Query planning

Denis Miginsky

The question of the day

For the following query

```
SELECT DISTINCT p.WARE, m.COMPANY
FROM MANUFACTURER m, PRODUCT p, CATEGORY c
WHERE c.CLASS='Raw food' AND m.RECIPE_ID=p.RECIPE_ID
  AND c.WARE=p.WARE
ORDER BY p.WARE ASC
LIMIT 10
```

what is the asymptotic complexity?

Possible query plan (1)

Let's implement SQL query as a functional program

One can use FILTER, MAP, TAKE, etc. as in any other functional language

X CROSS_PROD Y = [(x,y) | x <- X, y <-Y]   -- Haskell-like

Plan (pseudo-code)
MANUFACTURER CROSS_PROD PRODUCT
-> CROSS_PROD CATEGORY
-> FILTER c.CLASS='Raw food'
-> FILTER m.RECIPE_ID=p.RECIPE_ID
-> FILTER c.WARE=p.WARE
-> SORT_BY p.WARE
-> DISTINCT
-> MAP (p.WARE, m.COMPANY)
-> TAKE 10

Let:

M=size(MANUFACTURER) ~1000

P=size(PRODUCT) ~2000

C=size(CATEGORY) ~20

Q: What is the asymptotic complexity of this plan?

Complexity

Plan (pseudo-code)	Complexity	Cardinality
MANUFACTURER CROSS_PROD PRODUCT	O(M*P)	M*P
-> CROSS_PROD CATEGORY	O(M*P*C)	M*P*C
-> FILTER c.CLASS='Raw food'	O(M*P*C)	M*P*C' | C' < C
-> FILTER m.RECIPE_ID=p.RECIPE_ID	O(M*P*C')	P*C'
-> FILTER c.WARE=p.WARE	O(P*C')	~P*C'/C < P
-> SORT_BY p.WARE	O(P*C'/C*log(P*C'/C))	~P*C'/C
-> MAP (p.WARE, m.COMPANY)	O(P*C'/C)	~P*C'/C
->DISTINCT	O(P*C'/C*log(P*C'/C))	~P*C'/C
-> TAKE 10	10	10
TOTAL:	**O(M*P*C)**	**10**

What options are available to lower the complexity?

Option 1: algebraic properties

Fortunately, SQL is **not** a functional language. In fact, it is a level higher than a functional language. This means that:

1. Query can be compiled to the functional program in many ways

2. Since SQL is an implementation of relational algebra, more properties could be in use besides just a “crude” β-reduction

3. Such properties are the algebraic properties of the relational algebra’s operations: associativity and commutativity.

Some properties of relational algebra

Not so formally speaking:

- Joins are associative: **A** $\bowtie$ **B** and **A** $\bowtie$ **C** can be computed in any order (assuming any type of the inner or even the outer join)
- Inner joins are commutative (in fact, because formally they are not because we have to "revert" the θ-join predicate)
- Selections are associative with each other and even with joins (until there are enough attributes on the particular stage to apply the filter)

Thus: there are multiple variants to reorder the plan's elements.

Goals and hints for optimization

- **Goal:** achieve the total complexity as close as possible to the final cardinality of the query
- **Sub-goal:** keep the complexity as small as possible of each stage
- **Rule:** keep the cardinality as small as possible
- **Hint:** place the filters as early as possible (they always reduce the cardinality)
- **Hint:** place the joins (and other operations) with the lower cardinality as early as possible

Better plan (2)

Plan (pseudo-code)	Complexity	Cardinality
CATEGORY FILTER c.CLASS='Raw food'	O(C)	C’ < C
-> CROSS_PROD PRODUCT	O(P*C’)	P*C’
-> FILTER c.WARE=p.WARE	O(P*C’)	~P*C’/C < P
-> CROSS_PROD MANUFACTURER	O(M*P*C’/C)	~M*P*C’/C
-> FILTER m.RECIPE_ID=p.RECIPE_ID	O(M*P*C’/C)	~P*C’/C
-> SORT_BY p.WARE	O(P*C’/C*log(P*C’/C))	~P*C’/C
-> MAP (p.WARE, m.COMPANY)	O(P*C’/C)	~P*C’/C
-> DISTINCT	O(P*C’/C*log(P*C’/C))	~P*C’/C
-> TAKE 10	10	10
TOTAL:	**O(M*P*C’/C)**	**10**

Much better complexity! However, it is terrible yet (any non-linear is terrible).

Option 2: better algorithms

- There are multiple algorithms for the special cases of join (Cartesian product with the filter is far from the best)

- There are better algorithms for the special cases of selection rather than the crude **filter**

- Better algorithms require specific properties of the data organization

Indices

In general, the **index** is the special data structure that contains the entries from the original table (but, probably, not the whole entries), that helps with searching.

Tree index – the most common index, implementing multimap **index_attr ↠ row_id** (by a sort of balanced tree, usually a variant of B-tree) where **index_attr** – the attribute of the choice, **row_id** – internal DB identifier of the row (with the lookup complexity of **O(1)**).

The tree index has the following properties:

- The lookup and the insertion costs are **O(log(N))** per element
- The iteration cost is **O(1)** per element
- All the entries are ordered by **index_attr**

Join algorithms

- Nested loop join
- Hash join
- Merge join (sort-merge join)

Nested loops join

Nested loops join of **TAB1** and **TAB2** on predicate **P:**

For each **row1** from **TAB1**, for each **row2** from **TAB2**:
  result += (**row1**, **row2**) when **P(row1**, **row2)**

Assuming **M**=**size**(**TAB1**), **N**=**size**(**TAB2**)

Pros:

- Any predicate is supported
- No prerequisites on the data organization
- Preserves order of **TAB1**

Cons:

- Complexity **O(M*N)**

Hash join

Hash join of **TAB1** and **TAB2** on attributes **a1(TAB1)** and **a2 (TAB2):**

Assuming **IDX2**=**index**(**TAB2**, **a2**) (hash-table originally)

For each **row1** from **TAB1** (called the leading table)**:**
  result += **IDX2**[**a1**(**row1**)]

Pros:

- Complexity **O(N*log(N))** + **O(M*log(N))** (no first part when the index is pre-built)
- Preserves the order of **TAB1**

Cons:

- Limited predicates are supported (= for traditional hash-join, also <,> for DB-version)

Original vs DB hash-joins

Hash-join is usually recommended to be used when at least one table is original and has pre-built index.

When both tables have indices, a smaller one is preferred as the leading one.

<u>Complexity is asymmetrical for TAB1 and TAB2</u>.

	Original	DB
Index type	hash-table	tree-map
Recommended leading table	largest	smallest
Predicates	=	=, >, <

Merge-join

Merge join of **TAB1** and **TAB2** on attributes **a1(TAB1)** and **a2 (TAB2):**

Sort **TAB1** and **TAB2**.

Next use the algorithm similar to merging ordered arrays in merge-sort.

Pros:

- Complexity **O(M*log(M)) + O(N*log(N)) + O(M) + O(N)** (no first part when TAB1 and TAB2 either pre-sorted or have indices)
- The result is sorted

Cons:

- Limited predicates are supported (=, <, >)

Plan 2

Plan (pseudo-code)	Complexity	Cardinality
CATEGORY FILTER c.CLASS='Raw food'	O(C)	C' < C
-> NL_JOIN PRODUCT ON c.WARE=p.WARE	O(P*C)	~P*C'/C < P
-> NL_JOIN MANUFACTURER ON m.RECIPE_ID=p.RECIPE_ID	O(M*P*C'/C)	~P*C'/C
-> SORT_BY p.WARE	O(P*C'/C*log(P*C'/C))	~P*C'/C
-> MAP (p.WARE, m.COMPANY)	O(P*C'/C)	~P*C'/C
->DISTINCT	O(P*C'/C*log(P*C'/C))	~P*C'/C
-> TAKE 10	10	10
TOTAL:	**O(M*P*C'/C)**	**10**

The same plan, just re-written with nested loops join

Plan 3

Plan (pseudo-code)	Complexity	Cardinality
CATEGORY FILTER c.CLASS='Raw food'	O(C)	C’ < C
-> HASH_JOIN PRODUCT INDEX BY WARE ON c.WARE=p.WARE	O(P*C’/C)	~P*C’/C < P
-> HASH_JOIN MANUFACTURER INDEX BY RECIPE_ID ON m.RECIPE_ID=p.RECIPE_ID	O(P*C’/C)	~P*C’/C
-> SORT_BY p.WARE	O(P*C’/C*log(P*C’/C))	~P*C’/C
-> MAP (p.WARE, m.COMPANY)	O(P*C’/C)	~P*C’/C
->DISTINCT	O(P*C’/C*log(P*C’/C))	~P*C’/C
-> TAKE 10	10	10
TOTAL:	**O(P*C’/C*log(P*C’/C))**	**10**

Same as plan 2, but with a better join algorithm

Option 3: laziness

Why shall we process the entire result if we need only a few top rows?

Which of the following operations could be lazy?

- Nested loops join
- Hash join
- Merge join
- Selection/filter
- Projection
- Ordering
- Distinct
- Count

Plan 3 with laziness

Plan (pseudo-code)	Complexity	Cardinality
CATEGORY FILTER c.CLASS='Raw food'	O(C)	C' < C
-> HASH_JOIN PRODUCT INDEX BY WARE ON c.WARE=p.WARE	O(P*C'/C)	~P*C'/C < P
-> HASH_JOIN MANUFACTURER INDEX BY RECIPE_ID ON m.RECIPE_ID=p.RECIPE_ID	O(P*C'/C)	~P*C'/C
-> **SORT_BY** p.WARE	O(P*C'/C*log(P*C'/C))	~P*C'/C
-> MAP (p.WARE, m.COMPANY)	~10	~10
->DISTINCT	~30	10
-> TAKE 10	10	10
TOTAL:	**O(P*C'/C*log(P*C'/C))**	**10**

The complexity is the same, non-linear still.

The sorting instruction breaks the laziness. Can we fix this?

Plan 4

Plan (pseudo-code)	Complexity	Cardinality
CATEGORY INDEX BY WARE FILTER c.CLASS='Raw food'	~20	C’ < C
-> MERGE_JOIN PRODUCT INDEX BY WARE ON c.WARE=p.WARE	~100 = ~10*C/C’	~10
-> HASH_JOIN MANUFACTURER INDEX BY RECIPE_ID ON m.RECIPE_ID=p.RECIPE_ID	~100	~10
-> MAP (p.WARE, m.COMPANY)	~10	~10
->DISTINCT	~30	10
-> TAKE 10	10	10
TOTAL:	**~270**	**10**

That is the best plan I have for now. Can you do it better?