zTuned: Automated SQL Tuning through Trial and (Sometimes) Error

by

Herodotos Herodotou

Department of Computer Science
Duke University

Date: ____________________

Approved:

____
Shivnath Babu, Advisor

____
Kamesh Munagala

____
Vijayshankar Raman

Thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science in the Department of Computer Science
in the Graduate School of Duke University
2009

Abstract

SQL tuning—the attempt to improve a poorly-performing execution plan produced by the database query optimizer—is a critical aspect of database performance tuning. Ironically, as commercial databases strive to improve on the manageability front, SQL tuning is becoming more of a black art. It requires a high level of expertise in areas like (i) query optimization, run-time execution of query plan operators, configuration parameter settings, and other database internals; (ii) identification of missing indexes and other access structures; (iii) statistics maintained about the data; and (iv) characteristics of the underlying storage system. Since database systems, their workloads, and the data that they manage are not getting any simpler, database users and administrators often rely on intuition and trial and error for SQL tuning.

This work takes the position that the trial and error (or, experiment-driven) process of SQL tuning can be automated by the database system itself in an efficient manner; freeing the user or administrator from this burden in most cases. We formalize the problem of tuning a poorly-performing execution plan. We then describe the design of a prototype system that automates SQL tuning using an experiment-driven approach. Experiments are conducted with almost zero impact on the user-facing production database. The nontrivial challenge we addressed was to plan the best set of experiments to conduct, so that a satisfactory (new) plan can be found quickly.

Contents

Abstract **iii**

List of Tables **vi**

List of Figures **vii**

Acknowledgements **viii**

1 Introduction **1**

1.1 Optimizer Cost Models 1

1.2 SQL Tuning 3

1.3 Contributions 7

2 Related Work **10**

2.1 Query Hinting Mechanisms 10

2.2 Performance Monitoring and Diagnostics Infrastructures 12

2.3 Statistics Management and Execution Feedback 13

2.4 Automated Physical Database Design 14

3 Opening Up the Query Optimizer **15**

3.1 Explain Command 15

3.2 Explain_Plan Command 18

3.3 Explain_Plan Features 19

3.3.1 Implicit Operator Support 20

3.3.2 Cardinality Feature 20

4 Exploring the Execution Plan Space 22

4.1 Execution Plan Neighborhood 22

4.2 Exploring Within a Plan Neighborhood 25

4.3 Exploring Across Plan Neighborhoods 28

5 An Experiment-Driven Approach to SQL Tuning 30

5.1 Experiment-Driven Workflow 30

5.2 Experiment Design 32

5.3 Workbench for Experiments 33

6 zTuned: Automated SQL Tuning 37

6.1 Performing SQL Tuning Online 38

6.2 Performing SQL Tuning Offline 41

6.3 zTuned Settings 44

7 Experimental Evaluation 45

7.1 Environment 45

7.2 Effectiveness Evaluation 46

7.3 Evaluating Online Vs Offline Modes 49

7.4 Case Study with Skewed Data 52

8 Conclusion 54

A Appendix 55

A.1 Explain_Plan Design 55

A.2 zTuned Usage 58

Bibliography 59

List of Tables

7.1 Tuning Results for TPC-H Queries 50

7.2 Tuning Results in Online Vs Offline Modes 51

7.3 Tuning Results over Uniform Vs Skewed Data 53

List of Figures

1.1 Production Database in an Enterprise 3

1.2 A Sample Query used to Generate Alternative Execution Plans . . . 5

1.3 Alternative Plans for the Same Query 6

3.1 Explain Command for a Sample Query 16

3.2 Explain Output from Executing the Command in Figure 3.1 17

3.3 Explain_Plan Command with a Particular Plan for a Sample Query . 18

3.4 Explain_Plan Output from Executing the Command in Figure 3.3 . . 19

4.1 Plans Within and Across Neighborhoods 23

5.1 Experiment-Driven Workflow 31

5.2 Workbench for Experiments 34

6.1 Online Mode Implementation Workflow 39

6.2 Offline Mode Implementation Workflow 42

7.1 Variant of TPC-H Query 21 47

7.2 Execution Plans for Query in Figure 7.1 48

7.3 Running Times for the Optimal, Optimizer, and zTuned Plans 49

7.4 Tuning Improvement in Online Vs Offline Modes 51

7.5 Tuning Improvement over Uniform Vs Skewed Data 53

Acknowledgements

I would like to thank my advisor Shivnath Babu and the rest of my committee members for their guidance and advice during this work. I would also like to thank my collaborator Vamsidhar Thummala for his extensive work on implementing the new workbench described in Section 5.3 and for providing Figures 1.1 and 5.2.

1

Introduction

Databases are important building blocks in modern enterprises and part of a very elaborate framework that surrounds the production environment. As a result, the performance of Database Management System (DBMSs) has become critical to the success of the business applications that use them. One of the biggest factors in the performance of a DBMS is the speed of the SQL statements it runs. The *Query Optimizer* is responsible for ensuring the fast execution of queries in the system. For each query, the optimizer will (a) consider a number of different execution plans, (b) use a cost model to predict the execution time of each plan based on some data statistics and configuration parameters, and (c) use the plan with the minimum predicted execution time to run the query to completion.

1.1 Optimizer Cost Models

Optimizer cost models are usually very complex, and depend on cardinality estimations for complex expressions, database configuration parameters, specialized cost formulas for each operator in the physical query execution plan, and plan properties derived from the structure of the plan itself. There are several factors that cause currently used cost models to be inaccurate [14]:

- Cardinality estimations rely on data statistics that might be incorrect, stale or very coarse, usually in the form of histograms.

- Cardinalities for complex expressions are calculated based on statistical properties of the data with several simplifying (and usually invalid) assumptions like uniformity or independence assumptions.

- The cost formulas for each operator rely on specified constants, like the cost of a single disk I/O or the cost of processing a filter condition on a particular tuple. Such formulas might not be accurate or representative of the underlying hardware, especially with some of the new intelligent storage systems.

- The need to produce a unified cost number has lead to the creation of relationships among the costing parameters. For example, in the default settings of PostgreSQL database, the cost of CPU processing is set to be 0.01 times the cost of a single disk I/O. Again, such relationships might not be accurate or representative of the underlying hardware.

- Database configuration parameters affect the execution costs of query plans. For instance, the amount of buffer pool memory might affect the cost of a particular join since the number of tuples already cached in memory is unpredictable.

- The costing of a particular plan is done in isolation, without taking into consideration other queries executing in the database. Issues like lock contention, caching, and hotspots caused by concurrent query execution are not taken into consideration for costing purposes.

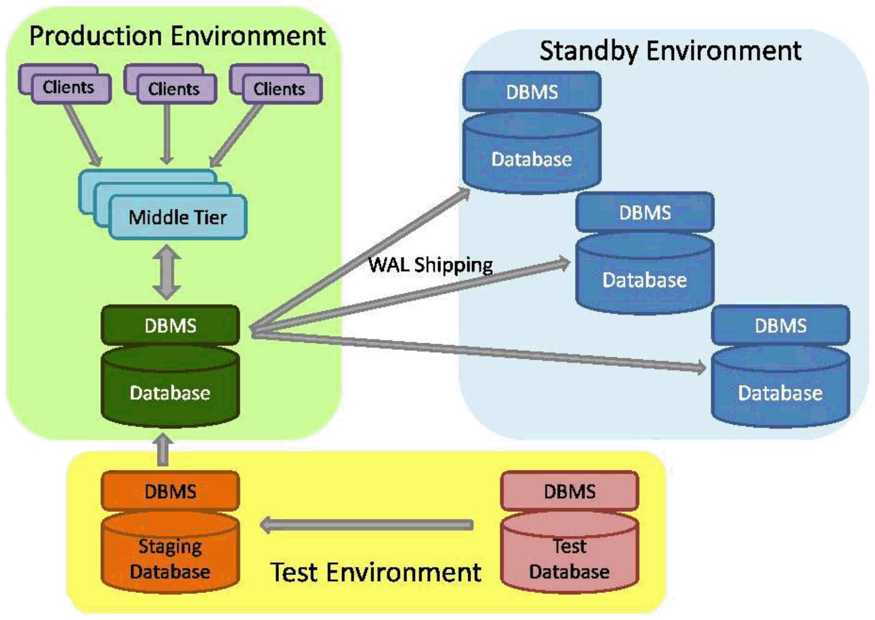


FIGURE 1.1: Production Database in an Enterprise

Inaccurate and incorrect costing may cause the query optimizer to select a suboptimal plan that, in many cases, can lead to a very poor performance for a particular query.

1.2 SQL Tuning

We saw that the rapid evolution of storage systems and complicated data patterns are causing estimates from traditional cost models to be increasingly inaccurate, leading to poorly performing execution plans [8]. Even when the system is well tuned, workloads and business needs change over time and the production database has to be kept in step. New optimizer statistics, configuration parameter changes, software upgrades and hardware changes are among a large number of factors that may cause a query optimizer to select different execution plans, perhaps with much worse performance than before. Hence, it is essential to perform *SQL tuning*, that is, to find better execution plans in order to make the offending queries execute faster.

SQL tuning has become a critical aspect of database administration and its success usually depends on the expertise of highly-skilled database administrators (*DBA*s) or time consuming trial-and-error steps. The typical database production framework can be seen in Figure 1.1. It consists of the production database, one or more standby databases for high availability, a test database used by DBAs and developers, and possibly a staging database for staging updates while they are moved from development to production. When called to tune a query, DBAs will use their experience, intuition, knowledge of the data being queried, several tips and tricks, or even guessing to complete the task. Initially, the DBA may collect some monitoring data on the production database in an attempt to diagnose the problem. However, data collection can increase the load on an already under-performing database, forcing the DBA to shift to the test database. The DBA's usual course of action would be:

1. Create a replica of the production environment on the test database. Mechanisms like workload capture and replay help here [17].

2. Get more insight into system behavior by performing experimental runs and collecting monitoring data on the test database. Multiple runs may be required because of system variability.

3. Form hypotheses regarding potential causes, and do further experiments to refine or confirm these hypotheses. For example, new indexes, statistics, or resources may be added, hints may be given to the query optimizer, configuration parameter settings may be changed, and so on.

4. When a fix is found, possibly after much trial and error, a careful validation is done to ensure that the fix will work on the production system. Finally, the fix is staged to the production system.

There are several tools available for a DBA to use in order to perform SQL tuning. Index advisors might recommend the creation of new indexes and configuration parameter tuning tools may find better parameter settings. However, in many cases, the DBA needs to manually experiment with different execution plans in an attempt to correct the mistakes of the query optimizer. A common mechanism found in most commercial databases used for this purpose is called *query hinting*. The DBA can use query hints to affect the choice of the execution plan from the optimizer. However, most query hinting mechanisms are very coarse and thus not useful with complex queries (more details on existing query hints can be found in Chapter 2).

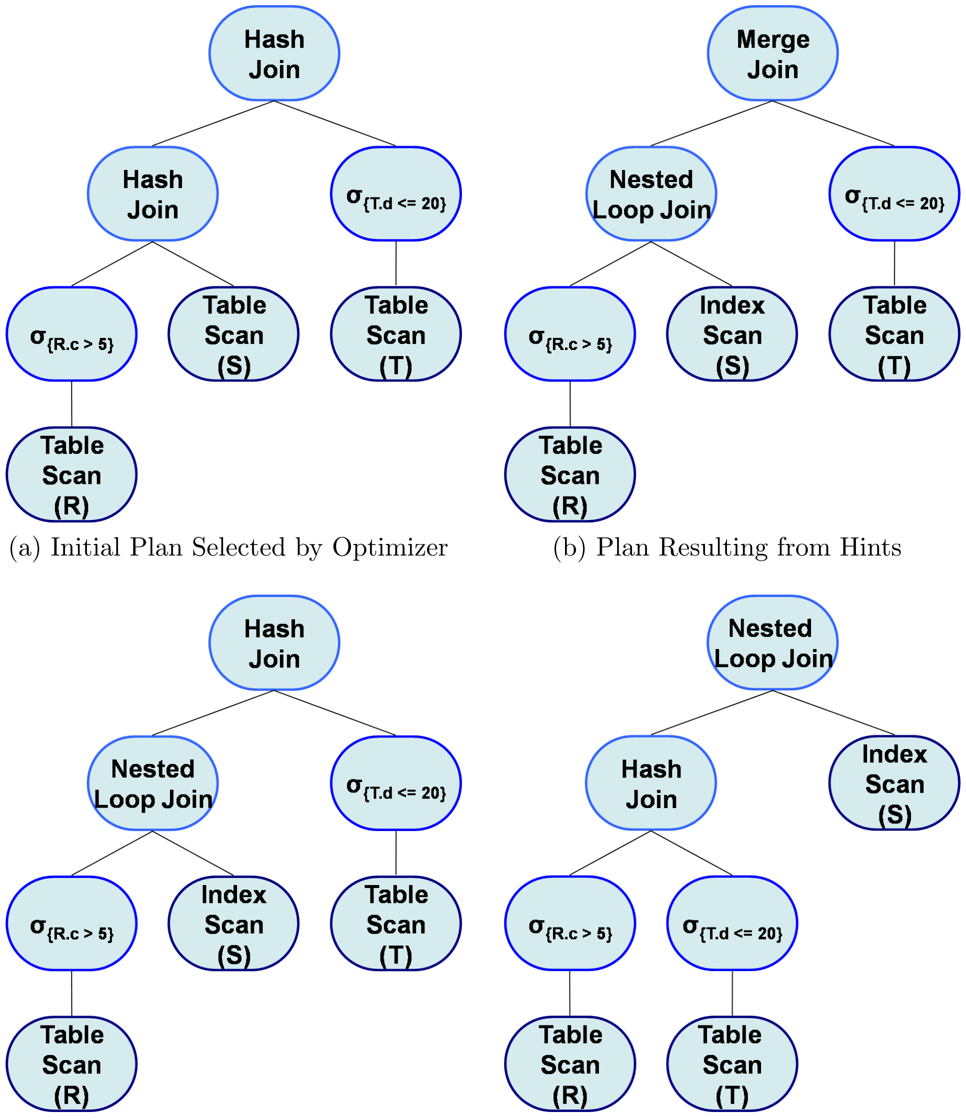
We will illustrate the use and weaknesses of hints with the following example. Consider the query shown in Figure 1.2.

```
SELECT a, b, c
FROM R, S, T
WHERE R.a = S.a AND R.b = T.b AND R.c > 5 AND T.d <= 20
```

FIGURE 1.2: A Sample Query used to Generate Alternative Execution Plans

Suppose the current execution plan selected by the query optimizer is the one shown in Figure 1.3a. Furthermore, suppose that the cardinality of $\sigma_{\{R.c>5\}}$ was severely overestimated and that the size of table S is large. Perhaps a better plan would involve an index nested loop join between tables R and S using an index on $S.a$, because of the small number of join tuples from R. In this case we would need to specify the use of a nested loop join over R and S.

However, most hinting mechanisms are very coarse and are hard to use in such cases. In particular, PostgreSQL allows a DBA to enable or disable the use of a join type. Here, we could disable the use of hash joins for this particular query.



(a) Initial Plan Selected by Optimizer (b) Plan Resulting from Hints

(c) Better Plan (d) Best Plan

FIGURE 1.3: Alternative Plans for the Same Query

The query optimizer could then choose the different plan seen in Figure 1.3b. The plan contains the nested loop join over R and S, but it also changed the upper hash join to a merge join, since the use of hash joins was disabled. The use of a merge join has an additional overhead from sorting the tuples from T that could overshadow the potential benefits from the nested loop join. In this case, the plan shown in Figure 1.3c would be a better choice, but it is almost impossible to force the optimizer to select it using hints. For this purpose, we introduced a new command in PostgreSQL, called *Explain_Plan* that would allow the DBA to fully specify this particular plan.

Despite advances in the particular tools and mechanisms that are available to the DBA, the overall trial-and-error process is very labor intensive, expensive, and requires an extensive knowledge of the database internals. In fact, DBA surveys estimate that this process consumes more than 50% of a DBA's day-to-day work time [16]. Clearly, there is a lot to be gained from automating the generation of alternative execution plans and performing SQL tuning in general.

1.3 Contributions

Our main contributions are the following:

1. We introduced a new command in PostgreSQL, called *Explain_Plan*. The Explain_Plan command allows for the full specification of a query execution plan to be costed and executed for a particular query. This command is a strong complement to the hinting mechanism of PostgreSQL and we believe that all commercial databases should provide this functionality.

2. We formulated the problem of SQL tuning using an experiment-driven approach for exploring the space of execution plans.

3. We developed an automated algorithm for planning experiments aimed at improving a poorly-running plan that is currently being used for a particular query. Each experiment in this context includes generating an alternative execution plan and either costing it or running it. The experiments collect cardinality statistics and execution costs in order to effectively search the large space of execution plans and reach to a good plan.

4. We developed and evaluated *zTuned*, a system that formalizes and automates the process of SQL tuning using the proposed experiment-driven approach.

Continuing the example introduced above, consider the case where the predicates for $\sigma_{\{R.c>5\}}$ and $\sigma_{\{T.d<=20\}}$ are negatively correlated, thereby resulting in a much smaller cardinality for $R \bowtie T$ than estimated by the optimizer. Some well-designed experiments can find this correlation and propose the plan shown in Figure 1.3d. This plan can be much better since the small cardinality for $R \bowtie T$ allows efficient index nested loop join with table S.

zTuned has the ability to generate different yet equivalent execution plans using a set of transformations. This plan generation occurs independently from the query optimizer allowing for a larger and perhaps different exploration of the plan space. In addition, since zTuned works outside the optimizer, it can potentially be used with any database that uses a cost-based optimizer and supports the specification of full execution plans.

zTuned can also be used in two different modes, *"online"* and *"offline"*. Online means that it can be used directly on the production system with small overhead. In the online mode, zTuned explores a smaller part of the plan space around the plan provided by the optimizer, in an incremental fashion. In the offline mode, zTuned has the ability to explore a much larger space using some experimental runs. Even though it might take more time to find a better plan, it will take considerably less time compared to the manual experimentation used today by DBAs.

The use of experiments is necessary in order to create a more holistic approach towards SQL tuning. When a performance problem arises, there is a large arsenal of potential *fixes* to choose amongst—updating statistics and cost models, changing current operator implementation, changing the entire plan shape, changing the physical design like indexes, or changing settings of configuration parameters. It is very challenging to select the right set of experiments to conduct that will lead to the best fix, especially if the fix is some combination of the above fixes. Our goal is to minimize the total cost of experiments that it takes to go from a current poor plan to a good plan. The experiments will take place on a new workbench, which takes advantage of underutilized resources like the standby system that most production environments have. Our approach imposes no overhead on the production system when run in offline mode and it guarantees that the recovery time for the standby system (in case the primary system fails) will not be affected.

Chapter 2 discusses related work on self-tuning database systems and explores the state of the art tools used today by DBAs to tune the performance of a database. Chapter 3 describes the mechanisms used to obtain the current execution plan from the query optimizer as well as to execute alternative plans. Chapter 4 introduces some new definitions and mechanisms that are essential in the efficient exploration of the plan space, and Chapter 5 provides an overview of how they can be utilized in order to select which experiments to conduct. It also introduces the new workbench for conducting the experiments when zTuned is used in the offline mode. All mechanisms and algorithms are used by our new automated SQL tuning tool, which is described in detail in Chapter 6. We evaluate the performance of zTuned in Chapter 7 and we conclude with Chapter 8.

2

Related Work

There has been an extensive work on providing DBAs the tools to correctly and efficiently tune a database system. Tuning database systems is a wide area of research that involves problems such as query hinting mechanisms, performance monitoring and diagnostics infrastructures, statistics management, and automating physical database design.

2.1 Query Hinting Mechanisms

Query hinting is a common mechanism found in many database systems that allow a DBA to influence the choice of a query execution plan from the optimizer. However, this support comes in varying degrees and is often scoped globally. PostgreSQL has a very coarse hinting mechanism [26] that effectively prevents an experienced DBA from fine-tuning a poorly performing query. All hints are in terms of enabling or disabling particular operators. For example, setting the parameter *enable_indexscan* to false, would completely disable index scans being used in any query plans for the current connection. The same can be done for all join operators. Hence, there is no way to force one join to be a hash join without at the same time forcing all other joins to be hash joins as well. The introduction of the Explain_Plan command solves this problem since a DBA can fully specify the execution plan for a given query. Even though this approach can be tedious for large queries, we note that a simple user interface (which is part of future work) would allow a DBA to simply modify any operator in a given execution plan.

The hinting mechanism in Microsoft SQL Server is more flexible since it includes support for specifying an access path for a table (i.e., force an index scan on a table) and also the ability to force a join order for all tables that appear in the query [22]. However, it has the same problem with join specification as PostgreSQL. These hints are again globally scoped and thus cannot constrain execution sub-plans. Moreover, SQL Server supports the ability to specify a full execution plan similar to Explain_Plan, but uses an XML interface [22]. Another unique feature of Explain_Plan that differentiates it from SQL Server's counterpart is the ability to specify the cardinality of a particular operator used for costing.

Oracle hints [24] can specify the first join to be used in the execution plan as well as the join operator to be used for a particular pair of tables. Even though Oracle hints can constraint particular joins, they do not support any finer scopes over an arbitrary set of tables. However, Oracle does support the injection of cardinality estimates in their hinting mechanism. IBM DB2 offer a different approach to query hinting through the creation of a particular table called PLAN_TABLE [19]. Hints are given to the query optimizer in the form of SQL queries over the PLAN_TABLE and can be used at the scope of a sub-query. Optimization hints however cannot force or undo query transformations, such as sub-query transformation to join or materialization or merge of a view or table expression.

Finally, it is worth noting some very recent work on query hinting called *Power Hints (PHints)* [23]. PHints provide a generic framework that supports hints with fine granularity and represent constraints over the plan search space. For example, it is possible to specify the use of two particular joins and a particular join order over three tables. With enough power hints a DBA could specify a plan completely, even though it is uncertain how intuitive or easy it would be to do so. This is the main advantage of Explain_Plan over Power Hints. Another difference from our work is that PHints are constrained to the search space of the query optimizer (assuming the database supports the interface of exporting the space of plans), whereas zTuned explores the space of plans on its own.

2.2 Performance Monitoring and Diagnostics Infrastructures

The support for database monitoring and diagnostics has increased over the last years. Several tools like HP OpenView [27] and IBM Tivoli [21] provide performance monitoring, whereas tools like DB2 SQL Performance Analyzer [18] and SQL Server Performance Tuning [2] provide extensive analysis of SQL queries without executing them. In [11] the authors proposed the "SQL Continuous Monitoring" infrastructure that builds on the server-side with the goal of supporting monitoring queries.

References [15] and [16] describe the new automated tools introduced with *Oracle Database 10g* and *11g*. These tools enable the database to monitor and diagnose itself on an ongoing basis, and alert the DBA when it finds any problems. The *Automatic Tuning Optimizer* is a new mode of the optimizer that is specifically used during designated maintenance sessions. When run in this mode, the optimizer generates additional information that can be used at run-time to speed performance. In particular, the tuning mode subsumes the behavior of the normal mode and has extended functionality, but comes with an additional overhead on the production system. The database creates SQL profiles for some queries that are identified as performing poorly. SQL profiles are metadata objects with configuration checks and better cardinality estimates produced using data samples. Based on predefined rules, performance tuning is invoked by the *Automatic Diagnostic Monitor*, which is able to analyze information in its performance data-warehouse.

The above mentioned tools are designed to facilitate the DBA in tuning and improving the performance of a database system. Our goal is the same but our approach is unique in the sense that we use an experiment-driven approach that will quantify and provide guarantees for the performance improvements. Moreover, their techniques impose an overhead on the production system, unlike our approach that does not affect it.

2.3 Statistics Management and Execution Feedback

Query execution feedback is a technique used in [1, 13] to improve the quality of plans by correcting cardinality estimation errors made by the query optimizer. Query feedback mainly consists of recording the number of rows produced by each operator during the execution of a particular query, and then relaying the new information back to the query optimizer. LEO's approach [29] extended and generalized this work to provide a general mechanism for repairing incorrect statistics and cardinality estimates of a query execution plan. The *Pay-as-you-go* framework [12] proposed more proactive monitoring mechanisms and plan modification techniques for gathering the necessary additional cardinality information from a given query execution plan. The central idea is to use some simple mechanisms to gather additional cardinality information that might be useful to the query optimizer in selecting a better execution plan in the future.

However, all of the above mentioned approaches are limited by the need to maintain a very low overhead on the production system. Furthermore, the cardinality of any relevant expression for the query that does not correspond to an operator in the current (or slightly modified) plan cannot be obtained through the above mechanisms. Our approach does not suffer from these limitations since all the experiments responsible for improving and correcting statistics will occur on the proposed workbench (on the backup system). Moreover, well-designed experiments can target the collection of any desired statistics, and thus explore very different parts of the execution plan space.

Another related research direction focuses on dynamic adjustments of query plans during their execution. Based on statistics collected during the query execution, a new operator introduced in [20] decides whether to continue or stop the execution and re-optimize the remaining plan. *RIO* [8] proposes proactive re-optimization techniques. RIO uses intervals of uncertainty to pick execution plans that are robust to deviations of the estimated values or to defer the choice of execution plan until the uncertainty in estimates can be resolved. The common feature of the above approaches is the possibility for a change in the execution plan while it is executing. On the contrary, our system will suggest a change in the execution plan that will affect future similar queries.

2.4 Automated Physical Database Design

References [4, 9, 10] explore several solutions to the problem of automating physical database design, primarily focusing on identifying the right set of indexes for a particular workload. References [3, 5, 25] tackled the related issues of view materialization, data partitioning, and table layouts for relational databases. Furthermore, several commercial tools were created to aid the database administrators in database tuning like Microsoft's *Database Engine Tuning Advisor* (DTA) [2], *DB2 Design Advisor* [30] and Oracle's *SQL Access Advisor* [15]. These tools are capable of providing useful recommendations regarding the physical design, that complement our approach to SQL tuning.

3

Opening Up the Query Optimizer

There are several tools and database mechanisms available for a DBA to leverage in order to perform SQL tuning. Index advisors might recommend the creation of new indexes and configuration parameter tuning tools may find better parameter settings. However, in many cases, DBAs need to manually experiment with different execution plans, in an attempt to correct the mistakes of the query optimizer. In order to search for a different plan, they first need to find out which plan the query optimizer selected along with the cardinality estimates that led to that selection.

3.1 Explain Command

PostgreSQL has a very useful command, called *Explain*, that can be used to display the generated execution plan. The execution plan shows how the tables referenced by the query will be scanned (by plain sequential scan, index scan, etc.) and if multiple tables are referenced, what join operators will be used to bring together the required rows from each input table. It also displays other operators like aggregates and sorting if needed.

In addition, it is crucial for the DBAs to gain some more insight into the reasons why that particular selection was made. The output from the Explain command is once again very useful as it contains the following information for each operator:

- Estimated startup time[1] before the first row can be returned.
- Estimated total time to return all rows.
- Estimated number of rows that this operator will produce.
- Estimated number of bytes for each row.

The Explain command offers an additional option, called *analyze*, that causes the statement to be actually executed, not only planned. The output then contains the following information:

- Actual startup time (in milliseconds) before the first row can be returned.
- Actual total time (in milliseconds) to return all rows.
- Actual number of rows that this operator will produce.
- Number of times this operator was used.

This information is particularly useful for seeing whether the optimizer's estimates are close to reality. Let us consider the Explain command shown in Figure 3.1.

```
EXPLAIN ANALYZE
SELECT  c_name, o_totalprice, c_acctbal
FROM    customer, orders
WHERE   c_custkey = o_custkey
    AND o_orderdate >= date '1996-01-01'
    AND o_orderdate < date '1996-01-10'
    AND o_totalprice < 1500
    AND c_acctbal < 1500;
```

Figure 3.1: Explain Command for a Sample Query

[1] Time is measured in terms of disk page fetches

The query shown in Figure 3.1 looks for all customers who placed an order during the first ten days of year 1996, and their account balance as well as the order price were less than \$1,500. This query was executed over a *TPC-H*[2] database with a scale factor of 1GB. The Explain output can be seen in Figure 3.2.

```
                                QUERY PLAN
---------------------------------------------------------------------
 Hash Join  (cost=5667.09..58413.08 rows=162 width=29)
            (actual time=221.371..973.714 rows=48 loops=1)
   Hash Cond: (orders.o_custkey = customer.c_custkey)
   ->  Seq Scan on orders  (cost=0.00..52725.00 rows=1550 width=14)
                     (actual time=1.230..763.017 rows=457 loops=1)
         Filter: ((o_orderdate >= '1996-01-01'::date) AND
                  (o_orderdate < '1996-01-10'::date) AND
                  (o_totalprice < 1500::numeric))
   ->  Hash  (cost=5471.00..5471.00 rows=15687 width=23)
             (actual time=208.153..208.153 rows=15728 loops=1)
         ->  Seq Scan on customer
                 (cost=0.00..5471.00 rows=15687 width=23)
                 (actual time=135.745..189.854 rows=15728 loops=1)
               Filter: (c_acctbal < 1500::numeric)
 Total runtime: 975.405 ms
```

FIGURE 3.2: Explain Output from Executing the Command in Figure 3.1

The query optimizer selected to perform a hash join over two sequential scan for the two tables. It is interesting to note that the number of output tuples from the sequential scan on table *orders* was overestimated by the optimizer, probably because of the complex filter condition. Perhaps, a better choice would have been to use an index nested loop join, using the index on the primary key of the *customer* table. Trying to use the existing hinting mechanism of PostgreSQL turns out to be quite troublesome. Initially, we set the flag *enable_hashjoin* to false. In this case, the optimizer selected to use a merge join, which has a worse running time from the original plan. We then set the flag *enable_mergejoin* to false. Then, the optimizer

[2] TPC Benchmark^{TM} H (TPC-H) is a decision support benchmark comprised by queries that simulate business intelligence queries.

selected to use a nested loop join but it used a bitmap heap scan over customer, which still did not lead to a better result.

3.2 Explain_Plan Command

This inadequate hinting mechanism has lead to the development of a new command in PostgreSQL, called *Explain_Plan*. The Explain_Plan command allows for the full specification of a query execution plan that can be costed and executed for a particular query. It is similar in nature with the Explain command but the input includes a string representation of the desired physical plan. It can be used for both costing a particular plan and executing it using the *analyze* option. The full syntax of the command is described in Appendix A.1.

```
EXPLAIN_PLAN ANALYZE
"{ nestjoin
    :outerjoinpath {
        seqscan
          :parent orders
          :filter orders.o_orderdate >= '1996-01-01' AND
                  orders.o_orderdate < '1996-01-10' AND
                  orders.o_totalprice < 1500 }
    :innerjoinpath {
        indexscan
          :parent customer
          :using  customer_pkey
          :filter customer.c_acctbal < 1500
          :indexCondition customer.c_custkey = orders.o_custkey }
}"
FOR
SELECT c_name, o_totalprice, c_acctbal
FROM   customer, orders
WHERE  c_custkey = o_custkey
   AND o_orderdate >= date '1996-01-01'
   AND o_orderdate < date '1996-01-10'
   AND o_totalprice < 1500
   AND c_acctbal < 1500;
```

FIGURE 3.3: Explain_Plan Command with a Particular Plan for a Sample Query

The command shown in Figure 3.3 shows how we can specify the full execution path we wanted to test, for the same query from Figure 3.1.

The output style of the Explain_Plan command is identical to the output from an Explain command. The output of the command in Figure 3.3 can be seen in Figure 3.4. It contains all the estimated and actual information per operator. It is interesting to note that the estimated total cost of the nested loop join is higher that the estimated total cost of the hash join seen in Figure 3.2. This is due to the incorrect cardinality estimation for the sequential scan over the *orders* table. However, the actual running time of the new plan is about 20% better.

```
                              QUERY PLAN
----------------------------------------------------------------------
 Nested Loop  (cost=0.00..61137.12 rows=162 width=29)
              (actual time=90.133..756.746 rows=48 loops=1)
   ->  Seq Scan on orders  (cost=0.00..52725.00 rows=1550 width=14)
                     (actual time=1.293..759.017 rows=457 loops=1)
         Filter: ((o_orderdate >= '1996-01-01'::date) AND
                  (o_orderdate < '1996-01-10'::date) AND
                  (o_totalprice < 1500::numeric))
   ->  Index Scan using customer_pkey on customer
                      (cost=0.00..6.03 rows=1 width=23)
                      (actual time=0.531..0.531 rows=0 loops=457)
         Index Cond: (customer.c_custkey = orders.o_custkey)
         Filter: (customer.c_acctbal < 1500::numeric)
 Total runtime: 759.557 ms
```

FIGURE 3.4: Explain_Plan Output from Executing the Command in Figure 3.3

3.3 Explain_Plan Features

We have seen how the Explain_Plan command can be used to fully specify the execution plan for a particular query. There are also two other important features in Explain_Plan, (a) the *implicit operator support* for certain operators and (b) the so-called *cardinality feature*.

3.3.1 Implicit Operator Support

When a query is send to the database for execution, the query optimizer is responsible to searching the space of possible execution plans to find the optimal one. In PostgreSQL, the query optimizer operates in two phases. During the first phase, it only considers the "Select-Project-Join" part of the query and builds the optimal execution plan bottom up. During the second phase, the optimizer takes into consideration any *group by* or *order by* clauses (among with more advance SQL options), and adds those operators to the plan as needed.

Following the same spirit, the input execution plan in the Explain_Plan command consists only of joins, scans and materialized nodes. This is the "important part" of the query that DBAs are mostly interested in modifying, and the part that usually matters the most. However, the Explain_Plan command can be used with queries that are more complex from simple Select-Project-Join queries. There is an implicit support for all aggregates, sort clauses, group clauses, having clauses etc. Implicit here means that a DBA does not need to specify these operators in the input physical plan. Instead, they are inferred from the provided query, and optimized during the second optimization phase.

3.3.2 Cardinality Feature

A unique feature of Explain_Plan is the ability to specify the expected number of tuples produced from each operator. For example, if the DBA knows that a join will produce 42 tuples based on execution history or data properties, she can specify it in the input plan. This information will be used instead of the optimizer's estimations for costing purposes. This feature is extremely useful in many situations and allows for a much deeper analysis of the database internals.

For instance, suppose the DBA has several plans in mind to experiment with. Without the cardinality feature, she would have to use the *analyze* option for each plan, which might be very time consuming for large queries. Instead, she can use Explain_Plan with the actual cardinalities per node and compare their estimated costs based on the database cost engine. Hence, she can quickly prune the space of plans she was initially considering and perhaps only use the *analyze* option on the most promising one. A similar technique is used by zTuned to effectively prune the search space.

With the cardinality feature we can additionally test and validate a query optimizer. We can now ask questions like: if the query optimizer had perfect information about the flow of tuples in a logical plan, would it still select the same physical plan? We could explore a chosen space of possible plans, cost each plan in the space and, find out the optimal plan in the space based on the current cost models. We could then compare this plan with the plan selected by the query optimizer (with likely invalid statistics and assumptions).

4

Exploring the Execution Plan Space

When a DBA attempts to correct the mistakes of the query optimizer, she needs to experiment with different execution plans. We have already seen that using hints to force the optimizer to use a specific plan is very difficult. The Explain_Plan command can be used to overcome this difficulty since it allows for the full specification of a plan to be costed and executed for a particular query. However, generating alternative plans manually and then using Explain_Plan to execute them is a tedious process. It is thus crucial to automate the process of generating alternative plans that are, of course, valid for the offending query.

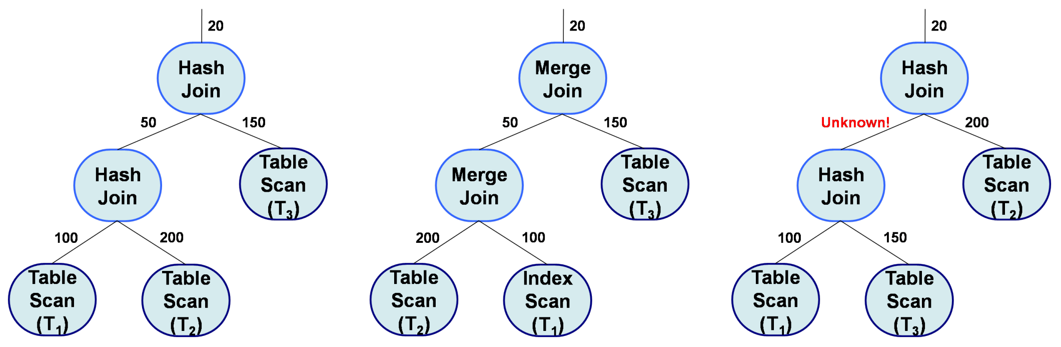
4.1 Execution Plan Neighborhood

In order to generate plans, we need to explore the execution plan space. We have developed algorithms to traverse the plan space in a structured and systematic way, while maximizing the use of all currently available information. The central concepts in our algorithms rely on the definitions of the *cardinality set* for an execution plan and the *plan neighborhood.*

Definition 4.1.1. A cardinality set for an execution plan is the set of cardinality values that are needed for costing that particular plan.

Consider the execution plan shown in Figure 4.1a. The numbers above each operator are the number of tuples produced from each operator i.e. the cardinality for that operator. These values are used by the database cost models to calculate the estimated cost for each operator and thus the entire plan. For example, in order to calculate the cost of a hash join, we need the cardinalities of the two child operators.

Definition 4.1.2. An Execution Plan Neighborhood is the set of all plans that are associated with the same cardinality set.



(a) Original Plan (b) Plan in the Same Neighborhood (c) Plan in a Different Neighborhood

FIGURE 4.1: Plans Within and Across Neighborhoods

In more practical terms, two plans belong in the same neighborhood if when we know the cardinalities for all operators in the first plan, we can induce the cardinalities for all operators in the second plan.

Consider the plans shown in Figure 4.1 and assume they are all valid for the same query. The first two plans belong to the same neighborhood since if we had the cardinalities from the plan in Figure 4.1a, we would know exactly what the cardinalities are for the plan in Figure 4.1b. Even though the two joins are different, the number of output tuples they will produce are going to be the same. The same is true when changing a table scan to an index scan and vice versa. Also note that the tables T_1 and T_2 are joined in a different order. Again, a join order change under a single join does not affect the number of tuples produced by the scans or the join itself.

Comparing the two plans from Figures 4.1a and 4.1c we see that the only change is the order change between tables T_2 and T_3. However, this causes plan 4.1c to belong to a different neighborhood. Given the cardinality estimates from plan 4.1a, we can induce the cardinalities for all the scans and the final join, but we cannot draw any conclusions about the cardinality of the join over tables T_1 and T_3.

The above definition of a plan neighborhood leads to two very interesting and useful properties.

1. *Cardinality Mapping* - Given two plans in the same neighborhood, there exists a one-to-one mapping between the cardinalities of each operator in the two plans. This property is a direct consequence of the neighborhood definition. Since we can use all cardinalities from one plan to induce the cardinalities of the other plan, there exists an appropriate mapping for this induction.

2. *Transitivity* - Suppose plan P_1 and plan P_2 belong to the same neighborhood, and suppose plan P_2 and plan P_3 also belong to the same neighborhood. Then, plan P_1 and plan P_3 must belong to the same neighborhood as well. By the previous property, there exists a one-to-one mapping between the cardinalities in the plans P_1 and P_2, and a one-to-one mapping between the cardinalities in the plans P_2 and P_3. Therefore, there must exist a one-to-one mapping between the cardinalities in the plans P_1 and P_3, i.e. plans P_1 and P_3 belong to the same neighborhood.

Lemma 4.1.3. *The space of all possible and equivalent plans can be partitioned into disjoint neighborhoods.*

Proof. First, we will provide a proof by contradiction that any two different neighborhoods are disjoint. Suppose that plan P_1 belongs to the neighborhood N_1 and plan P_2 belongs to a different neighborhood N_2. Also suppose that N_1 and N_2 are not disjoint and let plan P_3 be a common plan in the two neighborhoods. Then, by the transitivity property we conclude that plans P_1 and P_2 must belong to the same neighborhood. Therefore, N_1 and N_2 are the same neighborhood. However, this is a contradiction since we assumed that N_1 and N_2 were different. Therefore, any two different neighborhoods must be disjoint. Second, all possible plans must belong to a neighborhood. Even a single plan can form its own neighborhood since it can utilize its own set of cardinalities for all its operators. Since any two neighborhoods are disjoint and all plans must belong in some neighborhood, we conclude that the space of all possible and equivalent plans can be partitioned into disjoint neighborhoods. $\square$

4.2 Exploring Within a Plan Neighborhood

Instead of exploring the entire plan space, let us first consider the exploration of a single plan neighborhood. Exploration in this context is equivalent to the generation of all valid plans within a particular neighborhood. Hence, given a particular execution plan P_0, we can generate a set of different plans that belong in the same neighborhood as P_0. Note that based on our definition for a neighborhood, all generated plans must be equivalent to P_0, that is, they must be valid plans that produce the same actual output as P_0.

Let us return to Figure 4.1 and the previous discussion we had concerning these sample plans. We observed that changing a sequential scan to an index scan does not affect the number of output tuples. Of course, since we require that the generated plans are equivalent, any filter condition imposed to the sequential scan, is also transfered to the index scan. We also observed that changing a particular join from a hash join to a merge join, or changing the join order under the same join also does not affect the number of output tuples for that particular join.

We formalize the above observations into a set of transformation rules, which we call *intra-transformations*[1].

Definition 4.2.1. Intra-transformations are operator transformations that can be applied to a single node in the operator tree to generate a different execution plan within the same neighborhood.

All intra-transformations we are considering fall into the following broad categories:

1. *Scan Operator Transformations* - Transform a particular scan operator to a different one. For example, transform a sequential scan into an index scan.

2. *Join Operator Transformations* - Transform a particular join operator to a different one. For example, transform a hash join into a merge join.

3. *Single Join Order Transformations* - Swap the order of the outer and inner sub-plans of a particular join. For example, transform $A \bowtie B$ into $B \bowtie A$, where A and B represent any sub-plan.

It is important to note that some transformations might not be possible on a given plan. For example, a sequential join cannot be transformed into an index join unless there exists an index over that table. Also, some transformation might involve additional changes in order to preserve the correctness of the new plan. For instance, a merge join requires that the two sub-plans produce sorted output. If

[1] The prefix "intra" means "within" and in this context it refers to plans generated "within a particular neighborhood".

they don't, then a transformation to a merge join must add sort operators above the sub-plans. Certainly, such additions do not affect the number of tuples produced by the underlying operators and hence, the transformation still produces a plan in the same neighborhood.

Intra-transformations also have a set of interesting properties.

1. *Transitivity* - Suppose there exist a set of intra-transformations S_1 to transform plan P_1 to P_2 and another set S_2 to transform P_2 to P_3. Then, there exists another set of intra-transformations S_3 (for instance $S_1 \cup S_2$) to transform P_1 to P_3.

2. *Reversibility* - All intra-transformations are reversible. Suppose we have a plan P_1 and we perform a particular intra-transformation to produce plan P_2. Then, we can perform another transformation on P_2 to produce back P_1. Reversibility follows from the fact that any operator can be transformed to an equivalent operator. For example, a hash join can be transformed to a merge join and a merge join to a hash join. The single join order transformation is reversible on its own, that is, when applied twice on the same operator, the new plan is the same as the original plan.

Lemma 4.2.2. *The exploration of a neighborhood is independent from the initial plan, on which intra-transformations are applied.*

Proof. Suppose we have a plan P_0 and continually perform all intra-transformations to produce a set of plans $S = \{P_0, P_1, ..., P_n\}$ in the same neighborhood. Then, if we start from any plan P_i, $i = 1, 2, ..., n$, we will produce the same set S. To prove this lemma we will use a proof by contradiction. Suppose we start from plan P_k and we produce the set S'. S' must be a superset of S. We know that there exists a set of transformations to transform P_0 to P_k. By the reversibility property, we can transform P_k to P_0 and from there produce at least S. Now, suppose that $P_m \in S'$ but $P_m \notin S$. Then, there exists a set of transformations to transform P_k to P_m. By transitivity, P_0 can also be transformed into P_m. Thus, $P_m \in S$, which is a contradiction. Therefore, the exploration of a neighborhood is independent from the initial plan. □

4.3 Exploring Across Plan Neighborhoods

Intra-transformations are used to generate plans within a particular neighborhood. However, in order to explore the full plan space, we also need to generate plans that belong to different neighborhoods. For this purpose, we defined a new set of transformation rules, which we call *inter-transformations*[2].

Definition 4.3.1. Inter-transformations are transformations that can be applied across multiple operators and produce execution plans that belong to different neighborhoods.

Inter-transformations affect the structure of the execution plan. In particular, we are considering join order changes across multiple joins. For example, a join sequence $((A \bowtie B) \bowtie C)$ can be transformed into $(A \bowtie (B \bowtie C))$ to produce a new execution plan that belongs to a different neighborhood. Inter-transformations have analogous properties with the intra-transformations, namely transitivity and reversibility.

A combination of intra and inter transformations can be used to effectively and systematically explore the space of equivalent execution plans. In particular, given an initial plan, we can use intra-transformations to explore its neighborhood. Then, we can perform an inter-transformation to generate a new plan, that will serve as the initial plan in a different neighborhood. We can then repeat the process until we

[2] The prefix "inter" means "across" and in this context it refers to plans generated "across neighborhoods".

have explore the entire plan space. Of course, caution is needed to avoid returning to an already explored neighborhood to avoid an infinite loop. An easy solution is to generate all plans that belong in different neighborhoods together, before exploring within neighborhoods.

5

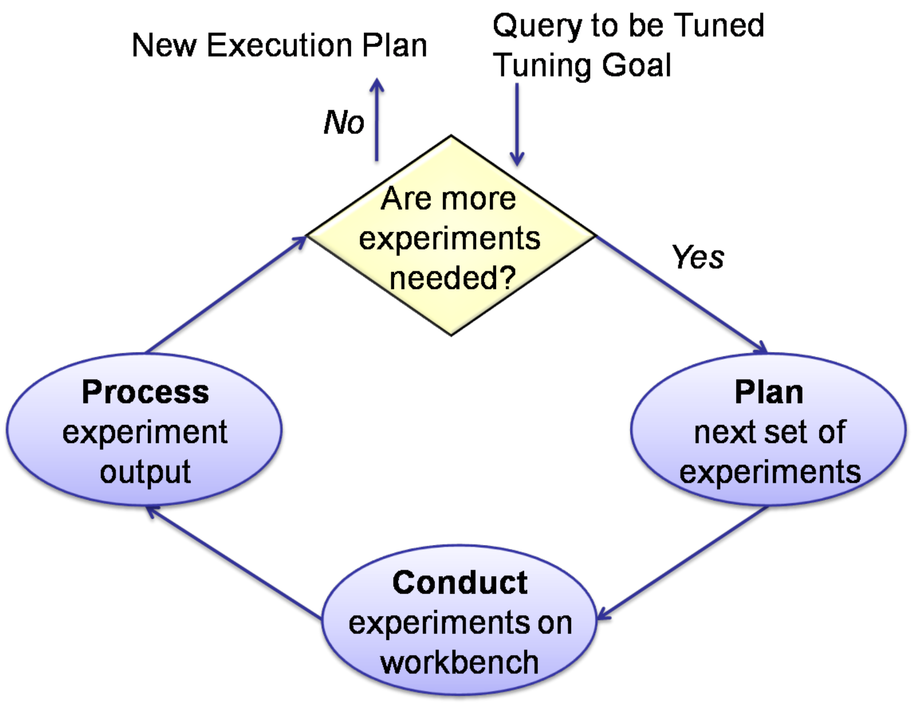
An Experiment-Driven Approach to SQL Tuning

We have seen how intra and inter transformations can be used to generate valid query execution plans within and across neighborhoods. We have also seen how the Explain_Plan command can be used to fully specify a query execution plan that could then be costed and executed. zTuned, our new SQL tuning tool, leverages the transformations and the capabilities of the Explain_Plan command to formalize and fully automate the process of SQL tuning using an experiment-driven approach.

5.1 Experiment-Driven Workflow

In order to explore the space of possible valid execution plans, a sequence of experiments will need to be performed. We define a single experiment to mean generating a plan through the use of transformations and running it using the Explain_Plan command with or without the "analyze" option. The experiments utilized estimated and actual cardinalities and execution costs, in order to effectively prune the large space of execution plans and reach to an optimal plan. We formalize this sequence through the experiment-driven *workflow* shown in Figure 5.1.

The first and most critical step in the workflow involves generating an *experiment*



New Execution Plan
Query to be Tuned
Tuning Goal
No
Are more experiments needed?
Yes
Process experiment output
Plan next set of experiments
Conduct experiments on workbench

FIGURE 5.1: Experiment-Driven Workflow

design; that is a sequence of experiments to generate alternative plans, collect samples, improve statistics, or verify performance improvements of a proposed solution. The total space of experiments is extremely large to cover and hence, our goal is to choose a smart ordering of experiments to conduct, in order to reach a better plan as fast as possible. At the same time, we wish to maximize the use of the available information we have collected so far. We describe the details of the experiment design in Section 5.2.

Conducting an experiment with a chosen setting of factors is nontrivial. These factors could be (i) workload-related (run in isolation or within a production workload replay), (ii) resource-related (for specified amounts of CPU and memory) or (iii) configuration-related (various database configuration parameters like buffer pool size). By default, zTuned uses the same settings with the production database, even though those could be adjusted if needed. Depending on which mode zTuned is used in (online Vs offline), experiments will either be conducted on the production system or on the proposed workbench discussed in detail in Section 5.3.

Finally, the output of these experiments must be processed to decide the next step. Analysis of the output data could lead to either a new query execution plan or to the design of the next set of experiments.

5.2 Experiment Design

Based on our experimental approach, we generate alternative plans until we find a better execution plan. Initially, we execute the query in isolation using the current execution plan and collect several pieces of information like time spent on each operator, estimated and actual cardinalities of data between the operators, disk I/O, memory utilization etc. Such information may be already available from plan execution history on the production database, in which case we avoid or limit the initial execution of the plan.

First, let us consider applying a set of intra-transformation on the initial plan P_0 in order to produce a set of plans $S = \{P_0, P_1, ..., P_n\}$ that belong to the same neighborhood. According to the definition of the neighborhood (Section 4.1) and given that we already know the cardinality of every operator in P_0, we induce the cardinalities for every plan in S. Then, we use the costing engine of the database (through the use of the Explain_Plan and its cardinality featured explained in Section 3.3.2) to estimate the execution cost of each plan in the neighborhood. Since all the plans are similar in structure and we used the same cardinalities for all of them, we can compare the estimated execution costs to select the optimal plan in the neighborhood.

It is important to remember here the potential causes of mistakes made by the query optimizer. These causes lie in two general categories: (a) cardinality mistakes, and (b) cost modeling mistakes. Previous studies have shown that cardinality mistakes are usually the main cause of incorrect costing. In general, the cost models are assumed to be correct enough. For now, we will make the same assumption as well and discuss later how zTuned could potentially relax it.

Essentially, this set of experiments explore a large part of the plan space using all the available cardinality information. This approach thus avoids executing all possible plans in the neighborhood, which would be prohibitively costly.

Intra-transformations are used to generate plans within a particular neighborhood. However, in order to explore the full plan space, we also need to generate plans that belong to different neighborhoods. For this purpose, we use inter-transformations on the original plan P_0 to produce a new set of plans that belong to different neighborhoods. These plans will serve as starting points for exploring different neighborhoods. It is worth remembering here that the space of all possible and equivalent plans can be partitioned into disjoint neighborhoods as shown in Section 4.2. Hence, with each neighborhood exploration, we effectively explore a different, disjoint set of execution plans.

When exploring different neighborhoods, we are faced with two problems. The first problem involves the selection of which neighborhood to explore next, given multiple choices. We would somehow need to decide which neighborhood is more promising to explore. The second problem occurs when we wish to compare two plans that were selected as optimal plans from two different neighborhoods. Both problems depend on the execution mode of zTuned and the solutions are described in Chapter 6.

5.3 Workbench for Experiments

Experiments may be run before or after the database goes into production use. Once the database is in production use, experiments can be run on: (i) the production database that serves user queries, (ii) the standby databases backing up the

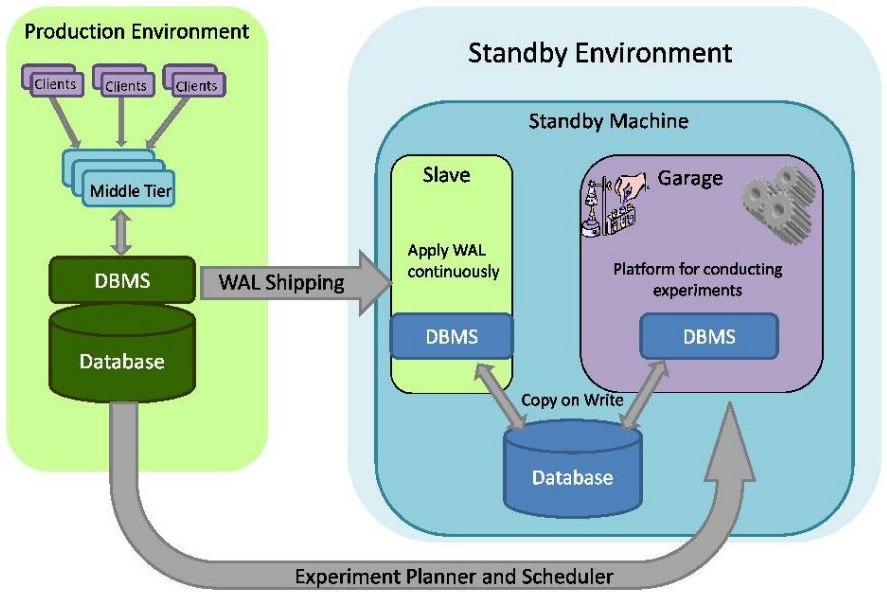


FIGURE 5.2: Workbench for Experiments

production database, or (iii) the test database used by DBAs and developers (see Figure 1.1).

Running experiments on the user-facing production database is a risky proposition unless the impact of experiments can be bounded or do not make an excessive use of database resources. For example, the DBA should be able to specify something like: "Use up to 10% of the system's resources to generate useful data through experiments, but do not harm the production workload by more than 5%". References [7, 8, 12] take this approach. Furthermore, fine-grained resource control and isolation through OS-level resource containers and virtualization [28] is a key enabler here.

However, many administrators will remain fearful of running experiments on the user-facing database system. We propose a novel *framework* for running online experiments that exploits the standby database environment (Figure 5.2) maintained for high availability [6]. This design emerged from the following observations:

- The competitive edge gained from high availability makes enterprises run databases with built-in redundancy. All commercial databases support one or more hot standby databases that are kept up to date with the (primary) production database through redo log shipping. If the primary fails, a standby will quickly take over as the new primary. Hence, the standby databases run the same hardware and software as the production database.

- Each standby database has very low utilization since it only has to apply redo log records. (Modern OS and storage systems provide very efficient mechanisms for applying redo logs.) Reference [16] mentions that enterprises that have 99.999% (five nines) availability typically have standby databases that are idle 99.999% of the time.

The standby databases are a valuable and underutilized asset that can be used for online experiments without impacting user-facing queries. However, the recovery time on failure should not be affected. Our workbench runs two resource-isolated *virtual machines* on each standby database: the *slave* and the *garage* (see Figure 5.2).

- The slave takes over the original responsibility of the standby database, and is responsible for applying the redo log records. The slave usually needs less than 1% resources. The remaining resources are given to the garage where experiments are run, but redo logs are not applied, in order to ensure consistency among each set of experiments.

- The slave runs continuously, while the garage is started on demand when experiments need to be done. When the garage is started, it is given a snapshot of the current database. Although both the slave and the garage logically have separate local copies of the database, only a single physical copy of the database exists on the standby system. The snapshots have copy-on-write semantics. Thus, the redo logs applied by the slave do not affect the garage's snapshot.

- When we are done with experiments or if the primary fails, the garage is torn down immediately. All resources are then released to the slave which will continue functioning as a pure standby or take over the primary as needed.

- Setting up the garage (including snapshots and resource allocation) takes less than a minute, and tear-down takes even less time. The whole process is so efficient that recovery time is not increased by more than a few seconds.

6

zTuned: Automated SQL Tuning

SQL tuning has become a critical aspect of database administration and its success usually depends on the expertise of highly-skilled database administrators (*DBA*s). However, many medium and small size enterprises do not have trained database administrators, rather they have general purpose information technology (IT) personnel. Even in the presence of DBAs, the SQL tuning task is extremely time consuming since in many cases it involves experimentation with different execution plans in an attempt to correct the mistakes of the query optimizer.

For this purpose, we developed *zTuned*, a system that formalizes and automates the process of SQL tuning using the experiment-driven approach presented in Chapter 5. zTuned utilizes several mechanisms, including the Explain and Explain_Plan commands discussed in Chapter 3. It has the ability to generate different yet equivalent execution plans using a set of transformations, and then through a smart exploration of the plan space, find a better execution plan. It is important to note that this process is similar to what a DBA would perform in order to manually diagnose and fix the issue. We automated this process in a way that provides confidence to the DBA that she can trust the system to perform SQL tuning automatically.

The plan generation occurs independently from the query optimizer, allowing for a larger and different exploration of the plan space. In addition, since zTuned works outside the optimizer, it can potentially be used with any database that uses a cost-based optimizer and supports the specification of full execution plans.

In an attempt to produce a system that is both efficient and practical, we gave zTuned the capability of performing SQL tuning in two different modes, *online* and *offline*. Given the different set of restrictions and requirements, we utilized the mechanisms and algorithms we developed in two different ways.

6.1 Performing SQL Tuning Online

Online SQL tuning refers to performing the tuning directly on the production system. Of course, the main concern is the additional overhead introduced by this process. Hence, our goal in this mode is to incur as small overhead as possible and search the plan space very efficiently and effectively. In the online mode, zTuned explores a smaller part of the plan and in particular, the neighborhood of the plan provided by the optimizer. This exploration occurs in an incremental fashion and it incurs a very small overhead. Figure 6.1 shows the flow of execution for zTuned in Online Mode.

Initially, zTuned collects the necessary statistics from the execution history that affect the poorly-performing query we are asked to tune. These statistics include past execution times as well as actual cardinalities for each operator. If this information is not available, zTuned must execute the offending query; but this is the only query execution that it will perform.

After obtaining the execution plan selected by the query optimizer, zTuned will attempt to explore the plan's neighborhood. It is important to remember here some of the essential properties of a plan neighborhood described in Section 4.1. In particular, all plans in the same neighborhood can utilize the exact same cardinality estimates for all their operators. Hence, zTuned uses the costing engine of the database

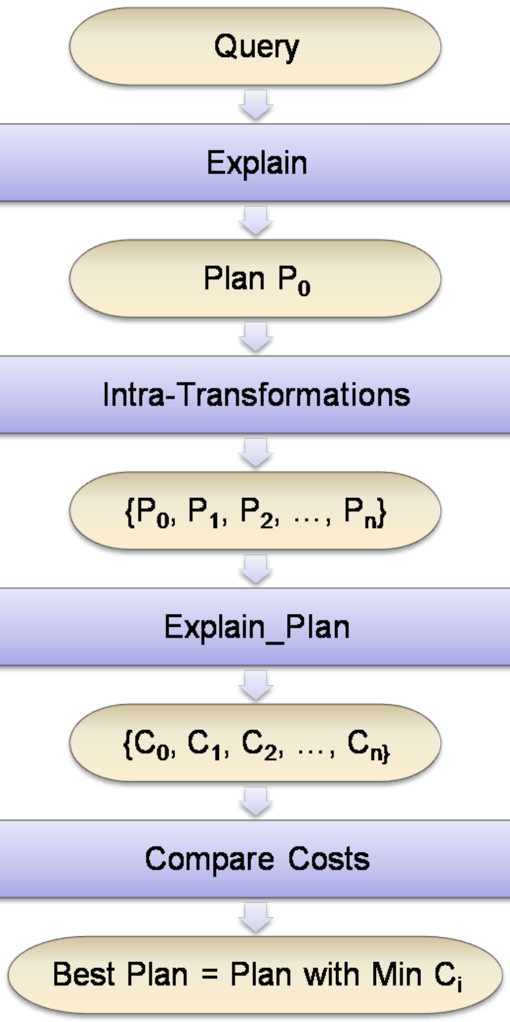

Query
Explain
Plan P_0
Intra-Transformations
$\{P_0, P_1, P_2, \ldots, P_n\}$
Explain_Plan
$\{C_0, C_1, C_2, \ldots, C_n\}$
Compare Costs
Best Plan = Plan with Min C_i

FIGURE 6.1: Online Mode Implementation Workflow

to estimate the execution cost of each plan it generates. Since all the plans are similar in structure and the same cardinalities are used, zTuned can safely compare the estimated execution costs in order to select the best execution plan within the neighborhood.

Even though a plan neighborhood consists of plans with similar structure and merely different operators, its size could be very large. Actually, the size of a plan neighborhood could be exponential in the number of tables, depending on the number of valid intra-transformations. Hence, it is also essential to explore a single neighborhood in a more systematic way. The idea is to order the operators in such a way that when transformations are applied to them, the generated plans are more likely to be better. We have devised two techniques that appear to work well in practice.

First, we consider the difference between the estimated (ec_i) and actual (ac_i) execution cost of each operator Op_i. However, the two costs cannot be directly compared because estimated costs are usually expressed in terms of disk page fetches, whereas real costs are expressed in terms of time. Instead, we perform a relative comparison between them based on the cost of the entire plan. Let E and A be the estimated and actual execution costs of the entire plan respectively. Then, $\frac{ec_i}{E}$ and $\frac{ac_i}{A}$ are the fractions of total estimated and actual costs attributed to Op_i. We define the *Costing Error* (E_i) for each operator Op_i to be:

$$E_i = \frac{\left|\frac{ec_i}{E} - \frac{ac_i}{A}\right|}{\frac{ac_i}{A}} \tag{6.1}$$

We rank the operators according to the costing error. Operators with higher costing error E_i are more likely to be the cause of the problem since they are an indication that the query optimizer has made a significant costing mistake. Hence, we perform all valid intra-transformations to the k operators with the highest ranking. The parameter k is an adjustable parameter and for our experiments, we used $k = 3$.

If the above set of experiments do not produce an execution plan that is better, a more detailed sensitivity analysis is needed. Based on this analysis, we compare the difference between the estimated (ed_i) and actual (ad_i) cardinalities of the different operators. We define the *Cardinality Error* (D_i) for each operator Op_i to be:

$$D_i = \frac{|ed_i - ad_i|}{ad_i} \tag{6.2}$$

Unlike the cost comparison, a higher cardinality error does not necessarily imply the need for a different operator. For each operator input, there exists a range of cardinality values for which that particular operator is the best. Therefore, only when the actual cardinality value is located outside that range, a different operator will be used. However, it is still a promising starting point for the neighborhood exploration, especially if the operator is located low in the tree. In this case, it is quite possible that this error propagated up the tree and thus affected the entire plan. Thus, we rank the operators according to the cardinality error and then perform all valid intra-transformations to the k operator with the highest ranking. We ensure that these operators are different from the k operators produced by the costing error ranking.

If the above set of experiments do not produce an execution plan that is better, the exploration continues until it covers the entire neighborhood. However, in order to explore the full plan space, we would also need to generate plans that belong to different neighborhoods. Unfortunately, zTuned is no longer able to use the original actual cardinalities. Even though zTuned could use as much information as possible and estimate the remaining cardinalities using the database statistics, comparing estimated costs would no longer be valid. Hence, in the online mode, zTuned does not explore different neighborhoods. If the original neighborhood does not include a better execution plan, the user is recommended to run zTuned in the offline mode.

6.2 Performing SQL Tuning Offline

In the offline mode, zTuned has the ability to explore a much larger space using more experiments that involve the execution of a particular plan. Even though it might take more time to find a better plan, it will take considerably less time compared to the manual experimentation used today by DBAs. Figure 6.2 shows the flow of execution for zTuned in Offline Mode.

Initially, zTuned uses the same exploration techniques outlined in Section 6.1 to

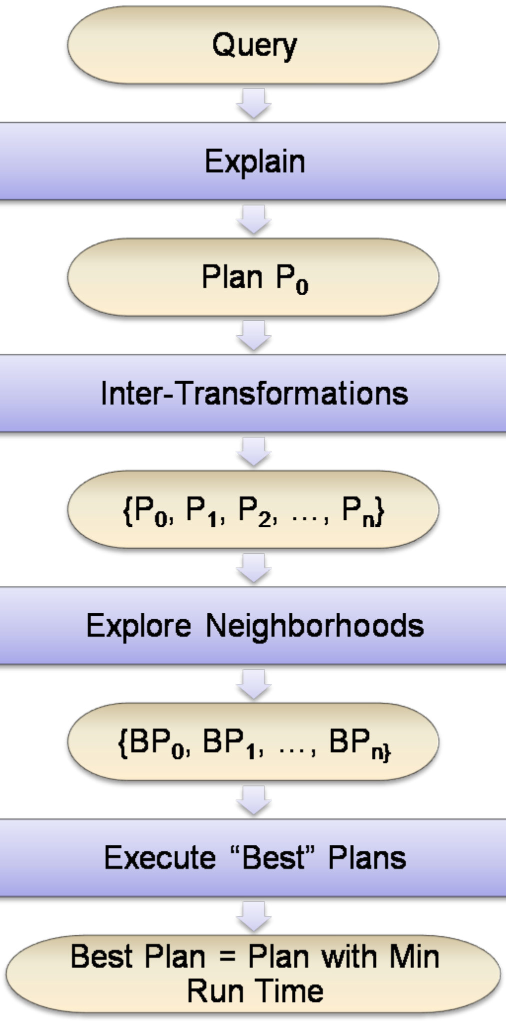

Query
Explain
Plan P_0
Inter-Transformations
$\{P_0, P_1, P_2, \ldots, P_n\}$
Explore Neighborhoods
$\{BP_0, BP_1, \ldots, BP_n\}$
Execute "Best" Plans
Best Plan = Plan with Min Run Time

FIGURE 6.2: Offline Mode Implementation Workflow

generate all possible plans in the neighborhood of the plan selected by the query optimizer. If this does not lead to a better execution plan, zTuned will then use inter-transformations to generate plans that belong to different neighborhoods.

However, we are no longer able to use the original running cardinalities since the generated plans belong in different neighborhoods. The idea here is to execute one or more of these further-away plans in order to collect the additional cardinalities needed, and then explore their neighborhoods with full information.

When exploring different neighborhoods, we are faced with three problems. The first problem involves the selection of which neighborhood to explore next, given multiple choices. After selecting the neighborhood, we need to select which plan from that neighborhood to execute initially in order to collect the necessary cardinalities. The final problem occurs when we wish to compare two plans that were selected as optimal plans from two different neighborhoods.

Continuing in the spirit of exploring the space incrementally, zTuned orders the plans generated by the inter-transformations based on the estimated running costs. It then fully explores each neighborhood, starting from the one with the smallest estimated running cost.

In the offline mode, zTuned has the ability to execute additional plans in order to collect any missing statistics that are needed to explore different neighborhoods. In Section 4.2, we proved that the exploration of a neighborhood is independent from the initial plan that is executed. Hence, we have two choices: (a) we can generate all plans in the neighborhood, cost them, and then execute the one with the smallest estimated cost or (b) execute the plan that was produced by the inter-transformation i.e., the plan that is closest to the plan selected by the query optimizer. We selected the latter choice because it would be inefficient to generate all plans and then store them or re-generate them during the neighborhood exploration. In addition, starting from the plan that is most similar to the plan selected by the query optimizer has some potential benefits that we plan to take advantage of in the future. In particular, since the two plans are so similar, we could materialize some subplans from the original plan and avoid the execution of the entire plan. Moreover, if we only need one additional cardinality value for our new plan, we might perform sampling instead of executing the entire plan.

Finally, zTuned executes the best plan from each neighborhood and uses the actual running times to compare them. These final runs also serve as validation for the improvement offered by the new plan.

6.3 zTuned Settings

In both modes, the user is able to provide tuning goals for each query. A tuning goal can be specified as an absolute running time threshold or a percent improvement over the running time of the poorly-performing plan. For example, the user may specify that a query must run under 5 minutes, or that she wishes that the query execution time is improved by 20%. In both cases, zTuned explores the space of plans until it finds a plan that satisfies the provided condition.

Alternatively, the user can bound the execution of zTuned in two ways. She can either specify a maximum time for zTuned to run or specify how many "better" plans to find. In the former case, the user could instruct zTuned to execute for 3 minutes and return the best plan it found so far. In the latter case, the user may instruct zTuned to return the first plan that it found to be better from the original plan. These settings are particularly useful when running the online mode. The full command to execute zTuned and all the options are outlined in Appendix A.2.

7

Experimental Evaluation

We performed an extended experimental evaluation of our automated SQL Tuning tool run in both online and offline mode. The purpose of the experimental evaluation is three fold. First, we evaluate the effectiveness of zTuned in finding better execution plans for poorly-performing queries. Second, we compare the online and offline modes of zTuned and discuss the trade offs between them. Finally, we provide an interesting study of the effects of skewed data on the performance of the query optimizer and zTuned.

7.1 Environment

Our experiments were run on a VMWare Virtual Machine running Ubuntu Linux 8.10, with an Intel Core Duo 2.53GHz CPU and 1GB of RAM memory. The database server we used was PostgreSQL v8.3.4. We used the TPC Benchmark H (TPC-H) database with a scale factor of 1GB. TPC-H is a decision support benchmark comprised of queries that try to simulate business intelligence queries.

By default, the TPC-H benchmark populates the tables with uniform data. However, it is unrealistic to believe that a given dataset does not contain any data skew.

For example, a typical international company will have different numbers of customers in each country. Therefore, we modified the TPC-H data generation tool (*DBGEN*) to produce skewed data that follow a zipfian distribution. We used the original DBGEN to generate all the data in a database, called *TPCH_Z0*, and the modified DBGEN to generate skewed data with a zipfian parameter of 3 in another database, called *TPCH_Z3*. We used an index advisor to produce indexes for the TPC-H workload in order to ensure a proficient physical design. We used the default value for all PostgreSQL configuration parameters. Finally, we run the *"Analyze"* command in PostgreSQL to update all the database statistics. By default, PostgreSQL collects the 10 most frequent values and creates histograms with 10 bins for almost all columns in all tables in the database.

The Explain_Plan command was implemented as an internal command in PostgreSQL and was written in C. As a result, some core PostgreSQL code was modified. zTuned is currently implemented as a stand-alone Java application. For our experiments, we used TPC-H queries generated from the 22 available templates. The TPC-H tool *QGEN* was used to generate all instances of the queries with different parameter values. The results presented below were obtain by executing zTuned in offline mode over the TPCH_Z3 database, unless otherwise noted.

7.2 Effectiveness Evaluation

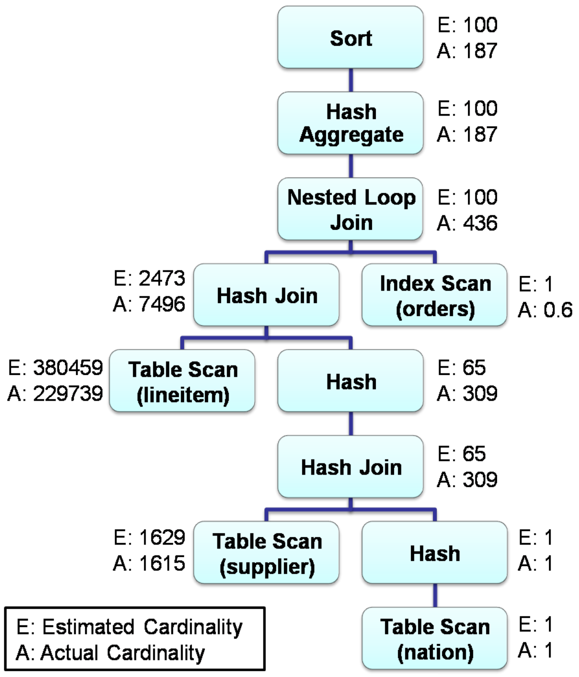
The first set of experiments targets the ability and efficiency of zTuned to find better execution plans for poorly-performing queries. First, we will present detailed results from tuning a specific query over TPC-H data. Second, we compare execution plans selected by the query optimizer to the ones selected by zTuned. For this purpose, we selected some queries and executed all possible plans. The truly optimal plan is the one that executes in the least amount of time. Finally, we discuss how zTuned can find better execution plans for a large variety of queries.

In order to get a better insight to why the query optimizer makes mistakes and how zTuned can correct them, we will use the sample query shown in Figure 7.1. This is a variant of query 21 from the TPC-H Benchmark. It identifies certain suppliers from Argentina whose products are part of a multi-supplier order (with current status of 'F') and with certain account characteristics. We generated and executed all possible plans for this particular query.

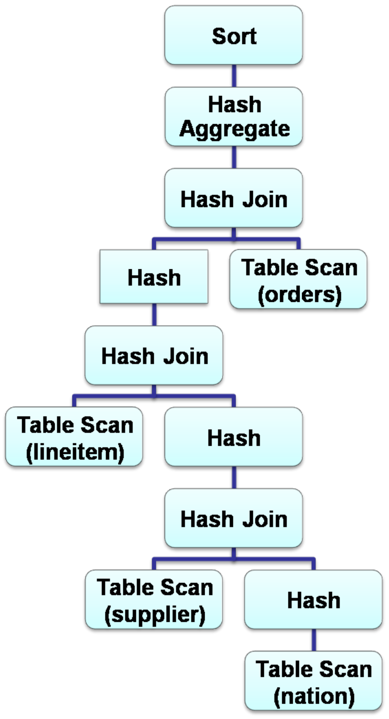
```
SELECT s_name, count(*) as numwait
FROM   supplier, lineitem, orders, nation
WHERE  s_suppkey = l_suppkey
   AND o_orderkey = l_orderkey
   AND o_orderstatus = 'F'
   AND s_nationkey = n_nationkey
   AND s_acctbal < 5000
   AND l_extendedprice > 80000
   AND n_name = 'ARGENTINA'
GROUP BY s_name
ORDER BY numwait desc, s_name;
```

Figure 7.1: Variant of TPC-H Query 21

The execution plan selected by the query optimizer is shown in Figure 7.2a. We see how the estimated cardinalities differ more from the actual ones for joins rather than for table scans. The optimizer does a reasonable job at estimating cardinalities for tables since all statistics are updated. However, the cardinalities of both joins were severely underestimated, which led the optimizer to select an index nested loop join at the top. The actual cardinality of the child hash join was much higher than expected and thus, the index over the *orders* table was hit a lot. The query from Figure 7.1 was then tuned using zTuned in the offline mode. In this case, zTuned selected the plan shown in Figure 7.2b, which performs about 20% better. Indeed, the new plan changes the index nested loop join into a hash join, which performs better given the larger number of tuples it needs to join.



(a) Plan Selected by Optimizer



(b) Plan Selected by zTuned

FIGURE 7.2: Execution Plans for Query in Figure 7.1

We repeated the above process for 3 more queries and the results are summarized in Figure 7.3. We note that in all cases, zTuned was able to find a plan that was either the truly optimal plan (found by exhaustive search), or much closer to the optimal plan compared to the plan selected by the query optimizer.

Finally, we generated a large set of TPC-H queries and tuned them using zTuned in offline mode. All queries were run three times and we report average times. Table 7.1 provides the results for 7 queries. The percent improvement that zTuned can offer varies greatly depending on the query. In general, larger queries see more improvement than smaller queries, since there is a higher probability for the query optimizer to make a mistake. When a query involves 3 or 4 joints, the incorrect cardinality estimates that are mainly based on invalid independence assumptions, have a larger effect on the selected plan. In addition, if a large mistake happens low

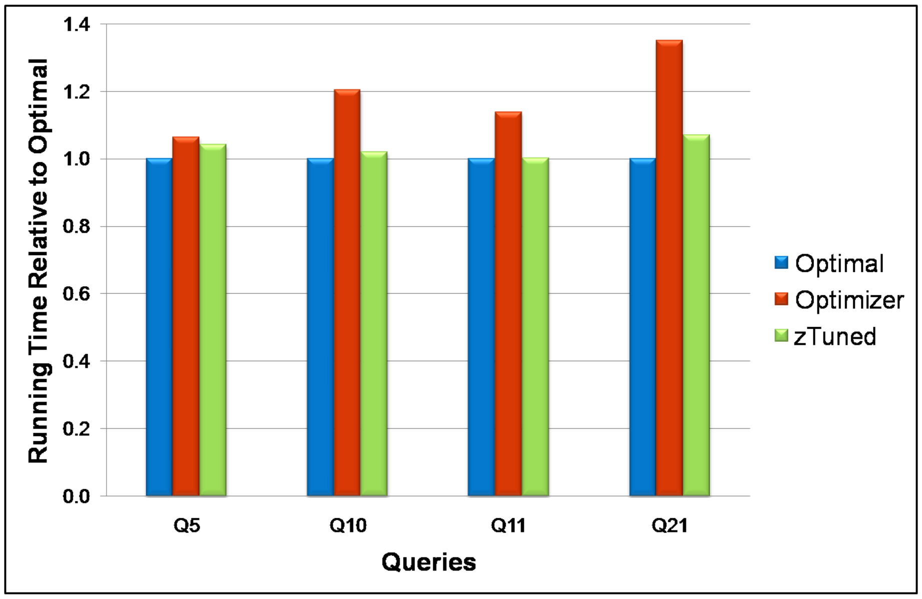


FIGURE 7.3: Running Times for the Optimal, Optimizer, and zTuned Plans

in the execution tree, then that mistake usually propagates up the tree. On average, zTuned is able to provide a plan with a 30% improvement over the plan selected by the query optimizer, and in some cases even up to 86% improvement.

The total number of plans explored in each tuning session also varies greatly. It is directly related to the number of tables accessed by the query. For instance, queries 8 and 9 involve joins over 6 tables and we see that zTuned explores 4000 and 7800 plans respectively. On the other hand, query 12 consists of a single join over two tables and the number of possible plans is just 12. Of course, tuning time is also directly related to the number of plans explored, as well as to the running time of the query.

7.3 Evaluating Online Vs Offline Modes

In an attempt to produce a system that is both efficient and practical, we gave zTuned the capability of performing SQL tuning in two different modes, *online* and *offline*. In the online mode, the cardinalities are collected from the plan selected

Query	Run Time of Optimizer Plan (sec)	Run Time of zTuned Plan (sec)	Percent Improvement	Tuning Time (sec)	Number of Plans Explored
5	41.76	40.64	2.68	225.84	1440
7	6.65	0.90	86.51	67.03	630
8	31.59	30.24	4.28	329.88	4000
9	224.88	95.28	57.63	884.92	7800
10	42.86	38.23	10.81	131.73	190
11	3.69	2.30	37.71	13.10	48
12	39.52	30.72	22.27	70.88	12

Table 7.1: Tuning Results for TPC-H Queries

by the query optimizer and then a single neighborhood is explored. Exploration within a neighborhood is only based on costing, and hence, we can explore a large number of plans very quickly. In the offline mode, we explore both within and across neighborhoods, using a combination of costing and execution in order to select a better plan. However, in most cases we don't actually execute more than 5-6 plans.

In this section, we present the results from tuning the same queries in both online and offline mode. Figure 7.4 summarizes the results whereas Table 7.2 provides more details.

As expected, the percent improvement achieved with offline mode is greater compared to online mode. Actually, in about half the cases, the online mode is not able to find a better plan and suggests running zTuned in offline mode. It is important to note the time it takes to explore plans in online mode, which is less than 1 second in many cases. This happens because the two heuristics described in Section 6.1 work well in practice. Operators with larger execution cost or cardinality mistakes are more promising to produce a better plan. Indeed, with just a few transformations for those operators, zTuned is able to produce plans with better performance. The tuning time for queries 5 and 8 is much larger because in this case, an entire neighborhood was fully explored unsuccessfully.

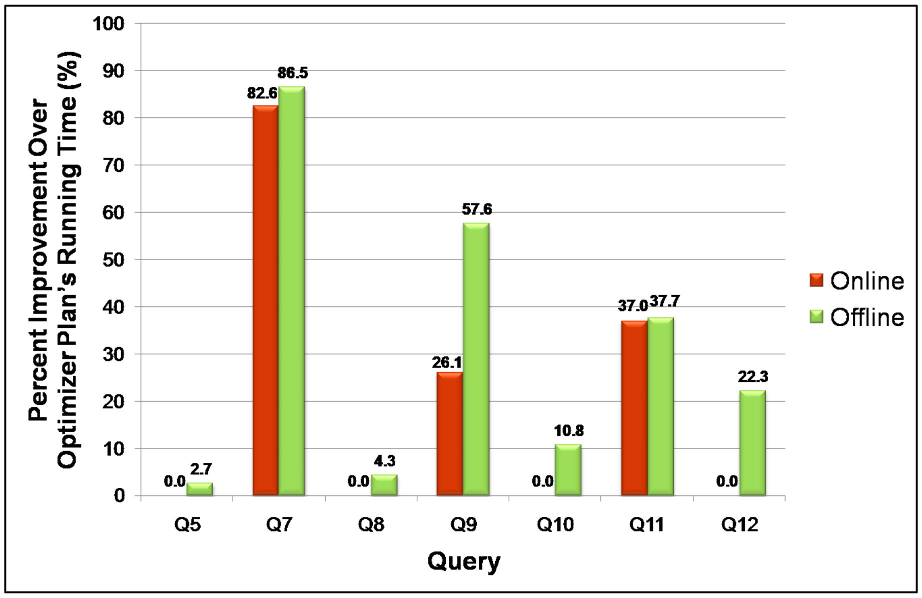


FIGURE 7.4: Tuning Improvement in Online Vs Offline Modes

Query	Run Time of Optimizer Plan (sec)	Online Mode		Offline Mode	
		Tuning Time (sec)	Percent Improvement	Tuning Time (sec)	Percent Improvement
5	41.76	50.71	0.00	225.84	2.68
7	6.65	0.15	82.57	67.03	86.51
8	31.59	104.97	0.00	329.88	4.28
9	224.88	0.16	26.14	884.92	57.63
10	42.86	0.06	0.00	131.73	10.81
11	3.69	0.10	36.99	13.10	37.71
12	39.52	0.24	0.00	70.88	22.27

Table 7.2: Tuning Results in Online Vs Offline Modes

For the offline mode, the tuning time is larger compared to the online tuning time since we actually execute some generated plans. However, the tuning times for those particular queries are still within minutes, which is significantly less compared to the time a DBA would need to find a better plan. In addition, note that the times we report are the times to explore the full space of plans, and not until the first better plan is found. zTuned provides both abilities but we chose to explore the full space for better evaluating its performance. Once again, there is a trade-off between the time spent and the percent of improvement that can be achieved.

7.4 Case Study with Skewed Data

In the presence of many joins, invalid independence assumptions have a strong affect on the selected execution plan. There is another class of invalid assumptions that can have similar negative effects even with few joins; namely, invalid uniformity assumptions. Most databases keep track of single table statistics with the use of histograms, which are usually relatively coarse. Hence, when the query optimizer needs to calculate cardinality estimates over predicates, it usually resorts to uniformity assumptions. Considering that the TPC-H Benchmark generates uniform data, evaluating zTuned only over the default TPC-H data seemed inappropriate. We thus modified DBGEN to generate skewed TPC-H data and match more realistic scenarios.

In this section we evaluate zTuned when asked to tuned the same queries over a database with uniform data and another with skewed data. Figure 7.5 summarizes the results whereas Table 7.3 provides more details.

The percent improvement for queries over skewed data was larger than over uniform data, as expected. Over uniform data, the cardinality estimates of the query optimizer were relatively close to the actual cardinalities, and hence it made better plan choices. In some cases, zTuned was in agreement with the query optimizer as to which plan is better, and in other cases the plan it found was marginally better. However, over skewed data, the query optimizer was making more mistakes on cardinality estimations, which led to many suboptimal plans. In these cases, zTuned was able to find much better plans.

Finally, note that the tuning times between the two cases are very similar most of the time, because the selected plans were similar. However, in some cases the selected

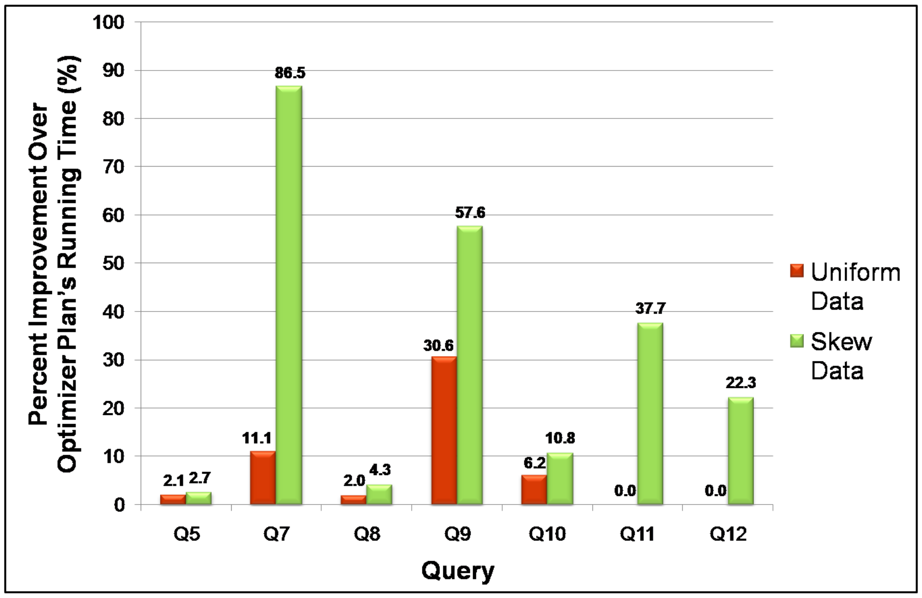


FIGURE 7.5: Tuning Improvement over Uniform Vs Skewed Data

Query	Uniform Data		Skewed Data	
	Tuning Time (sec)	Percent Improvement	Tuning Time (sec)	Percent Improvement
5	274.42	2.10	225.84	2.68
7	143.02	11.09	67.03	86.51
8	223.79	2.01	329.88	4.28
9	1143.52	30.58	884.92	57.63
10	101.74	6.22	131.73	10.81
11	17.18	0.00	13.10	37.71
12	69.79	0.00	70.88	22.27

Table 7.3: Tuning Results over Uniform Vs Skewed Data

plans were very different even though the queries and the database configuration parameters were the same. Since we run the "Analyze" option for both databases, each database had its own (and probably) different sets of histograms and statistics. Even with updated statistics, the query optimizer was making mistakes over both databases but even more mistakes over the skewed data. The ability of zTuned to find better plans however, is less affected by the statistics kept by the database.

8

Conclusion

SQL tuning is a critical aspect of database performance. Its success usually depends on the expertise of highly-skilled DBAs or time consuming trial-end-error steps. We developed a new command for the PostgreSQL database, called Explain_Plan, that is capable of costing and executing a fully specified physical plan. We then defined the concepts of plan neighborhoods, inter, and intra transformations in order to effectively and efficiently search the large space of execution plans.

Using the above mentioned mechanisms and techniques, we developed a system, called zTuned, that formalizes and automates the process of SQL tuning using an experiment-driven approach. We developed automated algorithms for planning experiments aimed at improving the poorly-running plan that is currently being used for a particular query. These experiments take place on a new workbench that takes advantage of underutilized resources, like the standby system that most production environments have.

Appendix A

Appendix

A.1 Explain_Plan Design

Command synopsis:

`EXPLAIN_PLAN (ANALYZE) (VERBOSE) "<Physical Plan>" FOR <Query>`

Parameters:

Analyze With this option the command will actually executes the plan and then displays the run time accumulated within each plan node along with the same estimated costs that the plain command shows

Verbose Shows the full internal representation of the plan tree, rather than just a summary. This option is only useful for debugging purposes.

Physical Plan The physical plan under consideration. The syntax is given below. Note that the quotes are important.

Query The actual SQL query that corresponds to the provided plan. This is required because it can provide useful information for the relations, indexes, and target lists (which is the information that the query should return) after the query is process by the parser. It is also used to infer aggregates, group by clauses, sort by clauses etc.

Physical Plan Syntax

Each plan node must appear within braces. The supported plan nodes and the corresponding keywords are:

Operator	Keyword
Sequential Scan	seqscan
Index Scan	indexscan
Nested Loop Join	nestjoin
Hash Join	hashjoin
Sort-Merge Join	mergejoin
Materialization	materialize

The syntax for each plan node is described next.

Sequential Scan

Attribute	Value	Notes
:parent	TableName	required
:filter	<expression>	optional
:cardinality	<integer>	optional

Index Scan

Attribute	Value	Notes
:parent	TableName	required
:using	IndexName	required
:indexCondition	<expression>	required
:filter	<expression>	optional
:cardinality	<integer>	optional

Nested Loop Join

Attribute	Value	Notes
:nestCondition	<expression>	optional if inner join is index scan
:filter	<expression>	optional
:outerjoinpath	<plan node>	required
:innerjoinpath	<plan node>	required
:cardinality	<integer>	optional

Hash Join

Attribute	Value	Notes
:hashCondition	<expression>	required
:filter	<expression>	optional
:outerjoinpath	<plan node>	required
:innerjoinpath	<plan node>	required
:cardinality	<integer>	optional

Merge Join

Attribute	Value	Notes
:mergeCondition	<expression>	required
:filter	<expression>	optional
:outersortkeys	<column list>	needed if outerjoinpath is not sorted
:innersortkeys	<column list>	needed if innerjoinpath is not sorted
:outerjoinpath	<plan node>	required
:innerjoinpath	<plan node>	required
:cardinality	<integer>	optional

Materialize

Attribute	Value	Notes
:path	<plan node>	required
:cardinality	integer>	optional

Expression Syntax

Operand	Definition
<expression>	A sequence of <subexpression> separated by AND/OR and grouped into parenthesis.
<subexpression>	<operand> <op> <operand>
<operand>	One of: <column name>, string, integer, or double
<column name>	It must have the form tablename.columnname
<op>	Any mathematical operators and LIKE
<column list>	A comma separated list of <column name>

Other Notes

- Aggregates, group clauses, having clauses, sort clauses etc are implicitely supported, that is they are induced from the provided query. The user should not specify them in the string representation of the physical plan.

- The cardinality value represents the expected number of output tuples for a particular node.

A.2 zTuned Usage

zTuned command:

```
    java zTuned [-v <n>] [-l logId] [-m mode]
                -c <config\textunderscore file>
                -q "<query\textunderscore text>"
                -f <query\textunderscore file>
```

Console verbosity (-v) options:

0 : No output, no error messages

1 : No output except error messages

2 : Basic output (default)

3 : Detailed output

Log id (-l) option:

A string to be appended to the log files.

If none, the current data and time is used.

Mode (-m) options:

online : Online Tuning (Default);

offline : Offline Tuning

Configuration (-c) option:

A configuration file for the database that specifies all the necessary information needed to connect to a database (url, name, port etc.)

Query (-q) option:

A query to tune

File (-f) option:

A file containing one or more queries to tune

Bibliography

[1] Ashraf Aboulnaga and Surajit Chaudhuri. Self-tuning histograms: building histograms without looking at data. In *SIGMOD '99: Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 181–192, New York, NY, USA, 1999. ACM.

[2] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollar, Arun Marathe, Vivek Narasayya, and Manoj Syamala. Database tuning advisor for microsoft sql server 2005. In *VLDB '04: Proceedings of the 30th International Conference on Very Large Data Bases*. VLDB Endowment, 2004.

[3] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, pages 496–505, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[4] Sanjay Agrawal, Eric Chu, and Vivek Narasayya. Automatic physical design tuning: workload as a sequence. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 683–694, New York, NY, USA, 2006. ACM.

[5] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 359–370, New York, NY, USA, 2004. ACM.

[6] S. Babu, N. Borisov, S. Duan, H. Herodotou, and V Thummala. Automated experiment driven management of (database) systems. In *12th Workshop on Hot Topics in Operating Systems (HotOS-XII)*, Monte Verit, Switzerland, may 2009.

[7] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, June 2004.

[8] Shivnath Babu, Pedro Bizarro, and David DeWitt. Proactive re-optimization. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 107–118, New York, NY, USA, 2005. ACM.

[9] Nicolas Bruno and Surajit Chaudhuri. Automatic physical database tuning: a relaxation-based approach. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 227–238, New York, NY, USA, 2005. ACM.

[10] Nicolas Bruno and Surajit Chaudhuri. An online approach to physical design tuning. *Data Engineering, International Conference on*, 0:826–835, 2007.

[11] Surajit Chaudhuri, A.C. Konig, and Vivek Narasayya. Sqlcm: a continuous monitoring framework for relational database engines. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 473– 484, 2004.

[12] Surajit Chaudhuri, Vivek Narasayya, and Ravi Ramamurthy. A pay-as-you-go framework for query execution feedback. In *VLDB '08: Proceedings of the 34th International Conference on Very Large Data Bases*, Auckland, New Zealand, 2008. Morgan Kaufmann Publishers Inc.

[13] Chungmin Melvin Chen and Nick Roussopoulos. Adaptive selectivity estimation using query feedback. *SIGMOD Rec.*, 23(2):161–172, 1994.

[14] Stavros Christodoulakis. Implications of certain assumptions in database performance evaluation. *ACM Trans. Database Systems*, 9(2):163–186, 1984.

[15] Benoit Dageville, Dinesh Das, Karl Dias, Khaled Yagoub, Mohamed Zait, and Mohamed Ziauddin. Automatic sql tuning in oracle 10g. In *VLDB '04: Proceedings of the 30th International Conference on Very Large Data Bases*, pages 1098–1109. VLDB Endowment, 2004.

[16] Robert G. Freeman and Arup Nanda. *Oracle Database 11g New Features.* McGraw-Hill Osborne Media, 2007.

[17] Leonidas Galanis, Supiti Buranawatanachoke, et al. Oracle database replay. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008.

[18] IBM Corporation. *DB2 SQL Performance Analyzer*, 2007. http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/topic/com.ibm.db2tools.anl.doc.iug/anlhome.htm.

[19] IBM DB2. *Giving optimization hints to DB2*, 2003. http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/ index.jsp?topic=/com.ibm.db2.admin/p91i375.htm.

[20] Navin Kabra and David J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. *SIGMOD Rec.*, 27(2):106–117, 1998.

[21] Günter Karjoth. Access control with ibm tivoli access manager. *ACM Trans. Inf. Syst. Secur.*, 6(2):232–257, 2003.

[22] Microsoft Corporation. *SQLServer Books Online: Query hint (transact-sql)*, 2007. http://technet.microsoft.com/en-us/library/ms181714.apsx.

[23] Ravishankar Ramamurthy Nico Bruno, Surajit Chaudhuri. Power hints for query optimization. In *ICDE2009 - 25th International Conference on Data Engineering*, Shanghai, China, 2009.

[24] Oracle Corporation. *Oracle(R) Database Performance Tuning Guide 10g Release 2 (10.2)*, 2008. http://download.oracle.com/docs/cd/B19306_01/server.102/b14211/hintserf.htm.

[25] Stratos Papadomanolakis and Anastassia Ailamaki. Autopart: Automating schema design for large scientific databases using data partitioning. *Scientific and Statistical Database Management, International Conference on*, 0:383, 2004.

[26] PostgreSQL. *Tuning Your PostgreSQL Server.* http://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server.

[27] Kenneth R. Sheers. Hp openview event correlation services. *Hewlett-Packard Journal*, 1996.

[28] *Dtrace, ZFS, and Zones in Solaris.* www.sun.com/software/solaris.

[29] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. Leo - db2's learning optimizer. In *VLDB '00: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 19–28, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[30] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy Lohman, Adam Storm, Christian Garcia-Arellano, and Scott Fadden. Db2 design advisor: integrated automatic physical database design. In *VLDB '04: Proceedings of the 30th International Conference on Very Large Data Bases*, pages 1087–1097. VLDB Endowment, 2004.