
Jens Teubner, DBIS Group
jens.teubner@cs.tu-dortmund.de

Winter 2015/16
Part VI

Query Optimization
We already saw that there may be more than one way to answer a given query.

Which one of the join operators should we pick? With which parameters (block size, buffer allocation, . . .)?

The task of finding the best execution plan is, in fact, the holy grail of any database implementation.
Plan Generation Process

- **Parser**: syntactical/semantical analysis
- **Rewriting**: optimizations independent of the current database state (table sizes, availability of indexes, etc.)
- **Optimizer**: optimizations that rely on a cost model and information about the current database state
- The resulting **plan** is then evaluated by the system’s **execution engine**.
Finding the right plan can dramatically impact performance.

\[
\text{SELECT } \text{L.L\_PARTKEY, L.L\_QUANTITY, L.L\_EXTENDEDPRICE} \\
\text{FROM LINEITEM L, ORDERS O, CUSTOMER C} \\
\text{WHERE L.L\_ORDERKEY = O.O\_ORDERKEY} \\
\text{AND O.O\_CUSTKEY = C.C\_CUSTKEY} \\
\text{AND C.C\_NAME = 'IBM Corp.'} \\
\]

In terms of execution times, these differences can easily mean “seconds versus days.”
Besides some analyses regarding the syntactical and semantical correctness of the input query, the parser creates an **internal representation** of the input query.

This representation still resembles the original query:

- Each **SELECT-FROM-WHERE** clause is translated into a **query block**.

\[
\text{SELECT } \text{proj-list} \\
\text{FROM } R_1, R_2, \ldots, R_n \\
\text{WHERE } \text{predicate-list} \\
\text{GROUP BY } \text{groupby-list} \\
\text{HAVING } \text{having-list}
\]

- Each \( R_i \) can be a base relation or another query block.
The parser output is fed into a **rewrite engine** which, again, yields a tree of query blocks.

It is then the **optimizer’s** task to come up with the optimal **execution plan** for the given query.

Essentially, the optimizer

1. **enumerates** all possible execution plans,
2. determines the **quality** (cost) of each plan, then
3. **chooses** the best one as the final execution plan.

Before we can do so, we need to answer the question

- What is a “good” execution plan at all?
Cost Metrics

Database systems judge the quality of an execution plan based on a number of cost factors, e.g.,
- the number of disk I/Os required to evaluate the plan,
- the plan’s CPU cost,
- the overall response time observable by the user as well as the total execution time.

A cost-based optimizer tries to anticipate these costs and find the cheapest plan before actually running it.

- All of the above factors depend on one critical piece of information: the size of (intermediate) query results.
- Database systems, therefore, spend considerable effort into accurate result size estimates.
Result Size Estimation

Consider a query block corresponding to a simple SFW query $Q$.

\[
\begin{array}{c}
\pi_{\text{proj-list}} \\
\sigma_{\text{predicate-list}} \\
\times
\end{array}
\]

\[
R_1 \quad R_2 \quad \cdots \quad R_n
\]

We can estimate the result size of $Q$ based on

- the size of the input tables, $|R_1|, \ldots, |R_n|$, and
- the selectivity $sel(p)$ of the predicate $\text{predicate-list}$:

\[
|Q| \approx |R_1| \cdot |R_2| \cdot \cdots \cdot |R_n| \cdot sel(\text{predicate-list})
\]
Table Cardinalities

If not coming from another query block, the size $|R|$ of an input table $R$ is available in the DBMS’s system catalogs. 

E.g., IBM DB2:

```sql
db2 => SELECT TABNAME, CARD, NPAGES
  FROM SYSCAT.TABLES
  WHERE TABSCHEMA = 'TPCH';
```

<table>
<thead>
<tr>
<th>TABNAME</th>
<th>CARD</th>
<th>NPAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORDERS</td>
<td>1500000</td>
<td>44331</td>
</tr>
<tr>
<td>CUSTOMER</td>
<td>150000</td>
<td>6747</td>
</tr>
<tr>
<td>NATION</td>
<td>25</td>
<td>2</td>
</tr>
<tr>
<td>REGION</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>PART</td>
<td>200000</td>
<td>7578</td>
</tr>
<tr>
<td>SUPPLIER</td>
<td>10000</td>
<td>406</td>
</tr>
<tr>
<td>PARTSUPP</td>
<td>800000</td>
<td>31679</td>
</tr>
<tr>
<td>LINEITEM</td>
<td>6001215</td>
<td>207888</td>
</tr>
</tbody>
</table>

8 record(s) selected.
Estimating Selectivities

To estimate the selectivity of a predicate, we look at its structure.

GORITHMS

\[ \text{column} = \text{value} \]
\[ \text{sel}(\cdot) = \begin{cases} \frac{1}{|I|} & \text{if there is an index } I \text{ on column} \\ 1/10 & \text{otherwise} \end{cases} \]

\[ \text{column}_1 = \text{column}_2 \]
\[ \text{sel}(\cdot) = \begin{cases} \frac{1}{\max\{|I_1|,|I_2|\}} & \text{if there are indexes on both cols.} \\ \frac{1}{|I_k|} & \text{if there is an index only on col. } k \\ 1/10 & \text{otherwise} \end{cases} \]

\[ p_1 \text{ AND } p_2 \]
\[ \text{sel}(\cdot) = \text{sel}(p_1) \cdot \text{sel}(p_2) \]

\[ p_1 \text{ OR } p_2 \]
\[ \text{sel}(\cdot) = \text{sel}(p_1) + \text{sel}(p_2) - \text{sel}(p_1) \cdot \text{sel}(p_2) \]
The selectivity rules we saw make a fair amount of assumptions:

- **uniform distribution** of data values within a column,
- **independence** between individual predicates.

Since these assumptions aren’t generally met, systems try to improve selectivity estimation by gathering **data statistics**.

- These statistics are collected offline and stored in the system catalog.
  - IBM DB2: `RUNSTATS ON TABLE ...`
- The most popular type of statistics are **histograms**.
Example: Histograms in IBM DB2

```sql
SELECT SEQNO, COLVALUE, VALCOUNT
FROM SYSCAT.COLDIST
WHERE TABNAME = 'LINEITEM'
  AND COLNAME = 'L_EXTENDEDPRICE'
  AND TYPE = 'Q';
```

<table>
<thead>
<tr>
<th>SEQNO</th>
<th>COLVALUE</th>
<th>VALCOUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>+00000000000996.01</td>
<td>3001</td>
</tr>
<tr>
<td>2</td>
<td>+00000000004513.26</td>
<td>315064</td>
</tr>
<tr>
<td>3</td>
<td>+00000000007367.60</td>
<td>633128</td>
</tr>
<tr>
<td>4</td>
<td>+00000000011861.82</td>
<td>948192</td>
</tr>
<tr>
<td>5</td>
<td>+00000000015921.28</td>
<td>1263256</td>
</tr>
<tr>
<td>6</td>
<td>+00000000019922.76</td>
<td>1578320</td>
</tr>
<tr>
<td>7</td>
<td>+00000000024103.20</td>
<td>1896384</td>
</tr>
<tr>
<td>8</td>
<td>+00000000027733.58</td>
<td>2211448</td>
</tr>
<tr>
<td>9</td>
<td>+00000000031961.80</td>
<td>2526512</td>
</tr>
<tr>
<td>10</td>
<td>+00000000035584.72</td>
<td>2841576</td>
</tr>
<tr>
<td>11</td>
<td>+00000000039772.92</td>
<td>3159640</td>
</tr>
<tr>
<td>12</td>
<td>+00000000043395.75</td>
<td>3474704</td>
</tr>
<tr>
<td>13</td>
<td>+00000000047013.98</td>
<td>3789768</td>
</tr>
</tbody>
</table>

SYSCAT.COLDIST also contains information like

- the \( n \) most frequent values (and their frequency),
- the number of **distinct** values in each histogram bucket.

Histograms may even be manipulated **manually** to tweak the query optimizer.
Join Optimization

- We’ve now translated the query into a graph of **query blocks**.
  - Query blocks essentially are a **multi-way** Cartesian product with a number of selection predicates on top.
- We can estimate the **cost** of a given **execution plan**.
  - Use result size estimates in combination with the cost for individual join algorithms in the previous chapter.

We are now ready to **enumerate** all possible execution plans, *e.g.*, all possible **3-way** join combinations for a query block.
How Many Such Combinations Are There?

- A join over \( n + 1 \) relations \( R_1, \ldots, R_{n+1} \) requires \( n \) binary joins.
- Its root-level operator joins sub-plans of \( k \) and \( n - k - 1 \) join operators (\( 0 \leq k \leq n - 1 \)):

  \[
  \begin{array}{c}
    \Join \\
    k \text{ joins} \\
    R_1, \ldots, R_k \\
    \downarrow \\
    \downarrow \\
    \downarrow \\
    n - k - 1 \text{ joins} \\
    R_{k+1}, \ldots, R_{n+1}
  \end{array}
  \]

- Let \( C_i \) be the number of possibilities to construct a binary tree of \( i \) inner nodes (join operators):

  \[
  C_n = \sum_{k=0}^{n-1} C_k \cdot C_{n-k-1}
  \]
This recurrence relation is satisfied by Catalan numbers:

\[ C_n = \sum_{k=0}^{n-1} C_k \cdot C_{n-k-1} = \frac{(2n)!}{(n+1)!n!} , \]

describing the number of ordered binary trees with \( n + 1 \) leaves.

For each of these trees, we can permute the input relations \( R_1, \ldots, R_{n+1} \), leading to

\[ \frac{(2n)!}{(n+1)!n!} \cdot (n + 1)! = \frac{(2n)!}{n!} \]

possibilities to evaluate an \((n + 1)\)-way join.
The resulting search space is **enormous**:

<table>
<thead>
<tr>
<th>number of relations $n$</th>
<th>$C_{n-1}$</th>
<th>join trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>120</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
<td>1,680</td>
</tr>
<tr>
<td>6</td>
<td>42</td>
<td>30,240</td>
</tr>
<tr>
<td>7</td>
<td>132</td>
<td>665,280</td>
</tr>
<tr>
<td>8</td>
<td>429</td>
<td>17,297,280</td>
</tr>
<tr>
<td>10</td>
<td>4,862</td>
<td>17,643,225,600</td>
</tr>
</tbody>
</table>

- And we haven’t yet even considered the use of $k$ **different join algorithms** (yielding another factor of $k^{(n-1)}$)!
The traditional approach to master this search space is the use of **dynamic programming**.

**Idea:**
- Find the cheapest plan for an $n$-way join in $n$ passes.
- In each pass $k$, find the best plans for all $k$-relation sub-queries.
- **Construct** the plans in pass $k$ from best $i$-relation and $(k - i)$-relation sub-plans found in earlier passes ($1 \leq i < k$).

**Assumption:**
- To find the optimal **global plan**, it is sufficient to only consider the optimal plans of its sub-queries.
Example: Four-Way Join

Pass 1 (best 1-relation plans)
Find the best access path to each of the \( R_i \) individually (considers index scans, full table scans).

Pass 2 (best 2-relation plans)
For each pair of tables \( R_i \) and \( R_j \), determine the best order to join \( R_i \) and \( R_j \) (\( R_i \Join R_j \) or \( R_j \Join R_i \)):

\[
\text{optPlan}(\{R_i, R_j\}) \leftarrow \text{best of } R_i \Join R_j \text{ and } R_j \Join R_i.
\]

\[\rightarrow 12 \text{ plans to consider.}\]

Pass 3 (best 3-relation plans)
For each triple of tables \( R_i, R_j, \) and \( R_k \), determine the best three-table join plan, using sub-plans obtained so far:

\[
\begin{align*}
\text{optPlan}(\{R_i, R_j, R_k\}) & \leftarrow \text{best of } R_i \Join \text{optPlan}(\{R_j, R_k\}), \\
\text{optPlan}(\{R_j, R_k\}) & \Join R_i, \quad R_j \Join \text{optPlan}(\{R_i, R_k\}), \ldots.
\end{align*}
\]

\[\rightarrow 24 \text{ plans to consider.}\]
Example (cont.)

Pass 4 (best 4-relation plan)
For each set of four tables \( R_i, R_j, R_k, \) and \( R_l \), determine the best four-table join plan, using sub-plans obtained so far:

\[
\text{optPlan}(\{R_i, R_j, R_k, R_l\}) \leftarrow \text{best of } R_i \Join \text{optPlan}(\{R_j, R_k, R_l\}),
\]
\[
\text{optPlan}(\{R_j, R_k, R_l\}) \Join R_i, \quad R_j \Join \text{optPlan}(\{R_i, R_k, R_l\}), \ldots,
\]
\[
\text{optPlan}(\{R_i, R_j\}) \Join \text{optPlan}(\{R_k, R_l\}), \ldots.
\]

\( \rightarrow \) 14 plans to consider.

- Overall, we looked at only 50 (sub-)plans (instead of the possible 120 four-way join plans; ↗ slide 218).
- All decisions required the evaluation of simple sub-plans only (no need to re-evaluate the interior of optPlan(\( \cdot \))).
Function: find_join_tree_dp(q(R₁, ..., Rₙ))

for i = 1 to n do
    optPlan({Rᵢ}) ← access_plans(Rᵢ);
    prune_plans(optPlan({Rᵢ}));

for i = 2 to n do
    foreach S ⊆ {R₁, ..., Rₙ} such that |S| = i do
        optPlan(S) ← ∅;
        foreach O ⊂ S do
            optPlan(S) ← optPlan(S) ∪
            possible_joins(optPlan(O), optPlan(S \ O));
        prune_plans(optPlan(S));

return optPlan({R₁, ..., Rₙ});

- possible_joins(R, S) enumerates the possible joins between R and S (nested loops join, merge join, etc.).
- prune_plans(set) discards all but the best plan from set.
find_join_tree_dp() draws its advantage from filtering plan candidates early in the process.

- In our example on slide 220, pruning in Pass 2 reduced the search space by a factor of 2, and another factor of 6 in Pass 3.

Some heuristics can be used to prune even more plans:

- Try to avoid Cartesian products.
- Produce left-deep plans only (see next slides).

Such heuristics can be used as a handle to balance plan quality and optimizer runtime.

DB2 UDB: SET CURRENT QUERY OPTIMIZATION = n
The algorithm on slide 222 explores all possible shapes a join tree could take:

- **Left-deep**
- **Bushy** (everything else)
- **Right-deep**

Actual systems often prefer **left-deep** join trees.\(^\text{15}\)

- The **inner** relation is always a **base relation**.
- Allows the use of **index nested loops join**.
- Easier to implement in a **pipelined** fashion.

---

\(^{15}\) The seminal **System R** prototype, *e.g.*, considered only left-deep plans.
Join Order Makes a Difference

- XPath evaluation over relationally encoded XML data\textsuperscript{16}
- \(n\)-way self-join with a range predicate.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{path_length_vs_exec_time}
\caption{Execution time vs. path length for XPath evaluation over 35MB XML data.}
\end{figure}

http://www.pathfinder-xquery.org/
Join Order Makes a Difference

Contrast the execution plans for a 8- and a 9-step path.

- **left-deep join tree**
- **bushy join tree**

DB2’s optimizer essentially gave up in the face of 9+ joins.
Dynamic programming still has exponential resource requirements:
- time complexity: $O(3^n)$
- space complexity: $O(2^n)$

This may still be too expensive
- for joins involving many relations ($\sim 10–20$ and more),
- for simple queries over well-indexed data (where the right plan choice should be easy to make).

The greedy join enumeration algorithm jumps into this gap.
Greedy Join Enumeration

1 Function: find_join_tree_greedy(q(R₁, ..., Rₙ))

2 worklist ← ∅;

3 for i = 1 to n do

4 worklist ← worklist ∪ best_access_plan(Rᵢ);

5 for i = n downto 2 do

6 // worklist = {P₁, ..., Pᵢ}

7 find Pⱼ, Pₖ ∈ worklist and △... such that cost(Pⱼ △... Pₖ) is minimal;

8 worklist ← worklist \ {Pⱼ, Pₖ} ∪ {(Pⱼ △... Pₖ)};

// worklist = {P₁}

8 return single plan left in worklist;

- In each iteration, choose the cheapest join that can be made over the remaining sub-plans.

- Observe that find_join_tree_greedy() operates similar to finding the optimum binary tree for Huffman coding.
Discussion

Greedy join enumeration:

- The greedy algorithm has $O(n^3)$ time complexity.
  - The loop has $O(n)$ iterations.
  - Each iteration looks at all remaining pairs of plans in worklist. An $O(n^2)$ task.

Other join enumeration techniques:

- **Randomized algorithms**: randomly rewrite the join tree one rewrite at a time; use *hill-climbing* or *simulated annealing* strategy to find optimal plan.

- **Genetic algorithms**: explore plan space by combining plans ("creating offspring") and altering some plans randomly ("mutations").
Consider the query

\[
\text{SELECT 0.O\_ORDERKEY, L.L\_EXTENDEDPRICE}
\text{ FROM ORDERS O, LINEITEM L}
\text{ WHERE 0.O\_ORDERKEY = L.L\_ORDERKEY}
\]

where table ORDERS is indexed with a \textbf{clustered index} OK\_IDX on column O\_ORDERKEY.

Possible table access plans are:

- \textbf{ORDERS}
  - \textbf{full table scan}: estimated I/Os: \(N_{\text{ORDERS}}\)
  - \textbf{index scan}: estimated I/Os: \(N_{\text{OK\_IDX}} + N_{\text{ORDERS}}\).

- \textbf{LINEITEM}
  - \textbf{full table scan}: estimated I/Os: \(N_{\text{LINEITEM}}\).
Since the **full table scan** is the cheapest access method for both tables, our join algorithms will select them as the best 1-relation plans in Pass 1.\(^{17}\)

To **join** the two scan outputs, we now have the choices

- **nested loops join,**
- **hash join,** or
- **sort** both inputs, then use **merge join.**

Hash join or sort-merge join are probably the preferable candidates here, incurring a cost of \(\approx 2(N_{\text{ORDERS}} + N_{\text{LINEITEM}}).\)

\[ \text{overall cost: } N_{\text{ORDERS}} + N_{\text{LINEITEM}} + 2(N_{\text{ORDERS}} + N_{\text{LINEITEM}}). \]

\(^{17}\)Dynamic programming and the greedy algorithm happen to do the same in this example.
A Better Plan

It is easy to see, however, that there is a better way to evaluate the query:

1. Use an *index scan* to access ORDERS. This guarantees that the scan output is already in **O_ORDERKEY** order.

2. Then only *sort* LINEITEM and

3. join using *merge join*.

→ **overall cost**: \[
  (N_{\text{OK_IDX}} + N_{\text{ORDERS}}) + 2 \cdot N_{\text{LINEITEM}}.\]

Although more expensive as a standalone table access plan, the use of the index pays off in the overall plan.
The advantage of the index-based access to ORDERS is that it provides beneficial physical properties.

Optimizers, therefore, keep track of such properties by annotating candidate plans.

System R introduced the concept of interesting orders, determined by

- ORDER BY or GROUP BY clauses in the input query, or
- join attributes of subsequent joins (⇒ merge join).

In prune_plans (), retain

- the cheapest “unordered” plan and
- the cheapest plan for each interesting order.
Join optimization essentially takes a set of relations and a set of join predicates to find the best join order.

By **rewriting** query graphs beforehand, we can improve the effectiveness of this procedure.

The **query rewriter** applies (heuristic) rules, without looking into the actual database state (no information about cardinalities, indexes, etc.). In particular, it

- **rewrites predicates** and
- **unnests queries**.
Predicate Simplification

Example: rewrite

```
SELECT  *
FROM    LINEITEM L
WHERE   L.L_TAX * 100 < 5
```

into

```
SELECT  *
FROM    LINEITEM L
WHERE   L.L_TAX < 0.05
```

- Predicate simplification may enable the use of **indexes** and simplify the detection of opportunities for join algorithms.
Additional Join Predicates

Implicit join predicates as in

```
SELECT * 
FROM A, B, C 
WHERE A.a = B.b AND B.b = C.c
```

can be turned into explicit ones:

```
SELECT * 
FROM A, B, C 
WHERE A.a = B.b AND B.b = C.c
AND A.a = C.c
```

This enables plans like

```
(A \Join C) \Join B .
```

((A \Join C) would have been a Cartesian product before.)
Nested Queries

SQL provides a number of ways to write *nested queries*.

- **Uncorrelated** sub-query:

  ```sql
  SELECT *
  FROM ORDERS O
  WHERE O_O_CUSTKEY IN (SELECT C_CUSTKEY
                          FROM CUSTOMER
                          WHERE C_NAME = 'IBM Corp.')
  ```

- **Correlated** sub-query:

  ```sql
  SELECT *
  FROM ORDERS O
  WHERE O.O_O_CUSTKEY IN
  (SELECT C.C_CUSTKEY
   FROM CUSTOMER C
   WHERE C.C_ACCTBAL < O.O_TOTALPRICE)
  ```
Taking query nesting literally might be expensive.

An uncorrelated query, e.g., need not be re-evaluated for every tuple in the outer query.

Oftentimes, sub-queries are only used as a syntactical way to express a join (or a semi-join).

The query rewriter tries to detect such situations and make the join explicit.

This way, the sub-query can become part of the regular join order optimization.

Summary

Query Parser
Translates input query into (SFW-like) query blocks.

Rewriter
Logical (database state-independent) optimizations; predicate simplification; query unnesting.

(Join) Optimization
Find “best” query execution plan based on a cost model (considering I/O cost, CPU cost, . . . ); data statistics (histograms); dynamic programming, greedy join enumeration; physical plan properties (interesting orders).

Database optimizers still are true pieces of art...
“Picasso” Plan Diagrams

### 5.5 Non-Monotonic Cost Behavior

The example switch-points shown earlier, were all non-monotonic behavior. Yet another example of such switch-points, where plans were switched to different left-deep plans bordering the line, when we move from one selectivity space to another. Here, we found that result cardinalities at the 26% switch-point, an additional step-down behavior in the middle section. Specifically, the common change in all plans across the switch-point is that result cardinalities in plan P1 (bright orange) are roughly equal cost, with the difference being that in plan P2, the SUPPLIER relation participates in a hash-join, whereas in P7, the PARTSUPP relation participates in a nested-loops join. However, the cost of this sort is very low, and there is an increase in the estimated cost of nearly as much at the top of the plan tree, whereas in P7, the hash-join occurs at the 26% switch-point, an additional step-down behavior in the middle section. Specifically, the common change in all plans across the switch-point is that result cardinalities in plan P1 (orange) jump up by a factor of 70 at 50% selectivity, the estimated cost decreases. The reason for this behavior is that the optimizer alters its plan selection at the same location. This is confirmed in the corresponding reduced plan diagram where the footprints disappear.

### 5.4 Speckle Pattern

A curious pattern, similar to footprints on the beach, shows up in Figure 9, obtained with Q7 on the OptA optimizer, and a specific example is shown in Figure 12 obtained for Q21 on OptB. Operating Picasso with Q17 on OptA (at its highest optimality), we see that although the result cardinalities jump up by a factor of 70 at 50% selectivity, the estimated cost decreases. The above example showed non-monotonic behavior. The reason for this behavior is that the optimizer alters its plan selection at the same location. This is confirmed in the corresponding reduced plan diagram where the footprints disappear. However, the cost of this sort is very low, and there is an increase in the estimated cost of nearly as much at the top of the plan tree, whereas in P7, the hash-join occurs at the 26% switch-point, an additional step-down behavior in the middle section. Specifically, the common change in all plans across the switch-point is that result cardinalities in plan P1 (orange) jump up by a factor of 70 at 50% selectivity, the estimated cost decreases. The reason for this behavior is that the optimizer alters its plan selection at the same location. This is confirmed in the corresponding reduced plan diagram where the footprints disappear.

### 5.3 Footprint Pattern

Figure 6: Duplicates and Islands (Query 5, OptC)

<table>
<thead>
<tr>
<th>Databases</th>
<th># Duplicates</th>
<th># Islands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>136</td>
<td>14</td>
</tr>
<tr>
<td>Reduced</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Original</td>
<td>80</td>
<td>15</td>
</tr>
<tr>
<td>Reduced</td>
<td>38</td>
<td>3</td>
</tr>
</tbody>
</table>

### Download Picasso at