Volume 5, Issue 12, December 2015 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

In-Memory Database Processing and Executing Engine

Bineeta Kumari Gupta
Oracle India Pvt. Ltd
India

Abstract: ***In- memory database primarily relies on main memory for computer data storage. Accessing data in memory eliminates seek time when querying the data, which provides faster and more predictable performance than disk. As it accesses only the main memory of the machine, the size of the databases are thus limited, and many optimizations are involved for storage and query processing. Query optimization process is responsible for planning statement executions with the purpose to take advantage of the strengths of the database management system. In this paper, we describe a SQL parser to generate execution plans optimized for the specific format: SELECT-FROM-WHERE queries. Also, it supports unions, intersections and nested queries of these forms. With enhancements in several areas – statistics, cost model, query transformation, access path and join optimization – the query optimizer plays a significant role in unlocking the full promise and performance of In-Memory database.***

Keywords— In-memory database, Query optimisation, Tree-reordering, B+ Tree, Join-Order Optimization

I. INTRODUCTION

In-memory databases are databases which use only the main memory (RAM) of the machine for storage. The size of these databases is thus limited, and many optimizations are involved for storage and query processing. The idea of this project was to explore optimization techniques for an in- memory database. Query optimization for in-memory databases is different from other databases, the major difference being that the disk seeks and write do not exist and thus their costs do not need to be considered while dealing with execution costs during optimizations.

The basic idea was to create a basic database processing and execution engine and efficiently implement query optimization techniques. Main focus was join-order optimizations using statistical sampling of table data for cost estimation. We have performed query tree reordering and join-order optimizations using statistical sampling of table data for cost estimation to find optimal execution plan for query.

The entire work is divided into Query processing, Optimisation and Efficient Execution.

II. DATABASE DESIGN

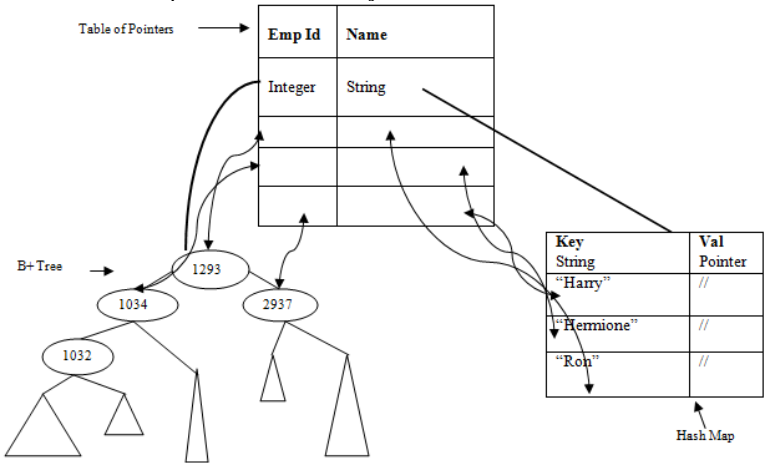
- **Data Types**

In our implementation we have considered for table entry fields, only two data types- Integer (INT) and String (VARCHAR). Integer and string data types would be used in separate contexts and for different purposes. For example, data entries can be sorted according to integer fields or integer field values may be used in comparisons.

On the other hand string fields would mostly be checked for equality. Other data types like real (float) or fixed char string can be handled in a similar ways.

- **Indexing**

As the implementation is in-memory, it is possible to index every field in the table while loading the tables to memory or inserting values into the tables. In our implementation, every field in the table is indexed as a secondary index.



Design for Storage and Indexing of Different Fields

For each INT field, a B+ Tree is created. Each node stores the integer value of the corresponding to the field in the data entry and a back pointer to the data entry of the table. Thus, for n data entries in the table, insertion, deletion and look-up are each O(log(n)). For each VARCHAR field, a hashmap is maintained which stores the string mapped to the back pointer to the data entry. In the table's data entry itself, individual string values are not stored, but only pointers to the position (iterator) of the corresponding entry in the hashmap. This saves space by reducing redundant storage.

III. QUERY PROCESSING

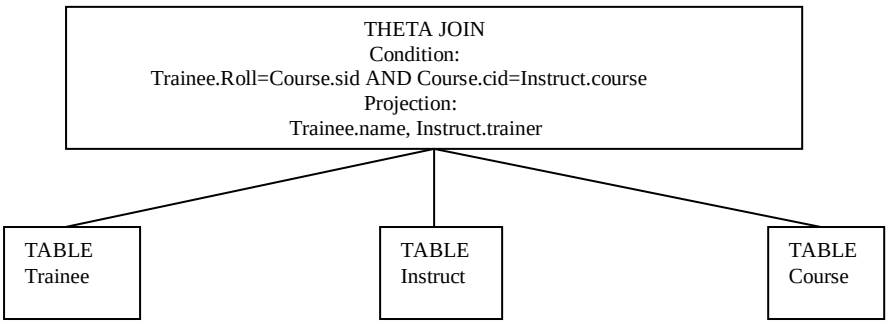
- **Query Tree Parsing**

First step, for any query is to get in the appropriate form. The input is taken from the standard in-put (stdin) in the form of SQL syntax queries. We support the basic - SELECT.. FROM.. WHERE- queries, unions, intersections, table renames and nested queries of these forms.

The query is stored in an object of query tree class. At the leaves of the tree are individual tables. Every other node is one of the types- union, intersect, except or theta join. For convenience of processing and optimisation, at every theta join node as well as at the table node (leaf), projection is done.

Query Q1 :

SELECT Trainee.name, Instruct.trainer FROM Trainee, Course, Instruct WHERE Trainee.roll=Course.sid AND Course.cid=Instruct.course



Tree Structure for Query Q1

- **Condition Tree**

The condition at the each theta join node, or at the leaf table node is given is also given in the form of a tree. The nodes of the condition tree are either table fields or values (integers or strings).

Query Tree Structure for Q1 (with condition tree)

query::

```
select:
        field: TRAINEE.NAME
         field: INSTRUCT.TRAINER
from:
      table: TRAINEE
      table: COURSE
      table: INSTRUCT
condition:
      [and]
         [=?]
            field: TRAINEE.ROLL
             field: COURSE.SID
         [=?]
            field: COURSE.CID
            field: INSTRUCT.COURSE
```

IV. QUERY OPTIMISATION

In our limited SQL grammar, we implement a optimizer which converts the parsed Query tree to an optimised, executable tree. The optimiser focuses on three aspects of optimisations:

- Percolating Selections
- Percolating Projections
- Join Order Optimisations

- **Query Tree Re-ordering**

1. Percolating Selections

We are reducing the size of the tree as early as possible through the selections so that the costly join operations operate on tables of smaller sizes. Another advantage of percolating conditions to the tables is using the index of the attributes, which decreases cost of execution. Selections are percolated down as per the equivalence rules. A selection after a join

can be made into a selection before a join if the attributes concerned in the selection process aren't involved in the join process.

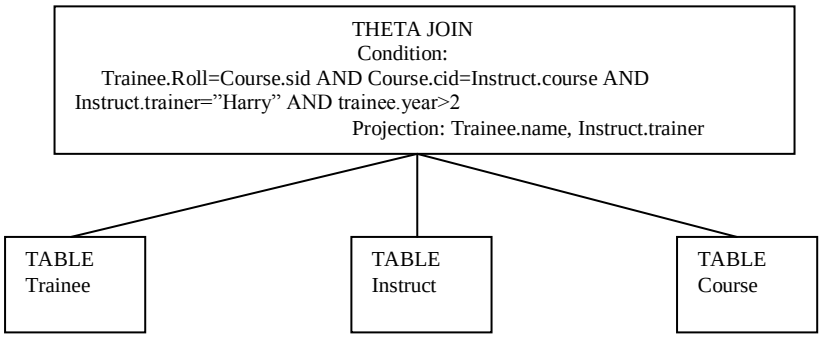
2. Percolating Projections

Again this also helps in reducing the size of the table so that the join order operations are again as less as possible. A check has to be maintained to make sure only valid projections are percolated down the tree.

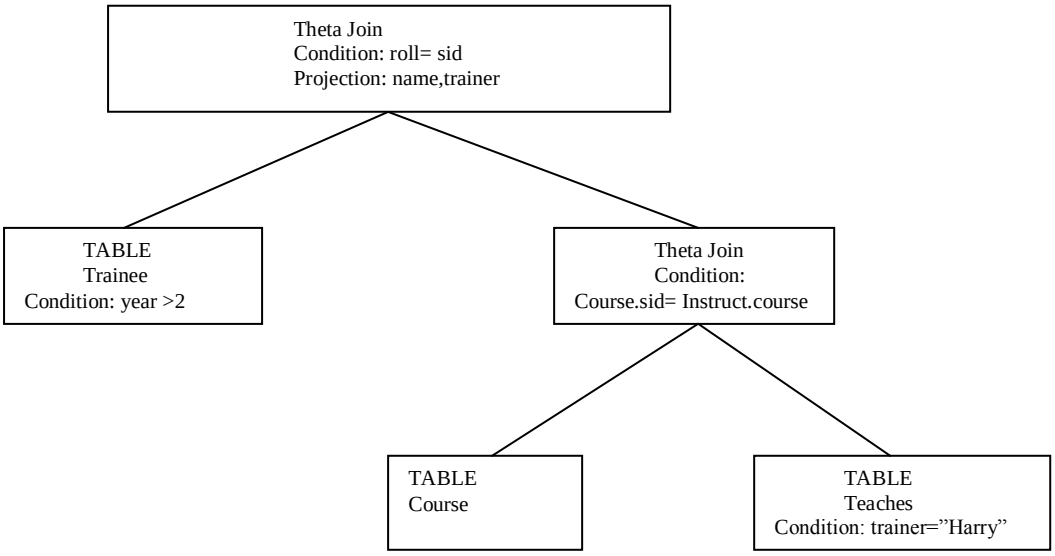
3. Example

Query Q2 :

SELECT Trainee.name, Instruct.trainer FROM Trainee, Course, Instruct WHERE trainee.roll=Course.sid AND Course.cid=Instruct.course AND Instruct.trainer="Harry" AND trainee.year>2



Unoptimized query tree for Query Q2



Optimized Query Tree for Q2 after Percolating Selections

- **Join Order Optimisations**

Joins are commutative as well as associative. We are basically using sampling to determine cost of smaller order joins and then following a bottom-up approach to determine the cost of larger order joins, by taking the smallest cost we can find in any arrangement of the larger join. Using a dynamic programming algorithm, we calculate the cost of of the subsets of join-orders.

Suppose we have to optimise n joins. What we do is we take samples from each of the n tables for joining such that the total table size after the joining is 5000. Now we have 2^N subsets of the join order and we find the cost of each of these subsets using the above samples we have taken. The cost of each individual subset is determined by recursively getting the cost of its subsets and then adding to it the cost it takes to join those subsets.

1. Cost Estimation

The cost evaluation of joining two subsets A and B of size a and b respectively is as follows:

If the join has a equality condition who LHS has fields of one subset and RHS has fields of the other subset (Sort-Merge Join),

$$\text{Cost} = k \times (a \times \log(a) + b \times \log(b))$$

Otherwise,

$$\text{Cost} = k \times (a \times b)$$

k = number of fields projected at each join

2. Sampling

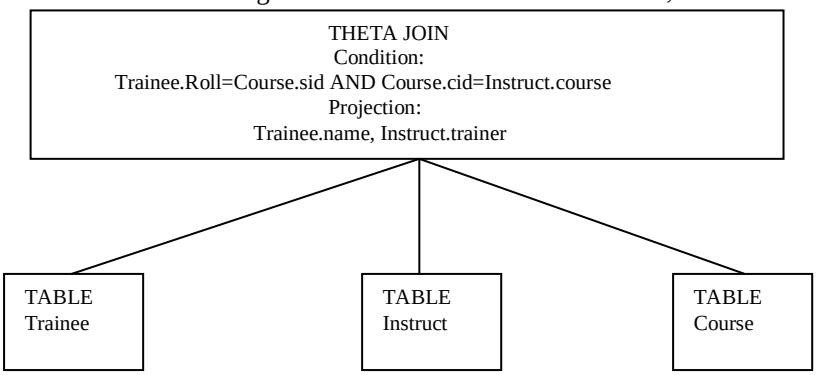
The sampling procedure is as follows: For all the tables involved in the query, a random sample of each table is extracted from the database. For the intermediate nodes in the query Tree, the samples from its children nodes are used for optimisation. When the minimum cost at a node exceeds the threshold value, the optimization at that node stops and we

set the cost of the node to the threshold value. When the cost of the tree is equal to the threshold value, then the query is too expensive to execute.

3. Example

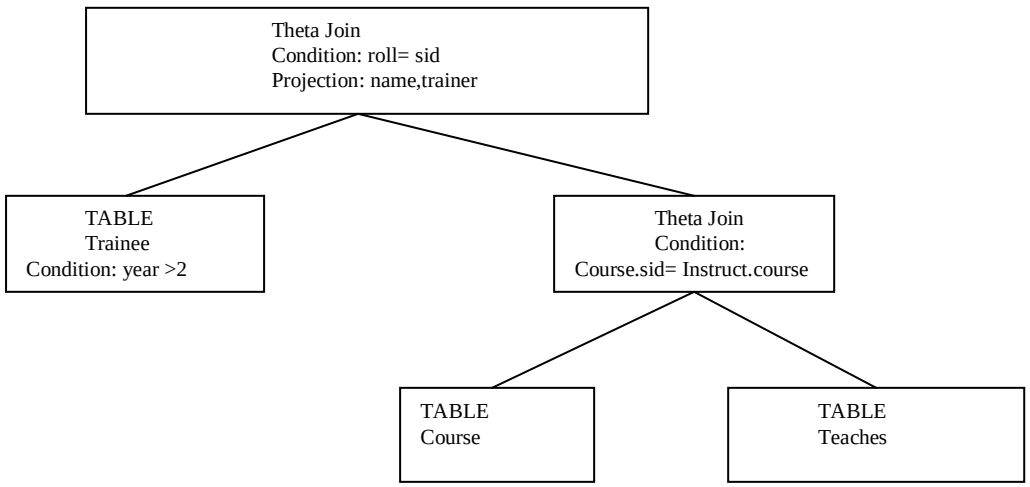
Consider again Query Q1 :

SELECT Trainee.name, Instruct.trainer FROM Trainee, Course, Instruct WHERE Trainee.roll=Course.sid AND Course.cid=Instruct.course with the following table size statistics: Trainee = 10000, Table = 5000, Instruct = 10



Unoptimized Query Tree for Q1

Table Size Statistics:
Trainee = 10000 Table = 5000 Instruct = 10



Optimized Query Tree for Q1 with Join Order Optimisations

V. TEST PLAN

The test plan includes 4 basic steps:

- Testing query processing: Testing whether query is parsed correctly.
- Testing join order optimisations: Trying test cases with upto 4-5 join nodes. Also, testing by varying table size statistics and analyzing join orders.
- Testing tree node percolation: Using test queries that would give percolation of selections and/or projections.
- Testing nested queries

- **Examples and Test Cases**

Query Q3 :

SELECT Trainee.RollNo, Taken.courseID FROM Trainee, Taken WHERE Trainee.RollNo > 1000 OR Taken.courseID <= 100

Output of Unoptimised Query Tree:

query::
select:
field: TRAINEE.ROLLNO
field: COURSES.COURSENO
from:
1. table: TRAINEE
2. table: COURSES condition:
[or]
[<?]
1000


```
            field: TRAINEE.ROLLNO
            [not]
            [<?]
               100
               field: COURSES.COURSENO
```

This is an example where the condition is a disjunction of two conditions over different tables. The query is thus, split as a union of two queries, each with one of the conditions.

Output of Optimised Query Tree:

```
union::
query::
      select:
            field: COURSES.COURSENO
             field: TRAINEE.ROLLNO
      from:
          1.  table: COURSES project to:
                        field: COURSES.COURSENO
          2.  table: TRAINEE
                    condition:
                       [<?]
                          1000
                          field: TRAINEE.ROLLNO
                          project to:
                         field: TRAINEE.ROLLNO
      condition:
            [always]
query::
      select:
            field: COURSES.COURSENO
               field: TRAINEE.ROLLNO from:
             1.  table: COURSES condition:
                          [not]
                             [<?]
                                100
                                field: COURSES.COURSENO
                                project to:
                           field: COURSES.COURSENO
             2.  table: TRAINEE
                       project to:
                            field: TRAINEE.ROLLNO
          condition:
                [always]
```

Query Q4 :

SELECT Trainee.firstname, Trainers.pname FROM Trainee, Taken, Trainers WHERE Trainee.rollno=Taken.Traineeid AND Taken.courseid=Trainers. AND Trainers.pname="Harry"

Output of Unoptimised Query Tree:

```
query::
    select:
          field: TRAINEE.FIRSTNAME
          field: TRAINERS.PNAME
    from:
        1.  table: TRAINEE
        2.  table: TAKEN
        3.  table: TRAINERS
    condition:
        [and]
           [=?]
              field: TRAINEE.ROLLNO
               field: TAKEN.TRAINEEID
           [=?]
            field: TAKEN.COURSEID
           field: TRAINERS.TEACHES
           [=?]
           field: TRAINERS.PNAME "Harry"
```


The optimisation for this query shows how projections are percolated downwards, how join order optimisation works and how conditions and sorting by index is applied at the appropriate-ate levels of the tree.

Output of Optimised Query Tree:

```
query::
    select:
        field: TRAINERS.PNAME
         field: TRAINEE.FIRSTNAME
    from:
        1. query:: select:
                    field: TRAINEE.ROLLNO
                    field: TRAINERS.TEACHES
                    field: TRAINERS.PNAME
                    field: TRAINEE.FIRSTNAME
                from:
                    1.  table: TRAINERS condition:
                                [=?]
                                   field: TRAINERS.PNAME "Harry"
                            project to:
                                field: TRAINERS.TEACHES
                                 field: TRAINERS.PNAME
                    2.  table: TAKEN
                            project to:
                                field: TRAINEE.ROLLNO field:
                                TRAINEE.FIRSTNAME
                condition:
                     [always] sort by:
                    field: TRAINERS.TEACHES
2. table: TAKEN
                project to:
                field: TAKEN.TRAINEEID
                field: TAKEN.COURSEID
 sort by:
                field: TAKEN.COURSEID
        condition:
            [and]
                [=?]
                   field: TRAINEE.ROLLNO
                    field: TAKEN.TRAINEEID
                [=?]
        field: TAKEN.COURSEID
         field: TRAINERS.TEACHES
```

VI. CONCLUSION AND FUTURE WORK

I have created a basic in-memory SQL database processing and execution engine and efficiently implement query optimisation techniques. This included building a SQL parser which converts a SQL query into a execution plan. Also, performed query tree reordering and join-order optimisations use statistical sampling of table data for cost estimation to find optimal execution plan for query.

For future work, same can be done for various other Query formats. Also, Lucene can be used for quick textual searching.

REFERENCES

[1] Oracle Database In-Memory: In-Memory Aggregation. Oracle White Paper, Oracle, 2015

[2] Lindstrom J., Raatikka, V., Ruuth, J., Soini, P., Vakkila, K. IBM solidDB: In-Memory Database Optimized for Extreme Speed and Availability. IEEE Data Eng. Bull. 36(2), pp. 14- 20, 2013

[3] Lahiri, T. et al. Oracle Database In-Memory: A Dual Format In-Memory Database. Proceedings of the 2015 IEEE 31st International Conference on Data Engineering, pp. 1253- 1258, 2015

[4] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. Acta Informatica, 9(1):1{21, 1977.

[5] H.V. Jagadish, Dan Lieuwen, Rajeev Rastogi, Avi Silberschatz, and S. Sudarshan. Dali: A high performance main-memory storage manager. In Procs. of the International Conf. on Very Large Databases, 1994. [JLR 94]

[6] In-Memory Data Management: An inflection point for Enterprise Applications by Von Hasso Plattner.

