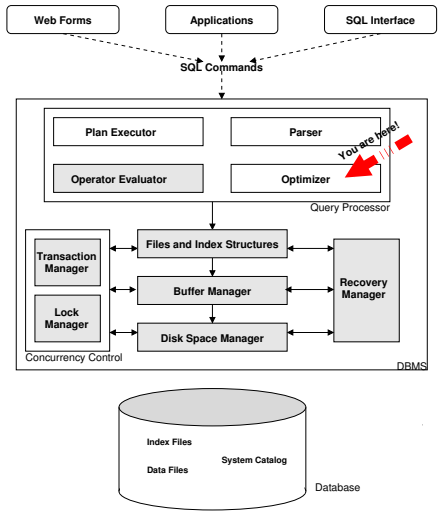
Module 15: Exploiting Interesting Orders

Module Outline



15.1 Motivation

We have seen that knowledge about "useful orders" of tuple streams (or other *interesting properties*) can speed up query processing. It is therefore desirable for a query optimizer, to exploit such information.

- The System/R optimizer could already exploit **interesting orders** for more efficient query execution plans. Order properties, attached to each pipeline (edge) in a query graph, capture sort orders originating from sorted storage, sorted index scans, or explicit sorting.
- Obviously, `order by` clauses, sort-merge-joins, duplicate elimination (`distinct` clauses), and `group by` clauses can benefit from such existing orders.

We will study two extensions of the System/R approach in this chapter.

1. DB2 built upon the System/R experience and covered many more cases.
2. Other work included more interesting properties, such as groupings and secondary orderings, thus improving query processing even further.

15.2 DB2 Order Optimization

DB2 extended the old System/R-style order optimization significantly:

- A uniform representation of order properties was developed that allows for the derivation of new order properties from given ones.
- Functional dependencies, key properties (a special case) as well as constant-selections have been taken into account in the derivation.
- Reduction and canonicalization of order properties allows for efficient checks of given vs. required orders.
- Each of the query plan operators is analyzed w.r.t. its consequences on the ordering of its output, given order information on its input(s).

15.2.1 Interesting Orders and Order Properties

We distinguish

Interesting Orders (IO): a required or useful ordering that could improve query processing speed or needs to be established, *e.g.*, for sorted output or to allow for a merge-join.

Order Properties (OP): a given ordering at some point in a query execution plan, *e.g.*, as a result of an ordered index scan.

Notice that for each (sufficiently complex) query, there will typically be multiple "interesting orders". Also, a particular order property could satisfy more than one interesting order at the same time. The optimizer has to detect, for instance,

- when an index produces an interesting order,
- the optimal place for sort operators,
- possibly avoid sorting,
- possibly sort ahead,
- opportunities for satisfying several interesting orders by a single sort operator.

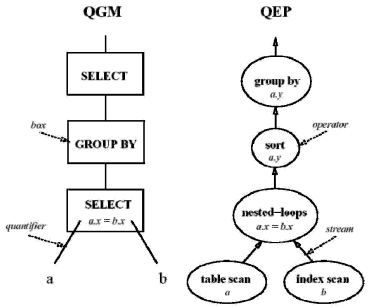
15.2.2 DB2 Approach

- Parse the input query and convert it to an intermediate form called the query graph model (QGM).
- Transform QGM into a semantically equivalent but more "efficient" QGM using heuristics, such as predicate push down, view merging, and subquery-to-join transformation.
- Perform cost-based optimization generate QEP and keep the least costly one.

Example: SQL, QGM, QEP

SQL query Q:

```
SELECT   a.y, sum(b.y)
  FROM   a, b
 WHERE   a.x = b.x
GROUP BY a.y
```



Attaching Interesting Order Information to the QGM

QGM boxes represent relational operators, arcs represent "quantifiers" (*e.g.*, table references).

DB2 applies an "order scan" of the QGM by traversing the QGM graph in a top-down manner to annotate the QGM with interesting orders:

1. Determine the input and output *requirements* for each QGM box:
   - output requirements from `ORDER BY`
   - input requirements from `GROUP BY`
2. Determine the interesting order for each `DISTINCT`
3. Determine the interesting order for merge-joins and subqueries
4. Top-down traversal to push down interesting orders, also trying to combine several interesting orders into one

Order specifications take the form of a sequence of attribute names (columns), as in "$\langle x, y, z\rangle$".

After QGM annotation, the generation of QEPs proceeds bottom-up, computing order properties on the go.

Reducing Order Specifications

Order specifications can often be reduced to simpler forms, so as to save effort and to make it easier to compare given and required orders.

Examples:

- Constant column values.

  Assume an interesting order $IO = \langle x, y\rangle$ and an order property of an input stream $OP = \langle y\rangle$. A naïve test between IO and OP would reveal that OP *does not satisfy* the required IO, hence an explicit `sort` operator would be introduced.

  If, however, a constant selection $\sigma_{x=10}$ has been applied to all records of the input stream, the interesting order IO can be rewritten into $IO_{\text{new}} = \langle y\rangle$, which is now satisfied by OP, so the sort can be avoided.

- Column equivalence.

  Assume $IO = \langle x, z\rangle$ and $OP = \langle y, z\rangle$. If we know that $x = y$ holds for all records in the input stream, we can conclude that $OP_{\text{new}} = \langle x, z\rangle$ already satisfies IO, so we don't need to sort.

  Generate column equivalence classes from these predicates.

- Take keys into account.

  Assume $IO = \langle x, y\rangle$ and $OP = \langle x, z\rangle$. If we know that x is key in the input stream, we can rewrite both $IO_{\text{new}} = \langle x\rangle$ and $IO_{\text{new}} = \langle x\rangle$, so, $IO = OP$ and we avoid the sort.

- Take Functional Dependencies into account.

  Keys are just a special case. DB2 rewrites keys and predicates into the form of FDs and uses those for "Order Reduction".

  For instance, a predicate $x = 10$ for an interesting order $IO = \langle x\rangle$ is transformed into an "empty headed FD" $\{\} \rightarrow \{x\}$, hence the implied new interesting order is $I_{\text{new}} = \{\}$.

DB2 Order Reduction Algorithm

The following algorithm is applied before checking IOs against OPs, and to get rid of spurious sort columns.

Algorithm: Reduce Order
Input: a set of FDs, applied predicates,
and an order property $O = \langle c_1, c_2, \ldots, c_n \rangle$
Output: the reduced version of O.

Rewrite O in terms of each column's equivalence class head;
Scan O backwards;
`for` (each column c_i scanned) `do`
    let $B = \{c_1, \ldots, c_{i-1}\}$, *i.e.*, the columns of O preceding c_i;
    `if` $B \rightarrow \{c_i\}$ `then`
        remove c_i from O;

Testing Order Satisfaction

After reducing both, an interesting order IO and an order property OP, the following algorithm tests whether OP already satisfies the required IO. If yes, no sorting is necessary. If not, then an explicit `sort` operator is added, but with a minimal number of sort columns.

Algorithm: Test Order
Input: an interesting order IO,
and order property OP
Output: **true**, if OP satisfies IO,
false otherwise.

Reduce OP and IO;
`if` (IO is empty or columns in IO are prefix of columns in OP) `then`
    return(**true**);
`else`
    return(**false**);

Covering Orders

The following algorithm is used during the top-down scan of the QGM to combine several interesting orders into one, producing an **order cover**.
An **order cover** of two interesting orders, IO_1 and IO_2, is another interesting order C, such that each order property OP satisfying C automatically satisfies both, IO_1 and IO_2, as well.

Example:

- $IO_1 = \langle x \rangle$ and $IO_2 = \langle x, y \rangle$ have a cover, namely $C = \langle x, y \rangle$.
- $IO_1 = \langle y, x \rangle$ and $IO_2 = \langle x, y, z \rangle$ do not have a cover.

  If, however, we assume predicate $x = 10$ holds for the input stream, then we can reduce: $IO_1^{\text{new}} = \langle y \rangle$ and $IO_2^{\text{new}} = \langle y, z \rangle$, such that now, we can find a cover: $C = \langle y, z \rangle$.

Algorithm for Computing Order Covers

Algorithm: Cover Order
Input: interesting orders IO_1 and IO_2
Output: the cover of IO_1 and IO_2,
or a return code indicating that no cover exists.

Reduce IO_1 and IO_2;
w.l.o.g., assume IO_1 is the shorter reduced order;
`if` (IO_1 is a prefix of IO_2) `then`
    return(IO_2);
`else`
    return("cannot cover IO_1 and IO_2");

Order Homogenization

The last component of the DB2 approach is the so-called order homogenization: when an interesting order is pushed down the QGM (so as to enable sort-ahead), some sort columns may have to be replaced by equivalent columns in the sub-graph.

Example:

SQL query Q:

```
SELECT   *
  FROM   a, b
 WHERE   a.x = b.x
ORDER BY a.x, b.y
```

- `order by` determines $IO = \langle a.x, b.y \rangle$
- Order scan tries to push down IO to both table accesses a and b as a sort-ahead
- For the table access to b, the equivalence class produced by $a.x = b.x$ is used to homogenize IO_b into $\langle b.x, b.y \rangle$
- IO cannot be pushed down to the access to table a, since $b.y$ is only available after the join.
- However, assume $a.x$ is a key in a that remains a key in the join, then $a.x \rightarrow b.y$, which then allows the reduction of IO_a to $\langle a.x \rangle$. This can now be pushed down to a.

Homogenization needs to be performed after order reduction. It can also make use of column equivalences that are established only *afterwards* (further up the QGM).

Algorithm for Order Homogenization

Algorithm: Homogenize Order
Input: interesting order IO
and target columns $C = \{c_1, \ldots, c_n\}$
Output: IO homogenized w.r.t. C, *i.e.*, IO_C
or a return code indicating that IO_C does not exist.

$IO_C \leftarrow$ Reduce(IO);
Using equivalence classes,
  try to substitute each column in IO_C with a column in C;
`if` (all columns in IO_C could be substituted) `then`
  return(IO_C);
`else`
  return("cannot homogenize IO to C");

Planning Phase

Once the DB2 optimizer has finished the order scan of the QGM, (required and interesting) order properties hang off the QGM boxes. Plan (QEP) generation proceeds bottom-up, considering alternative subplans for each box.

- More costly plans with comparable properties (keys, FDs, equivalences, predicates) are pruned.
- Interesting orders are used for pruning and to generate sort-ahead orders.
- For example, during join enumeration, try to sort outer operand for each interesting order. This could allow for `order by` to be pushed down the join tree.
- If no sorting is actually needed, this will be detected.
- Order push-down only works for join methods that preserve the order of the outer operand.
- Push-down needs homogenization w.r.t. the operand, and pushing beyond a merge join requires a cover with the merge join order.
- Pushing down sort-ahead orders increases join enumeration complexity by a factor of $\mathcal{O}(n^2)$ for n sort-ahead orders, since same join orders with different order properties need to be considered.

Property Propagation

During the plan generation phase, it is important to *propagate* the interesting properties (orders, keys, equivalences, and predicates) through the QEP operators and to *compare* two plans on the basis of such properties:

1. The *comparison* of two plans, P_1 and P_2 is done by overloading the symbol "$\leq$" to mean "less general or equivalent" for properties.

   Then, plan P_2 prunes plan P_1, iff $P_2.\text{cost} \leq P_1.\text{cost}$ and for each property x, $P_1.x \leq P_2.x$.

2. The *propagation* needs to be defined separately for each of the interesting properties in turn. We look at order only here.

Order Propagation

- Orders (if any) always originate from ordered index scans or sorting. Order propagates straight-forward through most relational operators, except for project and join:
  - If any column c_i of an order property $OP = \langle c_1, c_2, \ldots, c_n \rangle$ is projected away, then only the prefix $OP' = \langle c_1, c_2, \ldots, c_{i-1} \rangle$ propagates upwards.
  - For nested-loops and merge-join, the order of the outer stream is preserved. If the outer has a single record only, the order of the inner propagates. Hash joins destroy all incoming orders.
- The `Test Order` algorithm can be used to compare two order properties. If $int(OP)$ denotes a "cast" of OP into an interesting order, then

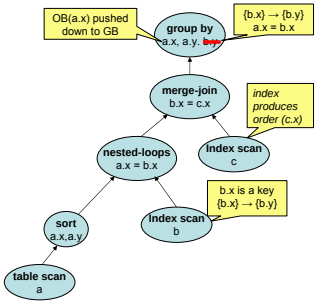
$$OP_1 \leq OP_2 \iff OP_2 \text{ satisfies } int(OP_2).$$

Propagation of key (and FD) properties is a little more complex ...

15.2.3 Example

Consider the example query

```
SELECT   a.x, a.y, b.y, sum(c.z)
  FROM   a, b, c
 WHERE   a.x = b.x  and
         b.x = c.x
GROUP BY a.x, a.y, b.y
ORDER BY a.x
```



- `Order by`'s interesting order $OB = \langle a.x \rangle$ can be pushed down and covered with `group by`'s interesting order $GB = \langle a.x, a.y, b.y \rangle$.
- This can be pushed down and covered with the merge-join order $MJ = \langle b.x \rangle$.
- Key $b.x$ gives rise to FD $b.x \rightarrow b.y$, which together with the equivalence resulting from $a.x = b.x$, allows for the reduction of GB to $GB' = \langle a.x, a.y \rangle$.
- As a result, the sort on $a.x, a.y$ simultaneously satisfies all interesting orders.

Sort-ahead (pushing the sort before the first join) is likely to produce the most efficient plan. With indexes on $b.x$ and $c.x$, it is probably more efficient than hash-based joins. With an ordered index on $a.x, a.y$, the sort could be eliminated.

15.2.4 More Complications

Actually, `group by` and `distinct` operators do not require an *exact* interesting order.

For example, the interesting order for `group by` $x, y,$ `sum(distinct`z`)` can be satisfied by $\langle x, y, z \rangle$ and $\langle y, x, z \rangle$. Also, the order could be ascending or descending.

A total of 16 different orders can satisfy the `group by` interesting order.[1]

The DB2 optimizer does not consider all of these different orders, rather, a *generalized* order specification is used that contains information on which columns can be permuted and whether or not each column can be ascending or descending.

Those "degrees of freedom" complicate all the algorithms shown above significantly ...

[1] 2^3 because of the permutations of x, y, z, each one asc/desc, giving 2^4=16

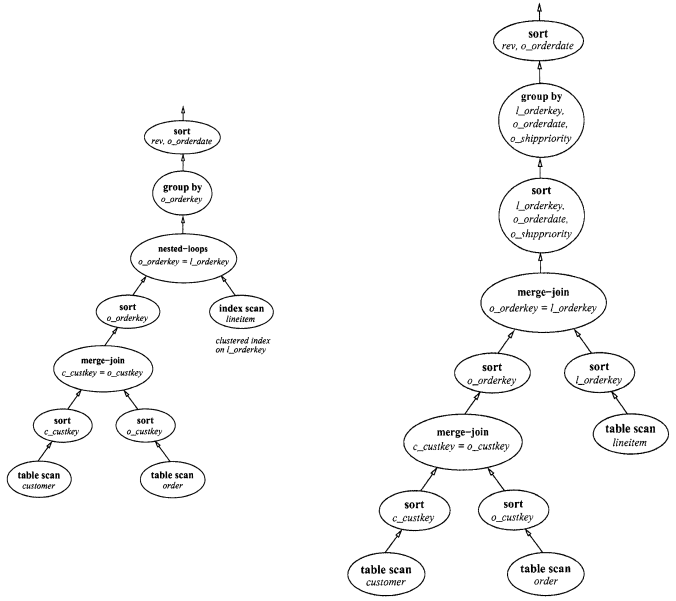
15.2.5 Experimental Results

The publications report significant improvements on "real customers' real queries", without giving the details. Essentially, just one particular TPC-D (OLAP-) benchmark query is analyzed in depth, Q3:

```
SELECT   l.orderkey, sum(l.extendedprice * (1 - l.discount)) as rev,
         o.orderdate, o.shippriority
  FROM   customer c, order o, lineitem l
 WHERE   o.orderkey = l.orderkey  and
         c.custkey = o.custkey  and
         c.mktsegment = 'building'  and
         o.orderdate < date('1995-03-15')  and
         l.shipdate > date('1995-03-15')
GROUP BY l.orderkey, o.orderkey, o.shippriority
ORDER BY rev desc, o.orderdate
```

For a 1GB TPC-D benchmark database with striped data (15 disks, 4 controllers), the DB2 optimizer kept the CPU at full speed (100% utilization) and obtained a factor of 2.04 improvement over DB2 without the order optimization (192s as opposed to 393s).

Here, we show the two QEPs, with and without order optimization:



Comparison

- By using the Reduce Order, Cover Order, and Homogenize Order algorithms, DB2 was able to push the sort for the `group by` below the nested-loops join. This made the index probes for the inner operand of the nested-loops join clustered, which allowed for prefetching and parallel I/O to the *lineitem* table, the largest of the TPC-D tables.
- The sort on $o.orderkey$ satisfied the `group by` order because fo the equivalence $o.orderkey = l.orderkey$ and the FD $o.orderkey \rightarrow \{o.orderdate, o.shippriority\}$.

  N.B. The redundant (functionally dependent) column had to be added to the `group by` clause in SQL, since otherwise it could not have been mentioned in the `select` clause.
- Without the order optimizations, DB2 could not identify that the sort on $o.orderke$ satisfies the `group by`. Further, without the awareness of equivalence classes, the same sort could not be exploited for an ordered nested loops join, hence, the optimizer generated a costly merge-join.

15.2.6 Summary of the DB2 Approach

We have seen order-aware optimization techniques employed in the DB2 query optimizer (Simmen et al., 1996).

- Push down sorts in joins (sort-ahead)
- Minimize the number of sorting columns
- Detect when sorting can be avoided
- Take advantage of predicates, keys, FDs, or indexes.

Techniques are general, they could be applied in any other query optimizer as well. Using such techniques can mean the difference between an execution plan that finishes in a few minutes versus one that takes hours to run.

15.3 Exploiting more Orders and Grouping

Only recently, renewed interest in the topic has led to new results on optimization potentials when we take into account not only simple order properties, but also *grouping* information as wells as *secondary* ordering/grouping.
This work (Wang and Cherniack, 2003) has been presented by a fellow student in a seminar presentation, that is repeated (in a revised from) here.

Even more work has been pubslished since, *e.g.*, (Neumann and Moerkotte, 2004b; Neumann and Moerkotte, 2004a).

Bibliography

Neumann, T. and Moerkotte, G. (2004a). A combined framework for grouping and order optimization. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, pages 960–971. Morgan Kaufmann.

Neumann, T. and Moerkotte, G. (2004b). An efficient framework for order optimization. In *Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, 30 March - 2 April 2004, Boston, MA, USA*, pages 461–472. IEEE Computer Society.

Simmen, D. E., Shekita, E. J., and Malkemus, T. (1996). Fundamental techniques for order optimization. In Jagadish, H. V. and Mumick, I. S., editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 57–67. ACM Press.

Wang, X. and Cherniack, M. (2003). Avoiding ordering and grouping in query processing. In *Proc. Intl. Conf. on Very Large Databases*, pages 826–837.