Analyzing Plan Diagrams of Database Query Optimizers

Thomas Mayer

KIT, Institut für Programmstrukturen und Datenorganisation (IPD),
D-76131 Karlsruhe, Germany

Abstract. This report concentrates on the visualization and analysis of the behavior of query optimizers. In the paper [6] "Analyzing Plan Diagrams of Database Query Optimizers" the authors visualize the query plans and corresponding estimated costs of a SQL-query with variable parameters in one- or two-dimensional selectivity spaces. Therefore, they introduce Plan and Cost Diagrams. The authors show that the number of query plans of each tested commercial database is surprisingly high. Moreover, they determine a non-motivated fragmentation of the Plan Diagrams. To improve this situation the authors introduce Reduced Plan Diagrams which decrease the number of query plans by summarizing the query plans of Plan Diagrams with limited additional estimated costs. Finally, the authors state that the assumptions of Parametric Query Optimization (PQO) do not hold in practice. They suggest the use of Reduced Plan Diagrams for PQO. [6]

1 Introduction

1.1 The Selectivity space

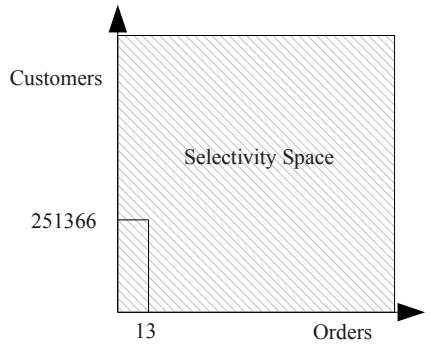

Fig. 1. Two-dimensional Selectivity Space

Relational selectivity space is a set of joined relations including conditions on attributes of these relations and the values of these attributes. The dimensionality of a selectivity space depends on the number of attributes with conditions. Using two conditions on 2 attributes, it is possible to navigate through that 2-dimensional selectivity space as demonstrated in figure 1: Given two joined relations *customers* and *orders*, we can use two conditions, *orderID = 13 and customerID = 251366* to select a single point. If we use "$\leq$" conditions instead, we can select from no data of any dimension (down-left corner) to all data of any dimension (top-right corner). In our example the conditions would look like *orderID* $\leq$ *13 and customerID* $\leq$ *251366*.

In the following, the word space will be a synonym for selectivity space. We will only consider two-dimensional spaces to make visualization easier. In practice, the selectivity space can have an arbitrary number of dimensions.

1.2 The Relational Database Management System

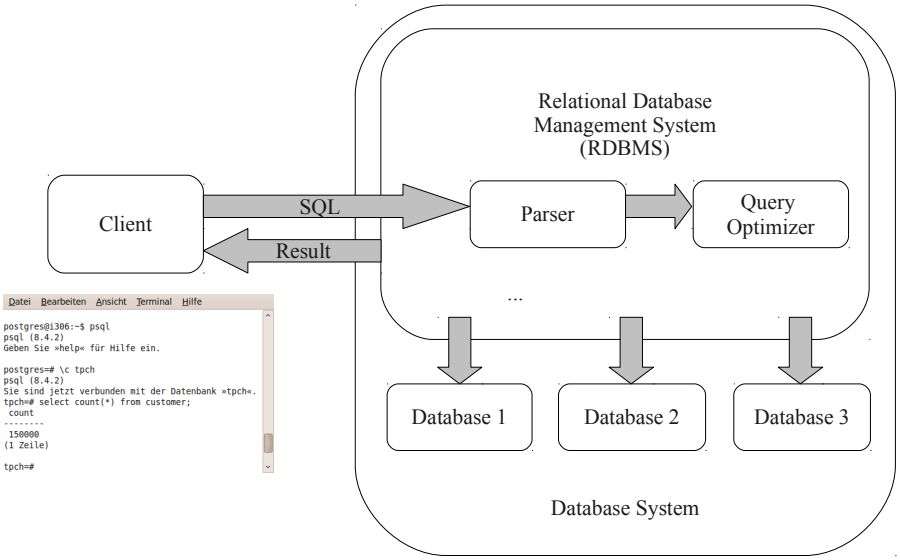

Fig. 2. RDBMS scheme

Relational Database Management Systems (RDBMS) offer the possibility to normalize the data into relations. For operations on that data, a declarative language called *Structured Query Language (SQL)* is usually used.

The RDBMS needs to carry out several steps to execute an SQL query. Figure 2 shows a typical execution stack. The parser checks for proper syntax and makes it usable for internal purposes. Afterwards, a query optimizer decides

how to execute the query (We will elaborate on this more later). Finally the query will be executed and the result of the query will be returned to the user as demonstrated in figure 2.

1.3 The Query Plan

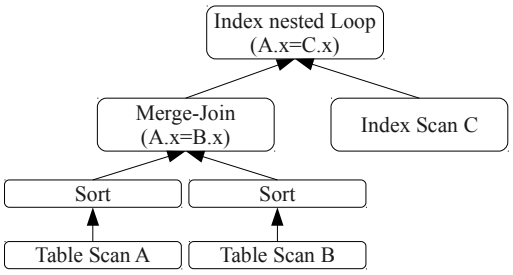

Fig. 3. Operator Tree

A query plan (or query execution plan) is a strategy to execute a SQL statement. Therefore, physical operators such as sort, sequential scan, index scan, nested-loop join and sort-merge join can be arranged in an operator tree. Such an operator tree represents a query plan [1]. The query plan does not affect the result of the query. But as the order can differ and different operators are possible to execute the query, many different query plans are possible as demonstrated in example 1. They span the *search space* [1].

Example 1. Given two relations A and B while A.id is a foreign key of B.Aid. *v_idx* is an index on the attribute *v*. Also given is a SQL query:

```
select A.id, B.id,u,v from A inner join B on (A.id=B.Aid) where v=0;
```

A

id	**u**
1	x
2	y
3	z

B

id	**Aid**	**v**
1	1	0
2	1	1
3	2	2
4	2	0
5	3	0
6	3	1

Result

A.id	**B.id**	**u**	**v**
1	1	x	0
2	4	y	0
3	5	z	0

The query can be executed as follows:

Plan P1: AB (first scan A, then join B). The estimated cost of scanning A is 3 because all rows have to be considered. As the relation B contains 6 rows, it might be estimated that every row in A has two child rows in B. This would mean that we can estimate the costs like this: $cost(AB) = 3 \cdot 2 = 6$.

Plan P2: BA (first scan B, then join A). The estimated cost of B scanning is about 3 because it might be estimated that when using index *v_idx* about 3 rows match the condition $v = 0$. As the join condition ($A.id = B.Aid$) corresponds to the foreign key relation, exactly one row in A will match to a row in B. As a result, we can estimate the costs as follows: $cost(BA) = 3 \cdot 1 = 3$. Both plans have the same result but the optimizer should choose plan P2 as it is expected to have less costs.

1.4 The Optimizer

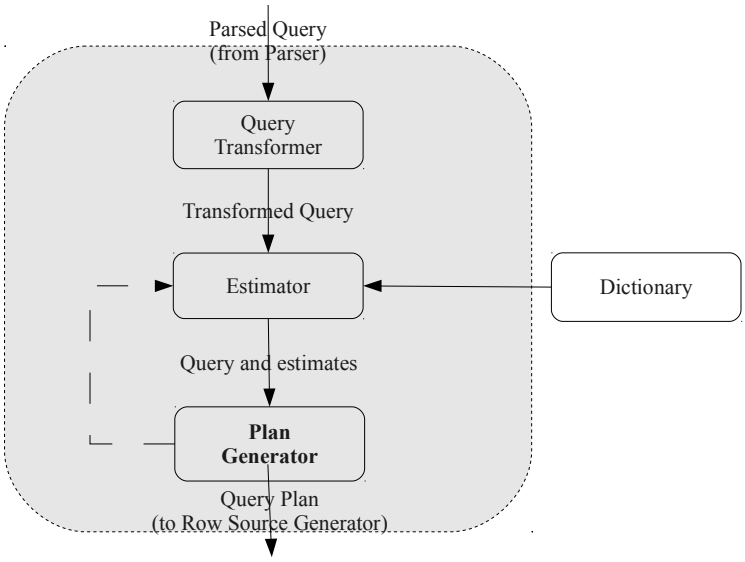

Fig. 4. Optimizer of a commercial RDBMS [4]

The optimizer is a part of the RDBMS which decides which is the estimated best strategy to execute the query. To have a closer look on optimizers, an implementation [4] of a commercial RDBMS as shown in figure 4 can be considered: First, the parsed query must pass the *query transformer* inside the optimizer. The query transformer rewrites the query using techniques like:

- view merging
- predicate pushing
- subquery unnesting (as demonstrated in example 2)
- query rewrite with materialized views
- or-expansion

Example 2. Given the relations of example 1. Using subquery unnesting, a query including a subquery

```
select A.id, u from A where A.id in (select Aid from B where v=0);
```

can be transformed into a query using a join instead.

```
select A.id, u from A inner join B on (A.id=B.Aid) where v=0;
```

Next, the transformed query passes the *estimator* [4]. This section estimates the costs (e.g. number of rows) of different operations which might be relevant to execute the query. Therefore, a dictionary (which contains statistical information about the data) can be used to be able to estimate the number of rows which match a where clause. It is a requirement for the estimations in example 1 that the dictionary contains information about the number of rows of each relation and spreading of values in attribute *v*. Finally, the *Plan Generator* is determining the expected optimal query plan to execute the query.

1.5 The Explain Statement in SQL

Many RDBMS offer the possibility to determine the query plan that the optimizer selects to execute a SQL query. Therefore, a user can send an *explain* statement to the RDBMS. An *explain* statement in SQL is a statement which consists of the keyword "explain" followed by a *select* statement. An example is given in example 3. The *select* statement will then be parsed by the parser of the RDBMS. Afterwards, the optimizer will decide which is the expected optimal query plan to execute the query as demonstrated in example 3 with the same data as in example 1. The corresponding query plan is the result of the *explain* statement and will be returned to the user.

Example 3.

```
explain
select A.id, B.id,u,v
from A inner join B on (A.id=B.Aid)
where v=0;

QUERY PLAN
------------------------------------------------------------
 Hash Join  (cost=1.07..2.18 rows=3 width=6)
   Hash Cond: (b.aid = a.id)
   ->  Seq Scan on b  (cost=0.00..1.07 rows=3 width=8)
         Filter: (p = 0)
   ->  Hash  (cost=1.03..1.03 rows=3 width=6)
         ->  Seq Scan on a  (cost=0.00..1.03 rows=3 width=6)
```

1.6 Introduction into the topic

[3] introduces a tool to visualize which query plan is used by a database at each point of the selectivity space. Using the visualization of query plans over the whole selectivity space, it is possible to determine the stability of query optimizers. An optimizer is relatively stable if only few query plans are used and if these query plans cover convex areas of the selectivity space.

It is expected behavior that more selectivity leads to more costs, so the costs should increase monotonically. Additionally, little more selectivity should lead to little more increase of costs. However, in case the estimated costs do not correspond to the execution costs of a plan, two things can go wrong when considering two neighbored points of two different plans in the space: First, the execution costs can decrease when increasing the selectivity. Second, the execution costs could increase dramatically, because the optimizer could not determine optimality regions in that way, that the plan is changed when a break even point from one plan to another plan is reached. Despite from real execution costs, highly intricate patterns and irregular boundaries in plan optimality regions indicate strongly non-linear cost models [5].

In such a RDBMS with a non-stable optimizer it can happen at several points of the selectivity space that a query is running much longer than expected by optimizer and user. The problem is that it is very hard to figure out under which circumstances such a problem occurs and how to debug it. The problem might even worsen if the data changes continuously. Additionally, a high number of query plans can mean that there is lots of overhead inside the optimizer [6].

As a result, there is to say that it would be great if the number of query plans and the number of their occurrences over the selectivity space could be reduced, while the cost increase for such a reduction should be very limited.

The authors compare the number of plans and the stability of optimizers. They do not consider the execution time of queries which means that they do not rate or compare the speed of databases.

2 The Visualisation of Query Plans and Costs

2.1 Plan Diagrams

By increasing the selectivity stepwise in each dimension, it is possible to determine the estimated costs at each point by sending an *expect* statement at every point of the selectivity space by varying one or more conditions until the space is covered. Picassso [3] is a tool which is automatically sending these *explain* statements to the RDBMS. The optimizer of the RDBMS will then return the corresponding query plan for every explain statement which represents one point in the selectivity space. Picasso then plots every plan in a different color as demonstrated in figure 6. Using a query grid of 100 x 100 means that 10,000 *explain* statements are sent to the RDBMS. For our purposes, a grid of 100 x 100 is enough.

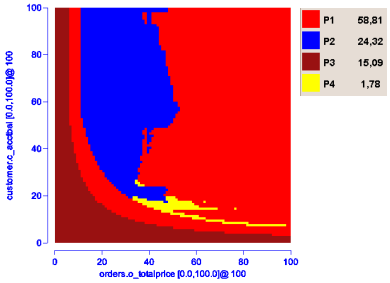

Fig. 5. Plan Diagram

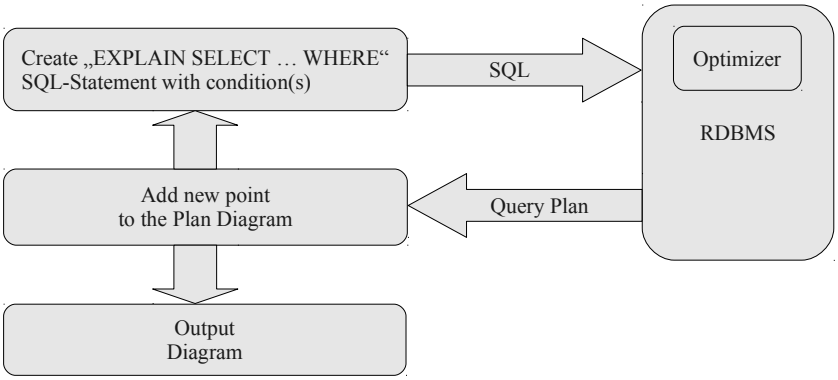

Fig. 6. Creation of Plan Diagrams

As a result, a Plan Diagram is created. The Plan Diagram shows which query plan will be used by the optimizer at which point of the space. This is demonstrated in figure 5.

2.2 Cost Diagrams

A Cost Diagram adds the "cost" dimension into the plan diagram. For a point in selectivity space, it shows the expected costs of the intended execution plan. Picasso is still coloring the surface of the 3-dimensional diagram so that one can see which plan is producing the costs as demonstrated in figure 7.

2.3 Reduced Plan Diagrams

The reduced plan diagram is generated by reducing the number of plans and the number of occurrences of plans without increasing estimated costs more

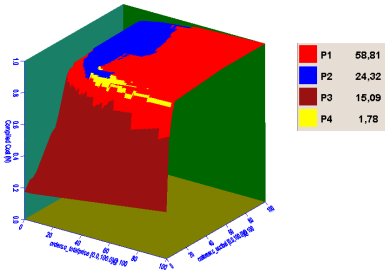

Fig. 7. Cost Diagram

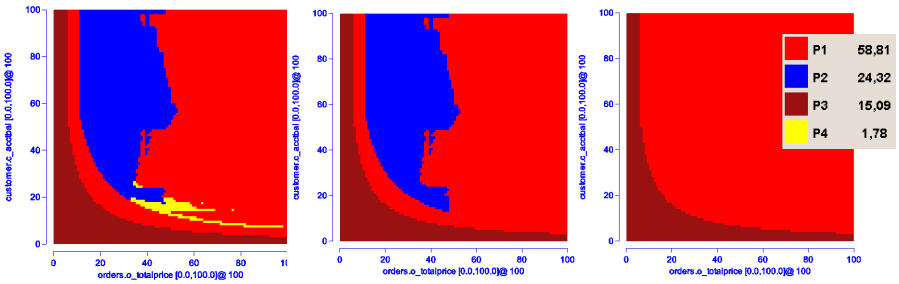

Fig. 8. Reduced Plan Diagram

than a plan optimality tolerance threshold (=maximum allowed cost increase) as demonstrated in figure 8.

According to the Cost Domination Principle [2], it is expected that the optimizer cost functions are monotonically non-decreasing with increasing base relation selectivities and result cardinalities [6]. As shown in figure 9, when passing the space in direction to less selectivity, the query plan of one point can be replaced by a query plan with more selectivity (we take the query plan with smallest costs in direction top or right) if that plan does not exceed the maximum allowed cost increase of the costs of the old plan. According to the Cost Domination Principle, the costs of the point with more selectivity represents an upper bound of the costs for the point with less selectivity. This procedure is very conservative, as we have less selectivity, but we calculate with the costs of the point with more selectivity.

Finally, an entire plan can be swallowed if and only if all its query points can be swallowed by either a single plan or a group of plans. The authors are ordering the plans in ascending order of size and then pass the list checking for the possibility of swallowing each plan.

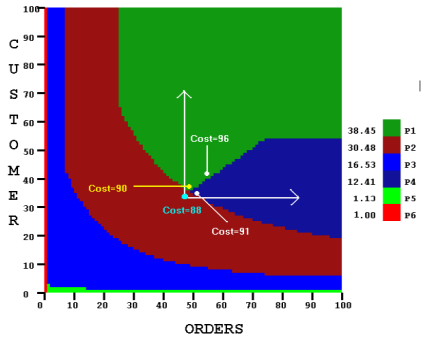

Fig. 9. Dominating Quadrant

2.4 Estimated Cost Diagram vs. Execution Cost Diagram

As already described, the estimated costs can be determined by sending an *explain select ...* statement to the RDBMS. The measurement of the costs is usually the number of rows that are expected to be affected. As the query itself is not executed, a 10x10 diagram can be created in 10s (using TPC-H query 9). The optimizer's decisions base on static estimations. That means that two neighbored points in the selectivity space with the same and also with different query plans should have nearly the same estimated costs, while increased selectivity should lead to more estimated costs. That is why the estimated cost diagram should always be monotonic.

To produce an execution cost diagram, Picasso will execute the select statement and measure the execution time. Therefore, the measurement of the costs is seconds and the creation time of an execution cost diagram is much higher (e.g. 250s for a 10x10 grid with TPC-H query 9). The execution cost diagram might not be monotonic, e.g. if the estimations were not precise.

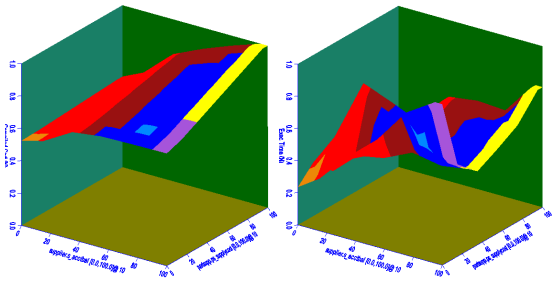

Fig. 10. Estimated Cost Diagram and Execution Cost Diagram

2.5 Examples of plan diagrams, cost diagrams, reduced diagrams

To get an idea how the diagrams can look like, we have a closer look on diagrams generated by Picasso using some TPC-H queries and a *postgresql* RDBMS.

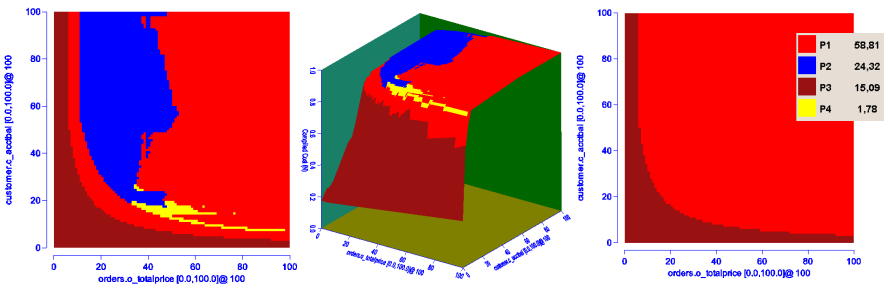

Fig. 11. TPC-H Query 7: Plan Diagram, Cost Diagram, Reduced Plan Diagram

The plan diagram of TPC-H query 7 in figure 11 uses four different query plans while the query plan with the smallest coverage covers only 1.78% of the space. There are several small segments and the diagram is not smooth. The corresponding cost diagram is monotonous but it does not look linear at all. In the reduced plan diagram the space is covered by only two query plans with an average cost increase of $\leq 0.72\%$ and a maximum cost increase of 6.73%.

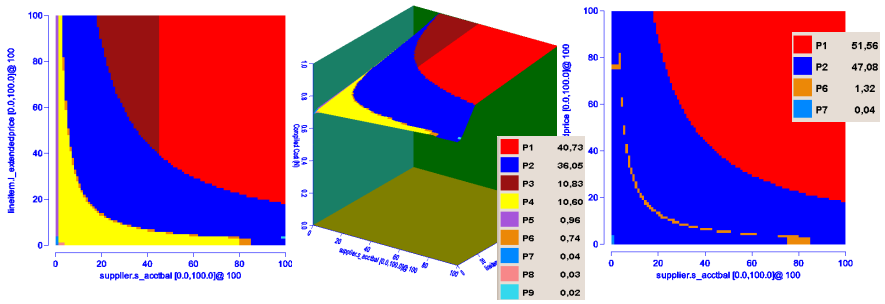

Fig. 12. TPC-H Query 8: Plan Diagram, Cost Diagram, Reduced Plan Diagram

The plan diagram of TPC-H query 8 in figure 12 uses nine different query plans while the query plan with the smallest coverage covers only 0.02% of the space. There are several small segments and the diagram is not smooth. The corresponding cost diagram is monotonous and it looks quite linear. In the

reduced plan diagram the space is covered by only four query plans with an average cost increase of $\leq 0.41\%$ and a maximum cost increase of 9.8%.

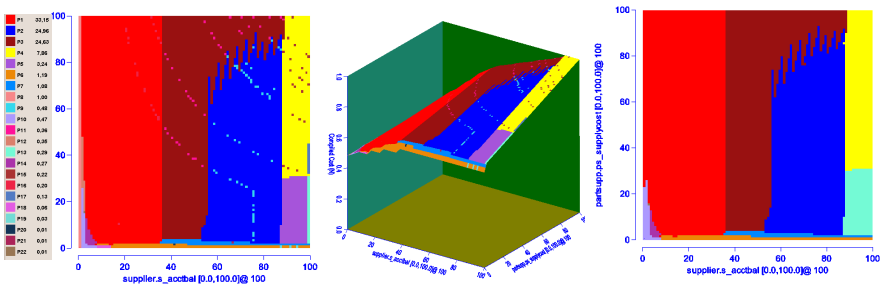

Fig. 13. TPC-H Query 9: Plan Diagram, Cost Diagram, Reduced Plan Diagram

The plan diagram of TPC-H query 9 in figure 13 uses 22 different query plans while the 14 query plans with the smallest coverage each cover less than 0.5% of the space. There are several small segments and the diagram is not smooth at all. The corresponding cost diagram is monotonous and it looks quite linear. In the reduced plan diagram the space is covered by only ten query plans with an average cost increase of $\leq 0.15\%$ and a maximum cost increase of 9.15%.

2.6 Comparison of Optimizers

TPC-H Query	#Plans	%plans for 80%
2	2	100.00%
5	8	37.50%
7	4	50.00%
8	9	33.33%
9	22	13.64%
10	9	22.22%
18	7	14.29%
21	4	25.00%
postgre	**8.13**	**37.00%**
OptA	**28.7**	**17.00%**
OptB	**24.5**	**23.00%**
OptC	**28.8**	**16.00%**

Table 1. Comparison of Optimizers

The commercial optimizers OptA, OptB, OptC use in average between 24.5 and 28.8 different query plans to cover the selectivity space for the mentioned TPC-H queries. In contrary, postgre only uses 8.13 plans in average to cover the selectivity space. The commercial optimizers need in average between 16% and 23% of the query plans to cover 80% of the selectivity space. In contrary, postgre needs in average 37% of the plans to cover 80% of the selectivity space. As a result there is to say, that the optimizer of postgre is much more stable than the tested commercial optimizers. However, although postgre's optimizer seems to make much less fine-grained plan choices, postgre's behavior is similar to the behavior of the tested commercial databases.

3 Linearity of Cost Diagrams

3.1 The Linearity of database operations

First, we want to consider join operations: If the join is done using a *nested loop join* [7], the costs are in $O(|R| \cdot |S|)$ for relations R, S. If $|R|$ is fixed and $|S|$ increased by factor a, then the costs increase linearly by factor a as long as we use the same plan. In case an index can be used for the join condition using an *indexed nested loop join* improves the situation. Now we run through relation R and search for corresponding rows in S using an index. The costs are then in $O(|R| \cdot \log |S|)$ which means that when increasing $|S|$ by factor a the costs increase by factor $\log a$. In case an index can be used for both attributes of a join condition the relations only have to be merged using a *merge-scan-join* [7] which is possible in $O(|R| + |S|)$. Increasing $|S|$ by factor a will increase costs by $\leq a$. A *hash-join* [7] can also be done in $O(|R| + |S|)$. Finally we can state that a set of join operators exists while these join operators have linear cost increase when the selectivity of one dimension is increased.

As the TPC-H queries also make use of *order by* clauses, we also have to consider the costs of sorting the result of a query. In some cases an index can be used for this task which would mean that sorting is possible in $O(n)$ when sorting n tuples. But in most cases, no index is available or the order was destroyed by other operations (e.g. hash join). In that case the result of a query can only be sorted in $O(n \log n)$ by using sort algorithms like *merge-sort*. However, we do not necessarily see this in cost diagrams: In case the expected result of the query is relatively small and fits into memory the sorting procedure could be considered relatively cheap compared to operations with lots of disk I/O like table scan etc. Additionally, more selectivity does not necessarily mean a bigger result, e.g. if in the query plan a *group by* is before a *sort by* operation as the last operation, the *sort by* operation will not become more expensive in case the number of tuples does not grow after the group by. Only the proceding operations might become more expensive. A *group by* is used in TPC-H query 7. However, if an increase of $|S|$ by factor a (with $a > 1$) increases the number of tuples in the result by factor a the corresponding cost for the sorting operation will be in $O(a|S| \log a|S|)$ which is not linear by definition.

3.2 Real world effects in Estimated Cost Diagrams

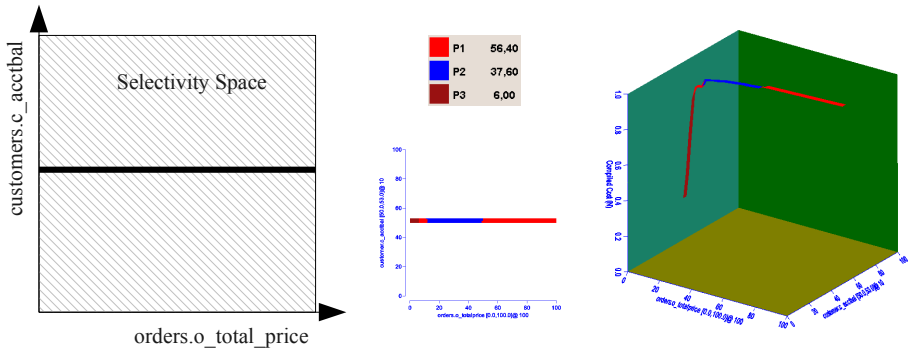

Fig. 14. TPC-H Query 7: Linearity in one dimension within one plan

When we consider figure 11 we could think that estimated costs were increasing more than linear when moving from point (0/0) to (8/8) of the selectivity space. The reason for that is that postgre is doing an indexed nested loop join when using plan P3 while $O(|R| \cdot \log |S|)$ can be more than linear in case *both* $|R|$ *and* $|S|$ are increased. Only if we consider the expected cost development in one dimension we get a linear increase of expected costs within one query plan as demonstrated in fig. 14. However, the different plans have different gradients. While P3 uses two nested loop joins, P1 and P2 use one hash join and one nested loop join which means that the gradient can be more flat and the estimated cost diagram even looks linear when increasing two dimensions. However, the plan P1 should look more stable and Plan P2 should not appear at all because Plan P1 is expected to be cheaper than P2 again when selectivity increases.

4 Relationship to Parametric Query Optimization (PQO)

The goal of *Parametric Query Optimization (PQO)* is to apriori identify the optimal set of plans for the entire relational selectivity space *at compile time*. At run time, the actual selectivity parameters can be used to identify the best plan. The expectation is that this is much faster than than optimizing the query from scratch [6]. As mentioned in [6], most of the literature is based on assuming cost functions that result in about the following:

1. *Plan Convexity*: If a plan P is optimal at point A and point B, then it is optimal at all points between the two points.
2. *Plan Uniqueness*: An optimal plan P appears at only one contiguous region in the entire space.

3. *Plan Homogeneity*: An optimal plan P is optimal within the entire region enclosed by its plan boundaries.

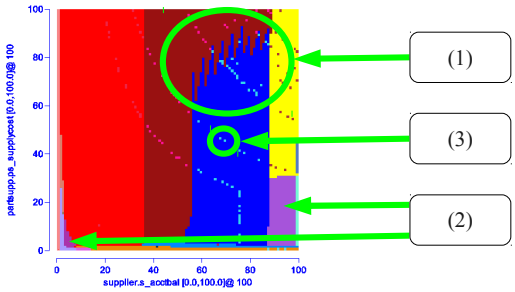

Fig. 15. Assumptions (1), (2), (3) do not hold true in practice.

However, [6] shows that all three assumptions do not hold true for three tested commercial optimizers. In figure 15 the same result is shown for postgre: No plan convexity is given for the blue plan (1). The plan uniqueness is not given for the violet plan, because it occurs twice (2). And the plan homogeneity is not given for the blue plan because within its plan boundaries there are some little dots (3).

[6] states that the gap between theory and practice concerning the assumptions is considerably narrowed if reduced plan diagrams are used for PQO. However, it is speculative, that all assumptions are completely fullfilled as the reduced plan diagram depends on the maximum allowed cost increase. E.g. figure 12 and figure 13 show that reduced plan diagrams do not necesarily correspond to the assumptions of PQO.

5 Conclusion

In this report we have analyzed the behavior of two-dimensional plan and cost diagrams produced by modern optimizers on queries based on the TPC-H benchmark. This report shows that many of the queries result in highly intricate diagrams with many different plans covering the space. Further was shown that typically 80 percent of the space are covered by 20-37 percent of the plans.

Next, we have analyzed when linearity of expected cost diagrams can be expected. In case selectivity is increased in the direction of one dimension, the expected costs increase linearly. Linearity is even given when increasing selectivity into two dimensions and a using a merge-scan-join or a hash join. When using different plans which should, at their boarder, have the same expected costs as their neighbors, the corresponding chart might not look linear as costs increase differently.

Finally, we have shown that the assumptions for PQO do not hold in practice. All tested optimizers had patterns that conflict to these assumptions. It would be better to use reduced plan diagrams instead for PQO because reduced plan diagrams are more stable and suitable to the assumptions for PQO.

6 Appendix

6.1 Testbed

All Diagrams have been produced in the following testbed environment (otherwise mentioned):

- Picasso 2.0
- TPC-H 2.9.0, default settings
- postgresql 8.4.2, default settings
- java-6-sun-1.6.0.15
- Ubuntu 9.10, Kernel 2.6.31-18-generic
- Pentium-M (32 bit), 1,86 GHz, 1GB RAM, 250 GB hard disk

6.2 Using TPC-H with postgresql

Installation and test of a postgresql RDBMS using ubuntu 9.10 32 bit:

```
$sudo apt-get install postgresql tofrodos
$sudo su postgres
$psql
psql (8.4.2)
[...]
postgres=# create database test;
CREATE DATABASE
postgres=# \c test
psql (8.4.2)
Sie sind jetzt verbunden mit der Datenbank test.
test=# create table newtable (field int);
CREATE TABLE
test=# \d
          Liste der Relationen
 Schema |   Name   |  Typ  | Eigentmer
--------+----------+-------+-----------
 public | newtable | table | postgres
(1 Zeile)
```

Installation of DBGEN and QGEN

```
tar -xvpzf tpch_2_9_0.tar.gz

$cp makefile.suite makefile
$vim makefile
CC      = gcc
# Current values for DATABASE are: INFORMIX, DB2, TDAT (Teradata)
#                                  SQLSERVER, SYBASE
# Current values for MACHINE are:  ATT, DOS, HP, IBM, ICL, MVS,
#                                  SGI, SUN, U2200, VMS, LINUX, WIN32
# Current values for WORKLOAD are:  TPCH
DATABASE= INFORMIX
MACHINE = LINUX
WORKLOAD = TPCH
```

Create the TPC-H data for the relations:

```
$./dbgen
$ls *.tbl
customer.tbl  lineitem.tbl  nation.tbl  orders.tbl  partsupp.tbl
part.tbl  region.tbl  supplier.tbl
```

generate queries (but they are not needed as they also come along with Picasso)

```
$mv queries/*.* .
$./qgen > queries.sql
```

load dbscheme and data:

```
$chmod  664 dss.ddl
$chmod  664 dss.ri
$chmod  664 *.sql
$sudo su postgres
$psql
psql (8.4.2)
[...]
postgres=# create database tpch;
CREATE DATABASE
postgres=# \c tpch
psql (8.4.2)
Sie sind jetzt verbunden mit der Datenbank tpch.
tpch=# \i dss.ddl
CREATE TABLE
CREATE TABLE

tpch=# \d
          Liste der Relationen
 Schema |   Name   |  Typ  | Eigentmer
--------+----------+-------+------------
```

```
 public | customer | table | postgres
 public | lineitem | table | postgres
 public | nation   | table | postgres
 public | orders   | table | postgres
 public | part     | table | postgres
 public | partsupp | table | postgres
 public | region   | table | postgres
 public | supplier | table | postgres
(8 Zeilen)
```

Import the TPC-H data:

```
copy customer from '/.../customer.tbl' with delimiter as '|';
```

postgresql requires that delimiter does not appear at the end of each line. Workaround to set a value at the end of each line: First, create a script *workaround.sh* and execute the script and the following sql statements for each relation:

```
#!/bin/bash
while read record
do
    echo "$record""0"
done

$./workaround.sh <customer.tlb >customer.tlbx

alter table customer add x int;
copy customer from '/.../customer.tblx' with delimiter as '|';
alter table customer drop x;
```

add indexes: edit dss.ri and outcomment this:

```
-- CONNECT TO TPCD;
```

remove all TPCD.

remove all constraint names. Example:

```
old:
ALTER TABLE NATION
ADD FOREIGN KEY NATION_FK1 (N_REGIONKEY) references REGION;
new:
ALTER TABLE NATION
ADD FOREIGN KEY (N_REGIONKEY) references REGION;

sudo su  postgres
\c tpch
\i dss.ri
-- create statistical summaries
vacuum analyze;
-- change password
alter user postgres with password '1234';
```

6.3 Setting up Picasso

PICASSO server and client: no spaces are allowed in current directory name!

```
tar -xvpzf picasso2.0.tgz
```

change rights and linewraps:

```
find picasso2.0 -type d -exec chmod 755 {} \;
find picasso2.0 -name "*.sh" -exec chmod 700 {} \;
find picasso2.0 -name "*.sh" -exec dos2unix {} \;

cd PicassoRun/
cd Unix
chmod 700 activatedb.sh

vim activatedb.sh
```

Insert as first line:

```
#!/bin/bash

$dos2unix activatedb.sh
$./activatedb.sh
```

edit compileServer.sh: Write classpath into one line.

```
$./compileServer.sh
```

Picasso Client:

```
$sudo apt-get install libjava3d-java
(Version 1.5.2 is installed)
$cd /usr/lib/jvm/java-6-sun/jre/lib/i386
$sudo ln -s /usr/lib/jni/libj3dcore-ogl.so
```

Remove all java exceptions by updating the client's libraries:

```
$cd Picasso/Libraries
$mv j3d* ./../.
$ln -s /usr/share/java/j3dutils.jar
$ln -s /usr/share/java/j3dcore.jar
$mv vecmath.jar ./../.
$ln -s /usr/share/java/vecmath.jar
$mv swing-layout-1.0.jar ./../.
$ln -s /usr/lib/jvm/java-6-sun-1.6.0.15/lib/visualvm/platform9/
   modules/ext/swing-layout-1.0.3.jar $swing-layout-1.0.jar

$cd PicassoRun/Unix
$./compileClient.sh
```

Run Picasso server and client:

```
$cd PicassoRun/Unix
```

Edit runServer.sh: Write classpath into one line:

```
$./runServer.sh
$./runClient.sh
Enter  localhost port 4444  ok
```

Setup database connection:

```
DBConnection->new
Connection Descriptor: pg
Machine: localhost
Engine: POSTGRES
Port: 5432
Database: tpch
Schema: public
User: postgres
Pasword: ****
```

References

1. Surajit Chaudhuri. An overview of query optimization in relational systems. *Proc. of the 17th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, 1998.
2. A. Hulgeri and S. Sudarshan. Anipqo: Almost non-intrusive parametric query optimization for nonlinear cost functions. *Proc. of 29th Intl. Conf. on Very Large Data Bases (VLDB)*, September 2003.
3. Indian Institute of Science. The Picasso Tool. `http://dsl.serc.iisc.ernet.in/projects/PICASSO`, 2005. [Online; accessed 22-February-2010].
4. Oracle. The Query Optimizer. `http://download.oracle.com/docs/cd/B19306_01/server.102/b14211/optimops.htm`, 2010. [Online; accessed 21-February-2010].
5. Naveen Reddy. Next generation relational query optimizers. *Master Thesis*, June 2003.
6. Naveen Reddy and Jayant R. Haritsa. Analyzing plan diagrams of database query optimizers. *Proc. of 31st Intl. Conf. on Very Large Data Bases (VLDB)*, September 2005.
7. Karsten Schmidt. Vorlesung Datenbankadministration. `http://wwwlgis.informatik.uni-kl.de/cms/fileadmin/users/kschmidt/db2-zert/07-AnfrageoptimierungI.pdf`, 2008. [Online; accessed 08-March-2010].
8. Janet L. Wiener, Harumi Kuno, and Goetz Graefe. *Performance Evaluation and Benchmarking*. Springer Berlin / Heidelberg, June 2003.