Query Expression trees and Optimization

Anusha Kothapally

Query processing is set of activities involved in data extraction from database while querying with human understandable language such as SQL. It includes activities such as translating high-level queries to expressions in machine understandable format, optimizing the expressions for efficient execution and final evaluation of the expression.

Major steps involved in query processing are:

1. Parsing and translation
2. Optimization
3. Evaluation

Parsing and translation: Here, parser checks the syntax of the input query, verifies relation names in query with relation names in database. Then parse-tree representation is constructed then query is translated to RA expression.

Optimization: Given a SQL query there are several ways to compute the final answer. Each relational operation can be computed with one of the several different algorithms. It would generate an annotated expression with detailed execution plan. Optimizer will take the translated RA query and using the statistical data in database it would compute the minimum cost execution plan. Each SQL query can be translated into several different RA queries with the same meaning. Different evaluation plan for given query results different costs. Amongst all equivalent evaluation plans system would construct the minimum cost execution plan for input query. Cost is estimated using statistical information from the database catalog.

Evaluation: The query evaluation engine takes a query evaluation plan, executes that plan and returns the answer to that query.

Multiple Algorithms for each RA operator: Each operation in the expression can be implemented with different algorithms. An evaluation plan is therefore needed to define exactly what algorithm should be used for each operation, and how the execution of the operations should be coordinated, choosing the cheapest algorithm for each operation independently is not necessarily a good idea. Although a merge join at a given level may be costlier than a hash join, it provide a sorted output that makes evaluating a later (such as duplicate elimination, intersection, or another merge join) operation cheaper.
Since, query execution may happen on different OS, and the algorithm, which gave best cost in one scenario, may not give best cost in the other scenario using statistical data. Therefore, based on context, environment different algorithm may be used to implement single RA operator.

Query Expression tree: It is a graphical representation of operations and relations a RA expression. Query expression tree is generated in parsing stage. Leaf nodes are relations and internal nodes are operations. An internal node is executed when its operands are available, once, it is executed node is replaced by result. Root node will be executed at the last. By applying query transformation rules on tree, we could generate several equivalent trees for given RA expression. It gives the visual representation of equivalent queries, which provides the order of execution. Once all equivalent trees generated, using statistical data least cost tree is identified. When we apply the algorithms and specify the order of execution on minimum cost query tree then the evaluation plan is ready for query compute.

Query Transformation rules: Optimizer should generate equivalent expressions for given query. Transformation rules used to specify how to transform an expression into another logically equivalent so that we could evaluate least cost equivalent expression. Two relational expressions are considered to be same if they same set of tuples and the order of tuples don't matter. Equivalence rule says, if two different expressions are equivalent we could replace one with other. Optimizer uses equivalence rules to transform expression into other equivalent expression. Below are query transformation rules.

- A conjunctive selection condition can be broken up into a cascade (sequence) of individual s operations
  $\sigma_{c1\wedge c2}(E) = \sigma_{c1}(\sigma_{c2}(E))$
- Commutative nature of σ: The σ operation is commutative:
- Cascade of ∏: In a sequence of ∏operations, all can be ignored except last one.
  $\prod_{L1}(\prod_{L2}(\ldots(\prod_{Ln}(E)))) = \prod_{L1}(E)$
- Selections can be combined with Cartesian products and theta joins
  $\sigma_{c1}(E1 \times E2) = E1 \mid X \mid_{c1} E2$
- Theta join operations are commutative.
  $E_1 \mid X \mid_{c1} E_2 = E_2 \mid X \mid_{c1} E_1$
- Natural join, Theta join operations are associative.
  $(E1 \mid X \mid E2) \mid X \mid E3 = E1 \mid X \mid (E2 \mid X \mid E3)$
- Select conditions distributes over the theta-join operation in below two conditions.
- When all attributes in c1 involve only of the expression
  $\sigma_{c1}(E1 \mid X \mid E2) = (\sigma_{c1}(E1)) \mid X \mid E2$
- If selection condition c1 involves only attributes of E1, and c2 involves only attributes of c2.
  $\sigma_{c1\wedge c2}(E1 \mid X \mid \sigma E2) = \sigma_{c1}(E1) \mid X \mid \sigma_{c2}(E2)$
- Project operation distributes over the theta-join.
- Set operation Union and Intersect are commutative but not set difference
- Set union, intersect are associative
- Selection operation distributes over the union, intersect and set difference.
- Project operation distributes over union

Evaluation of Expressions: We have two strategies to evaluate a RA expression in query expression tree.

1) Materialization- evaluates one operation at a time, starting at the lowest level (leaf nodes) in query tree. Use intermediate results materialized or stored in temporary relations to evaluate next level operation.
2) Pipelining -evaluates several operations simultaneously, and pass the result on to the next operation.

Cost of query evaluation can be measured in terms of a number of different resources, including disk accesses, CPU time to execute a query, and, in a distributed or parallel database system, the cost of communication. In general disk access are considered most costly. Therefore, optimizers would make an effort to reduce the resource consumption to reduce the resource consumption.

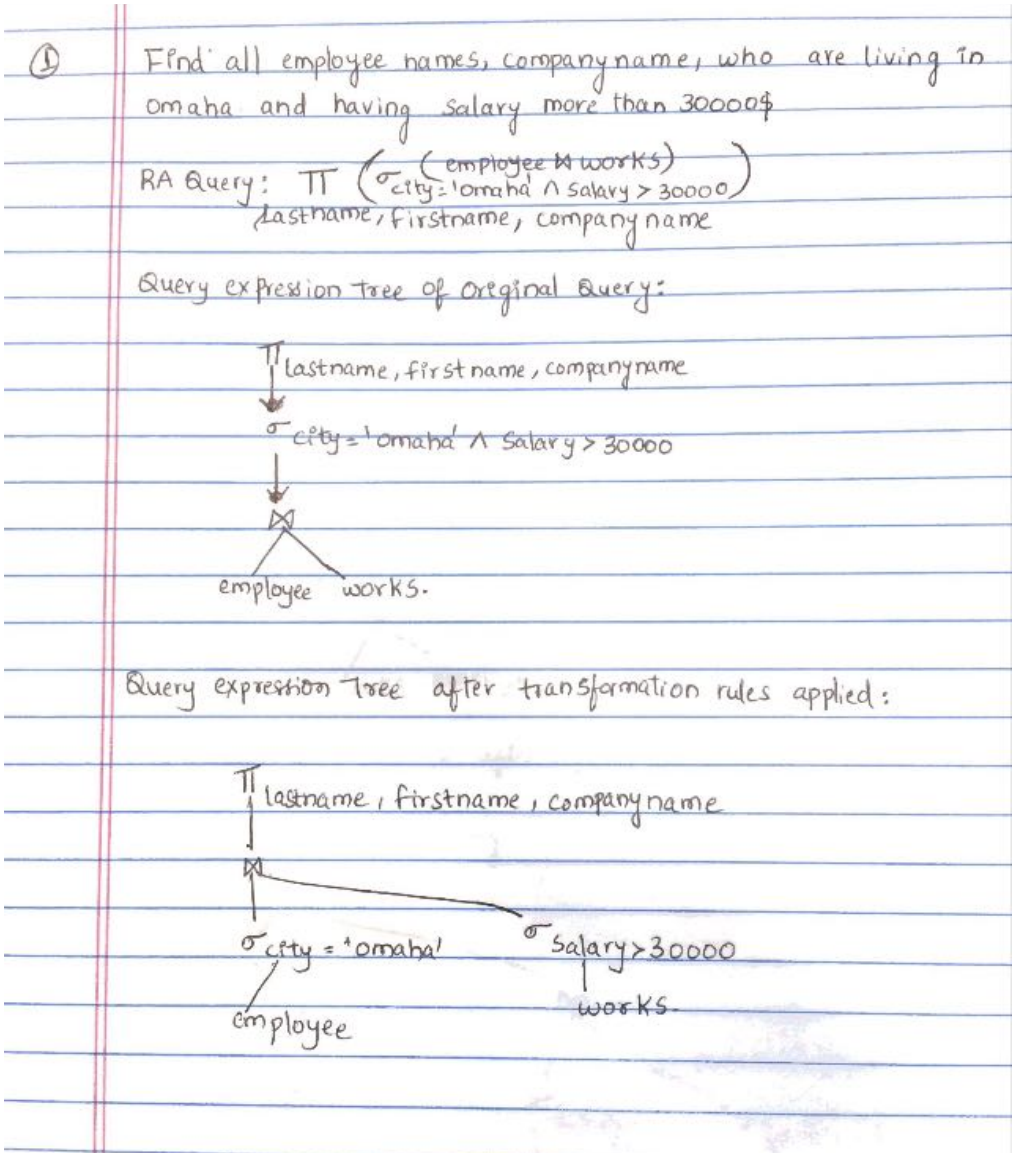
A relational algebra expression can be evaluated in many ways. An annotated expression specifying detailed evaluation strategy is called the execution plan. Among all semantically equivalent expressions, the one with the least costly evaluation plan is chosen.

Heuristics for Optimization:

- Do selection as early as possible. Use cascading, commutativity, and distributivity to move selection as far down the query tree as possible
- Perform operations that generate smaller tables first.
- Use associativity to rearrange relations so the selection operation that will produce the smallest table will be executed first
- If a cross product appears as an argument for a selection, where the selection involves attributes of the tables in the product, change the product to a join
- If the selection involves attributes of only one of the tables in the product, apply the selection to that table first
- Do projection early. Use cascading, distributivity and commutativity to move the projection as far down the query tree as possible.
- Examine all projections to see if some are unnecessary
- If a sub-expression appears more than once in the query tree, and the result it produces is not too large, compute it once and save it

Below are two RA queries with query transformation tree before and after transformation rules applied.

1)

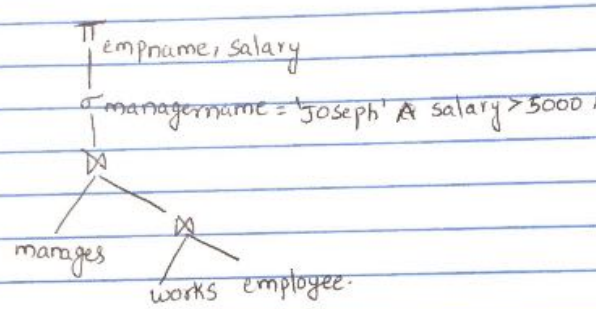


2)

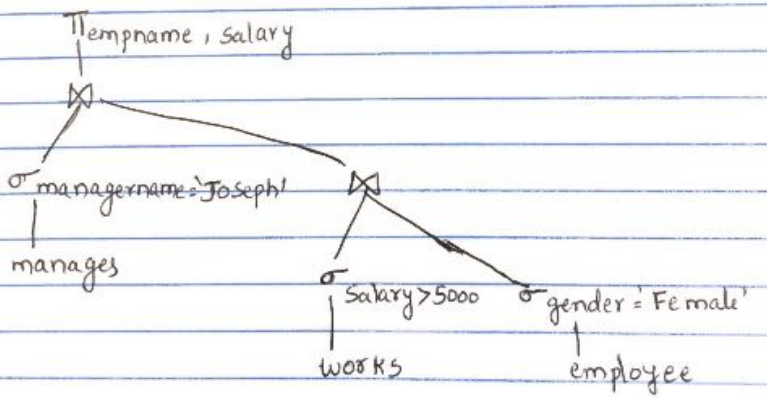
Find all employees name, salary who are working under manager 'Joseph', earns salary morethan 5000$ and their gender is Female.

RA Query: $\Pi_{empname, salary}\left(\sigma_{managername = 'Joseph' \wedge salary > 5000 \wedge gender = 'F'}\left(manages \bowtie works \bowtie employee\right)\right)$

Query expression tree of original query:



Query expression tree after applying transformation rules.



No SQL

NoSQL refers to non-SQL or non-relational or not only SQL. In the existing relational database it is being hard with more functionality and less performance to handle huge amount of and varieties of data, this problem was solved using NoSQL will less functionality and high performance, it uses storage and retrieval mechanism of data modeled in the means of other than tabular relations used in relational database. On demand of modern real-time applications such as Facebook, Google, Amazon.com NoSQL databases are widely used in big data. The data in NoSQL is non-relational; data is not related to each other. The way of emphasizing No SQL as not only SQL shows that they could support SQL like query language. The main focus of NoSQL is to provide Scalability, Performance and High Availability.

NoSQL database depends on the problem it has to address. It can handle both structured data and unstructured data such as media, text messages, blogs etc. Data structures used by NoSQL are different from those in SQL. Most of the NoSQL stores compromise ACID properties on transactions, although few of the NoSQL databases have made them central. Most of the NoSQL databases offer eventual consistency where changes are propagated to all nodes eventually. Eventual consistency may cause stale reads that means queries for data may not return updated data and also has chances of data loss. Concept of write-ahead may avoid data loss. Because of join functionality in RDBMS scalability is reduce hence NoSQL database do not implemented to support joins to improve Scalability. NoSQL do not support complex transactions, constraints at database level that provides high performance, if required we need to implement at application level.

Scenarios where we could use NoSQL databases is:

- The ability to store and retrieve great quantities of data is required.
- Storing relationships between the data is not important.
- When data is Unstructured
- If fast applications need to developed
- Constraints and validation logic is not required at database level

NoSQL databases further divided into sub categories.

1. Key Value Store: Every single item will be stored as attribute name, together with its value.
2. Document Oriented: It has where each key pair with a complex data structure.
3. Object database
4. Graph oriented etc.,

MongoDB

One of NoSQL products is MongoDB; it is an open source document-oriented database program. In MongoDB, records are documents, which behave a lot like JSON objects in JavaScript. This is different than SQL databases, where fields correspond to columns in a table and individual records correspond to rows.
Database: It is container for collections. A single MongoDB server can hold multiple databases.
Collection: It is a collection of documents. It is equivalent to a table in RDBMS. It doesn't enforce any schema as tables do in RDBMS.
Document: It is a set of key-value pairs. Documents with in a collection need not to have same structure or same set of fields.
Fields: Fields corresponds to columns in RDBMS. Field can be of type which MongoDB supports.

I have downloaded mongo db, installed and configured on my machine. I have written basic queries such as:

- Creating collections.
- Insert documents in it.
- Extracting the documents from collection.
- Extracting particular fields from documents.
- Extract documents under specific condition.
- Updating documents.
- Removing documents.
- Querying using aggregate functions.

We can connect to MongoDB server using shell, java, python, and node JS etc. Here, I have connected and executed queries through shell.

Creating a database: As we said each server could hold multiple databases. Using below command we can create database. Below command would create a new database if there is no database with mentioned name else, it would return the existing database.

```
> use myOrg
Result: switched to db myOrg
```

In order get the name of current db:

```
> db
Result: myOrg
```

Get all dbs in the server:

```
> show dbs
Result:
admin  0.000GB
local  0.000GB
myOrg  0.000GB
mydb   0.000GB
```

Create Collection:

```
> db.createCollection("company")
Result: { "ok" : 1 }
> show collections
Result:
company
employee
```

Drop a collection and database:

Create a test database

```
> use testdbcrud
Result: switched to db testdbcrud
```

Create a collection and insert document.

```
> db.testcollection.insert({"Key":"value"})
Result: WriteResult({ “nInserted” : 1 })
> show databases
admin      0.000GB
local      0.000GB
myOrg      0.000GB
mydb       0.000GB
testdbcrud  0.000GB
> db.testcollection.drop();
true
> db.dropDatabase()
Result: { “dropped” : “testdbcrud”, “ok” : 1 }
```

Insert a document to database:

```
> db.employee.insert ({
  “Lastname”:”Brady”,
  “Firstname”:”Dan”,
  “city”:”Omaha”,
  “Gender”:”M”
})
Result: WriteResult({ “nInserted” : 1 })

> db.employee.insert({
  “Lastname”:”Abraham”,
  “Firstname”:”John”,
  “Phone Number”:”98786754321”,
  “email”:”john@123.com”,
  “city”:”Lincoln”,
  “Gender”:”M”
})
Result: WriteResult({ "nInserted" : 1 })
```

Query all document in collection:

```
> db.employee.find().pretty()
Result:
{
        "_id" : ObjectId("59721c006ff92baba1dacc15"),
        "Lastname" : "Brady",
        "Firstname" : "Dan",
        "city" : "Omaha",
        "Gender" : "M"
}
{
        "_id" : ObjectId("5972217f6ff92baba1dacc18"),
        "Lastname" : "Abraham",
        "Firstname" : "John",
        "Phone Number" : "98786754321",
        "email" : "john@123.com",
        "city" : "Lincoln",
        "Gender" : "M"
}
```

Get documents from collection on condition. Query using where:

```
> db.employee.find({"Lastname":"Abraham"}).pretty()
Result:
{
        "_id" : ObjectId("5972217f6ff92baba1dacc18"),
        "Lastname" : "Abraham",
        "Firstname" : "John",
        "Phone Number" : "98786754321",
        "email" : "john@123.com",
        "city" : "Lincoln",
        "Gender" : "M"
}

> db.employee.find({"Lastname":{$ne:"Abraham"}}).pretty()
Result:
{
        "_id" : ObjectId("59721c006ff92baba1dacc15"),
        "Lastname" : "Brady",
        "Firstname" : "Dan",
        "city" : "Omaha",
        "Gender" : "M"
}
```

Get documents from collection under multiple conditions. Query using ‘and’:

```
> db.employee.find ({$and:[{"Lastname":"Abraham"},{"Gender":"M"}]}).pretty()
Result:
{
        "_id" : ObjectId("5972217f6ff92baba1dacc18"),
        "Lastname" : "Abraham",
        "Firstname" : "John",
        "Phone Number" : "98786754321",
        "email" : "john@123.com",
        "city" : "Lincoln",
        "Gender" : "M"
}
```

Update a document:

```
> db.employee.update ({"Firstname":"John"},{$set:{"Firstname":"Joseph"}})
Result: WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })

> db.employee.find().pretty()
Result:
{
        "_id" : ObjectId("59721c006ff92baba1dacc15"),
        "Lastname" : "Brady",
        "Firstname" : "Dan",
        "city" : "Omaha",
        "Gender" : "M"
}
{
        "_id" : ObjectId("5972217f6ff92baba1dacc18"),
        "Lastname" : "Abraham",
        "Firstname" : "Joseph",
        "Phone Number" : "98786754321",
        "email" : "john@123.com",
        "city" : "Lincoln",
        "Gender" : "M"
}
```

Remove a document:

```
> db.employee.remove({"Firstname":"Joseph"})
Result: WriteResult({ "nRemoved" : 1 })
> db.employee.find().pretty()
{
        "_id" : ObjectId("59721c006ff92baba1dacc15"),
        "Lastname" : "Brady",
        "Firstname" : "Dan",
        "city" : "Omaha",
        "Gender" : "M"   }
```

Project certain fields:

```
> db.employee.find ({},{"Firstname":1,"Lastname":1,"Phone Number":1,_id:0}).pretty();
Result:
{ "Lastname" : "Brady", "Firstname" : "Dan" }
{
        "Lastname" : "Abraham",
        "Firstname" : "John",
        "Phone Number" : "98786754321"
}
```

Limit to certain number of rows:

```
> db.employee.find().pretty().limit(1);
Result:
{
        "_id" : ObjectId("59721c006ff92baba1dacc15"),
        "Lastname" : "Brady",
        "Firstname" : "Dan",
        "city" : "Omaha",
        "Gender" : "M"
}
```

Sort the documents:

```
> db.employee.find().pretty().sort({"Lastname":1});
Result:
{
        "_id" : ObjectId("59722c4c6ff92baba1dacc19"),
        "Lastname" : "Abraham",
        "Firstname" : "John",
        "Phone Number" : "98786754321",
        "email" : "john@123.com",
        "city" : "Lincoln",
        "Gender" : "M"
}
{
        "_id" : ObjectId("59721c006ff92baba1dacc15"),
        "Lastname" : "Brady",
        "Firstname" : "Dan",
        "city" : "Omaha",
        "Gender" : "M"
}
```

Aggregate functions:

```
> db.employee.aggregate([{$count:"Lastname"}])
Result: { "Lastname" : 3 }
```

Observations:

In order to insert a record into database we first need to create a table with defined schema. Where as in Mongo we need not to create a collection, while inserting a document into collection if there is no collection exists with a name it would create a one automatically.

All the records in table will have same schema. But it is not required with MongoDB each document different fields with different documents.