Cartesian Products

• Cartesian Products.
• More Common Than You Think.

• New England Oracle User’s Group
• Boston chapter. March 5, 2008

• How To Get Them
• How to Solve Them
Who is Rodger Lepinsky?

• University Degrees in both Business (1993) and Computers (1995).


• Working in technology, primarily Oracle, since 1995.
• Designing databases since 1992.
• Analysis, Database Architecture, Design, Development, Warehousing, Tuning, DBA.

• Seen far too many cartesian products, and bad database designs.

• Working at a financial company in California.
Show of Hands - How Many People?

- Are developers, DBA, managers, other?
- Work in an environment that uses a modeling tool?
- Have their database schema on their cubical wall?
- Can determine the effective keys of views they’ve never seen before, without looking at the code?
- Have seen cartesian products in their systems?
Joke for Database Experts

• In relational databases, there is technically no such thing as a “relationship”.
• There are only foreign key constraints.
• :)  

• Alter table …
• Add constraint …
• Foreign Key … etc.
• But foreign key constraints are good!
Background of this Presentation

- Oracle is an incredible product!
- Unfortunately, often implemented -very- badly.
- Bad schemas, bad coding, reinventing the wheel.
- Including cartesian products, designed right into the system.
- See dbdebunk.com for some very bizarre quotes from “professionals”, revealing what they don’t know.
Background of this Presentation

- Became: High Speed PLSQL
- And: Database Design Mistakes to Avoid
  - including sections on Data Analysis, and Cartesian products
- Which has now become: a separate presentation on cartesian products
Background of this Presentation

• Reasoning:
  • Not much written on cartesian products, outside of academic circles.

  • There is little written regarding cartesian products, from a development, or a design standpoint.

  • No third party GUI tool or Oracle feature will save you from mistakes resulting in cartesian products.

  • Many database people are not aware of cartesian products.
Some Mathematics

• Relational databases and SQL actually have a solid foundation based in mathematics.

• Relational databases are based on set theory.
• The same set theory that you studied in junior high.
• Union, Minus, Intersect
• It also involves Relational Algebra, and Relational Calculus

• Cartesian products are defined in the mathematics.
SQL and Mathematics

- No other computer language has the same strict mathematical theory behind it as SQL does.
- Developed by Edgar Codd in 1970.
- This will be familiar if you studied advanced database.

- In Math:
  - \( \pi \) - Projection (SELECT clause)
  - \( \sigma \) - Restrict, or Select (WHERE clause)
  - R - Relation (FROM clause)

\[
\pi_{<\text{LNAME, FNAME, SALARY}>}(\sigma_{\text{DNO} = 5 \text{ AND } \text{SALARY} > 30000})(\text{EMPLOYEE})
\]
SQL and Mathematics

• In Math:
  • $\pi_{\text{LNAME}, \text{FNAME}, \text{SALARY}}(\sigma_{\text{DNO} = 5 \text{ AND } \text{SALARY} > 30000}(\text{EMPLOYEE}))$

• In SQL:
  • Select LNAME, FNAME, SALARY
  • from Employee
  • Where DNO = 5
  • AND SALARY > 30000
SQL and Mathematics

• What is this computer code in a mathematical statement?

• While (I <= J) loop

• If ( I * a >= x ) then
  • If ( ((x * b) - I) <= y ) then
    • Z = Z + (I * a);
  • End if;
• End if;

• J := J + 1;

• End loop ;
Mathematical Definition of a Cartesian Product - Set Theory

• A cartesian product is the multiplication of two sets of data.

• Cartesian product: Set1 x Set2
• Set1 contains: (1, 2, 3)
• Set2 contains: (6, 7, 8)

• The Cartesian product contains: (1,6),(1,7),(1,8),(2,6),(2,7),(2,8),(3,6),(3,7),(3,8)
• A relation is any subset of this cartesian product

• We specify the domains of these sets by applying conditions for their selection
• \( R = \{(x,y)| x \text{ is a member of Set1, } y \text{ is a member of Set2, } y = 6\} \)
• \( R = \{(1,6), (2,6), (3,6)\} \)
• Ref: http://cec.wustl.edu/~cse530/DatabaseManagementSystems-cs530a-4.ppt
Data Analysis in my Early Career

- My early career. Move a Cobol system to Oracle.

- Parts of the schemas were based on Cobol.
- In addition, two relational systems were married together.

- Result:
  - Poor schema.
  - Required lots of data analysis first.
  - Coding second.
Data Analysis

- Looking at the Primary Key (PK) and Foreign Key (FK) of the tables.

- Finding the effective keys of views.

- Counting the number of rows in each table.

- Group BYs on the fields I was joining and filtering on.
  - look for duplicates
  - look at data distributions
Data Analysis

• Looking at the column cardinality of fields:

  • Count (distinct FieldY )
  • Count (distinct FieldZ )
  • Count (distinct FieldY || FieldZ)

• And then joining tables on those fields.
Data Analysis

• Then asked:

• How many rows should I expect from this query?
  – What is the minimum number possible?
  – What is the maximum number possible?
  – What is the relevant range to expect?
A \leq x \leq B

• For: Count (*)
• What is \( A \leq x \leq B \) ???

• Where
• \( A = \) minimum possible count
• \( B = \) maximum possible count
• What can I reasonably expect?

• If you know statistics, this looks -similar- to a
• Probability Density Function, or a
• Continuous Uniform Distribution
• \( P(-.45 \leq X \leq .8) \)
A \leq x \leq B

- Given: Count (*)
- What is $A \leq x \leq B$ ???

- A few key factors affect the answer:
  - (long, boring analysis not shown)

- Do the tables have PK and FK?
  - if not, the query behavior and answers are different.

- Identifying versus Non-Identifying relationships?

- What are the counts of the tables?

- Do filter conditions exist?
A <= x <= B

- Simple rule of thumb to determine cartesian products.
- Every table has a Count(*) of the number of rows.
- If a schema has proper PK and FK, and a query is joined together correctly, the maximum number of rows possible is the largest count(*) of all the tables.
A \leq x \leq B

- Simple rule of thumb to determine cartesian products.

- I.e. Given three tables with FKS. Counts of 50, 500, and 5000 and a query properly joining the three tables:

  - Maximum count possible is 5000 rows.

  - Greater than 5000 rows, indicates a definite cartesian product.

  - (Although it’s possible to have fewer rows returned; say the 5000 rows is just a lookup/parent table.)
Cartesian Products in Queries

- 5 ways to get cartesian products in queries:
  - 1) No join conditions at all on multiple tables.
  - 2) Less than (N-1) join conditions.
  - 3) Joining on non-unique values (AKA Joining down the middle).
  - 4) Not including all concatenated fields of the PK in the join condition.
  - 5) Incorrect use of OR conditions.
Cartesian Products in SQL Queries

- Example schema.
- Deliberately vague to concentrate on the cartesian products, not how it might be (re)modeled.

### Table Counts

<table>
<thead>
<tr>
<th></th>
<th>Small Schema</th>
<th>Big Schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>UsesLookups</td>
<td>30</td>
<td>30,000</td>
</tr>
<tr>
<td>LookupTable</td>
<td>10</td>
<td>10,000</td>
</tr>
<tr>
<td>OneTable</td>
<td>20</td>
<td>20,000</td>
</tr>
<tr>
<td>FirstMany</td>
<td>60</td>
<td>60,000</td>
</tr>
<tr>
<td>UsesMany</td>
<td>120</td>
<td>120,000</td>
</tr>
<tr>
<td>SecondMany</td>
<td>240</td>
<td>240,000</td>
</tr>
</tbody>
</table>
Cartesian Products in SQL Queries - No Join Conditions

• 1) No join conditions at all on multiple tables

• Counts:
• LookupTable 10
• OneTable 20

• Maximum rows should be: 20

• From before, the maximum number of rows in the query, will be the number of rows in the many/child table.
Cartesian Products in SQL Queries - No Join Conditions

1) No join conditions at all on multiple tables

```sql
Select onetableid,
    l.lookuptableid
from onetable o, /* 20 rows */
    lookuptable l /* 10 rows */
```

ONETABLEID LOOKUPTABLEID
--------- ---------------
...          
  114        10
  115        10
  116        10
  117        10
  118        10
  119        10

200 rows selected.
Cartesian Products in SQL Queries - Less than (n-1) join conditions

2) More than 2 tables in the join, and the number of join conditions is less than (n-1), where n is the number of tables in the query. Example one is really a variation of this rule.
Cartesian Products in SQL Queries - Less than (n-1) join conditions

- The query should be:

- Select  l.lookuptableid,
- o.onetableid,
- f.firstmanyid

- From   lookuptable    l, /* 10 rows */
- onetable    o, /* 20 rows */
- firstmany    f /* 60 rows */

- Where   f.onetableid = o.onetableid
- and     o.lookuptableid = l.lookuptableid
- order by 1, 2, 3

- 3 tables,  2 join conditions.
- N   N-1

Copyright - Rodger Lepinsky
March 2008
Cartesian Products in SQL Queries - Less than (n-1) join conditions

• Correct Result:
  • ...  
  • 9   116   1  
  • 9   116   2  
  • 9   116   3  
  • 9   117   1  
  • 9   117   2  
  • 9   117   3  
  • 10  118   1  
  • 10  118   2  
  • 10  118   3  
  • 10  119   1  
  • 10  119   2  
  • 10  119   3  
  • 60 rows selected.

• The maximum part of the range.
• Table FirstMany has 60 rows.
Cartesian Products in SQL Queries - Less than (n-1) join conditions

• Remove one of the join conditions:

  • Select  l.lookuptableid,
  •  o.onetableid,
  •    f.firstmanyid
  • from    lookuptable    l, /* 10 rows */
  •         onetable       o, /* 20 rows */
  •    firstmany f /* 60 rows */
  • where   f.onetableid = o.onetableid
  • order by 1, 2, 3

• 3 tables, only 1 join condition.
• \( N^{(n-2)} \)
Cartesian Products in SQL Queries - Less than (n-1) join conditions

- Result:
  - ...
  - 10  117  1
  - 10  117  2
  - 10  117  3
  - 10  118  1
  - 10  118  2
  - 10  118  3
  - 10  119  1
  - 10  119  2
  - 10  119  3
  - 600 rows selected.

- 10 times more than the correct join.
Cartesian Products in SQL Queries - Less than (n-1) join conditions

• Remove a different join condition.

• Select l.lookuptableid,
  o.onetableid,
  f.firstmanyid

• from lookuptable l, /* 10 rows */
•    onetable o, /* 20 rows */
•    firstmany f /* 60 rows */
• where o.lookuptableid = l.lookuptableid
• order by 1, 2, 3

• 3 tables, only 1 join condition.
• N (N-2)
Cartesian Products in SQL Queries - Less than (n-1) join conditions

- Result:
  - ...
  - 10  119  3
  - 10  119  3
  - 10  119  3
  - 10  119  3
  - 10  119  3
  - 10  119  3
  - 10  119  3
  - 10  119  3
  - 10  119  3
  - 10  119  3
  - 10  119  3

- 1200 rows selected.

- 20 times more than the correct result.
How To See The (Not N-1) Cartesian Product

• Many tables, join conditions, and filter conditions can make it difficult to see a cartesian product. Ie.

  • Select Field1, Field2
  • From Table_A a, Table_B b, Table_c c
  • Where Field2 = abc
  • And Field3 = jkl
  • And A.PK = B.THE_pk_of_a
  • And field1 = xyz

• One join condition is missing.
• Table_C needs to be joined to either Table_A, or Table_B.
How To See The (Not N-1) Cartesian Product

• Apply the mathematics concepts we learned in high school.

• Ie. Let $x = (a+b)$

• Let the well joined part of the query, involving Table_A, and Table_B, become a view.
How To See The (Not N-1) Cartesian Product

- Create View V_Well_Joined_Query
- As
- Select Field1,
- Field2
- From Table_A a,
- Table_B b
- Where a.PK = B.THE_pk_of_a
- And Field2 = abc
- And Field3 = jkl
- And Field1 = xyz
How To See The (Not N-1) Cartesian Product

• Now, add Table_C back to the query:
  
  • Select Field1,
  •      Field2
  • From V_Well_Joined_Query,
  •      Table_C
  • Result is:
  • The classic cartesian product of no join conditions.

• 1) and 2) are really the same.
3) Joining tables on non-unique values.
Also called "Joining down the middle".

Neither field, or set of fields, is unique, and neither is a PK.
Here, joining on the field: LookupTableID

Note: There is NO correct way to join these two tables.
Cartesian Products in SQL Queries - Joining Down the Middle

• Query:

    • Select o.onetableid,
    •       u.useslookupsid,
    •       o.lookuptableid
    • from onetable o, /* 20 rows */
    •       useslookups u /* 30 rows */
    • where o.lookuptableid = u.lookuptableid
    • order by 1, 2

• Counts:
• OneTable: 20
• UsesLookups: 30
Cartesian Products in SQL Queries - Joining Down the Middle

- Result:
  - ...  
  - LooktableID
  - 116     74     9
  - 116     75     9
  - 116     76     9
  - 117     74     9
  - 117     75     9
  - 117     76     9
  - 118     77     10
  - 118     78     10
  - 118     79     10
  - 119     77     10
  - 119     78     10
  - 119     79     10
  - 60 rows selected.
Cartesian Products in SQL Queries - Joining Down the Middle

- Result: 60 rows returned.

- Not a full cartesian product of 600 rows.

- But, more rows than either of the two tables
  - (20 rows, and 30 rows).
Cartesian Products in SQL Queries - Joining Down the Middle

• If each side of the join condition, is NOT Unique, you will (eventually) get a cartesian product as shown above.

• May go undetected for a long time.

• Perhaps when written initially, there was not a lot of realistic data to work with, unique on that field.

• As the database grows, the data becomes non-unique, and you then get cartesian products.
Cartesian Products in SQL Queries - Not All Fields Included in the Join

• 4) Not including all the fields in the join condition

• Here there is a concatenated key in both tables. The fields are:
  • OneTableID
  • FirstManyID

• A proper join needs both fields.
Cartesian Products in SQL Queries - Not All Fields Included in the Join

- Proper join
- Two fields included.
- One join condition.

- Select `count(*)`
- from `firstmany f, /* 60 rows */`
- `secondmany s /* 240 rows */`
- `where f.onetableid = s.onetableid`
- `and f.firstmanyid = s.firstmanyid`

- `COUNT(*)`
- `----------`
- `240`

- As it should be.
Cartesian Products in SQL Queries - Not All Fields Included in the Join

- Remove one of the fields in the join condition.

- Select count(*)
  - from firstmany f, /* 60 rows */
  - secondmany s /* 240 rows */
  - where f.onetableid = s.onetableid

  COUNT(*)
  __________
  720

- Three times what it was before.
Cartesian Products in SQL Queries - Incorrect use of OR conditions

- 5) Incorrect use of OR conditions

- Query with the same two tables.

- Proper join, with no filter conditions (AND, OR) will give 240 rows.
Cartesian Products in SQL Queries - Incorrect use of OR conditions

- Add a single OR condition:

  - Select `count(*)`
  - from `firstmany f, /* 60 rows */`
  - `secondmany s /* 240 rows */`
  - where `f.onetableid= s.onetableid`
  - and `f.firstmanyid = s.firstmanyid`
  - or `secondmanyid = 4`

  - `COUNT(*)`
  - ---------
  - 3780
Cartesian Products in SQL Queries - Incorrect use of OR conditions

- Add one AND condition, and one OR condition:

```sql
-- Select count(*)
-- from firstmany f, /* 60 rows */
-- secondmany s /* 240 rows */
-- where f.onetableid= s.onetableid
-- and f.firstmanyid = s.firstmanyid
-- and secondmanyid = 3
-- or secondmanyid = 4

COUNT(*)
--------
3660
```
Cartesian Products in SQL Queries - Incorrect use of OR conditions

- So even though the number of join conditions are correct, we still get a cartesian product!

- Corrected:
  - Use brackets around the OR condition.
Cartesian Products in SQL Queries - Incorrect use of OR conditions

- Select count(*)
- from firstmany f, /* 60 rows */
  secondmany s /* 240 rows */
- where f.onetableid = s.onetableid
- and f.firstmanyid = s.firstmanyid
- and (secondmanyid = 3
- or secondmanyid = 4)

- COUNT(*)
- --------
- 120

- 120 < 240
- Filter condition present.
Cartesian Products in SQL Queries - Incorrect use of OR conditions

- Perhaps not a cartesian product, mathematically.

- However, the symptoms are the same, multiples more rows than should occur, so I include it here.

- If not a cartesian product, what is it?
Cartesian Products and Performance

- Most all cartesian products will give performance problems.

- Cartesian products are often only discovered because of performance issues.
Cartesian Product Performance Case Study One

- SQL Server Case from work.

- Fewer than N-1 join conditions.

- About six tables in the query.

- Used INNER JOIN syntax.
  - Table joins are done in the FROM clause.
  - There is no clear WHERE clause.
  - Filtration can also done in the WHERE clause.
Cartesian Product Performance Case Study One

- SELECT Count(*)
- FROM Table_A a
- JOIN Table_B b
- ON b.fieldx IN ( 'value1', 'value2' )
  - /* actually a Filter condition, not a join condition */

- Not a join condition, but actually a filter condition (ON).
- Second table to join on is not there.

- 27 minutes to complete.
Cartesian Product Performance Case Study One

• Corrected:

  • SELECT      Count(*)
  • FROM        Table_A   a
  • JOIN        Table_B   b
  • ON          a.joinfield = b.joinfield  /* join condition */
  • WHERE       b.fieldx IN ('value1', 'value2')  /* filter condition */

Time: 7 seconds.
From 27 minutes to 7 seconds.
About 231 times faster.
Cartesian Product Performance Case Study Two

- Not all the fields in the concatenated key included in the join.
- Oracle case from work.

- Query working behind a gui.
- Six or seven seconds to move the cursor to the next record.

- Very irritating to the user.

- Isolate the query.
Cartesian Product Performance Case Study Two

• Query had a number of filter conditions.
• NOT EXISTS

• Second table was an archive table.
• Same table structure of the first.

• Keep the join conditions
• Remove the filter conditions.

• Use Count(*)
Cartesian Product Performance Case Study Two

- **USING NOT EXISTS**

  ```sql
  SELECT count(1)
  FROM table_t t
  WHERE NOT EXISTS
  (  
    SELECT 1
    FROM table_s s
    WHERE t.joinfield1 = s.joinfield1
  )
  ```

- Using Not Exists took 8 hours.
Cartesian Product Performance Case
Study Two

• Rewrite the query using an outer join.
• My experience: Outer Joins are faster than Not Exists

• \[
\begin{align*}
\text{SELECT} & \quad \text{COUNT}(1) \\
\text{FROM} & \quad \text{table}_t \quad \text{t} \\
\quad & \quad \text{Table}_s \quad \text{s} \\
\text{WHERE} & \quad \text{t}.\text{joinfield1} = \text{s}.\text{joinfield1} \quad (+) \\
\text{AND} & \quad \text{s}.\text{joinfield1} \text{ IS NULL}
\end{align*}
\]

• Time: 10 hours.

• 25% longer. 2 hours longer.
• Something definitely wrong!
Cartesian Product Performance Case Study Two

- Determine the primary keys of the table(s).
- Two fields in the Primary key! Not one.
- Add the second field of the PK to the join condition.
Cartesian Product Performance Case Study Two

- Corrected: USING NOT EXISTS

- SELECT count(1)
- FROM table_t t
- WHERE NOT EXISTS
- (SELECT 1
- FROM table_s s
- WHERE t.joinfield1 = s.joinfield1
- AND t.joinfield2 = s.joinfield2
- )

Time: 4 seconds.
From 8 hours (50400 seconds) to 4 seconds.
12600 times faster!
Cartesian Product Performance Case Study Two

• Corrected. Using an Outer Join.

• SELECT COUNT(1)
• FROM table_t t,
  Table_s s
• WHERE t.joinfield1 = s.joinfield1 (+)
• AND t.joinfield2 = s.joinfield2 (+)
• AND s.joinfield1 IS NULL
• AND s.joinfield2 IS NULL

Time: One second!
From 10 hours (36000 seconds) to 1 second.
About 36000 times faster.
Cartesian Product Performance - Case Study Three

• One shop.

• Cartesian products found in numerous places.
• Views, packages, procedures, queries.
• Eventually, I just started listing them into a document.

• Extremely complex schema and code.
• DB design mistake of “too many fields in the PK”.
• Multiple levels of nested inline views in single views or queries.
Cartesian Product Performance - Case Study Three

- JSP front end app.
- A simple ad hoc query took over 2 hours!

- SQL Code fragments found in many JSP files.
- Different files/parts for:
  - SELECT here
  - FROM clause
  - WHERE clause
  - GROUP BY clause
  - ORDER BY clause
Cartesian Product Performance - Case Study Three

- To isolate the query.
- Start a query from the JSP front end.
- Watch the active processes in the Oracle data dictionary.
- Expand the address of the SQL statement(s) being executed.
- One single SQL statement was over 17,000 bytes long!
- Modified the parameters with hard numbers.
- Reran the query with autotrace/explain plan.
Cartesian Product Performance - Case Study Three

- Explain plan was over 100 lines long!
- (the longer the plan, the slower the query.)
- MERGE JOIN CARTESIAN found in numerous places.

- Solution:
  - Check the join conditions of the many tables in the query.
  - At least one field in the PK not included in the join condition.
  - Add the join condition.

- Result: about 2 second response time.
- At least 60 times faster.
Tuning Cartesian Products

- Only real solution to speed up cartesian products:
  - Correct the SQL code
  - Add the necessary join conditions.

- INDEXES REALLY DON'T HELP.

- STATISTICS REALLY DON'T HELP.
Indexes and Statistics - Performance Case Study Four

- Scenario: Miss one of the concatenated fields.

- Table row count
  - Firstmany: 60,000
  - Usesmany: 120,000
Indexes and Statistics - Performance Case Study Four

- Using a Proper join:

```sql
SELECT count(*)
FROM firstmany f, /* 60,000 rows */
     usesmany u /* 120,000 rows */
WHERE f.onetableid = u.onetableid
AND f.firstmanyid = u.firstmanyid

COUNT(*)
--------
120000
1 row selected.
Elapsed: 00:00:00.38

- Less than one half second response time.
Indexes and Statistics - Performance
Case Study Four

• Modify the query to get a cartesian product.

• Select count(*)
• from firstmany f, /* 60,000 rows */
• usesmany u /* 120,000 rows */
• where f.firstmanyid = u.firstmanyid
• /* Remove one join condition */

• COUNT(*)
• -------
• 2400000000
• 1 row selected.
• Elapsed: 00:08:36.15

• 2,400,000,000 = 2.4 billion rows.
• 08:36.15 = 8 minutes, 36 seconds
Indexes and Statistics - Performance Case Study Four

- Systematically add indexes and statistics.
- Results: Time response matrix in Minutes:seconds

<table>
<thead>
<tr>
<th>No Statistics</th>
<th>With Statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Index</td>
<td>8 min 36s</td>
</tr>
<tr>
<td>With Index</td>
<td>8 min 14s</td>
</tr>
</tbody>
</table>

- The fastest time, 4:46, is 286 times slower than the correct code.
- Notice that the index actually slowed down the response time when used with statistics. (5:07, versus 4:46)

- While there might be a marginal improvement with index and statistics, it's nowhere near what it would (should) be if the code were corrected.
Indexes and Statistics - Performance Case Study Four

- Some interesting statistics about this case study, in rows per second.

- Slowest time:
  - 2.4 billion rows / (8 minutes, 36 seconds) =
  - 4,651,162 rows per second

- Fastest time:
  - 2.4 billion rows / (4 minutes, 36 seconds) =
  - 8,391,608 rows per second

- The speeds are this fast, because there is no IO involved.
- When these queries ran, the CPU was utilized 100%.
- Environment: Redhat Linux. Oracle 9i. Intel 2.4 Ghz processor.
Cartesian Products and Statistics

• Why don't statistics work much with cartesian products?

• Consider the classic cartesian product:
  
  • Select count(*)
  • From Table_A,
  • Table_B

• Which particular fields would you like to run statistics for?
Cartesian Products and Statistics

• Statistics help with join and filter conditions.

• But, in the classic cartesian product,
  – When you have no join conditions, statistics don't help.
  – When you have no filter conditions, statistics don't help.

• When you have no join conditions, and no filter conditions, statistics really cannot help much.
Cartesian Products and Statistics

- Revisiting Case study one.
- Fewer than N-1 join conditions, using six tables and the INNER JOIN.

<table>
<thead>
<tr>
<th></th>
<th>Using Corrected Code</th>
<th>Using Uncorrected Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before Statistics update</td>
<td>Rows returned: 2</td>
<td>Rows returned: 36</td>
</tr>
<tr>
<td></td>
<td>Total Time: 7 seconds</td>
<td>Total Time: 27 minutes</td>
</tr>
<tr>
<td>After Statistics update</td>
<td>Rows returned: 2</td>
<td>Rows returned: 36</td>
</tr>
<tr>
<td></td>
<td>Total Time: 7 seconds</td>
<td>Total Time: 27 minutes</td>
</tr>
</tbody>
</table>

- From 27 minutes (1620 seconds) to 7 seconds.
- Approximately 231 times faster.

- Updating statistics made no difference.
- Correcting the code made all the difference.
Tuning Cartesian Products - Indexes

• Why don't indexes help much?
• Again consider the classic cartesian product:

  • Select  count(*)
  • From    Table_A,
  •          Table_B

• Which field(s) would like to index?

• Since there are no join conditions, and no filter conditions, an index really cannot help you much.
Tuning Cartesian Products - Indexes

- Case study two in more depth.
- Not all the fields in the concatenated key included in the join.
- Using NOT EXISTS.

<table>
<thead>
<tr>
<th></th>
<th>Using Corrected Code</th>
<th>Using Uncorrected Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>With No Index</td>
<td>Rows returned: 0</td>
<td>Rows returned: 70 billion</td>
</tr>
<tr>
<td>Total Time: 1 to 4 seconds</td>
<td>Total Time: 8 to 10 hours</td>
<td></td>
</tr>
<tr>
<td>After adding an Index</td>
<td>Rows returned: 0</td>
<td>Rows returned: 70 billion</td>
</tr>
<tr>
<td>Total Time: 1 to 4 seconds</td>
<td>Total Time: 8 to 10 hours</td>
<td></td>
</tr>
</tbody>
</table>

- Using an index made no difference.
- Correcting the code made all the difference.
Tuning Cartesian Products - Indexes

- Why don't indexes work with a cartesian product?

- Indexes take large sets of data, and order the data into smaller sets of values.
- Here, this is non unique data. Each value is found in 1 to 3 rows.
- If you search on a particular value, the index will narrow down the entire table to a few rows. In this case, a maximum of three rows.

- 15 rows. 7 unique values.
- Rows per value.
- Min: 1 row. Max: 3 rows.
- Average: $15/7 = 2.14$
Column Cardinality

• Column Cardinality - The number of unique values in a column or field.

• Here, a column cardinality of one.

• An index here does not help.

• If you search on X, you still search the whole table!
Column Cardinality

- Data with a low column cardinality.
- Here, a column cardinality of two.
- Two unique values: Y, N

- Again, an index here does not help much.

- If you search on Y, or N, you still search half the table!

- If the table is one million rows, you linear search through one half million rows.
Cartesian Products, Indexes, and Column Cardinality

- Returning to case study four.
- Miss one field in the join condition.
Cartesian Products, Indexes, and Column Cardinality

- What's the column cardinality of the FirstmanyID field?
- In Table: Firstmany

  - Select count(distinct FirstmanyID )
  - as cardinality,
  - count(*)
  - from Firstmany /* 60,000 rows */

  - CARDINALITY COUNT(*)
  - ----------- -----------
  - 3 60000
Cartesian Products, Indexes, and Column Cardinality

• What's the column cardinality of the Firstmany field?
• In Table: Usesmany

• Select count(distinct Firstmanyid )  
  as cardinality ,  
  count(*)  
• from Usesmany /* 120,000 rows */

• CARDINALITY    COUNT(*)  
• ---------------  -------  
•       3          120000
Cartesian Products, Indexes, and Column Cardinality

- In both tables, there are many rows of each value.

- `select Firstmanyid,`  
- `    count(*)`  
- `from Firstmany  /* 60,000 rows */`  
- `group by Firstmanyid`  

<table>
<thead>
<tr>
<th>FIRSTMANYID</th>
<th>COUNT(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20000</td>
</tr>
<tr>
<td>2</td>
<td>20000</td>
</tr>
<tr>
<td>3</td>
<td>20000</td>
</tr>
</tbody>
</table>
Cartesian Products, Indexes, and Column Cardinality

• Had we joined on the other field, Onetableid:

  • Select count(*)
  • from firstmany f, /* 60,000 rows */
  • usesmany u /* 120,000 rows */
  • where f.onetableid = u.onetableid

  • COUNT(*)
  • --------
  • 360000
  • 1 row selected.
  • Elapsed: 00:00:00.24
  • (About 1/4 second response time.)
Cartesian Products, Indexes, and Column Cardinality

• We still get a cartesian product (360,000 rows, not 120,000 rows), but we also get a very fast response time. Why?

• Select count(distinct onetableid) onetableid,
  count(*)
  from usesmany /* 120,000 rows */

<table>
<thead>
<tr>
<th>ONETABLEID</th>
<th>COUNT(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20000</td>
<td>120000</td>
</tr>
</tbody>
</table>

• Reason: the field Onetableid field actually has a column cardinality of 20,000 in both tables. 20,000 distinct values.
Cartesian Products, Indexes, and Column Cardinality

- Indexes do not help when the column cardinality of a field is low.
- Indexes help where the column cardinality of a field is high.

- The greater the column cardinality of the field(s) being joined on (the more unique data in the field), the faster the response time.
- A column cardinality equal to the number of rows in a table, can be used as a PK, or an alternate key.

- With a cartesian product, you get many rows, with low column cardinality.
- Therefore, indexes really do not help with a cartesian product.
Cartesian Products - Data Issues

- Cartesian Products also cause serious data issues.
- The same case studies from before returned an incorrect number of rows.
- Correct number of rows on left. Incorrect on right.

<table>
<thead>
<tr>
<th></th>
<th>Using Corrected Code</th>
<th>Using Uncorrected Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before Statistics update</td>
<td><strong>Rows returned:</strong> 2</td>
<td><strong>Rows returned:</strong> 36</td>
</tr>
<tr>
<td></td>
<td>Total Time: 7 seconds</td>
<td>Total Time: 27 minutes</td>
</tr>
<tr>
<td>After Statistics update</td>
<td><strong>Rows returned:</strong> 2</td>
<td><strong>Rows returned:</strong> 36</td>
</tr>
<tr>
<td></td>
<td>Total Time: 7 seconds</td>
<td>Total Time: 27 minutes</td>
</tr>
<tr>
<td></td>
<td>******************</td>
<td>******************</td>
</tr>
<tr>
<td>With No Index</td>
<td><strong>Rows returned:</strong> 0</td>
<td><strong>Rows returned:</strong> 70 billion</td>
</tr>
<tr>
<td></td>
<td>Total Time: 1 to 4 seconds</td>
<td>Total Time: 8 to 10 hours</td>
</tr>
<tr>
<td>After Adding an Index</td>
<td><strong>Rows returned:</strong> 0</td>
<td><strong>Rows returned:</strong> 70 billion</td>
</tr>
<tr>
<td></td>
<td>Total Time: 1 to 4 seconds</td>
<td>Total Time: 8 to 10 hours</td>
</tr>
</tbody>
</table>
Cartesian Products - Data Issues

• Case study five: Data problem caused by a Join Down the Middle.
• Two systems were redesigned to talk to each other.

• Many units sold on a single contract.
• Should be a simple one to many.
• But the two systems were connected with “a join down the middle”.

• If you started with a single machine, and navigated from one system to the other, you got three contracts!
• Impossible to know which contract was the one to update!

• The update actually had to be done manually!
Cartesian Products - Data Issues

• Case study six:
  • An XML / java programmer installed new code.
  • It created 28 rows for the same customer, not one!
  
  • The data was also tallied 28 times.
  • I.e. $28,000.00, not $1000.00.

  • Test cases: “The customer attempts to spend the phantom $27K we said they had in their account.”
  • “The CFO attempts to invest the phantom $27K we said was available.”
Cartesian Products - Operators

- Some Operators to be very aware of:
  - Distinct, Group By,
  - Union, Minus, Intersect
  - (some only exist in Oracle.)

- All of these operators eliminate duplicates.
- Cartesian products will not be obvious.
- Performance problems will still exist.
Cartesian Products - Operators

- Distinct, Group By,
- Union, Minus, Intersect

- When testing code for cartesian products, eliminate these operators during your tests.

- In the SELECT clause, use COUNT(*) instead of fields.
Cartesian Products - Operators

- For Union, Minus, or Intersect.
- Test each half of the query separately.
- Count the rows of each query.
- Then combine both queries with the operator.

Note: UNION ALL is different from UNION.

- UNION ALL will show every row, including every duplicate.
Cartesian Products - Operators
UNION ALL vs. Union

- Select u.lookuptableid
- from useslookups u /* 30 rows */
- UNION ALL
- select t.lookuptableid
- from onetable t /* 20 rows */

- ...
- 8
- 8
- 9
- 9
- 10
- 10

- 50 rows selected.
Cartesian Products - Operators
UNION ALL vs. Union

• Select u.lookuptableid
• from useslookups u  /* 30 rows */
• UNION
• select t.lookuptableid
• from onetable t  /* 20 rows */

• LOOKUPTABLEID
  --------------
  ...
  • 8
  • 9
  • 10

• 10 rows selected.
Cartesian Products - Operators

**INTERSECT**

- Select `u.lookuptableid`
- from `useslookups u` /* 30 rows */
- **INTERSECT**
- select `t.lookuptableid`
- from `onetable t` /* 20 rows */

- **LOOKUPTABLEID**
- -----------------
- ...
- 
- 8
- 9
- 10

- 10 rows selected.
Cartesian Products - Operators

MINUS

- Select u.lookuptableid
  - from useslookups u /* 30 rows */
- MINUS
- select t.lookuptableid
  - from onetable t /* 20 rows */
- no rows selected

- If any of these contained queries with cartesian products on both the top and bottom, all the cartesian products would be hidden from you in the result.
- Exception: Union All
A Group By is a Distinct

- Select distinct u.lookuptableid
- from useslookups u

<table>
<thead>
<tr>
<th>LOOKUPTABLEID</th>
<th>COUNT(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
</tr>
</tbody>
</table>

10 rows selected.

- Point to note: A GROUP BY, is a DISTINCT.

- Select u.lookuptableid, COUNT(*)
- from useslookups u
- GROUP BY u.lookuptableid

<table>
<thead>
<tr>
<th>LOOKUPTABLEID</th>
<th>COUNT(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
</tr>
</tbody>
</table>

10 rows selected.

- Remove COUNT(*) from the GROUP BY query
- Get exactly the same results as the DISTINCT.
Correcting a Cartesian Product

- No software (I know of) will help you correct a cartesian product in a SQL query.
- No command such as DBMS_???, alter, drop, show, etc.
- No GUI from Toad, or Embarcadero, etc.

- Even if there were, views would be problematic.
- Without access to a view’s underlying code, what’s the effective key of the view?

- Correcting cartesian products requires thought and analysis.
- And finally rewriting the query.
Ways to Recognize a Cartesian Product

- Poor query performance is often a good indicator.
- If a query is performing slowly, include checks for cartesian products in your tuning methodology.
- Start with the five ways to get a cartesian product.
TIP - Remove All Filter Conditions

• When debugging SQL, and developing in general:

• Change the SELECT clause to COUNT(*)

• Change the WHERE clause to ONLY include join conditions

• Remove ALL filter conditions

• Select Count(*)
• From Table_A, Table_B
• Where a.PK = B.The_PK_of_a
• /* ... no join conditions */
TIP - Remove All Filter Conditions

- Select Count(*)
- From Table_A,
  Table_B
- Where a.PK = B.The_PK_of_a
- /* ... no join conditions */

- Your query MUST perform reasonably before adding filter conditions.
- Otherwise, filter conditions are masking any issues.
Quick Check

• Count the number of rows in each table in the query.
• This is sometimes easier to do than looking up the PK.

• The rows returned by the query, need to be less than the counts of ANY of the tables in the query.
Ways to Recognize a Cartesian Product

- The primary keys of the child tables are coming back duplicated.

- You will expect the PK of the parent tables to be duplicated,

- But this should never happen to the PK of the child table.
Ways to Recognize a Cartesian Product

• A Group By and Count(*) on the child table’s PK returns:
  • Count(*) > 1

  • Select Child_PK_field,
  • Count(*)
  • From Parent_Table p,
  • Child_Table c
  • /* no join conditions */
  • Group by Child_PK_field
  • Having Count(*) > 1
Ways to Recognize a Cartesian Product

• Alternatively,
• Select Distinct of the child PK gives smaller count than count(*).

- Select count(Distinct Child_PK_fields),
  count(*)
- From Parent_Table p ,
  Child_Table c
- /* no join conditions */
Ways to Recognize a Cartesian Product - Statistics

- Statistics for a cartesian product query may indicate a massive number of IOs.

- In SQL Server or Sybase, the logical reads may be many-multiples more than the number of rows in any of the tables.
- Case study one returned 36 rows, but had millions of reads.

- In Oracle, you may see the phrase:
  - MERGE JOIN (CARTESIAN)
  - This may or may not indicate a cartesian product.
- Don’t rely on this. Check everything.
Basic Rules to Avoid Cartesian Products

• Rule one:

• Always have (n-1) join conditions

• ie.
• For three tables, you need (n -1) = 2 join conditions
Basic Rules to Avoid Cartesian Products

• Rule Two:
  • For every join condition between two tables, the set of fields for at least one table, MUST be unique.
  • Otherwise, it's a "join down the middle".

• This does not seem to be well known in the SQL world.
  • I have not seen this mentioned:
    • - in courses and textbooks
    • - in books
    • - in company documentation
    • - anywhere
Best Practices

• When writing queries, write the joins conditions first, and filter conditions at the end.
• Put each table, and each join condition on a separate row.
• Easier to eyeball N tables, and (N-1) join conditions.

• ALWAYS check what the primary keys of ALL the tables are!
• Just because something is named fieldname_id does not make it the primary key.
• Beware of tables that don't have a PK at all.
• If you are using views, determine what the effective key of the view is.
Best Practices

• Do not creatively eliminate join conditions.  Ie.
  
    • Select  count(*)
    • from    Table_A   A,
    •        Table_B   B
    • Where   A.Field2 = somevalue
    • and     B.Field1 = anothervalue

• Where one or more filter conditions are supposed to return only one row.
• Difficult to maintain, especially when there are many tables and filter conditions.
Best Practices - Tools

• Test your front end and ETL tools such as Brio, Crystal Reports, Business Objects, Informatica, etc., to see if you can create, insert or use a cartesian product, with any of the five methods.

• With Crystal Reports, it’s certainly possible to insert a SQL statement with a cartesian product in it.

• If the underlying SQL has a cartesian product, then the output will also be inaccurate.
Designing Cartesian Products Into Schemas

- Sometimes, cartesian products are designed right into schemas.

- So, it can be impossible to eliminate them from any SQL statement.
Designing Cartesian Products Into Schemas

• 1) Not using primary keys and foreign keys.
• 2) Not using UNIQUE constraints.
• 3) Using views that do not have a unique or effective key.
• 4) Joining down the middle with no PK, FK, or Unique value.
• 5) Not including all the fields in the "child" table.
• 6) Denormalize incorrectly on non-unique data.
• 7) Multiple lookup table scenario.
Designing Cartesian Products Into Schemas

1) Not using primary keys and foreign keys. This is probably the biggest culprit of cartesian products designed into schemas.

And the number one thing to avoid.

<table>
<thead>
<tr>
<th>Field_A</th>
<th>INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field_C</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field_A</th>
<th>INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field_D</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>
Designing Cartesian Products Into Schemas

- 1) Corrected.
- Correct the design by using proper PK and FK.
- Either identifying (left), or non-identifying (right) relationships.
Designing Cartesian Products Into Schemas

• Why do primary keys and foreign keys work so well?

• They enforce the rule:

• For every join condition between two tables, the set of fields for at least one table, MUST be unique.
Designing Cartesian Products Into Schemas

- 2) Not using UNIQUE constraints when they can be used.

- Foreign keys can be created on both PK and Unique constraints/indexes.

- UNIQUE constraints are good for alternate keys and denormalization (not that I’m advocating denormalizing).

- Unfortunately, this does not seem to be well known.
Denormalization with Alternate key / Unique field

- Say you have many different identifiers for a security in a lookup table:
- SEC_ID (PK), Internal company code, Symbol, CUSIP, SEDOL, external company codes.
Denormalization with Alternate key / Unique field

- Depending on the data, it’s sometimes possible to put a UNIQUE constraint on the denormalized fields.

- Then on the child tables, if you denormalize the data, you can then create a FK to these UNIQUE constrained fields.
Creating FK to PK and Alternate Key

- Create table TestFKOne
  - ( TestFKOnePK integer,
    - UniqueField integer
  )

- Table created.

- Create table TestFKMany
  - ( TestFKOnePK integer,
    - TestFKManyPK integer,
    - UniqueField integer
  )

- Table created.
Create FK to Primary Key

- alter table TestFKMany
- add constraint FK_TestFKOne
- FOREIGN KEY (TestFKOnePK)
- references TestFKOne

ERROR at line 4:
ORA-02268: referenced table does not have a primary key

- alter table TestFKOne /* CREATE PRIMARY KEY */
- add constraint PK_TestFKOne
- Primary key
- ( TestFKOnePK )

Table altered.

- alter table TestFKMany /* CREATE the FOREIGN KEY */
- add constraint FK_TestFKOne
- FOREIGN KEY (TestFKOnePK)
- references TestFKOne

Table altered.
Create FK to Alternate Key

- Create a FK to a Unique field:

  ```sql
  alter table TestFKMany
  add constraint FK_TestUnique
  FOREIGN KEY (uniquefield)
  references TestFKOne (uniquefield)

  ERROR at line 4:
  ORA-02270: no matching unique or primary key for this column-list
  ```

  ```sql
  alter table TestFKOne /* Create a UNIQUE constraint */
  add constraint CONS_unique /* This will create a unique index: CONS_unique. */
  UNIQUE (uniquefield)
  
  Table altered.
  ```

  ```sql
  alter table TestFKMany /* CREATE the FOREIGN KEY */
  add constraint FK_TestUnique
  FOREIGN KEY (uniquefield)
  references TestFKOne (uniquefield)
  
  Table altered.
  ```
Referential Integrity and Performance

• General rule of thumb:
  • Referential integrity enforcement is always fastest on OLTP (not batch) systems using primary keys and foreign keys.
• This is not well known.

<table>
<thead>
<tr>
<th>Performance</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fastest</td>
<td>Primary Keys and Foreign Keys</td>
</tr>
<tr>
<td>Slower</td>
<td>Triggers</td>
</tr>
<tr>
<td>Slowest</td>
<td>Application Logic outside of the database: java, client, etc.</td>
</tr>
</tbody>
</table>
Referential Integrity and Performance

- Removing referential integrity leads to cartesian products.
- Cartesian products cause performance problems.

- Case study seven:
- System loaded data from Cobol files.
- There were no PK and no FK in the database.
- The load did searches on every lookup table as it loaded.
- No PK, no indexes. -> slow retrieval time.
- To load 100,000 records took 6 hours!
- 4.6 rows per second!!!!
- And, there was still duplicate data after it was loaded.
- Joins resulted in cartesian products.
Designing Cartesian Products Into Schemas

- 3) Using views that do not have an effective key.

- Views are sometimes used for security in some systems.
  - Rather than giving access to the underlying tables.

- Effective key: a series of fields that uniquely identify the row.
  - Views are just stored queries.
  - You cannot create a PK, or FK on a view
  - (although a FK can reference a view as the parent table.)
Designing Cartesian Products Into Schemas

• Example. the Oracle data dictionary view: All_Objects

• Other than the field, Object_ID, the effective key seems to be:
  • Owner
  • Object_name

• But not in all cases.
Designing Cartesian Products Into Schemas

- Outside of Object_id, the effective key of All_Objects is actually:
  - Owner
  - Object_Name
  - SubObject_name
  - Object_type

- (Oracle 9i)
- Unless all four fields are included, you will eventually get a cartesian product, and duplicate data.
Designing Cartesian Products Into Schemas

- Variation:
  - Using views that have a built in cartesian product!

- Case study eight: Teradata database, wrong documentation.
  - The "ID" of the view was not unique.
  - The queries brought back duplicate data and were slow.

- The effective key was actually a combination of fields.
Designing Cartesian Products Into Schemas

- 4) “Joining down the middle” with no PK, FK, or Unique value.
- This is related to Rule 1. Not using PK and FK.
- Here, declaring the tables are “joined” on non-Unique Field_C.
Designing Cartesian Products Into Schemas

4) “Joining down the middle” with no PK, FK, or Unique value.

Variation:

- Have a PK, but join on non-unique, Field_C.
- Previously covered in SQL queries.
Designing Cartesian Products Into Schemas

- 5) Not including all the fields in the "child" table.
- This is one of the more bizarre "designs" I've seen.
- Parent1 is supposed to be the one table,
- And Child1 is supposed to be the many table.
- Not possible for this modeling tool to model it: the arrow had to be drawn in after.
Designing Cartesian Products Into Schemas

• The "parent" has four fields in the PK.
• But the "child" has only 2 of the same fields.
• Mathematically, the "child", is in fact, the "parent".
• For each unique value in the PK of Child1, you can have many values in Parent1.
Designing Cartesian Products Into Schemas

- Caused by not understanding the data, or the entities.
- Again, a variation of not using PK and FK.
- Oracle would not allow a FK to be created like this.
Designing Cartesian Products Into Schemas

- Another variation that I've seen.
- The tables actually do have a PK, and FK, in a non-identifying relationship.
- But then, they also have the same non PK fields.
- Here, Parent2_Field1 and Parent2_Field2.
- Instructions are to join on those fields, not with PK, and FK.
- This is a variation of denormalizing, and "join down the middle".

![Database Diagram showing the relationship between Parent2 and Child2 tables with non-identifying fields and the instruction to join on those fields instead of PK and FK.]

Copyright - Rodger Lepinsky
March 2008
Designing Cartesian Products Into Schemas

- An identifying relationship should be like this.
Designing Cartesian Products Into Schemas

- 6) Denormalize incorrectly and do joins on that field.
- More of a data issue.
- Say that there actually is a unique index on Field_C in the parent table, and a FK to it.
Designing Cartesian Products Into Schemas

- The data in the child table should get populated like this.
- The values in the child table sync with the values in the parent.

<table>
<thead>
<tr>
<th>DownTheMiddle1:</th>
<th>DownTheMiddle2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field_A</td>
<td>Field_C</td>
</tr>
<tr>
<td>AA</td>
<td>11</td>
</tr>
<tr>
<td>BB</td>
<td>22</td>
</tr>
<tr>
<td>CC</td>
<td>33</td>
</tr>
<tr>
<td>DD</td>
<td>44</td>
</tr>
<tr>
<td>Etc.</td>
<td></td>
</tr>
</tbody>
</table>
Designing Cartesian Products Into Schemas

• But, if the data in the child table gets populated like this, and you join on Field_C, not the FK:
  • Data errors
  • Cartesian products.

<table>
<thead>
<tr>
<th>Field_A</th>
<th>Field_C</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA</td>
<td>11</td>
</tr>
<tr>
<td>BB</td>
<td>22</td>
</tr>
<tr>
<td>CC</td>
<td>33</td>
</tr>
<tr>
<td>DD</td>
<td>44</td>
</tr>
<tr>
<td>Etc.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field_A</th>
<th>Field_C</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA</td>
<td>11</td>
</tr>
<tr>
<td>BB</td>
<td>11</td>
</tr>
<tr>
<td>CC</td>
<td>11</td>
</tr>
<tr>
<td>DD</td>
<td>11</td>
</tr>
<tr>
<td>Etc.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field_A</th>
<th>Field_C</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA</td>
<td>11</td>
</tr>
<tr>
<td>AA</td>
<td>22</td>
</tr>
<tr>
<td>AA</td>
<td>33</td>
</tr>
<tr>
<td>AA</td>
<td>44</td>
</tr>
<tr>
<td>Etc.</td>
<td></td>
</tr>
</tbody>
</table>
Designing Cartesian Products Into Schemas

• 7) Multiple lookup table scenario.
• This is the one situation I've found that does use PK and FK, and yet, still gives a cartesian product!
Designing Cartesian Products Into Schemas

- If you join the LookupTable to only OneTable, no problem.
- If you join the LookupTable to only UsesLookups, no problem.
- Proper PK and FK.
- Proper join conditions.
- No cartesian product.
Designing Cartesian Products Into Schemas

- But, if you join all three tables, on LookupTableId, it is the same as joining OneTable, and UsesLookups by themselves.

- The same case that we had before.
- Joining Tables on Non-Unique Values
- Join Down the Middle.
- Remember: there was no correct way to rewrite the SQL.
Designing Cartesian Products Into Schemas

• The design is mathematically correct. But the SQL is odd.
• It's similar to joining together, on the STATE field, the tables:
• CUSTomer, EMPloyee, or SUPPlier.
• It makes no sense to join those tables on the STATE field.
Designing Cartesian Products Into Schemas

- STATE and other lookup fields are only common attributes, neither a unique identifier. And thus, give a cartesian product.
- Other common, non-unique, attributes could be: Color, Address, First_Name, Department, Status, Type, Date.
- Scientific studies of clouds, industrial products, a person’s eyes, could all have the field: Color.
- Color is also just a common attribute. It doesn’t make sense to join the associated tables on this field.
- (Although it might make sense in poetry.
- “... skyblue eyes …”)
Most Bizarre Cartesian Product Design Award

- The design was supposed to have been a standard 1:M:M.
- Like this Identifying relationship.
Most Bizarre Cartesian Product Design Award

- Or a 1:M:M like this.
- Non-Identifying.

- Unfortunately, I can't remember the table design exactly. But I do remember a very strange join process.
Most Bizarre Cartesian Product Design Award

• I needed to join two tables: Was1 and Was3.
• But there was no FK between the two tables.
• Their solution: well just join on field Middle2.
• However, I noted that caused a cartesian product.
Most Bizarre Cartesian Product Design Award

- Their “solution”:
- Join Was1 to Was2 on the FK.
- Get the data from the field Middle1
- Join Was2 to Was3 on Middle1!!!
- What’s the problem here?
Most Bizarre Cartesian Product Design Award

- Ie. Use not one, but two "Joins down the middle".
- But that didn't work either.
- Mathematically, in this case, two wrongs do not make a right.
Good Uses for Cartesian Products?

- Usually, no good uses for cartesian products.
- Occasionally useful to insert data into test systems.
Good Uses for Cartesian Products?

- Select Table_A.PK,
  the_inline_view.the_rownum
  'Test data ' || the_inline_view.the_rownum
- From Table_A,
  (  
  Select rownum as the_rownum  
  From All_objects  
  Where rownum <= 25  
  ) as the_inline_view

- The second query is an inline view that returns 25 rows.
- No join condition to the first table -> cartesian product.

- This duplicates data from Table_A.
- For each row in Table_A, return another 25 rows.
- If there were 10 rows in Table_A, this query would return 250 rows.
Good uses for Cartesian Products?

• This could then be used as an “Insert as Select” statement.

• Insert into Table_B
  • (three fields of the same data types)
  • (Select cartesian product query)
Good uses for Cartesian Products?

- Another good use: stress and performance tests!

- Cartesian products work the CPU very hard.

- Use them as a background process while you test something else.
Help Stamp out Cartesian Products!

• Questions ???
Contact Info

- Rodger Lepinsky

rodger22@earthlink.net

rodger22@gmail.com