A. *TPC-H*

To test MALCOLM against TPC-H we created a query generator, which changes the parameters in the official benchmark queries. A key factor is to understand how long it takes before MALCOLM's prediction converges to the actual memory footprint. As a training set for each query, we randomised every selection point and range to produce 200 random versions of the query. As a test set we used the original queries.

Initial experiments led to the results such as shown in Figure 3, where the x-axis shows the number of training queries used and y-axis shows the deviation of the estimation from the actual footprint. Q1 is a simple large scan followed by an aggregation. The sole parameter is the value range. The experiment shows a steep learning curve,

[1]github.com/MonetDBSolutions/airtraffic-benchmark

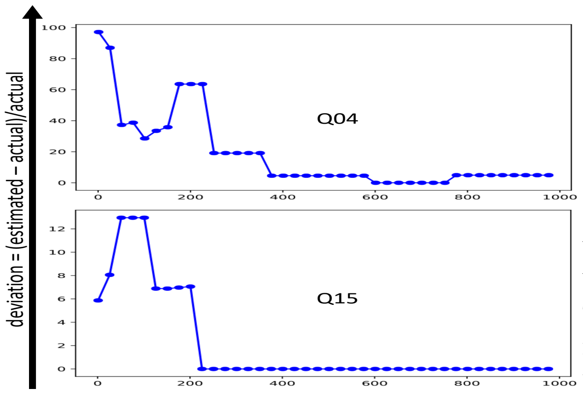

Fig. 4. Air traffic queries 4 and 15

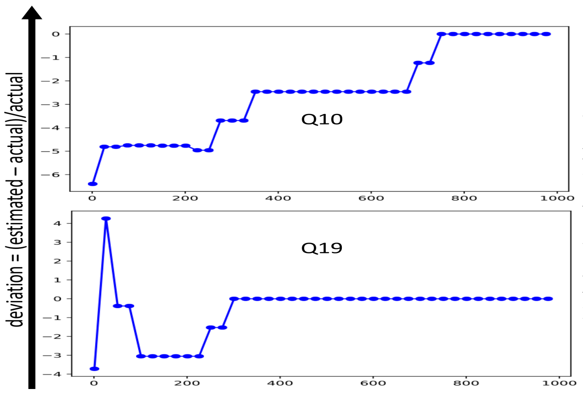

Fig. 5. Air traffic queries 10 and 19

which can be attributed to the uniform distribution. However, it also results in a little over fitting. Q6 is also mostly a simple scan and aggregate query, but here the number of tuples are less. This experiment shows that MALCOLM takes much longer to reach an almost perfect prediction of the memory footprint.

B. *Air Traffic*

The air traffic benchmark is a real-world example of business intelligence application. The data is represented by a single table, and it is highly skewed and sparse. The queries are mostly simple select-group-aggregate, but given the table size still hard to process. Figure 4 and Figure 5 show some of the results. Again we can observe both some under- and over-estimations, which all improve over time. Albeit they take more queries to stabilise. The graphs of Q04, Q15 and Q19 also illustrates how MALCOLM not necessarily improve monotonic. This is most probably due to the actual query load.

V. Related work

The approach taken in this project can best be compared with the long tradition in database query optimisers and database design wizards. They all collect query traces from an actual production system and use them to derive e.g. an optimal set of search accelerators [1]. This process seeks a balance between index creation and maintenance, but primarily deals with performance optimisation. The memory footprint is of less concern. Alternatively, it extends the work on gathering query traces to (semi-)automatically improve the cost model for query optimisation [2]. A better statistics improves both performance and resource use. All these systems are focused on a relative small and fixed compute cluster or database appliance.

In a more recent project [3] the authors gather sub-plans from the query trace log and use it as the building block for new queries. They show that reuse of good plans, i.e. based on past behaviour, leads to both a faster optimisation step and overall better performance. MALCOLM does not address the optimiser itself, but assumes that a physical plan has already been produced. It merely determines which virtual machine can handle the plan comfortably.

VI. Summary and Outlook

The MALCOLM prototype is a crucial tool in the design of cloud-based database management solution. The initial results are promising, a rather low number of queries are sufficient to get a good memory footprint estimate.

In the near future, we plan to extend MALCOLM to also look at the memory footprint in relationship of the parallel execution. Both memory footprint and degree of parallelism enables users to optimise their systems based on costs or response times.

Acknowledgments

This research has received funding from the European Union's Horizon 2020 research and innovation program under Grant Agreement no. 732366 (ACTiCLOUD).

References

[1] S. Chaudhuri and V. R. Narasayya, "Self-tuning database systems: A decade of progress," in *Proceedings of the 33rd International Conference on Very Large Data Bases, September 23-27*, 2007, pp. 3–14.
[2] V. Markl, G. M. Lohman, and V. Raman, "LEO: an autonomic query optimizer for DB2," *IBM Systems Journal*, vol. 42, no. 1, pp. 98–106, 2003.
[3] B. Ding, S. Das, W. Wu, S. Chaudhuri, and V. R. Narasayya, "Plan stitch: Harnessing the best of many plans," *PVLDB*, vol. 11, no. 10, pp. 1123–1136, 2018.