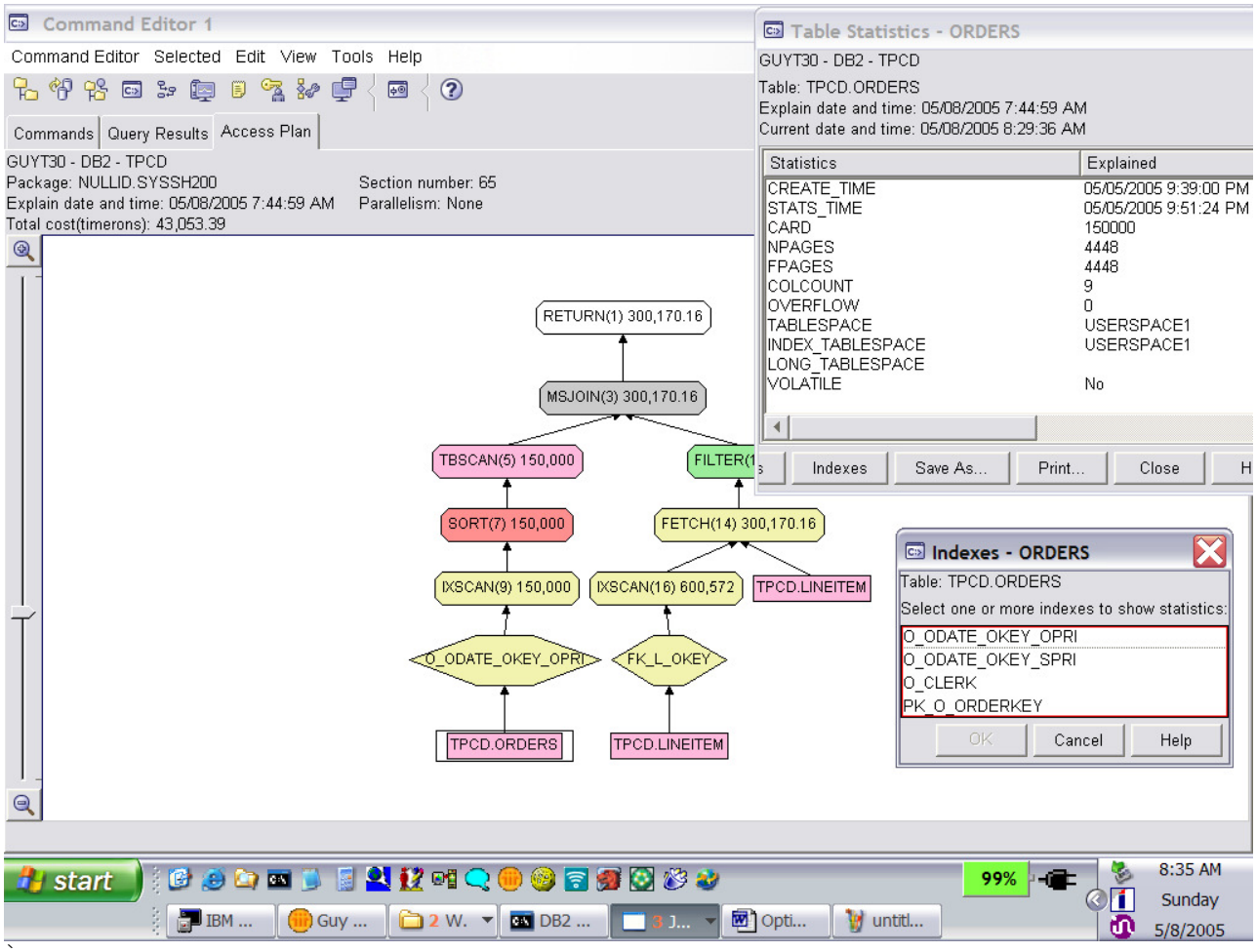
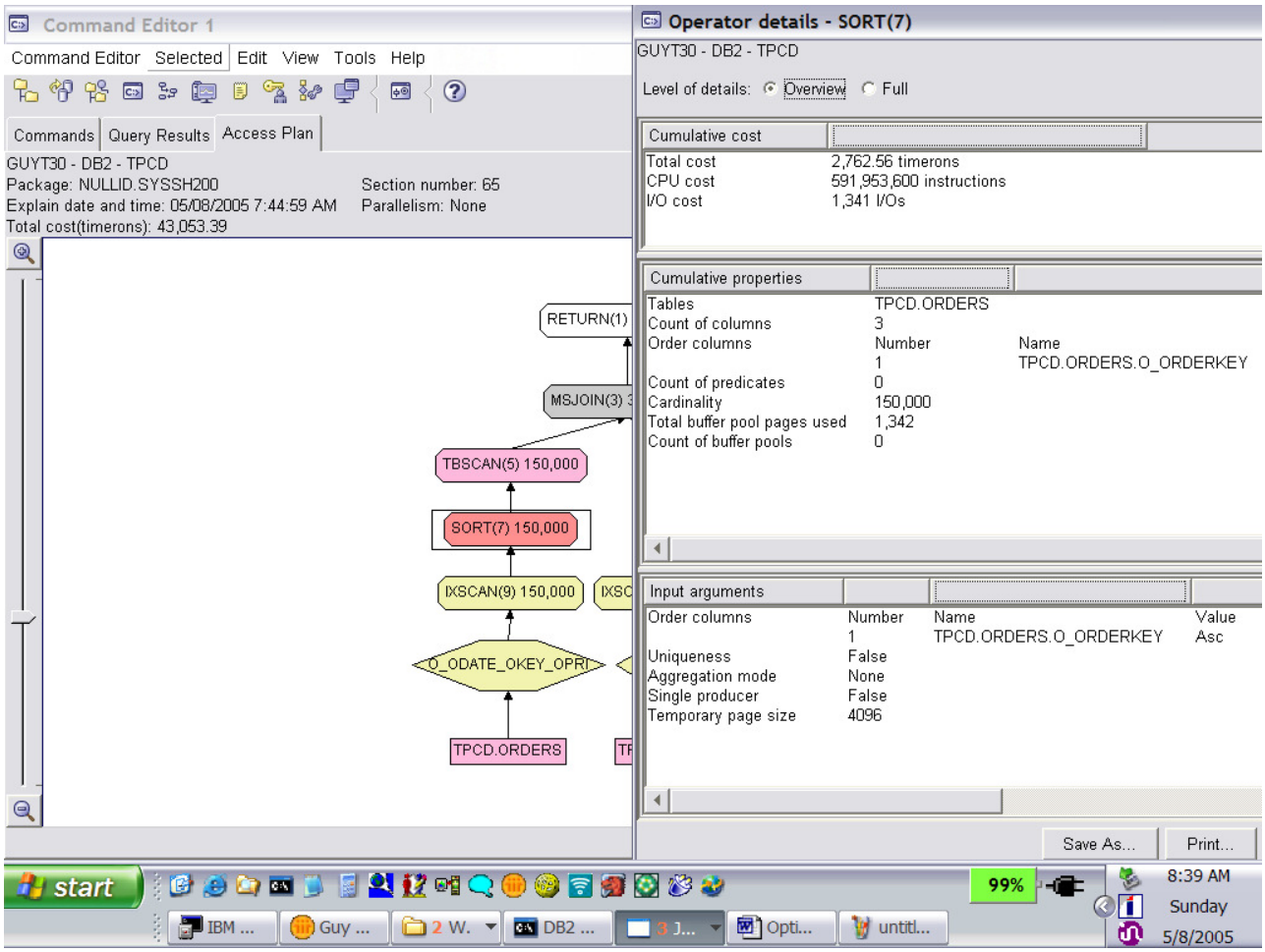
SCRIPT FOR DEMO OF DB2 FOR LUW OPTIMIZER

```
-- This script demonstrates plans produced by the DB2
-- for Linux, Unix, and Windows (LUW) Optimizer, via the Command Center within
-- the Control Center (these come free with DB2).  The Command Center
-- allows you to run DB2 queries and/or show the plans for those queries.
--
-- However, as of V9.7, with which I will be demo'ing, the support of
-- Visual Explain in the Command Center of the Control Center
-- is being deprecated in favor of a new version in Data Studio that
-- provides common Visual Explain support for both DB2 for LUW and
-- DB2 for z/OS (the mainframe).  The look and feel of the Data Studio
-- version is slightly different from the prior version, but I didn't
-- have time to replace the figures and description of how to invoke various
-- features.  However, the plans are almost all identical.
-- Unfortunately, the Data Studio version doesn't explicitly link multiple
-- SCANs of the same TEMP (known as Common Table Expressions), which is used
-- in the recursive query example, and heavily used in the CUBE and ROLLUP
-- examples. The correlation must be made by the viewer via the numbers
-- assigned to each TEMP, and indicated by a small [+] in the upper-left corner
-- of the TEMP operator.
-- GML  May 2011

-- Plans are represented as graphs of operators, common in the literature.
-- Data flows bottom up, with the leaves at the bottom being table or
-- index accesses.  Note that in DB2 plans are not limited to trees.
-- In fact, one of the queries is recursive, having a cycle in the graph
-- to produce it.

-- This demo uses the schema of the well-known TPC-H (previously known as TPC-D)
-- benchmark.  The naming convention for columns is that the first letter of
-- the table precedes the column name.  However, indexes not allowed by TPC-H
-- (but allowed by TPC-D) have been added to show off DB2's exploitation of them,
-- and some additional tables have been added to demo other DB2 Optimizer
features.

-- Right click on a LOLEPOP to see the details of its arguments, properties,
-- and cost in a separate window.  Right click on a table (rectangle) or an
-- index (diamond) to see its statistics.

CONNECT TO TPCD;

--
-- Basic plan graph:
--
SELECT O_ORDERKEY, O_ORDERDATE, O_ORDERPRIORITY, L_SHIPINSTRUCT
FROM TPCD.ORDERS O, TPCD.LINEITEM L
WHERE O_ORDERKEY = L_ORDERKEY AND L_SHIPDATE > L_COMMITDATE
ORDER BY O_ORDERKEY;

-- This simple join query is used to illustrate how to read plans in DB2's
-- Visual Explain, which is obtainable interactively from the Command Editor, a
-- part of the DB2 Control Center.  A separate window called "Overview" outlines
-- the part of this window that is being shown, which can be dragged and resized
-- to scroll over and zoom in on large graphs.
--
```

```
-- The tables referenced are shown as rectangles, usually at the bottom of the
-- graph. Indexes on those tables are shown as diamond-shaped hexagons.  All
-- other plan operators are shown as rectangles with rounded edges, each
-- containing the name of the operator, an operator number in parentheses
-- (numbered top-down, but not densely), and an estimated plan property of
-- choice, in this case cardinality (cost or I/Os could be shown).  Keep in mind
-- that the properties are the output result of that operator, and hence are
-- cumulative.  The edges of the graph show data flow, which is generally
-- bottom-up, as shown by the arrows.  The last operator is always a RETURN,
-- which shows the data returned to the user.
--
-- Double-clicking on any node shows more detailed information about that node.
-- For tables, that information is the statistics for that table.  Buttons on
-- this pop-up also list indexes on that table, the columns of that table that
-- were referenced by this query, etc.  Similarly, double-clicking on an index
-- gives its statistics, including buttons for the clustering information
-- provided by "Page fetch pairs", which empirically gives the number of pages
-- fetched for a scan with that index using different-sized buffer pools.
```

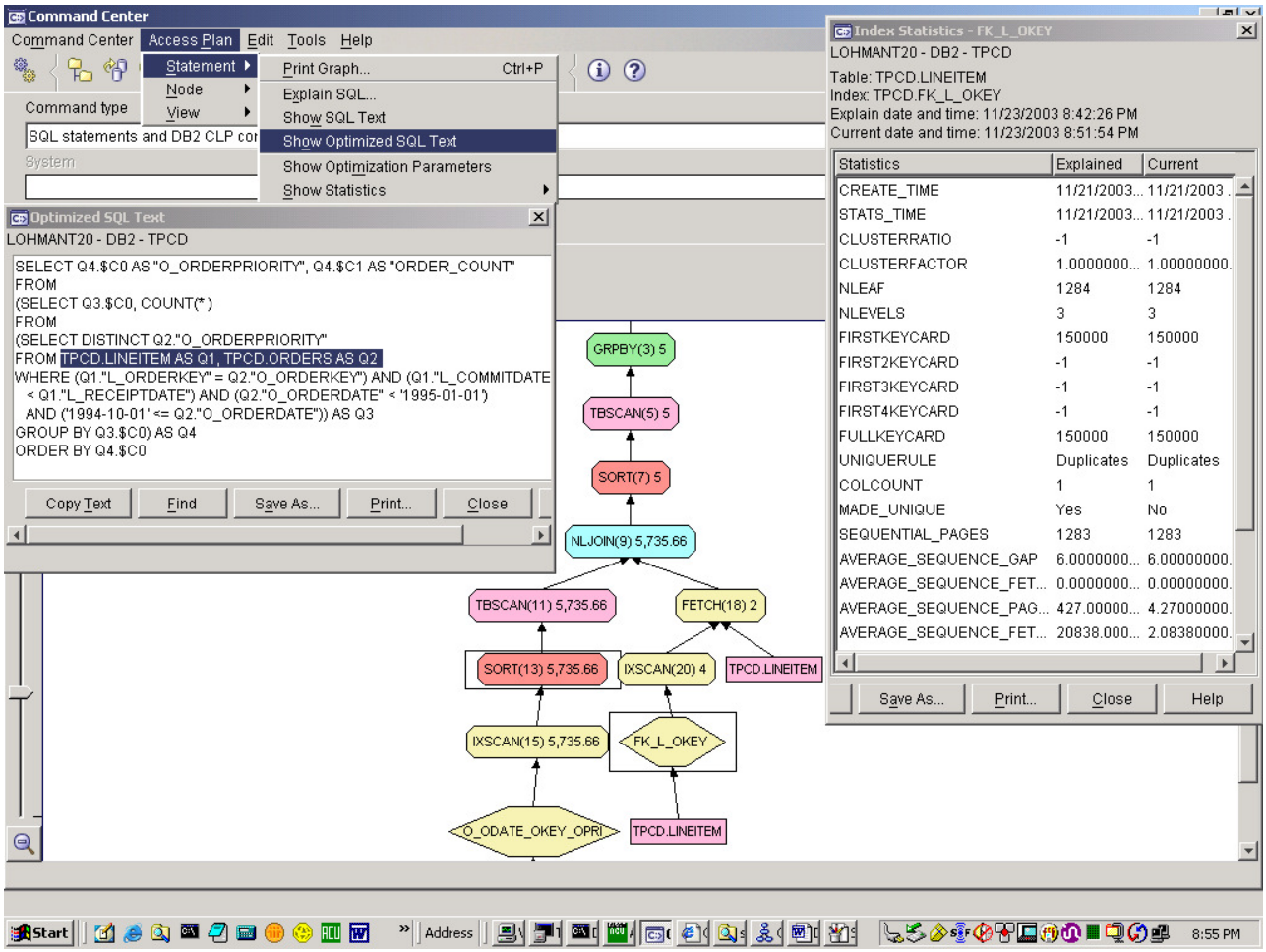


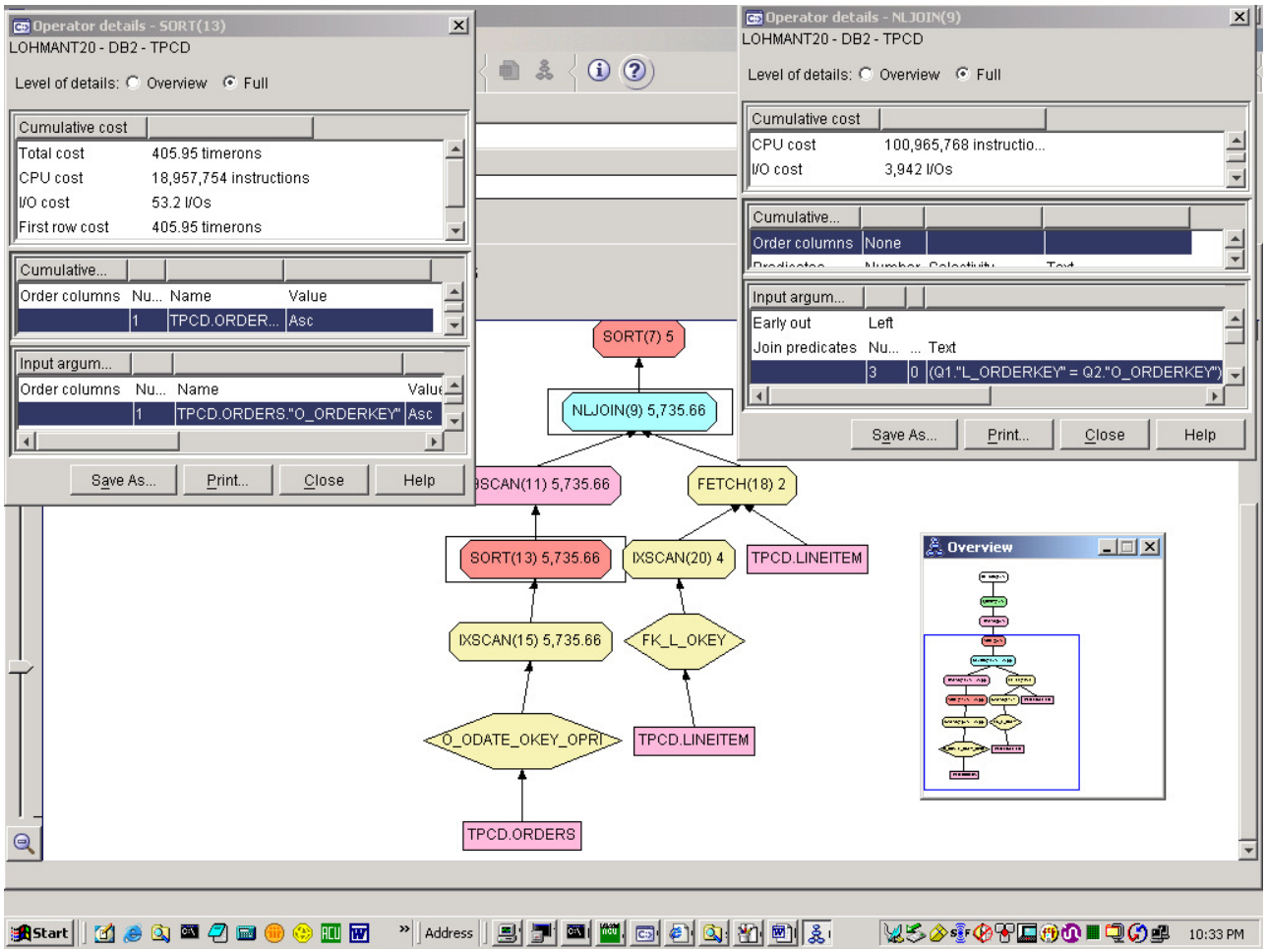

```
   -- For operators, the pop-up window contains 3 panes.  The bottom pane
   -- contains arguments of the operator.  The middle pane gives the output
   -- properties of that operator.  And the top pane gives the estimated
   -- cumulative cost of the entire plan up to and including this operator.
```




```
-- For example, for the SORT (7) operator, the most important argument is the
-- list of columns on which it will sort.  Note that the same column(s) become
-- the order columns of the property.  The estimated cumulative cost of the plan
-- so far is shown in the top pane.

-- This plan shows the (cumulative) cardinality in each box.  It's doing a merge
-- join, a good choice for bulk joining, and the Optimizer expects to return
-- 300,000 rows!  Also, note that the ORDER BY column (O_ORDERKEY) matches the
-- join column, so no SORT is needed after the join.  The outer (on the left)
-- is Orders and the inner table is Lineitem.  Both tables are doing an index
-- scan, but why is the SORT (7) necessary, if there's an index on the primary
-- key column?
--
-- NOTE: In V9.7, DB2 picks a different plan that uses the primary key index
-- on Orders and FETCHes the remaining columns, rather than using the
-- index-only index (O_ODATE_OKEY_OPRI) shown here and re-SORTing into the
-- join-key order.
--
-- The answer lies in the name of the index being used, and the
-- absence of a FETCH operator for the outer.  The index has all the columns
-- needed for this query, so we get "index-only access" with the IXSCAN (9),
-- saving any data page fetches, even though it is not applying any predicates
-- and not exploiting the order induced by the index (such an order is
-- "uninteresting" to this query, and so is lumped together as the "I don't
```

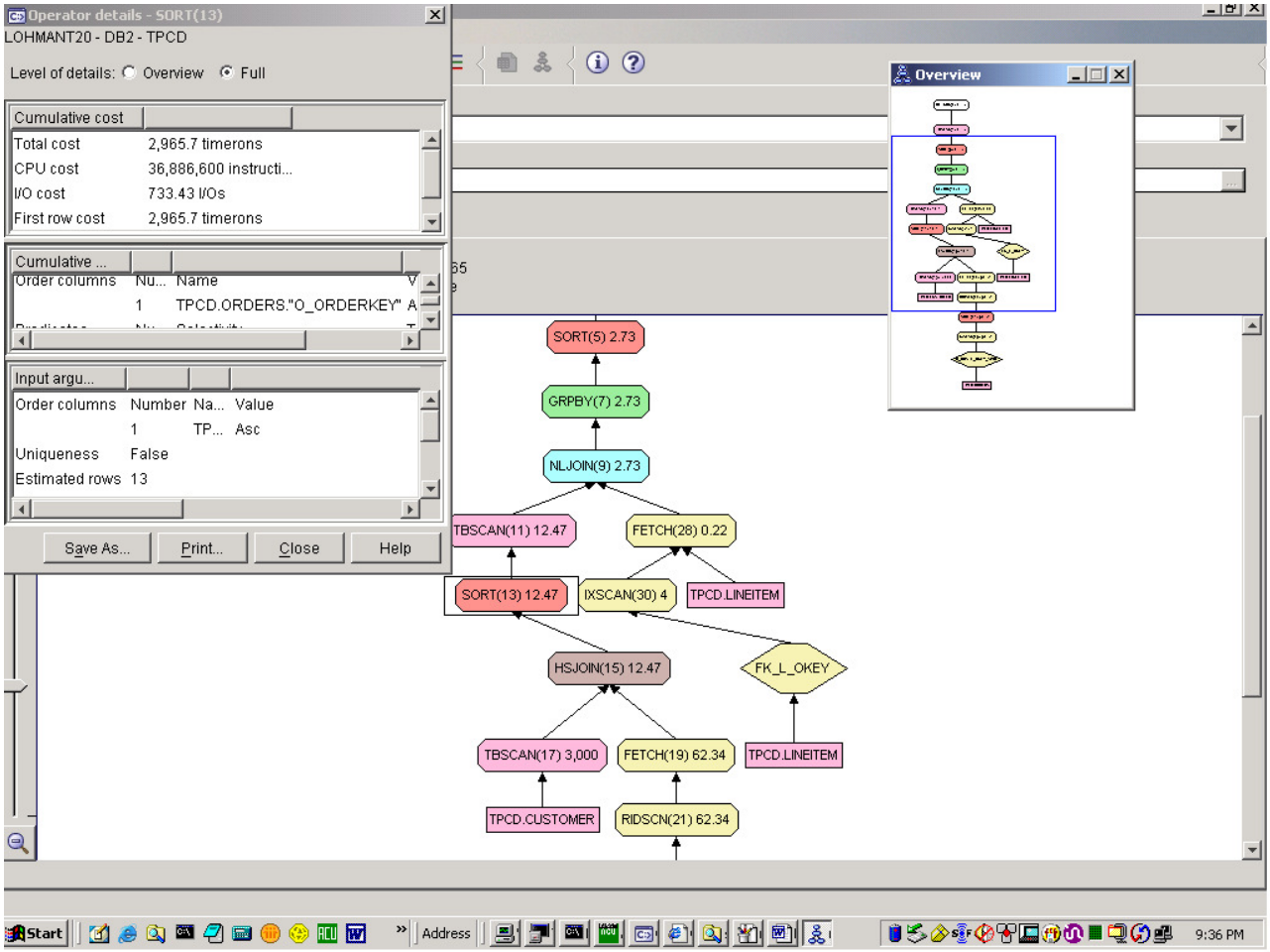
```
-- care" order).  As a result, we must sort in (7) to get the outer into
-- O_ORDERKEY order.  The inner, however, is using the foreign key index on
-- L_ORDERKEY to get the right (L_ORDERKEY) order (but to apply no predicaes).
-- It then FETCHes the data pages to get the rest of the columns referenced.
-- A FILTER (11) always appears on the inner of the merge join; its presence is
-- a historical artifact.




--
-- Query ReWrite: (Correlated) Exists Subquery to Join:
--
-- This transformation gives the Optimizer more latitude in its choices.
-- "Optimized SQL" shows the result of Query ReWrite, eliminating the SQ.
-- Note that NLJOIN is "early out" meaning that once a match is found for
-- an outer, we can stop the scan of the inner.
-- This patented technique avoids generating
-- a bunch of duplicates, only to have to eliminate them with a sort distinct.
-- This allows us to transform existential subqueries to joins, whether or
-- not the SELECT list of the subquery is a key (Oracle requires that to do
-- the transformation).

-- IXSCAN (15) is not producing any order.  Why?  It's applying the predicate
-- on O_ORDERDATE, so doesn't produce any "interesting" order.
-- Why the SORT in the outer, then?  For the ORDER BY?  No. That's SORT(7).
-- NLJOIN (9) doesn't require it, but DB2 does this "sort ahead" on the join
-- column because the inner is clustered on the join column.  By sorting the
-- outer on the join column, we do an "ordered nested-loop join".  This
-- benefits that NLJOIN by ordering the outer, so that the index's
-- (clustered) pages are referenced in a clean left-to-right sweep,
-- rather than randomly paging (and thus probably re-referencing many pages).

-- How does SORT(5) reduce the cardinality down to 5?  The aggregation is
-- pushed down into the SORT, combining rows as it finds rows with the same
-- value.  This quickly reduces the number of rows to be sorted, avoiding
-- spilling of runs.
-- Note that cardinalities of inners of NLJNs are the cardinality per outer row.

SELECT O_ORDERPRIORITY, COUNT(*) AS ORDER_COUNT
FROM TPCD.ORDERS
WHERE O_ORDERDATE >= DATE('1994-10-01') AND O_ORDERDATE < DATE('1994-10-01')
        + 3 MONTHS AND EXISTS
   (SELECT *
   FROM TPCD.LINEITEM
   WHERE L_ORDERKEY = O_ORDERKEY AND L_COMMITDATE < L_RECEIPTDATE)
GROUP BY O_ORDERPRIORITY
ORDER BY O_ORDERPRIORITY;
```


Command Center
Command Center
Access Plan
Edit
Tools
Help
Statement
Node
View
Print Graph... Ctrl+P
Explain SQL...
Show SQL Text
Show Optimized SQL Text
Show Optimization Parameters
Show Statistics
Command type
SQL statements and DB2 CLP cor
System
Optimized SQL Text
LOHMANT20 - DB2 - TPCD
SELECT Q4.$C0 AS "O_ORDERPRIORITY", Q4.$C1 AS "ORDER_COUNT"
FROM
(SELECT Q3.$C0, COUNT(*)
FROM
(SELECT DISTINCT Q2."O_ORDERPRIORITY"
FROM TPCD.LINEITEM AS Q1, TPCD.ORDERS AS Q2
WHERE (Q1."L_ORDERKEY" = Q2."O_ORDERKEY") AND (Q1."L_COMMITDATE
< Q1."L_RECEIPTDATE") AND (Q2."O_ORDERDATE" < '1995-01-01')
AND ('1994-10-01' <= Q2."O_ORDERDATE")) AS Q3
GROUP BY Q3.$C0) AS Q4
ORDER BY Q4.$C0
Copy Text
Find
Save As...
Print...
Close
GRPBY(3) 5
TBSCAN(5) 5
SORT(7) 5
NLJOIN(9) 5,735.66
TBSCAN(11) 5,735.66
FETCH(18) 2
SORT(13) 5,735.66
IXSCAN(20) 4
TPCD.LINEITEM
IXSCAN(15) 5,735.66
FK_L_OKEY
O_ODATE_OKEY_OPRI
TPCD.LINEITEM
Index Statistics - FK_L_OKEY
LOHMANT20 - DB2 - TPCD
Table: TPCD.LINEITEM
Index: TPCD.FK_L_OKEY
Explain date and time: 11/23/2003 8:42:26 PM
Current date and time: 11/23/2003 8:51:54 PM
Statistics
Explained
Current
CREATE_TIME 11/21/2003... 11/21/2003 .
STATS_TIME 11/21/2003... 11/21/2003 .
CLUSTERRATIO -1 -1
CLUSTERFACTOR 1.0000000... 1.00000000.
NLEAF 1284 1284
NLEVELS 3 3
FIRSTKEYCARD 150000 150000
FIRST2KEYCARD -1 -1
FIRST3KEYCARD -1 -1
FIRST4KEYCARD -1 -1
FULLKEYCARD 150000 150000
UNIQUERULE Duplicates Duplicates
COLCOUNT 1 1
MADE_UNIQUE Yes No
SEQUENTIAL_PAGES 1283 1283
AVERAGE_SEQUENCE_GAP 6.0000000... 6.00000000.
AVERAGE_SEQUENCE_FET... 0.0000000... 0.00000000.
AVERAGE_SEQUENCE_PAG... 427.00000... 4.27000000.
AVERAGE_SEQUENCE_FET... 20838.000... 2.08380000.
Save As...
Print...
Close
Help
Start
Address
8:55 PM



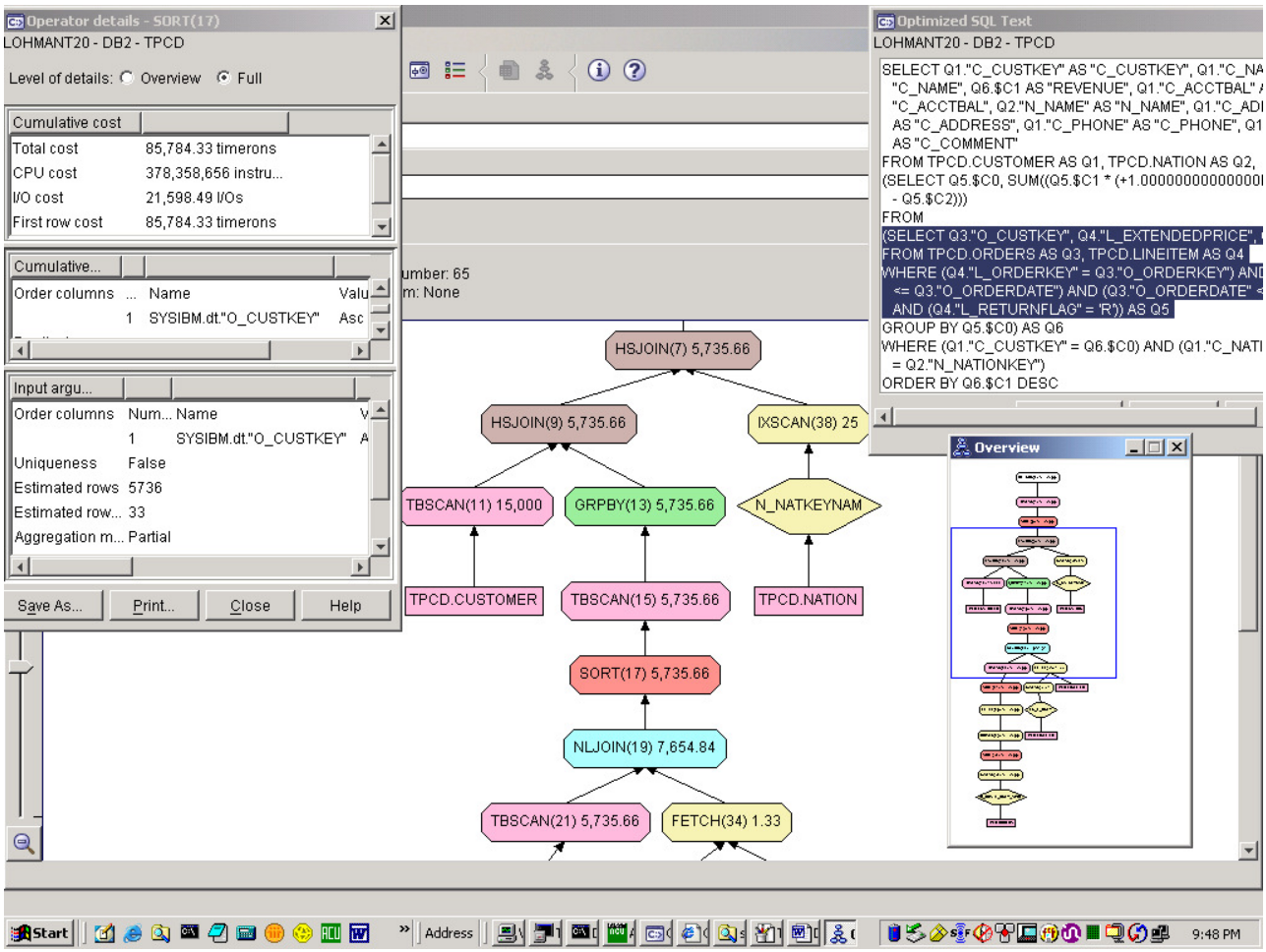

```
--
-- Order optimizations:
--
-- Shows several optimizations in order classes discussed in
-- Simmen, Shekita, and Malkemus,
-- "Fundamental Techniques for Order Optimization", SIGMOD 1996, pp. 57-67
--
-- Only need to do one SORT (13) for GROUP BY (7), on O_ORDERKEY.
-- But that's different than the query!  Here's why:
--   ++ Equivalence in GB order class: L_ORDERKEY = O_ORDERKEY is folded into GB
--   ++ Functional Dependencies: O_ORDERKEY determines other GB columns, saving
compares.
-- SORT (13) on O_ORDERKEY also benefits NLJOIN (9) as sort-ahead NLJOIN, whose
inner
--      is clustered (compare stats of index FK_L_OKEY vs. O_ODATE_OKEY_SPRI),
as above.
--
-- Also does list prefetch on O_ODATE_OKEY_SPRI, because it can apply predicates,
but
-- isn't terribly clustered (see PAGE FETCH PAIRS).
```

```
--
--
-- TPC-D query 3:
--

SELECT
  L_ORDERKEY,
  SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)) AS REVENUE,
  O_ORDERDATE,
  O_SHIPPRIORITY
FROM TPCD.CUSTOMER, TPCD.ORDERS, TPCD.LINEITEM
WHERE C_MKTSEGMENT = 'HOUSEHOLD'
  AND C_CUSTKEY = O_CUSTKEY
  AND L_ORDERKEY = O_ORDERKEY
  AND O_ORDERDATE < DATE('07/16/1991')
  AND L_SHIPDATE > DATE('07/16/1998')
GROUP BY L_ORDERKEY, O_ORDERDATE, O_SHIPPRIORITY
ORDER BY REVENUE DESC, O_ORDERDATE;
```



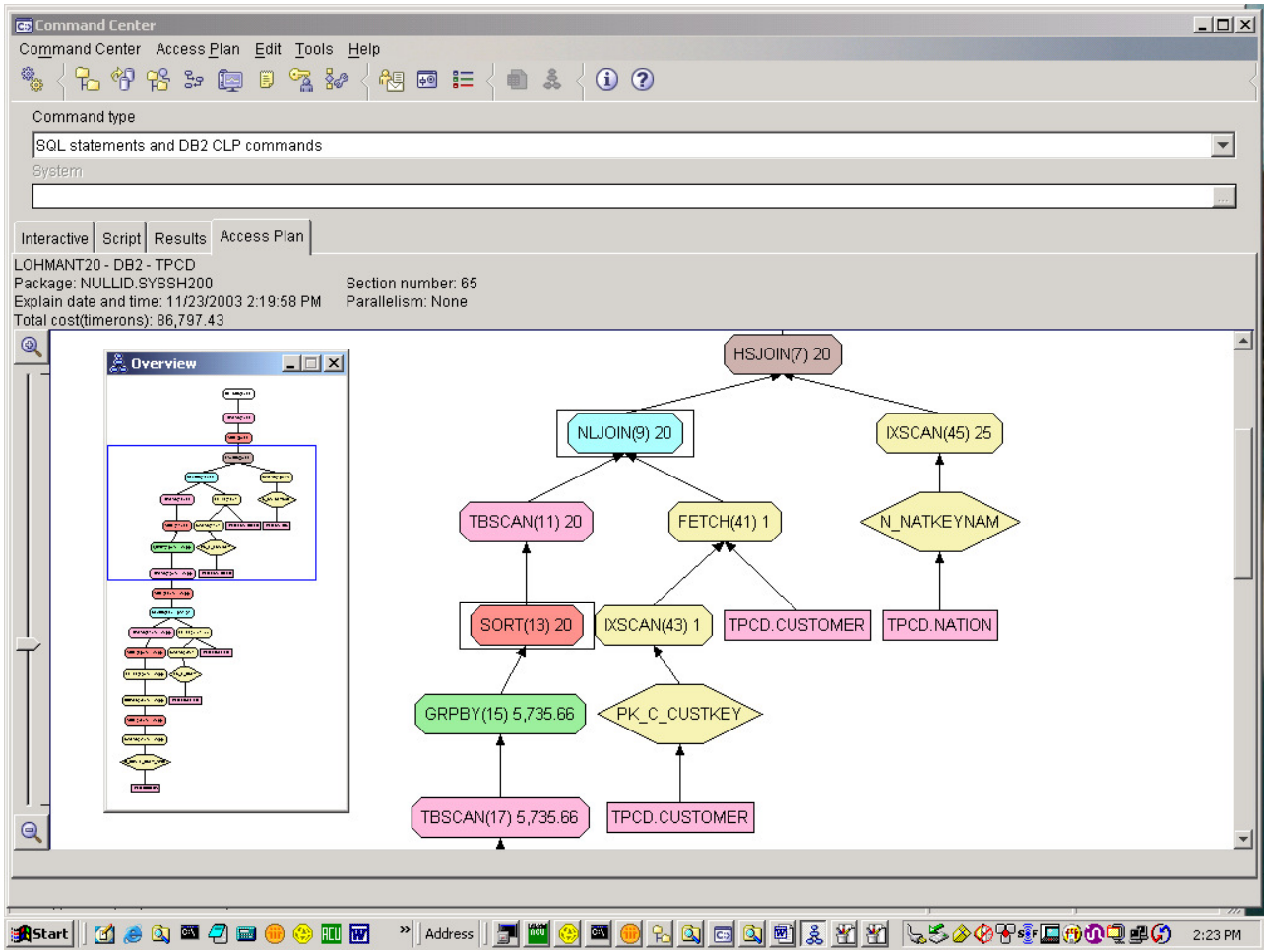

```
--
-- Aggregation re-ordering with join:
--
-- Done by Query ReWrite (show Optimized SQL), only when referential integrity
```

```
-- constraints assure us that joins will be 1-to-1, i.e., will neither
-- increase nor decrease the number of rows (show cardinality estimates by
-- changing Access Plan-->View-->Settings on Operator tab to show
-- "Name & Cardinality", then click "Apply" and "Close").
-- Here, referential integrity constraints assure that each LINEITEM row has one
-- and only one CUSTOMER row, and each CUSTOMER row has one and only one NATION.
-- Note that both of these remaining joins are HSJOINs.
--
-- Note also from the constant cardinalities of GROUP BY operator (13) and the
-- TBSCAN below it (operator 15) that that GROUP BY's aggregation is pushed into
-- the SORT (17) operator below it, as before (to confirm, click on SORT:
-- "Aggregation mode: Partial").
--
-- 'TPCD Returned Item Reporting Query (Q10)';
--

SELECT
  C_CUSTKEY,
  C_NAME,
  SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)) AS REVENUE,
  C_ACCTBAL,
  N_NAME,
  C_ADDRESS,
  C_PHONE,
  C_COMMENT
FROM TPCD.CUSTOMER, TPCD.ORDERS, TPCD.LINEITEM, TPCD.NATION
WHERE C_CUSTKEY = O_CUSTKEY
  AND L_ORDERKEY = O_ORDERKEY
  AND O_ORDERDATE >= DATE('08/03/1997')
  AND O_ORDERDATE <  DATE('08/03/1997') + 3 MONTHS
  AND L_RETURNFLAG = 'R'
  AND C_NATIONKEY = N_NATIONKEY
GROUP BY C_CUSTKEY, C_NAME, C_ACCTBAL, C_PHONE, N_NAME, C_ADDRESS, C_COMMENT
ORDER BY REVENUE DESC;
```



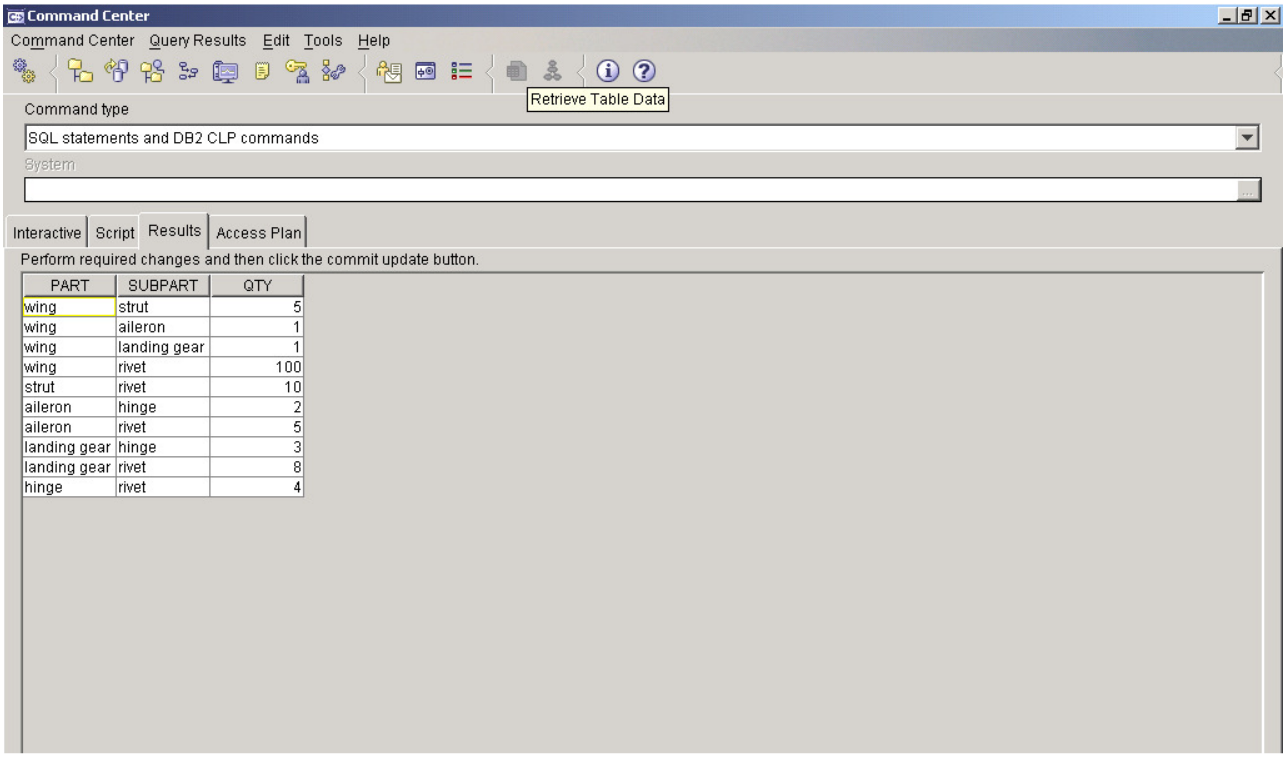

```
--
-- FETCH FIRST K ROWS ONLY:
--
-- Same query as before, *except* that we've added the "FETCH FIRST 20 ROWS
ONLY"
-- clause, to return the top 20 revenue producers.  In conjunction with the
-- GROUP BY reordering, this clause, which must be accompanied by an ORDER BY
-- clause, can be pushed through the joins with CUSTOMER and NATION because of
-- referential integrity constraints assuring one-to-one joins.
-- Note the cardinality estimate for the SORT (#7) is 20, and stays that way
-- through the joins.  Note also that this change in the cardinality estimate
-- changes the join with CUSTOMER: the GROUP BY portion becomes the outer (was
-- the inner (build) side) and changes from HSJOIN to NLJOIN, because only 20
rows
-- in outer.
-- Takes about 1 min. to execute and come up with 0 rows, close to estimate.
-- 'TPCD Returned Item Reporting Query (Q10)';
--

SELECT
  C_CUSTKEY,
  C_NAME,
```

```
  SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)) AS REVENUE,
  C_ACCTBAL,
  N_NAME,
  C_ADDRESS,
  C_PHONE,
  C_COMMENT
FROM TPCD.CUSTOMER, TPCD.ORDERS, TPCD.LINEITEM, TPCD.NATION
WHERE C_CUSTKEY = O_CUSTKEY
  AND L_ORDERKEY = O_ORDERKEY
  AND O_ORDERDATE >= DATE('08/03/1997')
  AND O_ORDERDATE <  DATE('08/03/1997') + 3 MONTHS
  AND L_RETURNFLAG = 'R'
  AND C_NATIONKEY = N_NATIONKEY
GROUP BY C_CUSTKEY, C_NAME, C_ACCTBAL, C_PHONE, N_NAME, C_ADDRESS, C_COMMENT
ORDER BY REVENUE DESC
FETCH FIRST 20 ROWS ONLY;
```



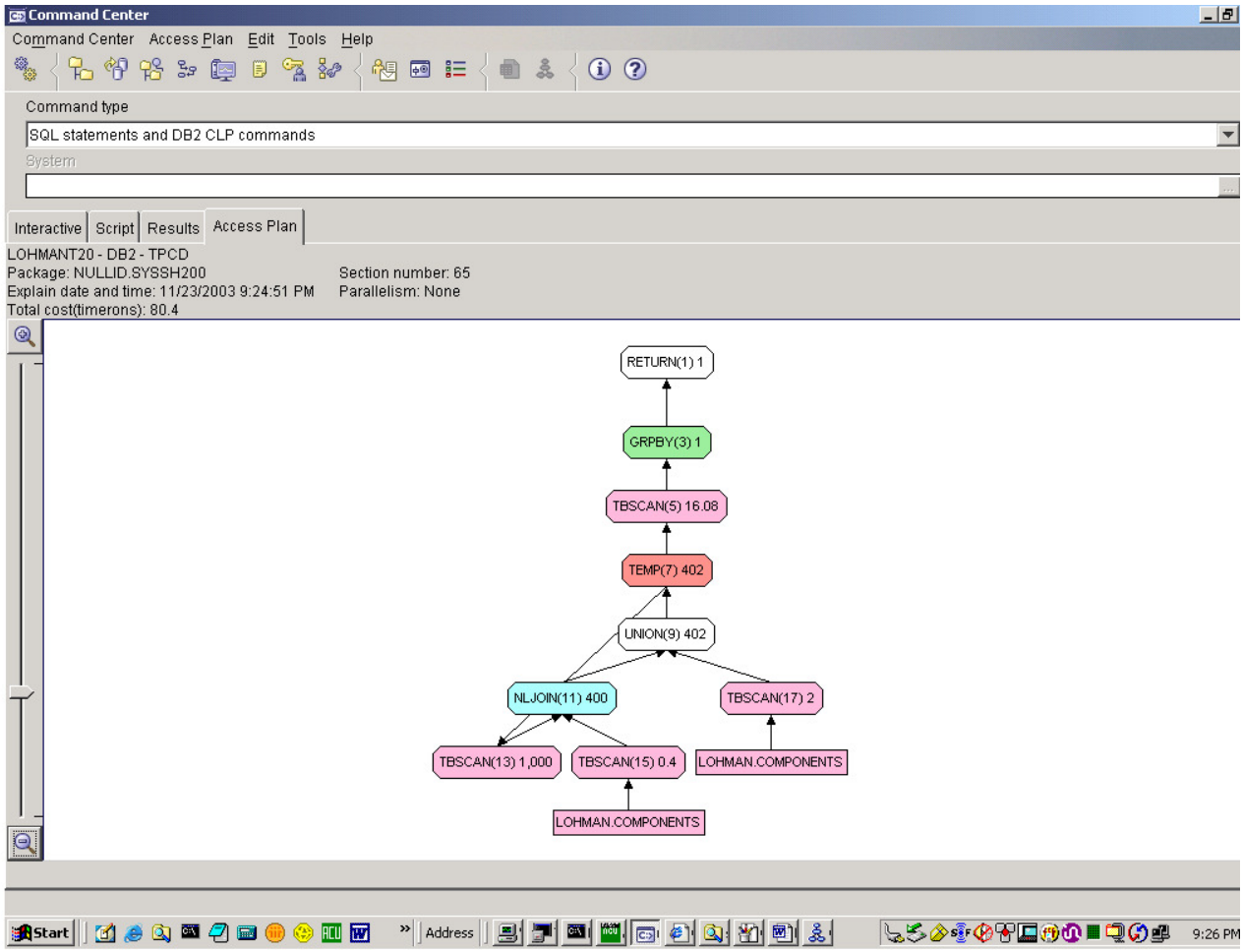

```
--
-- Recursive query:
--
-- First show the components table (classic part-subpart hierarchy, with the
-- quantity of each subpart making up the part.  Execute this query.
-- Taken from Don Chamberlin's book on DB2, a "toy" database.
```

```
--
SELECT * FROM GUY.components;
```

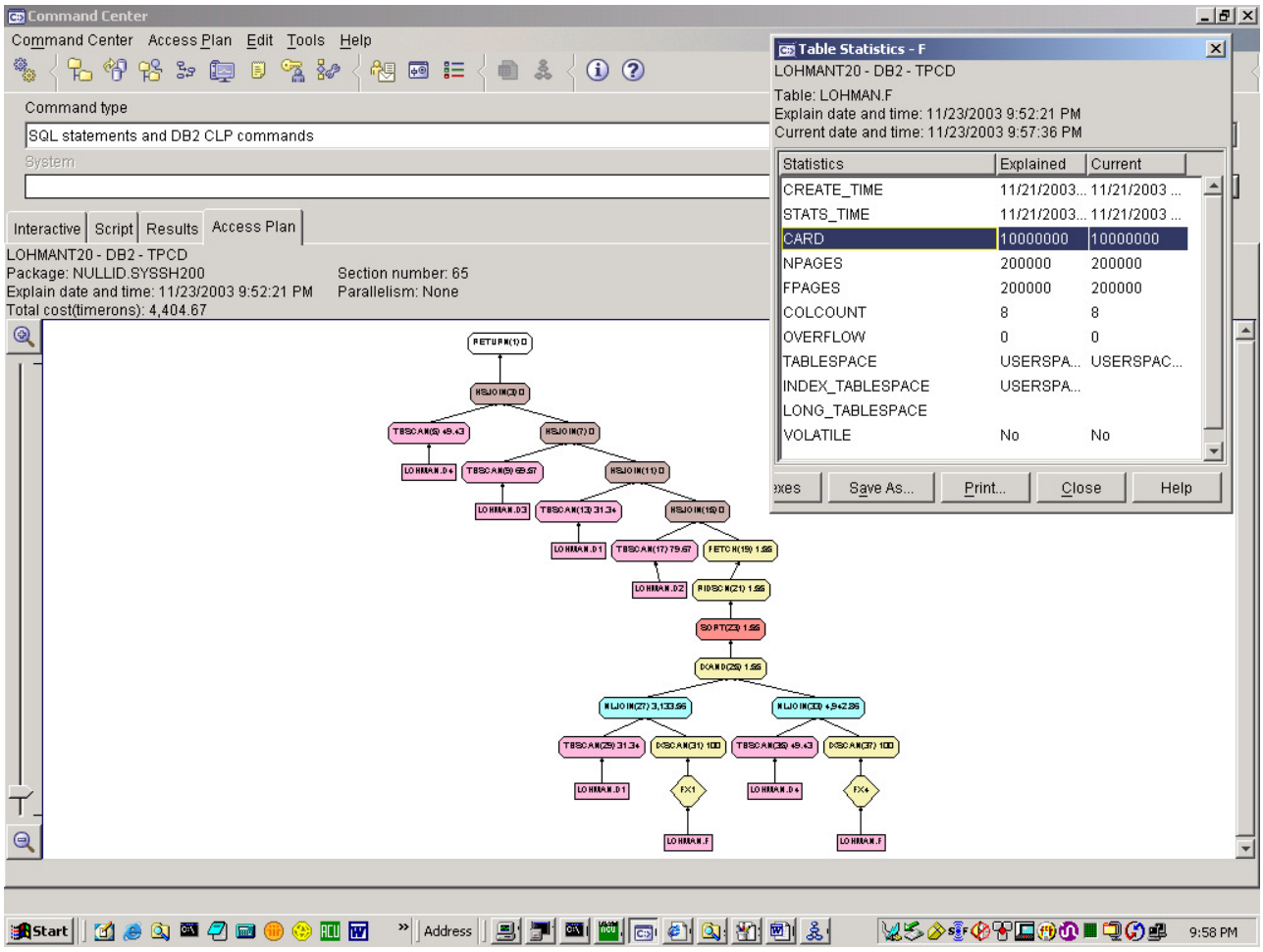


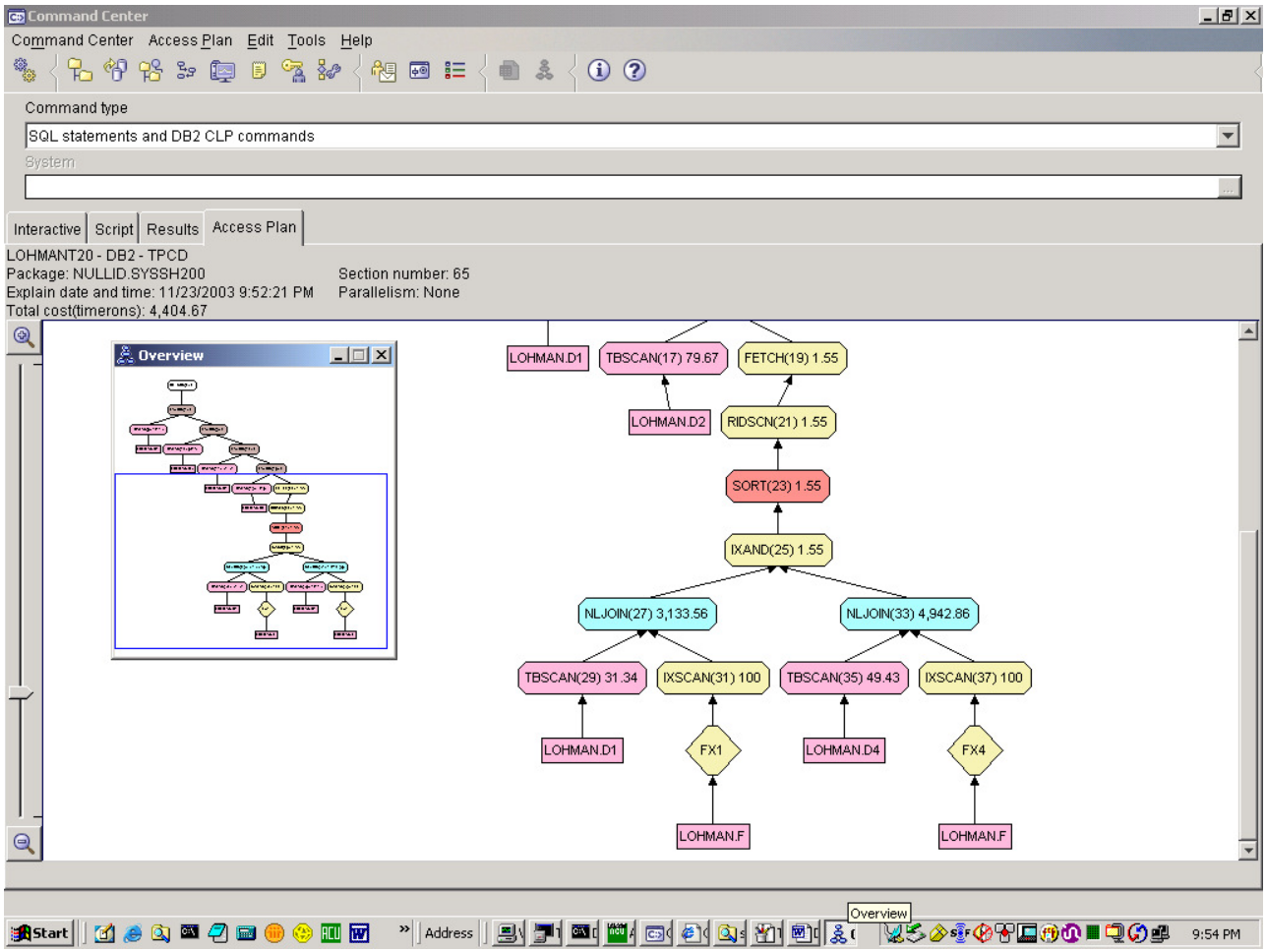

```
-- Now construct the parts explosion for all parts in a wing,
-- and count the number of rivets.
-- Note the recursive reference to wingparts in the WITH clause, which defines
-- a table expression (sort of a view within a query).
-- This syntax for table expressions and recursion is now in the SQL3 standard.
-- Execute this query to see the answer and warning of possibly infinite data.
-- It executes quite quickly on this small database.
-- We can't know a priori whether the data is infinite or not.
--
-- Now show the plan, and note how the "wing" row is accessed (#17) from
-- COMPONENT and materialized in a common sub-expression TEMP (#7), which
-- "clones" a row by permitting it to be accessed by more than one TBSCAN:
-- TBSCAN (5), which returns the row to the user, and TBSCAN(13) to NLJOIN (11)
-- that row with COMPONENTS in order to find its subparts.  These subparts
-- follow the same path, until the NLJOIN (11) finds no subparts, starving the
-- recursion to conclusion.
--
-- Contrast this with an infinite database, e.g.
-- flight segments between cities, and the infinite query selecting all paths
```

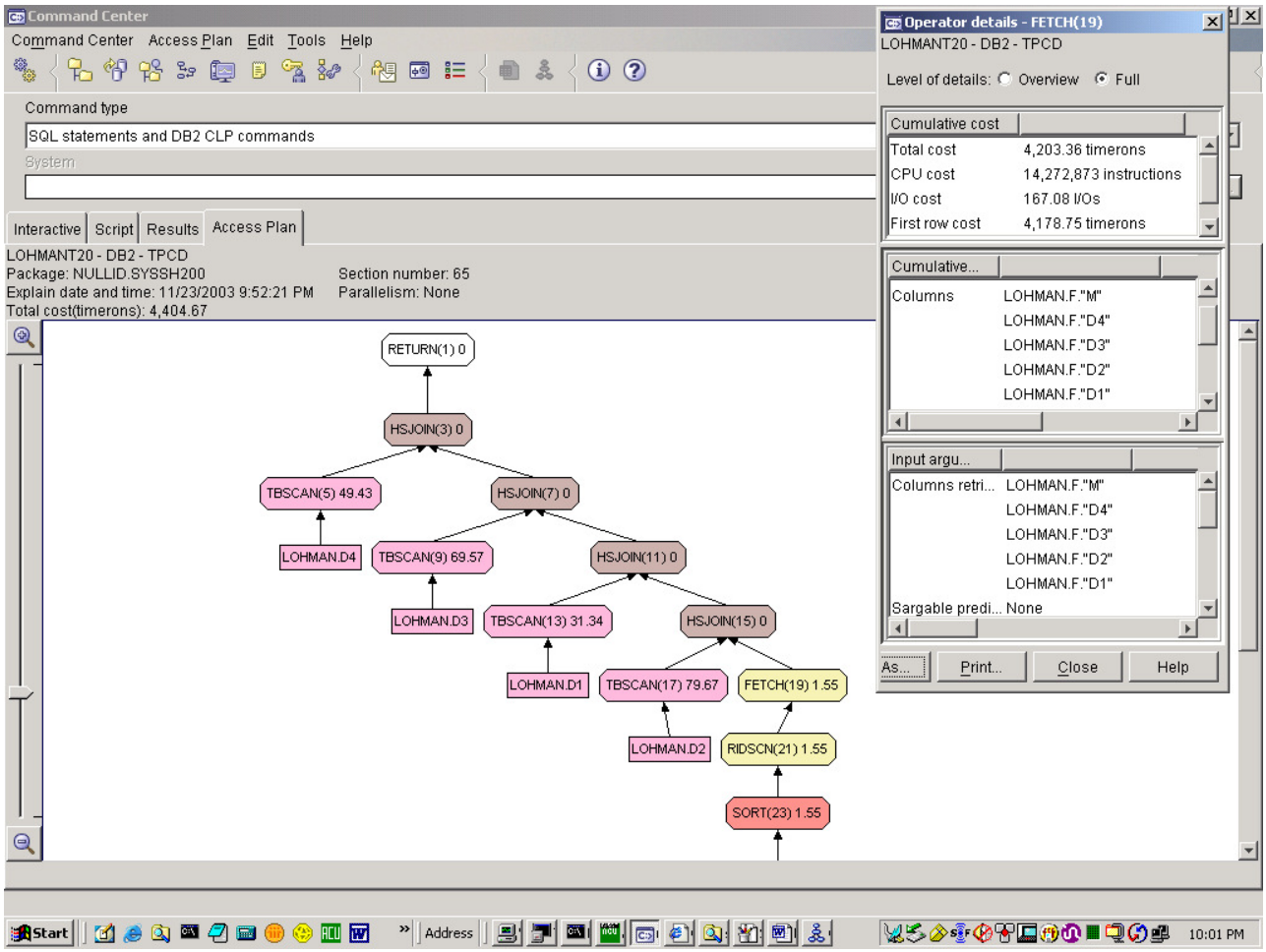
```
-- from JFK to SFO.  Without limitations on the number of flight segments, this
-- latter query would be infinite.
--
-- Note that this construction is superior to a "transitive closure" operator,
-- which would be adequate for this simple query but not for more complicated
-- recursive queries containing multiple joins, aggregation, and even recursion
-- within the recursion, which DB2 can support (nonlinear regression is
forbidden).
-- WARNING: cardinality estimates are still quite rustic.  Better statistics and
-- estimation techniques for recursive query cardinalities are research topics
-- that no-one seems to have attacked.

WITH wingparts(subpart, qty) AS
  ((SELECT subpart, qty
    FROM GUY.components
    WHERE part = 'wing')
  UNION ALL
   (SELECT c.subpart, w.qty * c.qty
    FROM wingparts w, GUY.components c
    WHERE w.subpart = c.part))
SELECT sum(qty) AS qty
FROM wingparts
WHERE subpart = 'rivet';
```



```
--
-- Star join:
--
-- Classic star schema from OLAP: a very large fact table, F, and 4 dimension
-- tables D1, D2, D3, and D4, each joined to the fact table by a key column
-- that needs to be indexed on the fact table in order for our special
-- "star join" strategy to be considered.
-- Get the access plan for this query to show that strategy.
-- Typically, the join of each dimension reduces the fact table's cardinality
-- only by a few orders of magnitude, leaving a still huge number of rows to be
-- joined with the other dimensions.  It is the intersection of each of these
-- joins between a dimension and the fact table that reduces the rows to a
-- manageable number. This is what our "star join" strategy does with the IXAND
-- operator (#25).  However, we don't actually do the IXAND by intersecting RIDs.
-- Instead, we use in-memory Bloom filters, whose bits are set “on” by one
branch's
-- hashing the fact table's RIDs to a bit position, and the next branch probing
-- the same Bloom filter with its resulting RIDs.  This is more efficient than
-- RID-list processing, but has the downside of possible collisions.  So after
-- we SORT the RIDs (#23) and
-- FETCH the data pages (#19), we have to *re*-join each of the dimension
-- tables to the result.  SORT #23 orders the RIDs in page order, for less --
seeking.
-- Not completely sure why we always build (right branch) with the composite,
-- and probe (left branch of the HSJOIN) with the dimension table, as the
-- estimated cardinality is quite small, so it's not to use the full memory
-- for each hash join.  This right-deep tree is non-pipelined.
--
-- Note that NLJOIN #27 and NLJOIN #33 each effectively constitute a semi-join
-- of the fact table with one of the dimensions.  DB2 chose to do semi-joins
-- with only the two most selective dimensions;
-- the semi-joins with the other dimensions were
-- deemed to be too expensive compared to the amount they filtered the fact
-- table rows.
--

select F.D1, F.D2, F.D3, F.D4,M
from GUY.F F, GUY.D1 D1, GUY.D2 D2, GUY.D3 D3, GUY.D4 d4
where F.D1=D1.key AND D1.attr<80 AND D1.keya<40 AND
      F.D2=D2.key AND D2.attr<80 AND
      F.D3=D3.key AND D3.attr<70 AND
      F.D4=D4.key AND D4.attr<50;
```

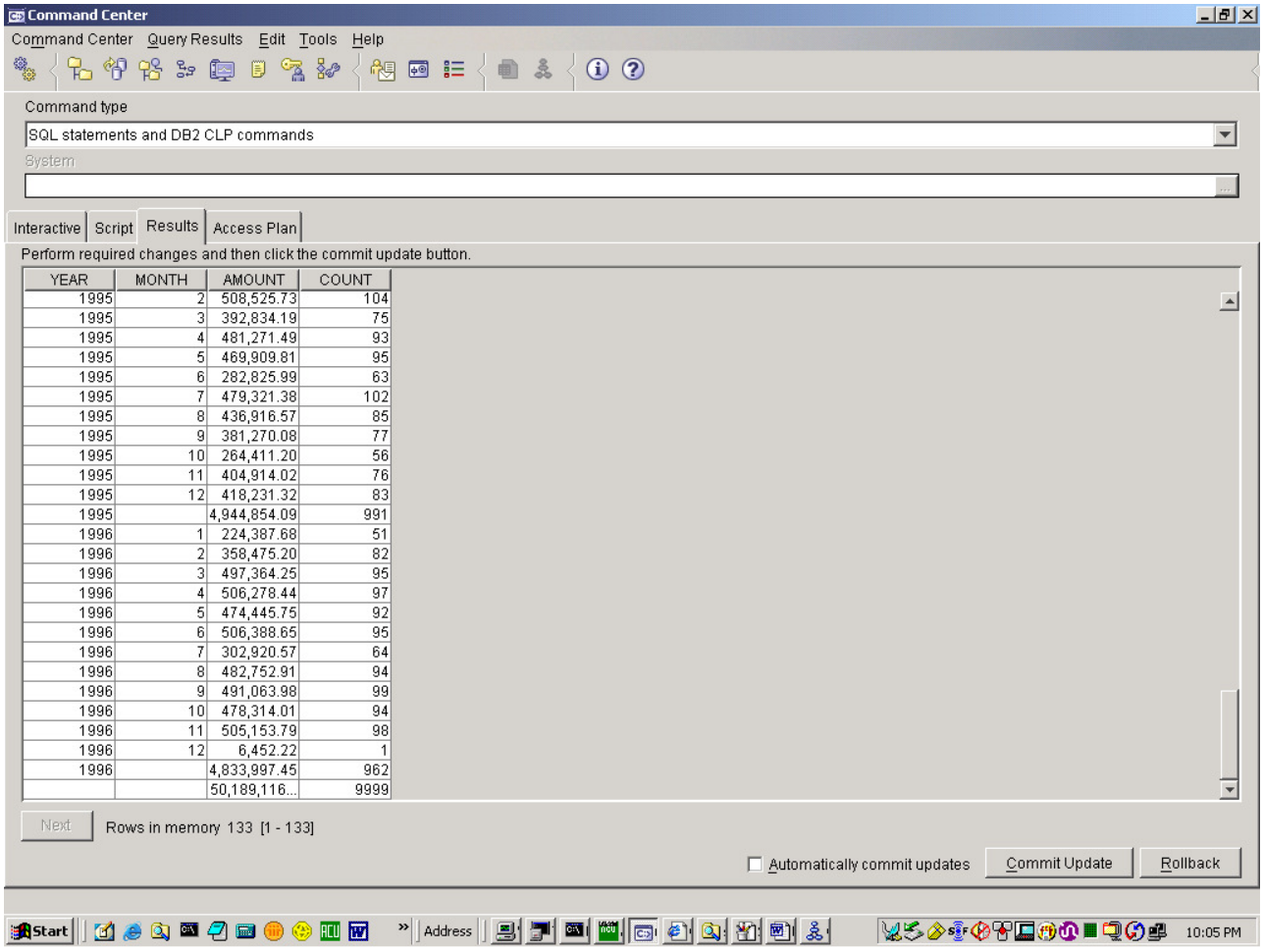

Command Center
Command Center Access Plan Edit Tools Help
Command type
SQL statements and DB2 CLP commands
System
Interactive Script Results Access Plan
LOHMANT20 - DB2 - TPCD
Package: NULLID.SYSSH200
Explain date and time: 11/23/2003 9:52:21 PM
Total cost(timerons): 4,404.67
Section number: 65
Parallelism: None
Table Statistics - F
LOHMANT20 - DB2 - TPCD
Table: LOHMAN.F
Explain date and time: 11/23/2003 9:52:21 PM
Current date and time: 11/23/2003 9:57:36 PM
Statistics
Explained
Current
CREATE_TIME
11/21/2003...
11/21/2003 ...
STATS_TIME
11/21/2003...
11/21/2003 ...
CARD
10000000
10000000
NPAGES
200000
200000
FPAGES
200000
200000
COLCOUNT
8
8
OVERFLOW
0
0
TABLESPACE
USERSPA...
USERSPAC...
INDEX_TABLESPACE
USERSPA...
LONG_TABLESPACE
VOLATILE
No
No
Save As...
Print...
Close
Help
RETURN(1)
LOHMAN.D1
LOHMAN.D2
LOHMAN.D3
LOHMAN.D4
LOHMAN.F
FX1
FX4
Start
Address
9:58 PM


Command Center
Command Center Access Plan Edit Tools Help
Command type
SQL statements and DB2 CLP commands
System
Interactive Script Results Access Plan
LOHMANT20 - DB2 - TPCD
Package: NULLID.SYSSH200
Section number: 65
Explain date and time: 11/23/2003 9:52:21 PM
Parallelism: None
Total cost(timerons): 4,404.67
Overview
LOHMAN.D1
TBSCAN(17) 79.67
FETCH(19) 1.55
LOHMAN.D2
RIDSCN(21) 1.55
SORT(23) 1.55
IXAND(25) 1.55
NLJOIN(27) 3,133.56
NLJOIN(33) 4,942.86
TBSCAN(29) 31.34
IXSCAN(31) 100
TBSCAN(35) 49.43
IXSCAN(37) 100
LOHMAN.D1
FX1
LOHMAN.D4
FX4
LOHMAN.F
LOHMAN.F
Start
Address
Overview
9:54 PM

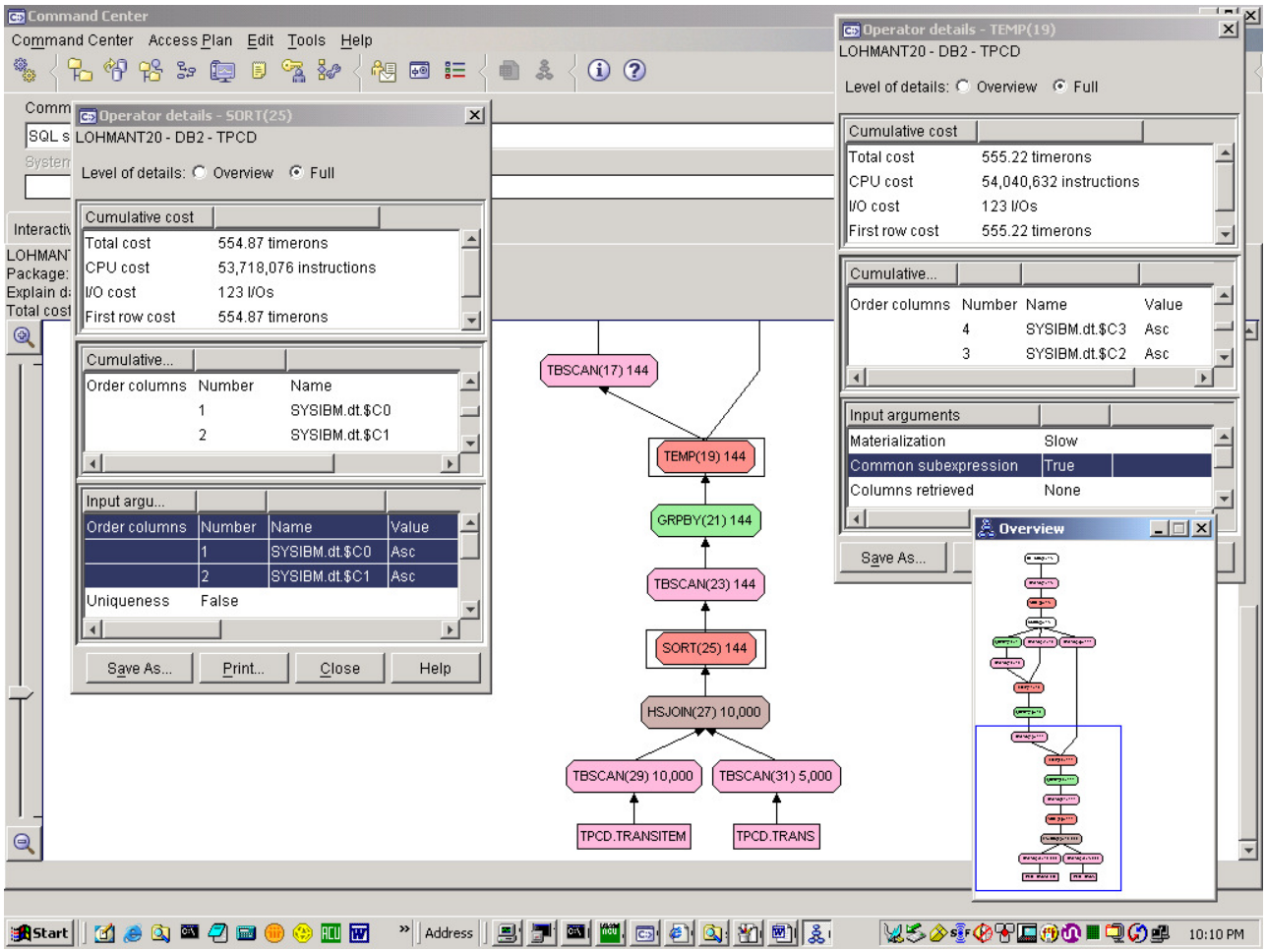


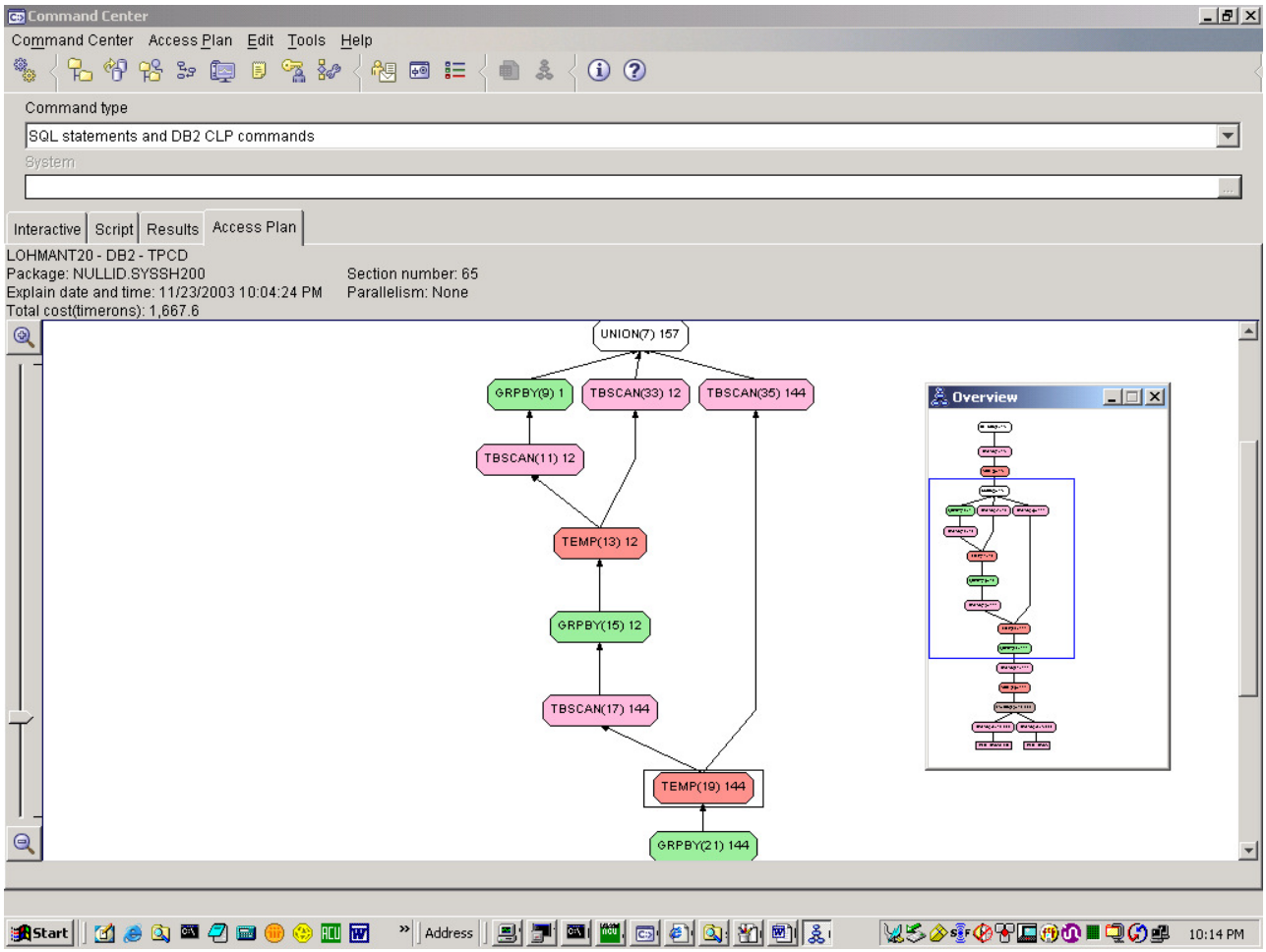

```
--
-- ROLLUP, 1 dimension (time):
--
-- This query returns the total payments for each
-- month of the year, for each year. It also returns the
-- grand total.
--
-- Show syntax of ROLLUP in GROUP BY clause.  Implies strict nesting.
-- Execute this query: limited output and it's fast.
-- Show how subtotals for each month are further totalled for each year,
-- where month is shown as NULL (for NULLable columns, an extra column
-- is introduced). Grand total for all years, the last row, has a null
-- month *and* year.
-- This is query q1 from Hamid Pirahesh's "Business Intelligence" demo.
--
-- In the plan, note how there is only one SORT (#25).  Click on it to show
-- the two columns (expressions) that are its  arguments.
-- Next the aggregation for the months is done in GROUP BY #21.
-- These aggregates are fed to a common sub-expression TEMP (#19), which
-- effectively clones the row.  One copy is fed directly to the UNION,
-- while the second goes to the right to be aggregated by year (#15).
-- This trick is repeated with CSE TEMP #13 and GROUP BY #9 to aggregate
```

```
-- the grand total.  The UNION #7 pulls all these aggregates together.
--
-- Most of this is done in Query ReWrite; to illustrate, pulldown
-- Access Plan-->Statement-->Show optimized SQL text and full-size the window.


SELECT Year(t.pdate) as year, month(t.pdate) as month,
       SUM(ti.amount) as amount, count(*) as count
   from tpcd.trans t, tpcd.transitem ti
   where t.id = ti.transid
   group by rollup(year(t.pdate), month(t.pdate))
ORDER BY year, month;
```



YEAR	MONTH	AMOUNT	COUNT
1995	2	508,525.73	104
1995	3	392,834.19	75
1995	4	481,271.49	93
1995	5	469,909.81	95
1995	6	282,825.99	63
1995	7	479,321.38	102
1995	8	436,916.57	85
1995	9	381,270.08	77
1995	10	264,411.20	56
1995	11	404,914.02	76
1995	12	418,231.32	83
1995		4,944,854.09	991
1996	1	224,387.68	51
1996	2	358,475.20	82
1996	3	497,364.25	95
1996	4	506,278.44	97
1996	5	474,445.75	92
1996	6	506,388.65	95
1996	7	302,920.57	64
1996	8	482,752.91	94
1996	9	491,063.98	99
1996	10	478,314.01	94
1996	11	505,153.79	98
1996	12	6,452.22	1
1996		4,833,997.45	962
		50,189,116...	9999

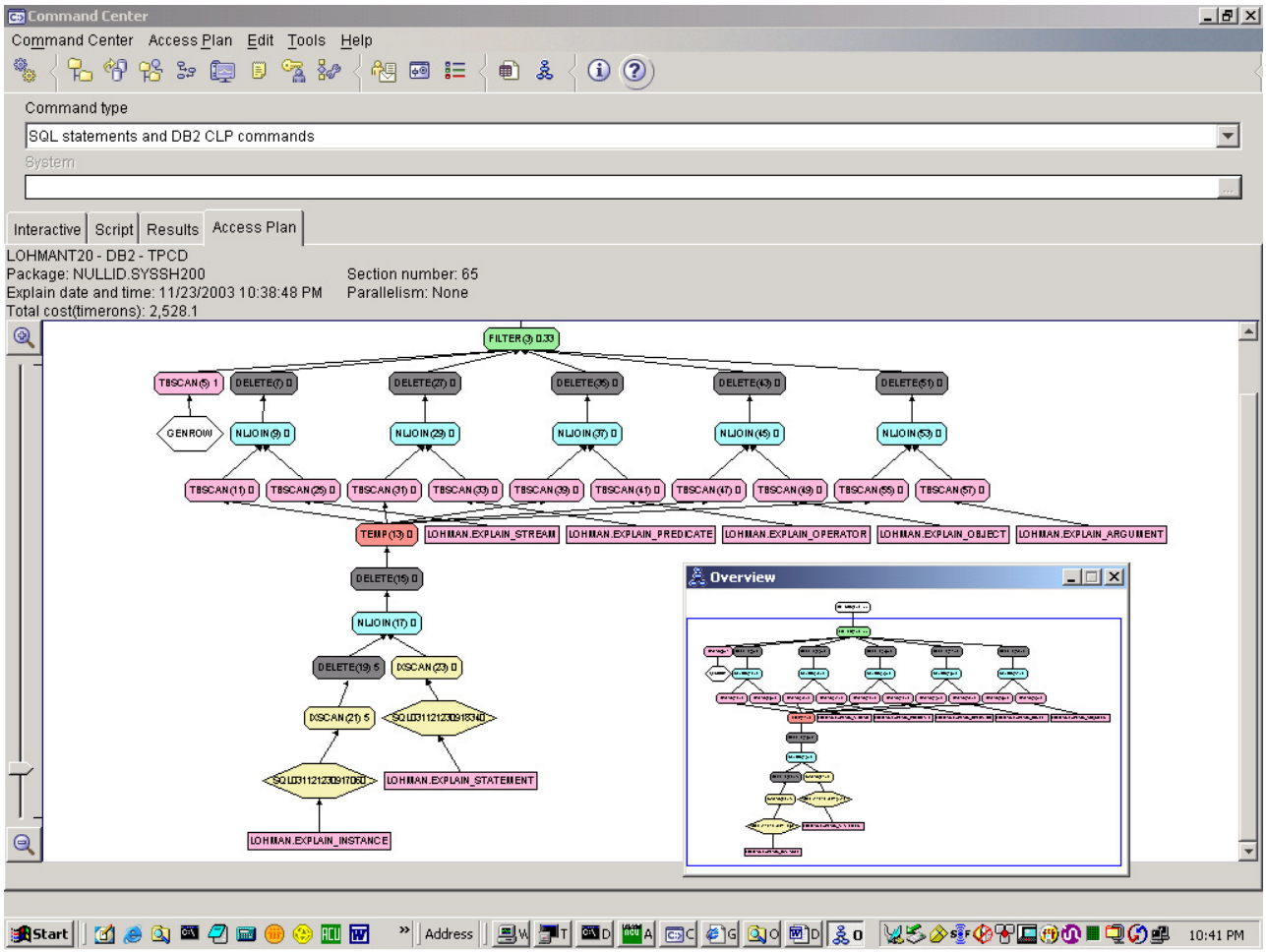

Command Center
Command Center Access Plan Edit Tools Help
Operator details - SORT(25)
LOHMANT20 - DB2 - TPCD
Level of details: Overview Full
Cumulative cost
Total cost 554.87 timerons
CPU cost 53,718,076 instructions
I/O cost 123 I/Os
First row cost 554.87 timerons
Cumulative...
Order columns Number Name
1 SYSIBM.dt.$C0
2 SYSIBM.dt.$C1
Input argu...
Order columns Number Name Value
1 SYSIBM.dt.$C0 Asc
2 SYSIBM.dt.$C1 Asc
Uniqueness False
Save As... Print... Close Help
TBSCAN(17) 144
TEMP(19) 144
GRPBY(21) 144
TBSCAN(23) 144
SORT(25) 144
HSJOIN(27) 10,000
TBSCAN(29) 10,000
TBSCAN(31) 5,000
TPCD.TRANSITEM
TPCD.TRANS
Operator details - TEMP(19)
LOHMANT20 - DB2 - TPCD
Level of details: Overview Full
Cumulative cost
Total cost 555.22 timerons
CPU cost 54,040,632 instructions
I/O cost 123 I/Os
First row cost 555.22 timerons
Cumulative...
Order columns Number Name Value
4 SYSIBM.dt.$C3 Asc
3 SYSIBM.dt.$C2 Asc
Input arguments
Materialization Slow
Common subexpression True
Columns retrieved None
Save As...
Overview
Start
Address
10:10 PM



Appendix: More Examples!

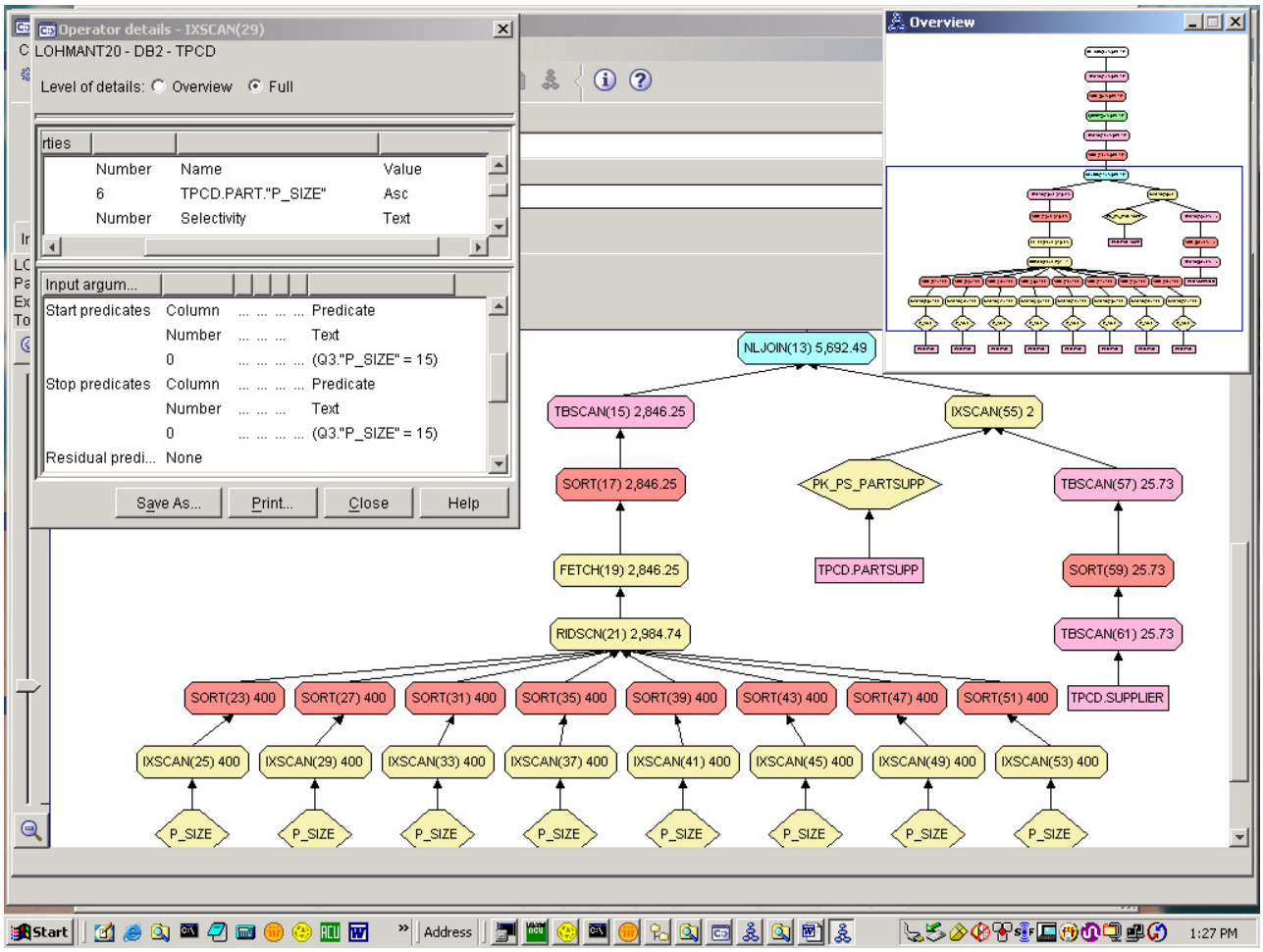
```
--
-- Query Global Semantics:
--
-- Query Global Semantics brings in from the catalogs for each table in
-- the FROM clause any semantic information relevant to that table, such
-- as referential integrity (RI), constraints, and triggers.  It compiles
-- that semantic information as part of the query itself, because these
-- additional steps might affect how we process the query overall, and the
-- whole query should be optimized as one unit.
--
```

```
-- This simple example, deleting all rows from the EXPLAIN_INSTANCE table,
-- shows how referential integrity constraints significantly complicate the
-- plan, and how rows DELETEd are flowed upward to subsequent steps.
-- The additional DELETEs are to orphaned rows in other EXPLAIN tables,
-- such as EXPLAIN_OPERATORS, EXPLAIN_STREAMS, etc.

DELETE FROM EXPLAIN_INSTANCE;
```



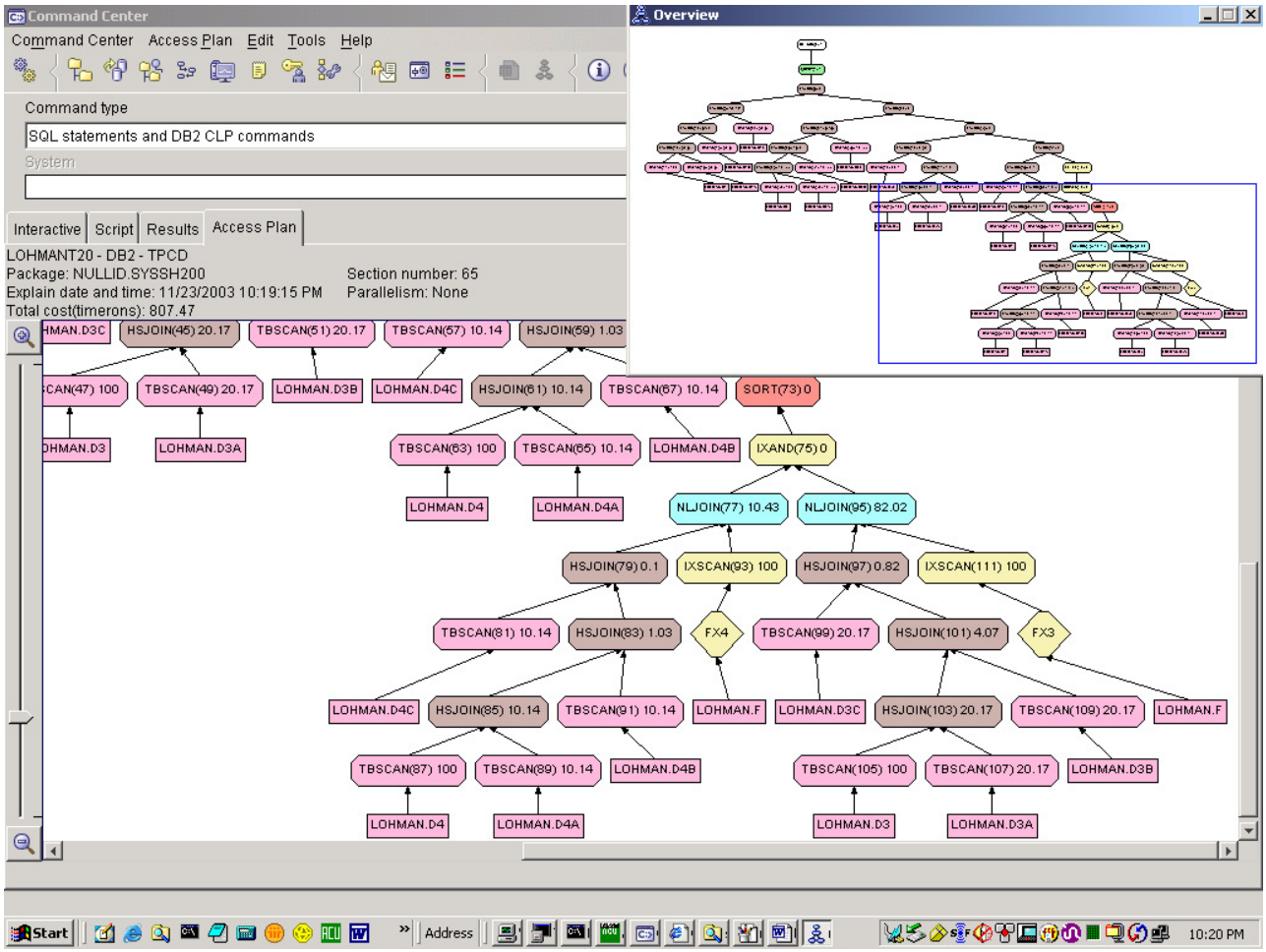

```
--
-- Query ReWrite and "Twinning":
--
-- "Twinning" of IN-list predicate
-- and "OR"d version of IN-list, to enable probes to index on column SIZE
-- or original version (for a table scan).  Cost decides which is used.
-- Zoom in to show individual start/stop predicates applied by index on SIZE.
-- How can IXSCAN (55) have two inputs, one an index and one a TBSCAN?
-- The input on the right is a NOT IN subquery that cannot be transformed
-- to a join.
-- Since the ORDER BY and GROUP BY are on different columns, we have to
-- SORT twice (SORT (5) and SORT (11)).

SELECT P_BRAND, P_TYPE, P_SIZE, COUNT(DISTINCT PS_SUPPKEY) AS SUPPLIER_CNT
```

```
FROM TPCD.PARTSUPP, TPCD.PART
WHERE P_PARTKEY = PS_PARTKEY AND P_BRAND <> 'Brand#22' AND P_TYPE NOT LIKE
        'MEDIUM ANODIZED%' AND P_SIZE IN (8,15,48,5,19,50,11,6) AND
        PS_SUPPKEY NOT IN
   (SELECT S_SUPPKEY
   FROM TPCD.SUPPLIER
   WHERE S_COMMENT LIKE '%Better Business Bureau%Complaints%' )
GROUP BY P_BRAND, P_TYPE, P_SIZE
ORDER BY SUPPLIER_CNT DESC, P_BRAND, P_TYPE, P_SIZE;
```



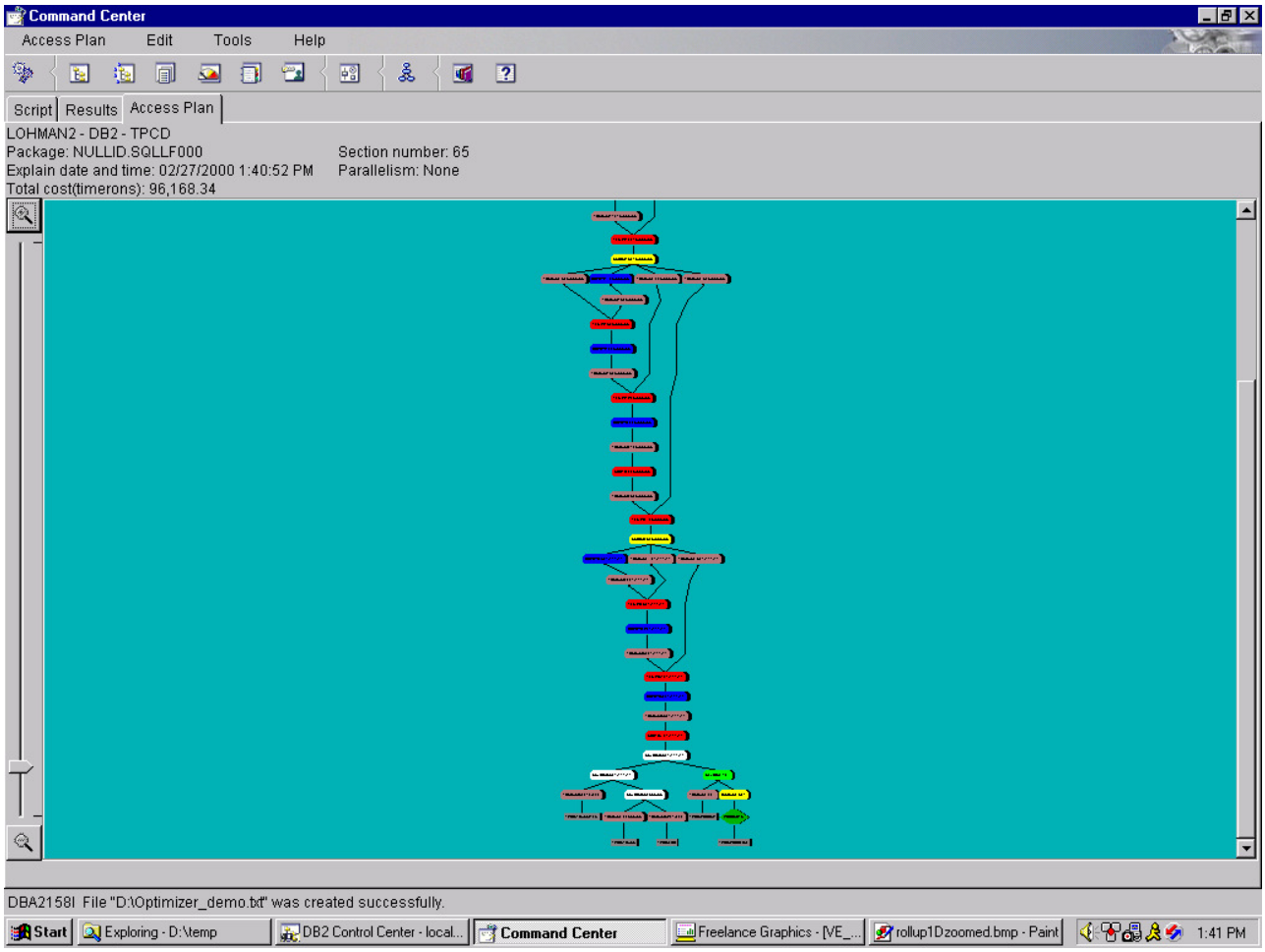

```
--
-- Snowflake query:
--
-- Each dimension table now itself has 3 dimensions.
-- Resulting plan has
-- LOTS of table scans and hash joins among snowflake tables, because
-- the dimension tables don't have indexes.
-- It's hard to tell in this rather large plan, but it's basically
-- the same plan as before, with each dimension requiring three
-- additional joins to the snowflake tables.
-- And the semi-joins are on D3 and D4, instead of D2 and D3 in the previous
plan,
```

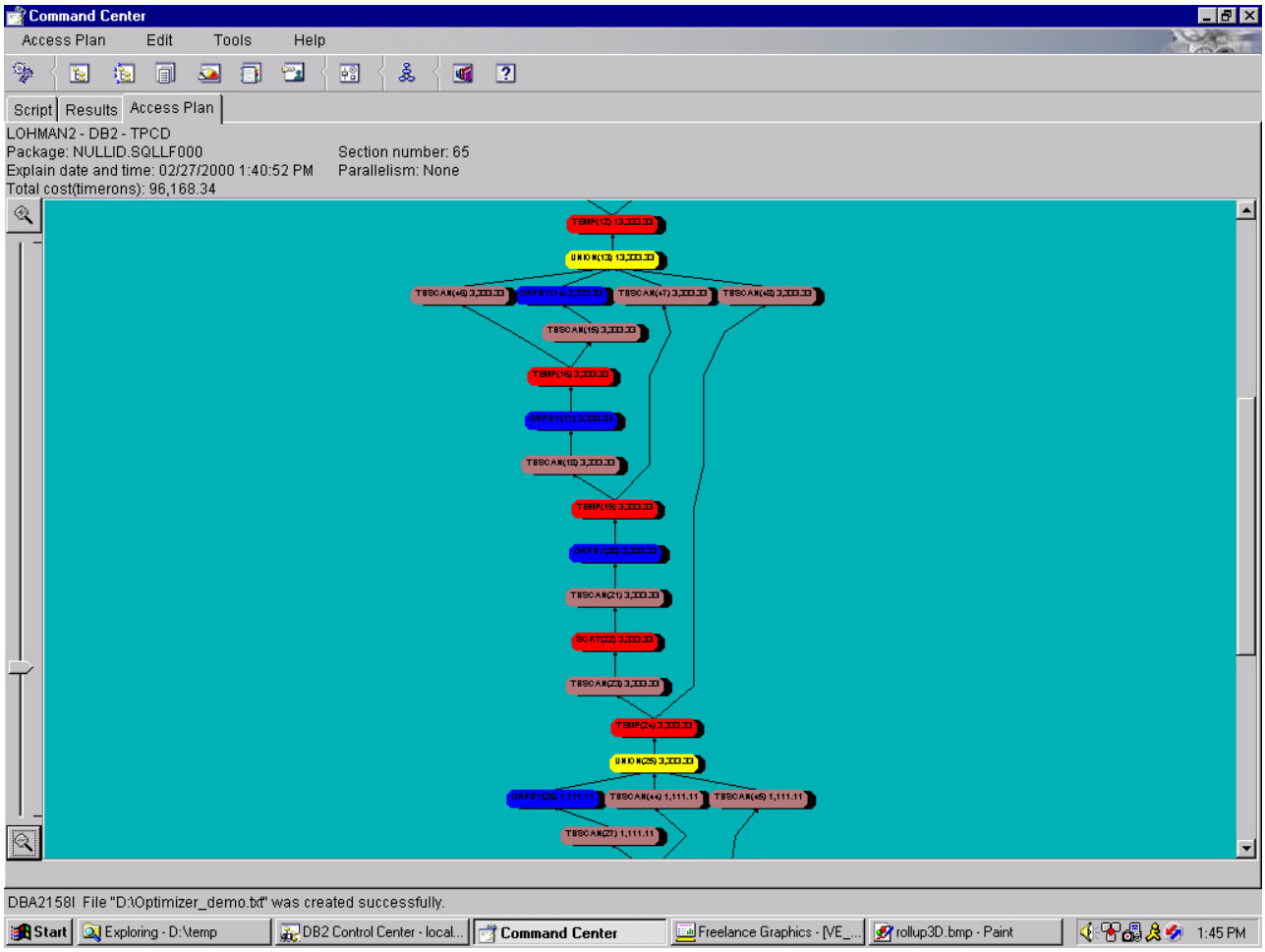
```
-- because they are the most selective.
-- Plus, the back-joins to all tables is bushier, probably because this many
tables
-- caused the Optimizer to revert to the Greedy join enumeration technique.
--
select count(*)
from   F,
       D1, D1a, D1b, D1c,
       D2, D2a, D2b, D2c,
       D3, D3a, D3b, D3c,
       D4, D4a, D4b, D4c
where
       F.D1 = D1.key and
       D1.keya = D1a.key and D1a.attr <= 80 and
       D1.keyb = D1b.key and D1b.attr <= 80 and
       D1.keyc = D1c.key and D1c.attr <= 80
and
       F.D2 = D2.key and
       D2.keya = D2a.key and D2a.attr <= 40 and
       D2.keyb = D2b.key and D2b.attr <= 40 and
       D2.keyc = D2c.key and D2c.attr <= 40
and
       F.D3 = D3.key and
       D3.keya = D3a.key and D3a.attr <= 20 and
       D3.keyb = D3b.key and D3b.attr <= 20 and
       D3.keyc = D3c.key and D3c.attr <= 20
and
       F.D4 = D4.key and
       D4.keya = D4a.key and D4a.attr <= 10 and
       D4.keyb = D4b.key and D4b.attr <= 10 and
       D4.keyc = D4c.key and D4c.attr <= 10;
```




```
--
-- ROLLUP, in 3 dimensions:
--
-- This is an example of a multidimensional analysis.
-- The query gets the sum and count of sales:
--   for each level of product hierarchy
--      for each level of location hierarchy
--         for each level of time hierarchy, composed of year, month.
-- Illustrate this by highlighting the individual ROLLUPs for:
--    product ID and product group
--    location (city, state, country)
--    time (month, year)

-- RUNNING THIS QUERY IS NOT RECOMMENDED: the output is voluminous, so
-- takes several minutes.
--
-- The plan looks much like the 1-dimensional case, replicated 3 times in
-- sequence, and connected by two UNIONs, #21 and #45.
-- As before, only one sort is needed for the multiple roll-up aggregations
-- done on each dimension, SORTs #61, #43, and #19.
--
-- This is q4 of Hamid Pirahesh's Business Intelligence demo.
```

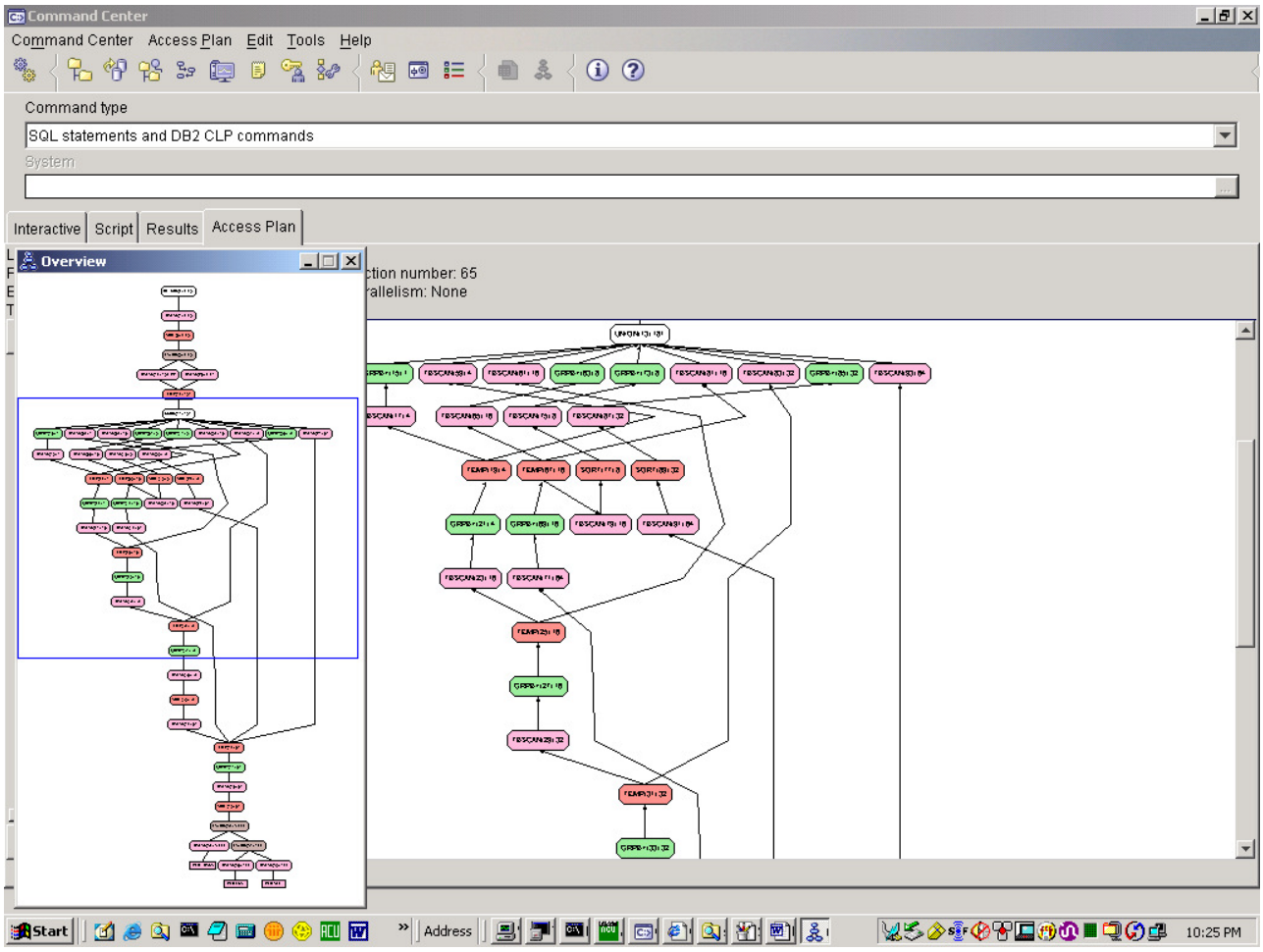
```
--
-- Most of this stacking is done in Query ReWrite; to illustrate, pulldown
-- Access Plan-->Statement-->Show optimized SQL text and full-size the window.
-- The Optimizer figures out that only one SORT is needed.


SELECT case l.id  when 0 then 'Electronics'
                  when 1 then 'Medical'
                  when 2 then 'Clothing'
            end as prodline,
       case pg.id when 1 then 'VCR'
                  when 2 then 'Coats'
                  when 4 then 'Antibiotics'
                  when 8 then 'Camcoder'
            end as pgroup,
       loc.country, loc.state, loc.city,
       year(pdate) as year, month(pdate) as month,
       SUM(ti.amount) as amount, count(*) as count
FROM tpcd.transitem ti, tpcd.trans t,
     tpcd.loc loc,
     tpcd.pgroup pg, tpcd.prodline as l
 where ti.transid = t.id
   and ti.pgid = pg.id and pg.lineid = l.id
   and t.locid=loc.id
   and year(t.pdate) between 1995 and 1997
group by rollup(l.id, pg.id),
         rollup(loc.country, loc.state, loc.city),
         rollup(year(pdate), month(pdate))
Order by prodline, pgroup,
         country, state,city,
         year, month;
```


Command Center
Access Plan
Edit
Tools
Help
Script
Results
Access Plan
LOHMAN2 - DB2 - TPCD
Package: NULLID.SQLLF000
Section number: 65
Explain date and time: 02/27/2000 1:40:52 PM
Parallelism: None
Total cost(timerons): 96,168.34
DBA2158I File "D:\Optimizer_demo.txt" was created successfully.
Start
Exploring - D:\temp
DB2 Control Center - local...
Command Center
Freelance Graphics - [VE_...
rollup1Dzoomed.bmp - Paint
1:41 PM




```
--
-- CUBE and grouping sets, multidimensional:
--
-- Analyze correlations with any combination of the other columns
-- First do multiple groupings and keep the results in table 'summary'
-- We only need to change the GROUP BY clause.
-- We don't expect users to come up with this rather complex SQL:
-- query tools can do that!
-- Note that the summary table size is typically small (in 100s of tuples),
-- so this query can be executed in just a few seconds.
-- To compute the correlation, we need to juxtapose the count of each
-- combination with the the count of each account status. For this, we
-- do a self-join or a subquery at the end.
--
-- This is q9 of Hamid Pirahesh's Business Intelligence demo.
--
-- The plan is much broader, doing the usual ROLLUP for one dimension,
-- but the bypassing branches from the CSE TEMPs now criss-cross to
-- other dimensions.
-- One branch bypasses all aggregation, while the other 3 each represent
-- one of the three dimensions in the cube.  As with ROLLUP, we SORT once
```

```
-- and aggregate multiple times for each dimension, but this time in
-- parallel rather than in sequence.
--
-- Most of this stacking is done in Query ReWrite; to illustrate, pulldown
-- Access Plan-->Statement-->Show optimized SQL text and full-size the window.
-- The Optimizer figures out that only one SORT is needed.

with tpcd.summary as
  (select t.status,  count(*) as count,
   c.marital_status, grouping(c.marital_status) as ms_ind,
   c.income_range, grouping(c.income_range) as ir_ind,
   c.residence, grouping(c.residence) as r_ind
   from tpcd.trans t, tpcd.acct a, tpcd.cust c
   where t.acctid = a.id and a.custid = c.id
   group by grouping sets( (t.status),
                           cube(
                                (t.status, c.marital_status),
                                (t.status, c.income_range),
                                (t.status, c.residence)
                               )
                         )
  )
select correl.status, total.count totalcount,
     integer(100*float(correl.count)/float(total.count)) as correlation,
     correl.marital_status, correl.income_range, correl.residence
from tpcd.summary correl,
     table (select total.count from tpcd.summary total
            where correl.status=total.status
              and total.ms_ind = 1 and total.ir_ind = 1 and total.r_ind = 1
           ) as total
where not (correl.ms_ind = 1 and correl.ir_ind = 1 and correl.r_ind = 1)
order by correl.status, correlation desc;
```


Command Center
Command Center Access Plan Edit Tools Help
Command type
SQL statements and DB2 CLP commands
System
Interactive Script Results Access Plan
Overview
ction number: 65
rallelism: None
Start
Address
10:25 PM