Interactive Physical Design Tuning

Nicolas Bruno, Surajit Chaudhuri

Microsoft Research, USA
{nicolasb,surajitc}@microsoft.com

Abstract— In the last decade, automated physical design tuning became a relevant area of research. The process of tuning a workload became more flexible but also more complex, and getting the best design upfront became difficult. We propose a paradigm shift for physical design tuning, in which sessions are highly interactive, allowing DBAs to quickly try different options, identify problems, and obtain physical designs in an agile manner.

I. INTRODUCTION

In the last decade, several academic and industrial institutions addressed the problem of recommending a set of physical structures that would increase the performance of a given workload on a database system. The physical design problem has been traditionally stated as follows: *Given a workload W and a storage budget B, find the set of physical structures, or configuration, that fits in B and results in the lowest execution cost for W*. This problem is very succinctly described and understood. Consequently, it has recently received considerable attention resulting in novel research results (e.g., [1], [2], [4], [5], [6], [7], [10], [11], [12], [15], [16], [18]) and industrial-strength prototypes in all major DBMS (e.g., [3], [13], [17]).

Existing solutions are mostly "monolithic": the normal process starts with a full input specification (i.e., a workload and a size constraint) and directly returns the final output configuration to deploy. Other interaction patterns are very unnatural or simply unavailable. Often, sophisticated DBAs find this model lacking in flexibility to address several real-world scenarios. The process of tuning a database is generally interactive and requires stronger feedback loops than what is offered in current DBMSs (we illustrate these claims in Section IV with concrete examples).

We believe that a change of paradigm is required for next-generation physical design tools. As far as we know, in this demonstration we showcase the first prototype that suggest highly interactive tuning sessions and results in better and richer interactions with DBAs. Our prototype exposes intuitive data structures and composable algorithms that can be examined and processed in a shell-like environment. The prototype borrows ideas from hierarchical file systems to expose and manipulate information, composable/pipelined scripts to easily build complex algorithms, and customized visualizations to obtain quick feedback in a tuning session. We show how these architectural design points enabled us to easily implement functionality not present in commercial products (e.g., [8]).

In the remaining of this paper we present the architecture of our prototype (Section II), give additional details on the
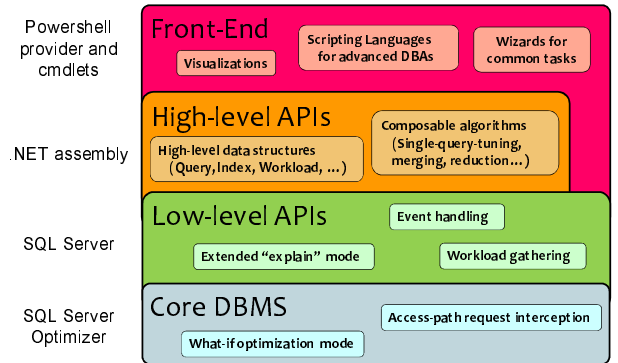

Fig. 1. Architectural layers for interactive physical design tuning.

scripting engine of our prototype (Section III), and present an sample session using our prototype (Section IV).

II. AN ARCHITECTURE FOR INTERACTIVE TUNING

Figure 1 shows a layered architecture for next generation physical design tools, including:

Core DBMS: The lowest layer resides within the database system and provides native support for operations such as what-if optimization [11] (with the additional fast variant of [9]) and access-path request interception functionality, as described in [5]. We implemented the Core DBMS layer by instrumenting Microsoft SQL Server query optimizer to support, in addition to what-if optimization, the access-path request interception and quick reoptimization techniques of [5], [9].

Low-level APIs: The low-level APIs expose, in formats that are simple to consume (e.g., XML), the functionality of the Core DBMS layer (and also the DBMS itself). As an example, they expose primitives to manipulate what-if mode, and also richer explain modes after optimizing queries, which surface optimization information required at higher levels. These APIs also encapsulate existing DBMS functionality, such as the ability to monitor and gather workloads.

High-level APIs: The previous two layers are, in some form or another, already present in current commercial DBMS. Physical design tools are typically built on top of the low-level APIs and only expose a rigid functionality (e.g., point to a workload, set the storage constraint, and optimize). Instead, we suggest a high-level API layer that exposes the internal representation and mechanisms used by current physical design tools in a modular way. Basic concepts such as queries, indexes, and access-path requests are exposed to higher layers to be used in different ways. In addition to these data structures, the high-level API layer exposes composable and simple

algorithms that tuning tools currently rely on. For instance, this layer exposes mechanisms to merge two indexes, or obtaining the best set of indexes for a single query. These primitive data structures and algorithms are not necessarily meant to be consumed by DBAs, but instead provide a foundational abstraction for applications to be built on top. We implemented the high-level API layer by introducing a new .NET assembly that encapsulates and exposes classes and algorithms relevant to physical design tuning. Among the classes that the assembly exposes are `Database`, `Table`, `Index`, `Column`, `Query`, `Configuration` and `Request`. We made these classes rich in functionality, so for instance the class `Index` has methods that return merged and reduced indexes, and methods that create hypothetical versions of the index in the database. The class `Query` has methods that evaluate (optimize) it under a given configuration, and methods that return its set of access-path requests.

Front-ends: Front-ends are based on both the low- and high-level APIs and deliver functionality to end-users. A very powerful interaction model, which we describe in detail in the next section, is a scripting platform to interact with physical database designs. The scripting language understands the data structures and algorithms exposed by the underlying layers and allows users to write small interactive scripts to tune the physical design of a database. Common tasks, such as minimizing cost for a single storage constraint (or other functionality provided by current physical design tools), can be seen as just pre-existing scripts that can be accessed using graphical user interfaces by relatively inexperienced DBAs.

III. A Scripting Engine for Physical Design Tuning

Our front-end is based on Windows PowerShell, an interactive scripting language that integrates with the Microsoft .NET Framework. Windows PowerShell provides an environment to perform administrative tasks by execution of cmdlets (which are basic operations), scripts (which are composition of cmdlets), stand-alone applications, or by directly instantiating regular .NET classes [14]. Its main features are:

Tight integration with .NET: Windows PowerShell leverages the .NET framework to represent data, and understands .NET classes natively. This means that our high-level API classes written in the .NET framework are easily available as first-class citizens in Windows PowerShell. We can then load all definitions directly into Windows PowerShell and start exploring, in a rudimentary but already interactive form, the physical design of a database, as shown in Figure 2.

Object pipelines: As shown in the previous example, and similar to Unix shells, cmdlets can be pipelined using the | operator. However, unlike Unix shells, which typically pipeline strings, Windows PowerShell pipelines .NET objects. This is a very powerful mechanism. For instance, consider the following script taken from Figure 2:

```
> $db.Tables.Values | where { $_.Pages -gt 500 } |
    sort Pages | select Name, Pages, Columns
```

obtains the list of all tables in the database, pipes the result (which is a list of `PDTCore.Table` .NET objects) to

```
> # create a new Database object from namespace PDTCore
> # pointing to server nicolasb02 and database tpch01g
> $db = New-Object PDTCore.Database("nicolasb02", "tpch01g")

Name            : tpch01g
Tables          : LINEITEM, CUSTOMER, SUPPLIER, NATION...
Configurations  : base, initial

> # obtain information on tables larger than 500 pages
> $db.Tables.Values | where { $_.Pages -gt 500 } |
    sort Pages | select Name, Pages, Columns

Name      Pages Columns
----      ----- -------
PARTSUPP  1573  {PS_PARTKEY, PS_SUPPKEY...}
ORDERS    2288  {O_ORDERKEY, O_CUSTKEY...}
LINEITEM  11486 {L_ORDERKEY, L_PARTKEY...}

> # create a new query
> $q = [PDTCore.Query]::Create($db,
        "SELECT * FROM LINEITEM WHERE L_ORDERKEY=15")

> # evaluate the query under the base configuration
> $db.Configurations["base"].Eval($q)

Name     Config Query Source Cost      SelectInfos Update
----     ------ ----- ------ ----      ----------- ------
base_Q1  base   Q1    Server 0.0032831 1           False
```

Fig. 2.   Using .NET classes in PowerShell.

the `Where` cmdlet, which understands the semantics of the objects and filters all those that contains less than 500 Pages. The resulting `PDTCore.Table` objects are pipelined to the `Sort-Object` cmdlet, which sorts them by the property `Pages` in descending order. In turn, the result of this cmdlet (i.e., an ordered list of tables) is passed to the `Select-Object` cmdlet, which returns a tuples with the name, number of pages, and number of columns of each table.

Data Providers: PowerShell has the ability to expose hierarchical data models by means of *providers*, which are then accessed and manipulated using a common set of cmdlets. As an example, the file system is one such provider. When situated in some node inside the file system provider, we can use `dir` to obtain the subdirectories or files in the current location, access contents of elements using `type`, and navigate the provider using `cd`. Windows PowerShell natively exposes the registry and the environment variables as providers. We implemented a PowerShell provider that exposes all the information about a tuning session in a hierarchical and intuitive object model. By using this provider, we can navigate and manipulate the state of a tuning session easily, almost as if it were a file system, as illustrated in Figure 3.

Cmdlets: Although the examples above are compelling, they are still not user friendly (we need to call the .NET methods directly). Also, they require considerable effort before actually tuning a database design. In addition to the provider, we augmented the .NET classes and methods with composable cmdlets. The examples in Figure 4 illustrate some cmdlets.

Scripts: The power of our implementation comes from scripts (either user defined or several of the ones that we already built). For instance, the script below is a simplified implementation of a common operation called *refinement*, in which a given input configuration is repeatedly "relaxed" via merging and reduction operations until it fits in the available

```
> # create a new provider on server nicolasb02
> # and default database tpch1g
> New-PDTDrive -Name P -Server nicolasb02 -Database tpch1g

> # go into the tables of tpch1g
> cd p:/tpch1g/tables

> # return all tables that start with part
> dir part* | sort Rows

Name        Database    Rows      Pages    Cols Indexes
----        --------    ----      -----    ---- -------
part        tpch1g      200000    3618     9    2
partsupp    tpch1g      800000    15628    5    2

> # get all indexes in the base configuration that have
> # more than two key columns
> dir P:/tpch01g/configurations/base/indexes |
    ? { $_.NumKeys -gt 1 }

Name                    Table      Keys Includes
----                    -----      ---- --------
PK__LINEITEM__07F6335A  LINEITEM   2 14
PK__PARTSUPP__0519C6AF  PARTSUPP   2 3
```

Fig. 3.   Navigating a custom provider for physical design tuning.

storage, so that the expected cost of the resulting configuration is as good as possible [6]. At each iteration, we calculate all possible transformations (using a cmdlet) and obtain the one that is expected to result in the smaller execution cost, repeating this process until a valid configuration is reached:

```
function Refine-Configuration() {
   Param ([PDTCore.Query[]] $Workload,
          [PDTCore.Configuration] $Configuration,
          [double] $Size)
   $act = $Configuration
   while ($act.Size -gt $Size) {
     $tr = Get-Transformations $Workload -Config $act |
             sort Cost | select -first 1
     $act = $tr.Apply()
   }
   return $act
}
```

In addition to `Refine-Configuration`, we implemented other common algorithms, such as the relaxation-based tuning approach in [5] and also a version that handles the constraint language of [8]. This last script takes as inputs a workload and a set of constraints as defined [8]. It heavily uses the .NET classes exported by the high-level APIs and is implemented as a PowerShell script in fewer than 100 lines of code. It is interesting to note that this PowerShell script corresponds to a rather non-trivial algorithm but is easily implemented reusing primitives exposed by lower layers.

Visualizations: An interesting side effect of using a composable, interactive script language is that we can very easily include third-party add-ins that offer specific functionality. One such example is PowerGadgets (`http://www.powergadgets.com`), which provides simple cmdlets to display data graphically, and is shown in Figure 5.

IV. A SAMPLE INTERACTIVE TUNING SESSION

In this section we present an annotated trace of a session that we conducted using our prototype. Figure 6 illustrates how an interactive approach can benefit DBAs by providing flexibility and control during physical design tuning. The example

```
> # load queries stored in a file. Note that results
> # can later be accessed via P:/tpch1g/queries
> Get-Query -Path D:/workloads/tpch-first-3.sql

Reading queries from D:/workloads/tpch-first-3.sql...
Name    Database    Type     Rows      Requests SQL
----    --------    ----     ----      -------- ---
Q0      tpch1g      Select   5.74262   6        SELEC...
Q1      tpch1g      Select   100       63       SELEC...
Q2      tpch1g      Select   10        26       SELEC...

> # create two indexes and merge them
> $i1 = New-Index -Table lineitem -Keys l_orderkey
> $i2 = New-Index -Table lineitem -Keys l_partkey
                   -Includes l_tax, l_orderkey
> $i3 = $i1.merge($i2)
> $i3.keys

Name         Table       Width
----         -----       -----
l_orderkey   lineitem    4
l_partkey    lineitem    4

> # create a new configuration
> $c = New-Configuration -Indexes $i1, $i3

> # evaluate the three most expensive queries
> # under the new hypothetical configuration
> dir p:/tpch1g/queries | sort -desc Cost |
    select -first 3 | Eval-Query -Configuration $c

Name      Config    Query     Cost
----      ------    -----     ----
C2_Q2     C2        Q2        143.075
C2_Q1     C2        Q1        6.05483
C2_Q0     C2        Q0        120.384
```

Fig. 4.   Cmdlets for physical design tuning.

```
> dir p:/tpch1g/tables |
    Out-Chart -gallery pie -label Name -values Pages
```
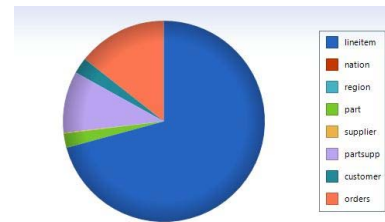

Fig. 5.   Using visualizations.

uses the provider, cmdlets and scripts described earlier, and others that are new (specially concerning visualization). We expect that advanced DBAs create their own scripts to further customize the physical design tuning experience. Moreover, special-purpose front-ends, which perform very specific but common tasks, can be deployed on top of the scripting language and shown to less advanced DBAs as graphical user interfaces.

REFERENCES

[1] S. Agrawal, S. Chaudhuri, and V. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2000.

[2] S. Agrawal, E. Chu, and V. Narasayya. Automatic physical design tuning: workload as a sequence. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2006.

[3] S. Agrawal et al. Database Tuning Advisor for Microsoft SQL Server 2005. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2004.
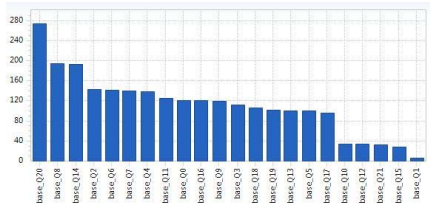
```
> #create a provider and set the location at the root
> New-PDTDrive -Name P -Server nicolasb02 -Database tpch1g
> cd P:

> #load TPC-H workload
> $w = Get-Query -Path D:/workloads/tpch-all.sql

> # get all query costs in decreasing cost order
> $w | Eval-Query -Configuration (Get-Conf base) |
    sort -desc cost | out-chart -values Cost -label Name
```



```
> # show the top-3 most expensive queries
> $expW = $w | Eval-Query -Configuration $c |
    sort cost -desc | select -first 3 | % {$_.query}
Name       Database      Type       Rows        Requests
----       --------      ----       ----        --------
Q20        tpch1g        Select     100         45
Q8         tpch1g        Select     172.421     48
Q14        tpch1g        Select     999.809     8

> # for each expensive query, infer the best indexes
> $bestIdx = $expW | % {$_.Requests} | % { $_.BestIndex }
Name         Table      Keys Includes
----         -----      ---- --------
⌐PDT_I17     orders     1    1
⌐PDT_I18     orders     1    1
⌐PDT_I19     lineitem   1    3
...

> # create a new configuration with all these best indexes
> # and compare this configuration with base for all queries
> $bestC = New-Configuration -Indexes $bestIdx -Name "MyC"
> Compare-Conf -Workload $w -C1 $bestC -C2 (Get-Conf base)
```
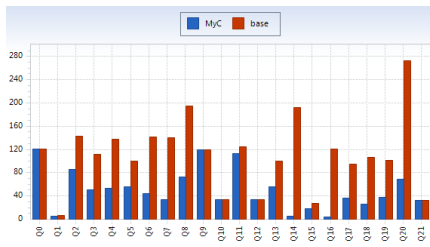


```
> # bestC surely is better, but what is its size?
> $bestC.size, (get-configuration base).size
3535.44, 1234.52

> # bestC is 2.9 times larger than base, refine it to 2.5GB
> $refC = Refine-Conf -Conf $bestC -Size 2500 -Workload $w
Name       Database      Size       Cost          Indexes
----       --------      ----       ----          -------
C11        tpch1g        2454.976   1080.27443    27

> # show all configurations evaluated by Refine-Conf
> dir P:/tpch1g/conf | out-chart -values size -xvalues cost
    -gallery scatter
```
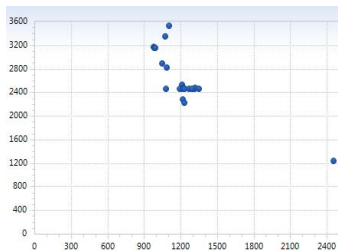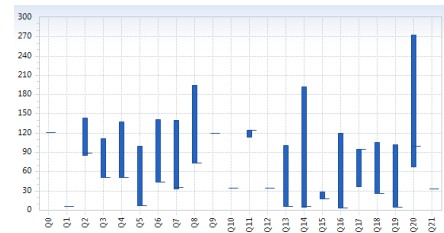

Fig. 6.   Interactive Physical Design Tuning Example

```
> # refC has good time/space tradeoff; compare queries in refC
> # against all other configurations. Display for each query,
> # min/max cost and current cost under refC
> Contrast-Conf -Workload $w -CS (dir P:/tpch1g/conf) -C $refC
```



```
> # Q17 and Q20 are the only two that could be improved
> # given more space (Q17 is really bad with 90 units)...
> # Use Constraints to tune again so that no query is worse
> # than 1.2x the cost under refC, but additionally
> # Q17 is expected to execute in fewer than 60 units.
> # Also try to get as close as possible to 2000MB
> $ct1 = "FOR Q in W ASSERT cost(Q,C) <= cost(Q,refC)*1.2"
> $ct2 = "ASSERT cost(W['Q17'], C) <= 60"
> $ct3 = "SOFT ASSERT size(C) = 2000"
> Tune-Constraints -W $w -Constraints $ct1, $ct2, $ct3
> ...
```

Fig. 6.   (cont.) Interactive Physical Design Tuning Example

[4]   S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2004.

[5]   N. Bruno and S. Chaudhuri. Automatic physical database tuning: A relaxation-based approach. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2005.

[6]   N. Bruno and S. Chaudhuri. Physical design refinement: The "Merge-Reduce" approach. In *International Conference on Extending Database Technology (EDBT)*, 2006.

[7]   N. Bruno and S. Chaudhuri. To tune or not to tune? A Lightweight Physical Design Alerter. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2006.

[8]   N. Bruno and S. Chaudhuri. Constrained physical design tuning. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2008.

[9]   N. Bruno and R. Nehme. Configuration-parametric query optimization for physical design tuning. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2008.

[10] S. Chaudhuri and V. Narasayya. An efficient cost-driven index selection tool for Microsoft SQL Server. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1997.

[11] S. Chaudhuri and V. Narasayya. Autoadmin 'What-if' index analysis utility. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 1998.

[12] S. Chaudhuri and V. Narasayya. Index merging. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 1999.

[13] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin. Automatic SQL Tuning in Oracle 10g. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2004.

[14] Microsoft. Windows Powershell, 2006. Accessible at http://www.microsoft.com/windowsserver2003/-technologies/management/powershell/default.mspx.

[15] S. Papadomanolakis and A. Ailamaki. An integer linear programming approach to database design. In *Workshop on Self-Managing Database Systems*, 2007.

[16] G. Valentin, M. Zuliani, D. Zilio, G. Lohman, and A. Skelley. DB2 advisor: An optimizer smart enough to recommend its own indexes. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2000.

[17] D. Zilio et al. DB2 design advisor: Integrated automatic physical database design. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2004.

[18] D. Zilio, C. Zuzarte, S. Lightstone, W. Ma, G. Lohman, R. Cochrane, H. Pirahesh, L. Colby, J. Gryz, E. Alton, D. Liang, and G. Valentin. Recommending materialized views and indexes with IBM DB2 design advisor. In *International Conference on Autonomic Computing*, 2004.