Ian Kenny

December 5, 2016

Databases

Lecture 11

In this lecture

- More on performance.
- Query processing and optimisation.

Introduction

When a user submits an SQL query, the DBMS doesn't simply execute that query exactly as it is formulated. The DBMS attempts to select an optimal formulation for the query.

Once the DBMS has formulated an 'optimal' query, it then selects the way in which the data will be accessed. For example, it will select where it needs to do a full file scan or if it can use an index for all or part of a query.

Query optimisation is the process for selecting an optimal formulation of a query, and selecting an optimal plan for accessing the data. There are many possible SQL formulations for the same query, and many possible relational algebra expressions for a single SQL query.

A DBMS might especially optimise the 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.

Query processing

The activities on the previous slide are referred to as *query processing*. Query processing involves

- Parsing the query to ensure it has legal syntax.
- Checking 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)
- Selecting an optimal query formulation from the given query.
- Formulating a plan to execute the query.
- Executing the query.

We are largely concerned with 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 *query tree* is created in this phase. This a relational algebra expression that represents the query, as it was formulated.

This phase also transforms the predicate(s) in the query into *Conjunctive Normal Form (CNF)* in which all boolean expressions are represented as a sequence of expressions that contain only disjuncts, connected by conjuncts. For example $(A \vee B) \wedge (A \vee C)$, to take advantage of well-known algorithms for manipulating such expressions.

This phase also performs a form of semantic analysis on the predicate(s) to check for contradictions, etc. For example $(artist = `DavidBowie' \wedge artist = `TheBeatles')$. (in this case the expression might be replaced with the value FALSE - it is DBMS specific).

Predicate simplification

The query decomposition phase also attempts to simplify the predicate(s) in the query. Some well-known Boolean algebra rules can be applied, e.g.

$$
\begin{aligned}
&p \wedge \mathit{false} \equiv \mathit{false}\\
&p \wedge \mathit{true} \equiv p\\
&p \wedge \neg p \equiv \mathit{false}\\
&p \vee \mathit{false} \equiv p\\
&p \vee \mathit{true} \equiv \mathit{true}\\
&p \vee \neg p \equiv \mathit{true}\\
&p \vee (p \wedge q) \equiv p\\
&p \wedge (p \vee q) \equiv p
\end{aligned}
$$

For example

$(p \vee q) \wedge (\neg p \vee r \vee q) \equiv (p \vee q) \wedge (q \vee r)$[1]

[1] This simplification is shown as an example. It does not necessarily follow from the rules above.

Database statistics

DBMSs maintain *database statistics*. This is data about the tables in a database. It is often called the *Catalog*. This information is used to assist query optimisation.

For each relation, 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.

Transformation rules for relational algebra

Using transformation rules, we can transform a relational algebra expression into a more efficient expression. Some examples follow. This is not an exhaustive list of rules. There are many and many others could be derived.

Rule 1: Cascade of selections

We can apply a conjunct of expressions in a select condition individually or as nested single selections.

$$\sigma_{p \wedge q \wedge r}(R) = \sigma_p(\sigma_q(\sigma_r(R)))$$

Which way round we perform this transformation depends on the context. It might be quicker to apply a separate sequence of selections as a single selection with multiple conditions (if the table is small, for example). However, it might be better to do it the other way round, depending on the cardinality of the relation and the relative restrictiveness of the selection predicates. For example, to apply a *highly restrictive* condition first. This might enable us to load the rest of the data into the RAM to complete the operation, rather than swapping pages of the relation file in and out of the buffer pool.

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 (Π_A) removes B and C hence those projections were 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.

$$\Pi_{A,B}(\sigma_{A=x \wedge B=y}(R)) = \sigma_{A=x \wedge B=y}(\Pi_{A,B}(R))$$

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 Cartesian 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. Instead of doing the full join or product, we can restrict one of the relations first before doing the join or product, where the condition applies to only one of the relations.

Rule 6: Projection and theta join and Cartesian 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 $\bar{A}$ contains attributes from R and $\bar{B}$ contains attributes form 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 Cartesian product

$$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$$
$$(R \times S) \times T = R \times (S \times T)$$

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, for example, rule 4, to reorder an expression to apply selection as soon as possible. Selection predicates on the same relation should be moved/kept together.

2. **Reorder joins to apply more selective conditions first.** The associativity of the binary operators allows us to reorder expressions containing joins so that the most restrictive condition is applied first. This will result in a smaller operand being created for subsequent join operations.

Heuristic strategies for query optimisation

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.

Examples: the music database

Consider the following Catalog information for the music database.

N(Album) = 5000, S(Album) = 10
N(Customer) = 30000, S(Album) = 16
N(Sales) = 40000, S(Album) = 8
N(Review) = 500, S(Review) = 4

Example 1

Consider the following query

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

One possible relational algebra expression for this query is

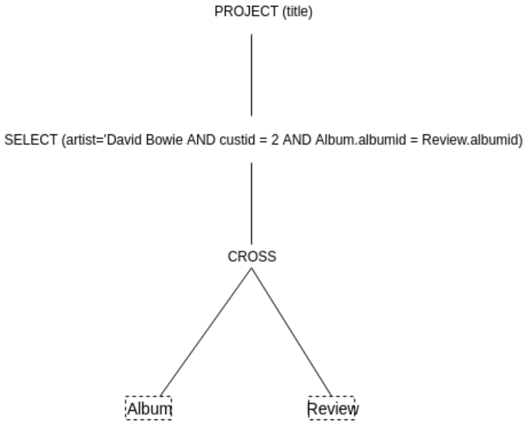
$$\Pi_{title} \\ (\sigma_{artist=\text{‘}DavidBowie\text{'} \wedge custid=2 \wedge Album.albumid=Review.albumid} \\ (Album \times Review))$$

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;
```

Example 1: 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

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

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(Album)$\times$ N(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

$$\Pi_{title}\\ (\sigma_{artist='DavidBowie' \wedge custid=2 \wedge Album.albumid=Review.albumid}\\ (Album \times Review))$$

Is equivalent to

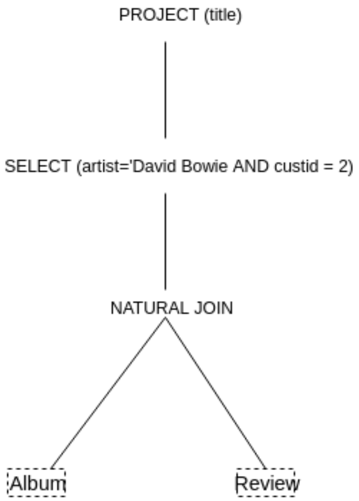
$$\Pi_{title}(\sigma_{artist='DavidBowie' \wedge custid=2}(Album \bowtie Review))$$

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.



Example 1

$$\Pi_{title}(\sigma_{artist='DavidBowie' \wedge custid=2}(Album \bowtie Review))$$

If we continue to look at the new expression, we note that the condition $artist = 'DavidBowie'$ applies only to the Album table and $custid = 2$ applies only to the Review table. We could apply those selections earlier. This might be worth doing since for the Album table N(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.

$$\Pi_{title}(\sigma_{artist='DavidBowie'}(Album) \bowtie \sigma_{custid=2}(Review))$$

Example 1

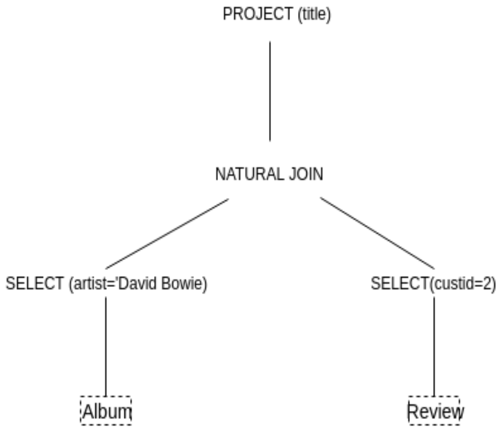
$$\Pi_{title}(\sigma_{artist='DavidBowie'}(Album) \bowtie \sigma_{custid=2}(Review))$$

And an equivalent SQL SELECT command for this would be

```
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.



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

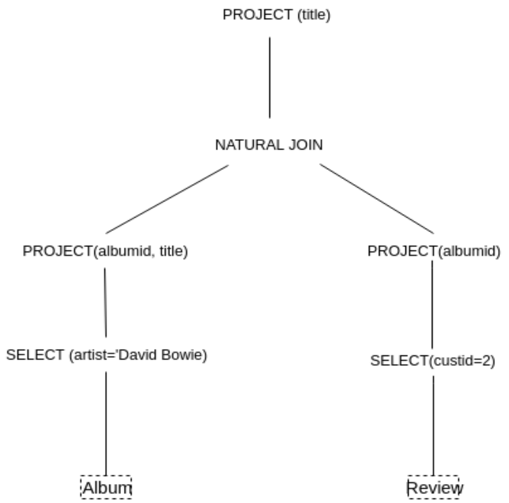
$$\Pi_{title} \\ (\Pi_{albumid,title}(\sigma_{artist=`DavidBowie'}(Album)) \\ \bowtie \\ (\Pi_{albumid}(\sigma_{custid=2}(Review)))$$

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.



Example 1

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. That is perhaps the one that should be preferred by us in this case.

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

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.[2]

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 3×10^{15} rows!

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

[2]Currency was never defined in the music database but that doesn't matter for the example.

Example 2

A possible relational algebra expression for this query is

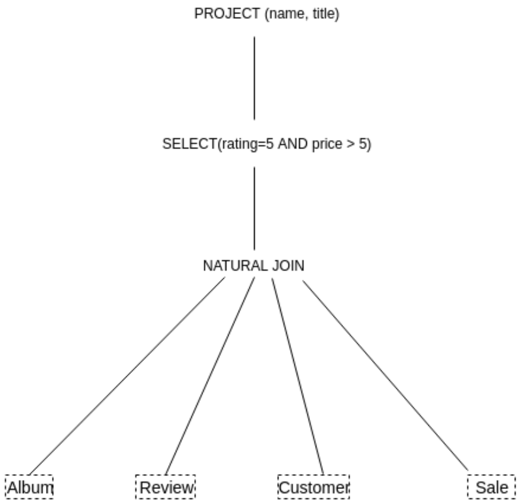
$$\Pi_{name,title}(\sigma_{rating=5 \wedge price>5}(Customer \bowtie Sale \bowtie Album \bowtie Review))$$

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.

If we compare them we see that $Customer \bowtie Sale$ is an operation on 30000*40000 tuples. $Sale \bowtie Album$ is an operation on (only) 40000*5000 tuples. $Album \bowtie Review$ is an operation on 5000*500 tuples.

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.

Example 2: relational algebra expression tree

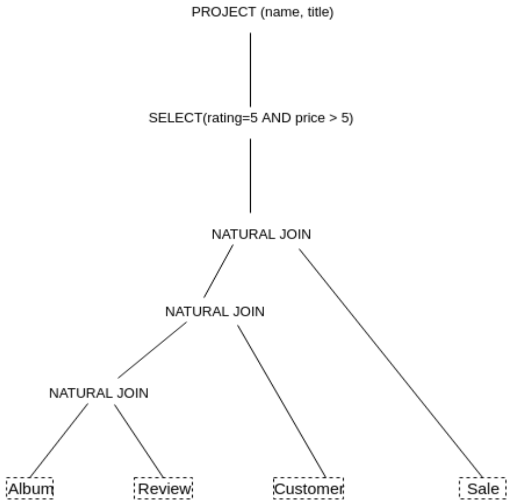


Example 2

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 the smallest 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.

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

Example 2: relational algebra expression tree

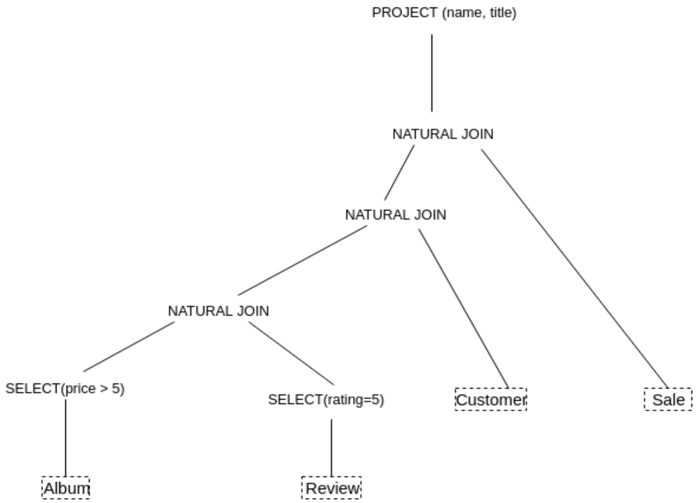


Example 2

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 $\mathit{rating} = 5$ 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 $\mathit{price} > 5$ 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.

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

Example 2: relational algebra expression tree

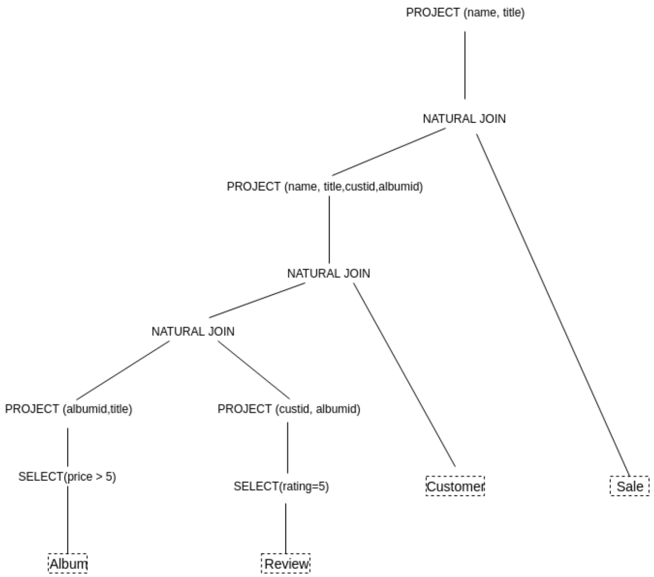


Example 2

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.

$$\Pi_{name,title} (Sale \bowtie (\Pi_{custid,albumid,title,name}(Customer \bowtie (\Pi_{albumid,title}(\sigma_{price>5}(Album)) \bowtie \Pi_{albumid,custid}(\sigma_{rating=5}(Review))))))$$

Example 2: relational algebra expression tree



The role of indexes

Of course, if an index exists for attributes in a relation, then a join doesn't necessarily even require the full table. It can use the index. This will be quicker since, as discussed, the index is usually far smaller than the table. Joins involving indexes will be faster as a result.