Ian Kenny

December 1, 2017
Introductory Databases

Query Optimisation
Introduction

Any given natural language query can be formulated as many possible SQL queries.

There are also many possible relational algebra queries for any given SQL query.

The actual file data in the database can be accessed in various ways too. The DBMS could exhaustively search the actual database data files, use indexes, hashes, etc. The DBMS has only a finite amount of RAM available to it. If a query involves constant swapping of data between RAM and persistent storage, the query could execute very slowly.

The DBMS will seek to minimise the length of time taken by a query. Query optimisation is the process for selecting an optimal formulation of a query, and selecting an optimal plan for accessing the data.
As an aside, a DBMS might especially optimise the fetching of the first $n$ rows of data to be presented to the user, where $n$ is small compared to the total number of rows in the actual result. The DBMS presents these first rows and then works on fetching and presenting the rest.

This is commonly-seen behaviour on websites.
Query processing

When a DBMS is presented with a query, it must do the following:

- Parse the query to ensure it has legal syntax.
- Check that all of the relations and attributes used in the query actually exist.
- Query decomposition: in which a logical representation of the query is created and manipulated (but not the yet an optimal form for the query)
- Select an optimal query formulation for the query, based on the many possible formulations.
- Formulate a plan to execute the query.
- Execute the query.

We are going to discuss the selection of an optimal formulation for a query.
Query decomposition

The query decomposition phase produces a logical representation of the query. The language for this is generally the relational algebra.

A relational algebra expression is created in this phase. This is a relational algebra expression that represents the query as it was formulated (i.e. non-optimised). We will see examples of that soon.
Database statistics

Query optimisation is also aided by database statistics. This is data about the tables in a database. It is often called the Catalog.

For each relation in the database, the Catalog will include

$N(R)$: The number of rows in relation $R$ (the cardinality of $R$).
$S(R)$: The number of attributes in a row in $R$ (the degree of $R$).

In addition to other information about the spread of the values in each attribute, which attributes have indexes, the size of each row in bytes, etc.

We will see how this information is used shortly.
Query optimisation

Query optimisation involves transformations applied to the relational algebra expression implied by a query.

As stated above, the DBMS creates an initial relational algebra representation of a query. This is not yet optimised.

The DBMS then applies a number of transformational rules to this initial expression to create a more optimal (hence efficient) expression.

We will now look at some of these transformational rules. This list is not exhaustive. There are other rules and others could be derived.
Rule 1: Cascade of selections

A conjunct of boolean expressions in a Select condition can be applied simultaneously or as a series of nested single selections, e.g.

\[ \sigma_{p \land q \land r}(R) = \sigma_p(\sigma_q(\sigma_r(R))) \]

Which way round we perform this transformation depends on the database statistics. If the SQL query implies the expression on the RHS, it might be better to transform it to the form on the LHS. This might be appropriate if the relation \( R \) is ‘small’. In this case it might be quicker for the DBMS to load each row of the table and apply all of the conditions simultaneously.
Rule 1: Cascade of selections

\[ \sigma_{p \land q \land r}(R) = \sigma_p(\sigma_q(\sigma_r(R))) \]

However, it might be better stick with the expression on the RHS, depending on the number of rows in the relation (i.e. if \( R \) is ‘large’) and on the relative restrictiveness of the Select expressions.

For example, if one of the Select expressions is highly restrictive then it can be applied first to minimise the amount of data that needs to manipulated for the second and third expressions.

If this were to reduce the amount of data significantly, it might enable the DBMS to load the remaining data into the RAM to complete the operation, rather than swapping data between file and the RAM.
Memory implications

Recall that the DBMS only has a finite amount of RAM available to it. It can load in a certain proportion of the data but the rest must remain on disk. Disk operations are very slow compared to RAM operations (which is itself slower than cache memory). Thus, it is very important that the DBMS minimise the amount of data that needs fetching from disk.

If a highly restrictive condition can be applied first, it might be possible for the DBMS to load the remaining data into the RAM to complete the operation, rather than swapping data between file and the RAM.
Rule 2: Commutativity of selections

Selections on the same relation are commutative.

\[ \sigma_p(\sigma_q(R)) = \sigma_q(\sigma_p(R)) \]

Again, we might perform this transformation in order to apply a more restrictive selection first.
Rule 3: Remove unnecessary projections

In a series of projections, only the last one is required.

\[ \Pi_A \Pi_B \Pi_C (R) = \Pi_A (R) \]

The final projection in this sequence (\( \Pi_A \)) removes \( B \) and \( C \) hence those projections are unnecessary.
Rule 4: Commutativity of selection and projection

If a selection predicate contains only attributes on the projection list, the selection and projection operations commute.

\[ \Pi_{A,B}(\sigma_{A=x \land B=y}(R)) = \sigma_{A=x \land B=y}(\Pi_{A,B}(R)) \]

In terms of the amount of data removed, selection is on average more restrictive than projection hence it may be better to perform that operation first.
Rule 5: Associativity of theta join and cross product

\[ \sigma_p(R \bowtie_r S) = (\sigma_p(R)) \bowtie_r S \]
\[ \sigma_p(R \times S) = (\sigma_p(R)) \times S \]

Again, this may allow us to reduce the amount of processing. If the condition \( p \) applies only to relation \( R \) then we do not need to apply it to the result of the join, we can apply it only to \( R \) and then join the result of \textit{that} operation to \( S \). This applies to the cross product too.

\footnote{theta join is specified here since it is simply a more general form of the natural join.}
Rule 6: Projection and theta join and cross product distribute

If relations $R$ and $S$ are to have the join or cross operator applied, and the projection list applied to the join or cross operation contains only attributes in either $R$ or $S$ (but there must be some attributes from $R$ and $S$), the projection operation distributes over the join or cross.

If $A$ is a set of attributes from $R$ and $B$ is a set of attributes from $S$ then

\[
\Pi_{A\cup B}(R \bowtie_r S) = (\Pi_A(R) \bowtie_r (\Pi_B(S)))
\]
\[
\Pi_{A\cup B}(R \times S) = (\Pi_A(R) \times (\Pi_B(S)))
\]
Rule 7: Associativity of natural join and cross product

\[(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)\]
\[(R \times S) \times T = R \times (S \times T)\]

Again, this rule allows us to perform the most restrictive operations first. We can reorder the operations to perform the operation that removes the most data first. Thus, subsequent operations will be performed on the least possible amount of data, which will be faster.
Heuristic strategies for query optimisation

The above rules can be summarised in the following heuristic strategies.

1. **Perform selection as early as possible.** Selection reduces the cardinality of relations hence should be performed as early as possible. We can use the above rules to reorder an expression to apply selection as soon as possible. Selection predicates on the same relation should be kept together, which may require reordering the expression.

2. **Reorder joins to apply more selective joins first.** The associativity of the binary operators allows us to reorder expressions containing joins so that the most restrictive operations are applied first. This will result in a smaller operand being created for subsequent join operations.
3. **Perform projections as early as possible.** Again, a projection operation reduces the amount of data in the results. Using projection as early as possible to remove *unwanted* attributes will result in less data being passed to subsequent operations. Projections involving attributes on the same relation should be kept together.

4. **Project away unwanted attributes.** Rather than carrying pointless attributes through a query, get rid of them. The only attributes that are actually needed are the ones needed in the result and those needed for joins.
Example: the music database

Consider an example a ‘music’ database with the following relations

**Album**: has data about music albums - artist, title, price, etc.

**Customer**: has data about customers.

**Sales**: has data about which albums have been sold to which customers.

**Review**: has data about customer reviews of albums.
Example: the music database

For reference, the relational schemas for the relations in the music database are as follows:

**album**(albumID int, artistid int, title varchar (255), label varchar (255), year numeric(255), genre varchar(255), price decimal(6, 2))

**customer**(custID int, fname varchar(255), lname varchar(255), houseNum varchar(255), postCode varchar(255))

**sales**(salesRef int, custID int, albumID int, saleDate date)

**review**(albumID, custID, rating numeric, text varchar (1024))
Consider the following Catalog information for the music database.

\[ \text{N(Album)} = 5000, \ S(\text{Album}) = 10 \]
\[ \text{N(Customer)} = 30000, \ S(\text{Album}) = 16 \]
\[ \text{N(Sales)} = 40000, \ S(\text{Album}) = 8 \]
\[ \text{N(Review)} = 500, \ S(\text{Review}) = 4 \]

Remember:

\[ N(R): \text{ The number of rows in relation } R \text{ (the cardinality of } R) \]
\[ S(R): \text{ The number of attributes in a row in } R \text{ (the degree of } R) \]
Example 1

Consider the following query

*Find the album titles of all albums by David Bowie reviewed by the customer with id = 2.*

The initial relational algebra expression for this query might be

\[
\Pi_{\text{title}}
(\sigma_{\text{artist} = 'David Bowie'} \land \text{custid} = 2 \land \text{Album.albumid} = \text{Review.albumid}
(\text{Album} \times \text{Review}))
\]

An SQL statement for this query is

```
SELECT title
FROM album, review
WHERE artist='David Bowie'
and album.albumid = review.albumid and review.custid = 2;
```
Relational algebra expression tree

This expression can be represented as a relational algebra expression tree. The leaves of the tree are the relations and the internal nodes are the operators. The root of the tree is the final operation/result.

```
PROJECT (title)
  |
  SELECT (artist='David Bowie AND custid = 2 AND Album.albumid = Review.albumid)
     |
     CROSS
       |
       Album
       |
       Review
```
Example 1

\[ \Pi_{\text{title}} (\sigma_{\text{artist} = 'DavidBowie' \land \text{custid} = 2 \land \text{Album.albumid} = \text{Review.albumid}} (\text{Album} \times \text{Review})) \]

As you can see, this is an inefficient query. It performs a cross product, which is a very expensive operation. In the case of the Album and Review relations, this will result in a relation that has \( N(\text{Album}) \times N(\text{Review}) \) tuples = 2,500,000. This is way too many and is completely unnecessary.

What we can see in this query is that since we are ‘manually’ joining on Album.albumid = Review.albumid, we could replace the cross with a natural join. This is an optimised operation in the DBMS.
Example 1

Hence

\[ \Pi_{title} \left( \sigma_{\text{artist} = 'David Bowie'} \land \text{custid} = 2 \land \text{Album.albumid} = \text{Review.albumid} \right) \left( \text{Album} \Join \text{Review} \right) \]

Is equivalent to

\[ \Pi_{title}(\sigma_{\text{artist} = 'David Bowie'} \land \text{custid} = 2)(\text{Album} \Join \Join \text{Review}) \]

And an equivalent SQL SELECT command for this would be

```
SELECT title 
FROM album NATURAL JOIN review 
WHERE custid = 2 AND artist = 'David Bowie';
```
Example 1: relational algebra expression tree

The revised tree for this expression is shown below.

```
PROJECT (title)
|
SELECT (artist='David Bowie AND custid = 2)
|
NATURAL JOIN
|
Album
Review
```
Example 1

\[ \Pi_{\text{title}}(\sigma_{\text{artist}=\text{DavidBowie}} \land \text{custid}=2(\text{Album} \bowtie \text{Review})) \]

If we continue to look at the new expression, we note that the condition \( \text{artist} = \text{DavidBowie} \) applies only to the Album table and \( \text{custid} = 2 \) applies only to the Review table. We could apply those selections earlier. This might be worth doing since for the Album table \( N(\text{Album}) = 5000 \) but only a few of those albums will be by David Bowie. The Review table is already relatively small, but we could apply the selection early there too.

\[ \Pi_{\text{title}}(\sigma_{\text{artist}=\text{DavidBowie}}(\text{Album}) \bowtie \sigma_{\text{custid}=2}(\text{Review})) \]
Example 1

\[ \Pi_{\text{title}}(\sigma_{\text{artist} = 'David Bowie'}(\text{Album}) \bowtie \sigma_{\text{custid} = 2}(\text{Review})) \]

And an equivalent SQL SELECT command for this would be

```sql
SELECT title
FROM
  (SELECT * FROM album
   WHERE artist = 'David Bowie') AS t1
NATURAL JOIN
  (SELECT * FROM review WHERE custid = 2) AS t2;
```
Example 1: relational algebra expression tree

The revised tree for this expression is shown below.

```
PROJECT (title)
     /
|-- NATURAL JOIN
  |   /
  |  SELECT (artist='David Bowie')
  |   /
  |   'Album'
  |   /
  |  SELECT (custid=2)
  |   /
  |   'Review'
```
Example 1

However, this expression still hasn’t been fully optimised. We note that in the previous expression (and the SQL query), the project operations on the two tables Album and Review don’t filter any attributes. In the relational algebra expression, there are no projection operations applied to those relations directly, and in the SQL statement the clause is SELECT * (all).

We can filter out early any attributes that we don’t need. We must keep those that are required in subsequent operations, of course.
Example 1

\[ \Pi_{\text{title}} (\Pi_{\text{albumid}, \text{title}} (\sigma_{\text{artist}=\text{DavidBowie}}(\text{Album})) \bowtie (\Pi_{\text{albumid}} (\sigma_{\text{custid}=2}(\text{Review})))) \]

And an equivalent SQL SELECT command for this would be

```
SELECT title
FROM
  (SELECT albumid, title FROM album
   WHERE artist = 'David Bowie') AS t1
NATURAL JOIN
  (SELECT albumid FROM review WHERE custid = 2) AS t2;
```
Example 1: relational algebra expression tree

The revised tree for this expression is shown below.

```
PROJECT (title)
  |
  /
|
NATURAL JOIN
  |
  /
|
PROJECT(albumid, title)  PROJECT(albumid)
  |
  /
|
SELECT (artist='David Bowie')  SELECT(custid=2)
  |
  |
  Album  Review
```
Of course, all of the SQL statements for this example are equivalent. The whole point is that the DBMS will attempt to transform a query into a more efficiently processable form. Of the four SQL statements in this example, the second one is the most natural and most readable.

```
SELECT title
FROM album NATURAL JOIN review
WHERE custid = 2 AND artist = 'David Bowie';
```

This query is the most readable, and should perhaps be the one that the user creates, but the point here is that the DBMS will probably not choose to implement the query that way. This doesn’t matter.
Example 1

Are any other optimisations possible?
Example 2

Consider the following query:

List the names of customers and the titles of the albums for customers who bought an album worth more than £5 and gave it a rating of 5.

Firstly consider the tables that are needed. We will need to somehow join Customer, Album, Review and Sale. We could, of course simply do the cross product. However, that would produce a result with $3 \times 10^{15}$ rows!

We need to consider an order for the joins that as quickly as possible reduces the cardinality of the intermediate results.
Example 2

A possible relational algebra expression for this query is

\[ \Pi_{name,title}(\sigma_{rating=5 \land price>5}(Customer \bowtie Sale \bowtie Album \bowtie Review)) \]

This is better than computing the cross product because the join operation is optimised. However, as it stands, the join operation here is still expensive and not guided by consideration of the relative sizes of the relations, for example.
Example 2: relational algebra expression tree

PROJECT (name, title)

SELECT(rating=5 AND price > 5)

NATURAL JOIN

Album

Review

Customer

Sale
Example 2

\[ \Pi_{name, title}(\sigma_{rating=5 \land price>5}(Customer \bowtie Sale \bowtie Album \bowtie Review)) \]

If we compare them we see that \textit{Customer} \bowtie \textit{Sale} is an operation on 30000*40000 tuples; \textit{Sale} \bowtie \textit{Album} is an operation on (only) 40000*5000 tuples; and \textit{Album} \bowtie \textit{Review} is an operation on 5000*500 tuples\textsuperscript{2}.

The DBMS uses this information to decide the order of evaluation of the join operations. If you consider the rules introduced earlier, the natural join is an associative operation. We can change the order of evaluation.

\textsuperscript{2}Are there other pairings that have an even lower number of tuples?
Example 2

In order to reduce the cardinalities of the intermediate results we can reorder the joins, i.e. force a particular order of operation. Since the Review and Album tables have a ‘small’ natural join, we can join those first. Since the Customer relation has the next lowest cardinality, we can join the result of the first join to that. Finally, we can join to the largest relation - the Sale relation.

$$\Pi_{name, title}(\sigma_{rating=5 \land price > 5}(Sale \bowtie (Customer \bowtie (Album \bowtie Review))))$$
Example 2: relational algebra expression tree

![Relational Algebra Expression Tree](image-url)
Example 2

However, as we saw with exercise 1, we also need to make selections as early as possible. Currently, this expression delays selection until near the root of the expression hence misses an opportunity to reduce the cardinalities of the intermediate results yet further. Indeed, if we apply the selection $rating = 5$ to the Review table, we remove all other ratings, and may be left with a small table as a result. The Review table is already far smaller than the other tables. We can also apply the condition $price > 5$ to the Album title as soon as possible but that may have less of an impact if many of the albums cost more than that.

\[ \Pi_{name, title} (Sale \bowtie (Customer \bowtie (\sigma_{price>5}(Album) \bowtie (\sigma_{rating=5}(Review)))) \)
Example 2: relational algebra expression tree

```
PROJECT (name, title)
       /
      /
NATURAL JOIN
/    /
/     /
NATURAL JOIN
/     /
/     /
NATURAL JOIN
/     /
/     /
SELECT(price > 5)
|     |
|     |
Album

|     |
|     |
SELECT(rating=5)
|     |
|     |
Review

|     |
|     |
'Sales'
```

Example 2

Finally, we note that at each stage of the processing of the expression, we only need the attributes that are needed for the natural join and any attributes that are desired in the final result, if they are different to the join condition. Thus, we can push projections down the expression towards the leaves to remove unwanted attributes.

$$\Pi_{name, title}$$
$$\quad (Sale \bowtie (\Pi_{custid, albumid, title, name}(Customer \bowtie$$
$$\quad \quad (\Pi_{albumid, title}(\sigma_{price>5}(Album)) \bowtie \Pi_{albumid, custid}(\sigma_{rating=5}(Review))))))$$
Example 2: relational algebra expression tree

```
PROJECT (name, title)
  |    |
  |
NATURAL JOIN
  |    |
  |
PROJECT (name, title, custid, albumid)
  |   |
  |    |
NATURAL JOIN
  |    |
  |
NATURAL JOIN
  |    |
  |
PROJECT (albumid, title)
  |    |
  |
SELECT(price > 5)
  |    |
  |
Album

PROJECT (custid, albumid)
  |    |
  |
SELECT(rating=5)
  |    |
  |
Review

Customer

Sale
```