Master’s Thesis

Optimizing query execution to improve the energy efficiency of database management systems

Tobias Flach
tobias.flach@arcor.de

Submitted on August 31, 2010

Supervised by Prof. Dr. Felix Naumann
Abstract

Increasing energy costs became one of the critical issues in database centers in the recent years. The consciousness to turn towards energy-preserving technologies have put concepts like power-awareness into the spotlight. But especially databases lack the capability of managing the energy consumption while operating and past research solely focused on improving performance characteristics. This master’s thesis investigates potential modifications of the PostgreSQL query optimizer and executor to improve the overall energy efficiency of the query processing engine. For this, this study introduces an energy cost model on the optimizer level and the use of dynamic voltage and frequency scaling on the executor level. Additional concepts like deadlines are implemented to exploit the full range of functions and they are subsequently combined and used to maximize the positive impact of the algorithms designed for energy efficiency on the energy consumption. The extended framework is capable of reducing active energy costs significantly which is proven by applying the TPC-H benchmark and a concluding discussion of additional extensions including query scheduling shows that even further energy savings can be achieved.
Zusammenfassung

## Contents

1 Introduction 5  
1.1 Optimization potentials ................................. 6  
1.2 Related Work ........................................ 7  

2 Query planning in PostgreSQL 9  
2.1 Construction of query plans ................................ 10  
2.2 Cost calculation ....................................... 13  
2.3 Selection and execution of the optimal plan ............... 15  

3 Energy-efficient query planning 16  
3.1 Computing energy efficiency .............................. 16  
3.2 System power costs .................................... 18  
3.3 Possible energy savings .................................. 22  
3.4 Implementation ......................................... 28  
3.5 System configuration .................................... 32  
3.6 Effects on the TPC-H benchmark ......................... 33  
3.7 Visualization of query plan changes .................... 36  
3.8 Effects on query plans .................................. 40  

4 Dynamic Voltage / Frequency Scaling 46  
4.1 Internal power states .................................... 48  
4.2 DVFS and Linux ........................................ 50  
4.3 Experimental results with DVFS ......................... 51  

5 Combining planning and scaling 56  
5.1 Deadlines .............................................. 56  
5.2 Experiment design & results ............................. 61  
5.3 Complexity analysis .................................... 68  

6 Outlook 71  
6.1 Potential extensions .................................... 71  
6.2 Query scheduling ....................................... 72
1 Introduction

Performance is the primary keyword which comes to mind when we talk about the capabilities of database management systems (DBMS). The requirements to these systems are manifold and required years and decades of research and development to provide database systems as we use them today. Among the performance requirements are a low latency which means that queries are executed in a very short time. This is especially necessary for on-line transaction processing (OLTP) scenarios where hundreds and thousands of transactions like updating or inserting new tuples into a relation have to be executed in real-time [GR92]. Another requirement is scalability which means that with an increasing database size the system requirements increase linearly or even less to prevent performance deterioration. Over the years the processing time for complex queries declined continuously due to the improvements of hardware and software components like query optimization which was studied extensively in the past [Cha98].

As already indicated, an integral part of a high-performance DBMS is a sophisticated module for query processing, which includes algorithms for selecting and executing an optimal plan for a query. As mentioned in the beginning, these algorithms are trimmed towards performance only. However, recent studies indicate that a good performance is no longer the sole quality criterion for a ‘good’ DBMS. According to the Claremont report on database research [AAB+09] power-awareness of a DBMS will become one of the core research areas in the future and several reasons can be found that support the emerging importance of energy efficiency in this sector. First, energy costs are still rising and become a priority issue of many companies [CDQ+05]. Second, a high energy consumption is almost always correlated with a higher need for cooling in server environments [Saw04]. The cooling cost can even be larger than the remaining operation costs. Finally green computing has become an important marketing aspect for promoting sustainable practices within a company which was discussed by multiple management studies like [Shr95] and [Ban01]. The second argument is the most important one because a high energy consumption has many implications on the design of a data center. I already mentioned the correlation with cooling requirements which can easily double the energy costs for operating a server. Furthermore, if we look at server farms we have to recognize the overall architecture of the farm regarding space and location requirements of the units. If a server produces heat the cooling components have to make sure that the server itself stays cool and furthermore that the heat can dissipate somehow from the farm. Cooling limitations consequently result in increasing restrictions concerning the organization of the server inside a structure [RPS+02]. A Gartner study [Kum10] shows that the main issues in today’s data centers are space, power, and cooling problems and that these issues are expected to become more severe with a recovering economy and equipment sprawl.

One way to deal with these issues is to improve the energy efficiency of the data center when queries are executed. For this we now turn to a discussion of optimization potentials
and afterwards provide an overview about related work.

1 Introduction

1.1 Optimization potentials

To improve the energy efficiency during query execution one of two possible approaches come to mind: we can either reduce the time consumed or we can reduce the energy consumed for executing the query. The prior aspect has been exhaustively discussed in publications in past years and decades which led to the development of high-performance relational DBMS with System R being the basis for most of them [CAB+81]. The latter aspect was ignored in most studies but gets an increasing attention in recent research projects (the next section will elaborate on them) and will also be the focus of this thesis. Optimization potentials in this area arise for several reasons:

1. A system consumes a high amount of power while it is idling and only a small additional amount of power, called active power, when it is executing commands.

2. Components like the central processing unit (CPU) and the hard disk drive (HDD) consume different amounts of power.

3. A recent study by Barroso & Hölzle [BH07] predicts an increasing energy proportionality of system components. This means that the energy consumption will become more proportional to the utilization of the component. If a perfect proportionality could be achieved this means that if a component has a utilization of 50% its power consumption will be 50% (of the power consumed during peak load times) as well. As a result the energy efficiency degrades slower with a decreasing utilization level which creates the incentive of processing data on components with lower power requirements to maximize energy savings.

To exploit the potentials to the full extent and to reduce the complexity of the problem we make the following assumptions:

- The system runs at all times.

- The system is not running at high capacity.

Both assumptions are reasonable in the context of database centers which are accessible from multiple locations. Especially global companies which hold offices in many different time zones usually require access to their databases around the clock and even local companies use the night time for executing maintenance and batch jobs on their databases [Gra96]. In addition to that a significant overhead may be produced by the start-up and shutdown sequences of whole database servers. The feasibility of the second assumption
results from a publication by Barroso & Hölzle [BH07] who showed that servers rarely 
operate at peak and usually have utilization levels of 10 to 50 percent.

The facts in conjunction with the assumptions made can be used to reduce the energy 
consumption of the system by employing a query optimizer which produces plans which 
consume less energy than the standard plans and executing them with energy-saving 
settings.

The goal of this thesis is to implement an energy efficiency model which will be integrated 
into the query optimization process and to combine the query executor with a technique 
called dynamic voltage / frequency scaling (DVFS) which reduces the energy consumption 
during the execution of the selected query plan. Hereby we will focus on the open-source 
DBMS PostgreSQL operated on a desktop environment. Considerations about server 
environments will be discussed throughout the study as well.

The remainder of this thesis is organized as follows. The next section gives a broad 
overview of related techniques used for improving energy efficiency in database envi-
nronments looking on both, the hardware and the software side. Chapter 2 discusses 
how queries are generally processed in DBMS and in particular in PostgreSQL including 
the construction of query plans, the cost calculation, and the selection and execution 
of the optimal plan. Based on that chapter 3 will introduce techniques to improve the 
planning process towards the production of energy-efficient plans. Experiments involv-
ing the TPC-H benchmark will be explained to show the effects of the modifications. 
Chapter 4 discusses DVFS and the subsequent chapter evaluates the combination of 
both techniques, energy-efficient query planning and the employment of DVFS during 
query execution. The last chapter gives an outlook including a presentation of possible 
extensions.

1.2 Related Work

A general overview about energy efficiency considerations in the DBMS research area 
is provided by Harizopoulos et al. [HSMR09]. In addition to a motivation for consid-
ering energy efficiency in future research they mention potential research areas where 
improvements seem feasible like resource managers or new system architectures. Supple-
mentary insights about possible improvements can be found in the publications by Liu 
et al. [LWL+09], Graefe [Gra08], and Gounaris [Gou09]. Another study discussing the 
challenge of reducing energy costs in data centers was conducted by Poess & Nambiar 
[PN08]. In particular, they developed a power consumption model and showed consump-
tion trends based on existing TPC-C benchmarks. Various other publications discuss 
additional approaches to reduce the energy footprint in data centers like Lang et al. 
[LPN09] who consider leveraging data replications to reduce the energy consumption.
The development of an energy-efficient query optimizer which will be discussed in chapter 3 bases primarily on the work of Xu et al. [XTW10]. They propose a power model that can be calibrated to introduce considerations about energy consumption into the query optimization process. The model introduces several parameters into the cost functions for accessing single relations and join operations and adjust the parameters manually based on multiple test runs with different workloads. Subsequently, they obtain parameter values which settled at a stable level and provide a reliable means to predict the power consumption of the system. However their study is flawed since they draw the wrong conclusions from power consumptions\(^2\). While a low power consumption can be an indicator for power savings the overall energy consumption might be higher if the processing time increased too.

The idea of energy-efficient query optimization was also discussed by Alonso & Ganguly [AG92]. They evaluate the processing in both, a standalone and a client-server architecture, with the focus on the latter one. A higher complexity arises in this case due to the different environments used. While a laptop client will try to delegate the major processing operations to the server to conserve battery power, for the server the opposite approach seems feasible to maximize throughput. Their study evaluates optimization techniques to solve this dilemma.

The technique of dynamic voltage / frequency scaling which will be discussed in chapter 4 was discussed among others by Lang & Patel [LP09], Jiang et al. [JWX+09], and Pedram [Ped96]. Especially the first two publications evaluate the possible impact of using DVFS in a database environment while the third one studies the problem on the hardware level. The technique gained additional importance in the area of sensor networks [QWBH06], [GLG+05], [YG03] and thermal management of laptop and multicore environments [MD06].

\(^1\)Recently a supplement to this study has been published by Xu [Xu10] where the focus is put on the definition of formal control-theoretic methods to introduce power awareness into a DBMS.

\(^2\)In particular, they use the power consumption per tuple which is highly misleading for the following reason. A tuple is usually only processed once while a power consumption indicates the energy consumed in a particular time frame. Thus, the power consumption of a tuple (in Watt) indicates how much energy is consumed by the tuple per second while it should be an absolute value expressed by an energy consumption measure.
2 Query planning in postgresQL

Before we turn to the introduction of a model which is capable of making query processing more energy-efficient it is necessary to take a look at the current model, its goals, and a few relevant implementation details.

As with every other DBMS postgresQL tries to optimize query execution towards performance which means that it tries to select and execute the plan which is expected to have the lowest response time among all available plans for a query. The algorithm to do this consists of three elemental steps [GMUW08]:

1. Construction of query plans
2. Cost calculation
3. Selection and execution of the optimal plan\(^3\)

Figure 1 shows the workflow between the parts of the DBMS which are responsible for these steps with step 1 being done by the plan generator, step 2 by the performance cost calculator and step 3 by the plan evaluator and executor.

Figure 1: Interactions between the DBMS components involved in processing a query

The following sections discuss every step mentioned above as far as it relevant to the introduction of a modified query planning model in the next chapter.

\(^3\)Here the optimal plan denotes the plan chosen by the query optimizer. Exhaustive research in the past years and decades developed numerous techniques to improve the probability that the chosen plan is in fact the plan which executes fastest. Nevertheless it is possible that the query optimizer chooses an inferior plan sometimes. Common reasons are outdated statistics, falsely calibrated cost parameters, or uncaught correlations [Cha98].
2 Query planning in PostgreSQL

2.1 Construction of query plans

The PostgreSQL query optimizer performs a near-exhaustive search over the space of alternative strategies. This means that almost every plan which is capable of satisfying the specified query is analyzed in the optimization process. For example, consider the following query run against a database satisfying the specifications of the TPC-H benchmark:\(^4\):

\[
\begin{align*}
\text{SELECT} & \phantom{=}\text{DISTINCT} \ l_{\text{partkey}} \\
\text{FROM} & \phantom{=}\text{lineitem, orders} \\
\text{WHERE} & \phantom{=}l_{\text{orderkey}} = o_{\text{orderkey}}
\end{align*}
\]

The query accesses two tables defined in the TPC-H schema and returns all keys of parts which were ever shipped in an order. A valid query plan could consist of a two sequential scans iterating over both tables and joining elements together in a nested loop. As an additional feature the plan could also sort the set of result tuples after using one of the two named scan methods. Although sorting is not required by the query description it does not invalidate a plan and is often considered by the DBMS if it makes subplan processing (e.g. selection on a table) by a parent node (e.g. joining two tables) cheaper. Alternatively, index scans in conjunction with a merge join could be used to match the query description. In this case explicitly sorting the result would definitely increase the overall cost which leaves two reasonable plans to be checked\(^5\).

A plan in PostgreSQL is defined as a tree where the root node produces the final result. The child nodes have an operator type and further attributes (for example a column condition) required to match the query description. Dependencies between the nodes are represented by the tree structure. Figure 2 shows the tree for a valid plan of the query described above. Note that both scans\(^6\) can be executed in parallel.

The following list describes some of the operator (or node) types provided by PostgreSQL:

**Sequential scan** The relation is scanned sequentially. All tuples are fetched from the database by sequentially reading the corresponding pages and analyzed afterwards.

**Index scan** The relation is scanned by using an established index.

---

\(^4\)See http://www.tpc.org/tpch/ for further details.

\(^5\)Here, an additional aggregate node is required to fulfill the DISTINCT clause.

\(^6\)orders_pkey is an automatically created index on the primary key of the orders tables. l_orderkey_idx is an explicitly created index on the l_orderkey attribute of the lineitem table.
2 Query Planning in PostgreSQL

Figure 2: Sample plan tree produced by PostgreSQL for a query involving a join on two tables and a DISTINCT clause

**Bitmap scan** A bitmap is created containing tuple locations satisfying a query condition. A bitmap for multiple conditions is created by applying a bitwise AND/OR to the bitmaps for single conditions. The final bitmap contains the locations for the tuples satisfying all conditions and these tuples are subsequently fetched from the database and the conditions checked again.

**TID scan** The relation is scanned by reading the tuples specified in a constraint of the form `ctid = expression` or `expression = ctid`, i.e. when the tuple ID (TID) is explicitly used for defining the wanted tuple(s). This is different from using a sequential scan or index scan since the TID explicitly specifies where a tuple is located.

**Sort** A sort operation is executed on one or more columns of an existing result (for example result set from a scan).

**Aggregation** The aggregate operator is used when multiple values should be combined producing a new value for a group, for example the average of the values or the number of tuples in the group.

**Merge/Hash Join** Among other possibilities tables can be joined by using a merge join or hash join algorithm. The merge join requires that the tables to be joined are sorted on the join attribute. The hash join on the other hand does not have this
requirement but it needs a hash table on one of the two tables to find matching tuples in a short time.

**Nested loop** Another possibility to join two tables is the usage of a nested loop where the cross product is checked by running an (outer) loop on one table and an (inner) loop on the other table.

Further details about these and additional plan (or subplan) types are discussed for example in PostgreSQL guides by Douglas & Douglas [DD05] or Eisentraut & Helmle [EH08].

The types mentioned in the list reveal one aspect about the plan nodes: not every node type is substitutable by any other node type. Node types can be arranged in groups which are alternatives for each other under particular conditions. For example, the sequential scan can be replaced by an index or bitmap scan if an index exists on the conditioned column, or a TID scan if a TID expression is provided. In contrast to this there is no alternative for the sort operator. This is self-explanatory but it should be emphasized at this point that there is not an unlimited number of alternative plans for all queries.

Despite that fact it is possible that the optimizer has to check a large number of alternative plans. A representative example is a join between \( n \) tables with the same join attribute. Under the assumption that we only consider one join type we still need to check all possible \( n! \) join orders and as such the computational effort has a growth larger than any exponential function\(^7\). For that reason, PostgreSQL provides an alternative optimizer to the standard near-exhaustive optimizer: the genetic query optimizer (**GEQO**). GEQO was introduced by researchers from the Institute of Automatic Control at the University of Mining and Technology, in Freiberg, Germany, and solves the join order problem by employing a heuristic optimization method which operates through nondeterministic, randomized search. Since GEQO does not modify the cost calculation process it is not incorporated any further in this study. Additional information about GEQO and genetic algorithms in general can be found in the PostgreSQL documentation and in [BNKF98].

Altogether the following list names some of the common differences between plans for the same query:

**Scan type** The optimizer can choose between scan types like sequential, index and bitmap scan. If an index exists on at least one of the condition columns it is usually used if the selectivity of the query and/or the index is small enough to make fetching all tuples sequentially too expensive.

**Join type** As with different scan types the optimizer can also choose between different

\(^7\)The growth rate is actually somewhere in between that of exponential and double-exponential functions.
join types like hash join, merge join, or nested loops. Again the choice depends on additional constraints like requirements for sorting or table cardinalities.

**Join order** Theoretically joins can be executed in any order but the DBMS tries to keep the target cardinality as small as possible. Statistics are used to predict the cardinality of a particular join.

### 2.2 Cost calculation

Once a plan tree is constructed the corresponding performance cost can be calculated by using the *cost function* defined for each node type. In the typical case where the tree consists of more than one node the cost of the child nodes is used as an argument for the parent node’s cost function. Consequently, the total cost for executing the plan is associated with the root node. Hereby, the performance or time cost ($C_T$) for executing a node (ignoring the costs of child nodes) is computed as follows:

$$C_T = \sum_{\text{type}} n_{\text{type}} \cdot f_{T,\text{type}}$$

where $n_{\text{type}}$ is the number of operations of type $\text{type}$ required to execute the node and $f_{T,\text{type}}$ the performance cost factor for the same type. Table 1 gives an overview of the operation types defined in PostgreSQL, the components on which they are executed and the default values of the performance cost factors.

<table>
<thead>
<tr>
<th>type</th>
<th>Component</th>
<th>$f_{T,\text{type}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular tuple processing</td>
<td>tup</td>
<td>CPU</td>
</tr>
<tr>
<td>Indexed tuple processing</td>
<td>idx</td>
<td>CPU</td>
</tr>
<tr>
<td>Operator / function processing</td>
<td>op</td>
<td>CPU</td>
</tr>
<tr>
<td>Sequential page fetch</td>
<td>seq</td>
<td>HDD</td>
</tr>
<tr>
<td>Random page fetch</td>
<td>rnd</td>
<td>HDD</td>
</tr>
</tbody>
</table>

Table 1: Overview of the PostgreSQL’s operation types and associated system components default performance cost factors

It can be seen that disk operations are much more time-consuming than CPU operations but we have to keep in mind that one disk operation processes one page which may store multiple tuples depending on the row size in contrast to a cpu operation which processes only one tuple or expression.

---

8The variable $f_{T,\text{type}}$ has two index parameters with the first one indicating that it is a performance / time factor. Another factor will be introduced in chapter 3.
To compute the number of operations of each type cost functions are defined for every node type which use statistics about the database and cost estimates for child nodes to get an estimate for the startup and total performance cost (which is the sum of the startup and the runtime performance cost). The startup cost is hereby defined as the time consumed before a single output element can be produced. For example, for a sort operator this is the runtime cost of all child nodes since sorting can only be done when the whole dataset is available whereas an index scan has usually no startup cost since the first element can be accessed immediately. Table 2 shows a simplified version of the cost functions for selected operators (derived from findings of Xu et al. [XTW10]) with \( n \) being the number of loops, \( c \) being a constant and \( C_T(\text{outer-/inner-path}) \) being the cost estimates inherited from the child nodes (the outer and inner path specified for the join algorithm).

<table>
<thead>
<tr>
<th>Operator</th>
<th>Cost function ((C_T))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential scan</td>
<td>( n_{\text{seq}} \cdot f_{T,\text{seq}} + n_{\text{tup}} \cdot f_{T,\text{tup}} )</td>
</tr>
<tr>
<td>Index scan</td>
<td>( n_{\text{seq}} \cdot f_{T,\text{seq}} + n_{\text{idx}} \cdot f_{T,\text{idx}} + n_{\text{rnd}} \cdot f_{T,\text{rnd}} + n_{\text{tup}} \cdot f_{T,\text{tup}} + c )</td>
</tr>
<tr>
<td>Bitmap scan</td>
<td>( n \cdot (n_{\text{seq}} \cdot f_{T,\text{seq}} + n_{\text{idx}} \cdot f_{T,\text{idx}}) + n_{\text{rnd}} \cdot f_{T,\text{rnd}} + n_{\text{tup}} \cdot f_{T,\text{tup}} + c )</td>
</tr>
<tr>
<td>Merge join</td>
<td>( n \cdot (C_T(\text{outer-path}) + C_T(\text{inner-path})) )</td>
</tr>
<tr>
<td>Hash join</td>
<td>( C_T(\text{outer-path}) + n \cdot C_T(\text{inner-path}) + c )</td>
</tr>
<tr>
<td>Nested loop join</td>
<td>( C_T(\text{outer-path}) + n \cdot C_T(\text{inner-path}) )</td>
</tr>
</tbody>
</table>

Table 2: Overview of simplified versions of some cost functions used in PostgreSQL to compute the performance cost of query plans

A powerful tool for analyzing the cost factors of a query plan is the EXPLAIN statement in PostgreSQL. This statement produces a textual representation of the optimal query plan for a given query by displaying the nodes of the plan tree as well as additional information like the startup and total performance cost (first and second cost value), the expected cardinality (rows), and the row size in bytes (width) for each node. The textual representation for the plan produced for the query defined earlier (plan tree shown in figure 2) looks as follows (as mentioned earlier output cardinality and row size as well as overall startup and total cost are defined in the root node, i.e. in the first row):

```
HashAggregate (cost=124271.53..124860.04 rows=58851 width=4
  -> Merge Join (cost=1.98..119786.77 rows=1793902 width=4)
    Merge Cond: (orders.o_orderkey = lineitem.l_orderkey)
    -> Index Scan using orders_pkey on orders
      (cost=0.00..18583.79 rows=448500 width=4)
    -> Index Scan using l_orderkey_idx on lineitem
      (cost=0.00..77659.94 rows=1793902 width=8)
(5 rows)
```

The EXPLAIN command is an important part of the query analysis tool which will be introduced in chapter 3.7.
2.3 Selection and execution of the optimal plan

The last step in the query processing pipe is the selection, scheduling and execution of the optimal plan. The selection algorithm is relatively straightforward: the optimizer compares the cost values for the available plans and selects the plan with the lowest cost. Should two plans have the same total cost the plan with the lower startup cost is chosen. Once the optimal plan is selected it is executed immediately. At present there is no scheduling involved in the execution phase but the PostgreSQL source includes a scheduling agent which provides the opportunity to schedule jobs. However, this agent is not installed by default. Chapter 6.2 will discuss the energy savings potentials of using a query scheduler.
3 Energy-efficient query planning

The standard model has the purpose to maximize the performance of the DBMS or in other words to minimize response times when executing queries. In contrast to this approach the energy-efficient model which will be introduced in this chapter uses energy efficiency as an additional aspect which should be regarded in the query planning and execution process. Achieving a higher energy efficiency can be seen from two different vantage points. We can either reduce the time consumed for executing a query which is exactly what DBMS vendors are doing by increasing the performance of their systems or we can reduce the energy consumed while the process is running. Since the performance aspect has been discussed extensively in the past we will focus solely on the energy consumption aspect in the following sections.

3.1 Computing energy efficiency

To understand how a higher energy efficiency can be achieved it is important to understand how it is actually calculated. Initially we need to compute the power consumption of the system which can be done by using the following formula:

\[ P = V \cdot I \]  

\( V \) is the voltage the system is using and \( I \) the current drawn from the power supply unit. In this chapter we assume that both variables cannot be influenced by changing system settings (we will discuss the opportunities of increasing energy efficiency by using voltage and frequency scaling in chapter 4). Based on this formula we can compute the energy consumption of the system for a given time frame:

\[ E = P \cdot t \]  

Since the power consumption varies for every query plan and usually also during the execution of a query plan we need to modify the formula above by using the power function \( P(p, t) \) which returns the power consumption for a query plan \( p \) at a given moment in time. Hereby \( t = 0 \) is defined as the moment where the query execution starts but the function is not limited to the query execution time. After finishing the query execution the function will return the idle power consumed by the system. Subsequently the integral of the power function, where \([0, T] \) is the time frame (usually the runtime of plan \( p \)), gives the overall consumption:

\[ E(p, T) = \int_{0}^{T} P(p, t) \, dt \]  

The energy efficiency of a query plan \( p_1 \) compared to a different plan \( p_2 \) for the same query can now be computed by taking the quotient of the energy consumptions of both
3 Energy-efficient query planning

plans:

\[
\text{Eff} = \frac{E_1(p_1, T)}{E_2(p_2, T)}
\]

Note that both energy consumption functions get the same parameter value \( T \). Here \( T \) equals the maximum of the two runtimes of the query plans \( p_1 \) and \( p_2 \) and consequently the energy function of the query plan which finishes first returns the system’s idle power until the second query plan finishes as well. Assuming that \( T_1 \) is the runtime for query plan \( p_1 \), \( T_2 \) the runtime for query plan \( p_2 \) and \( T = T_1 = \max(T_1, T_2) \) (i.e. \( p_2 \) is the plan with the lower response time) we get the following formula for the energy efficiency:

\[
\text{Eff} = \frac{\int_0^{T_1} P(p_1, t) \, dt}{\int_0^{T_2} P(p_2, t) \, dt + (T_1 - T_2) \cdot P_{\text{idle}}}
\]

The metric charges the faster plan an additional energy cost for the time idling until the other plan has finished which is done based on the assumptions made earlier:

- The system runs at all times. As a result it still consumes power, the idle power, after completing a query.

- The system is not running at high capacity which means that on the one hand it consumes unused idle power between two query executions. On the other hand it means that as long as all queries can be executed within predefined constraints (for example deadlines) a lower response time only does not automatically influence the overall energy efficiency.

This allows the system to execute a plan with a higher performance cost as long as the energy consumption in the same time frame is lower compared to the plan employed by the standard query processor. However, we have to admit that this metric is not expedient for every system. As mentioned by Barroso & Hölzle [BH07], a mentionable portion of the server systems used in the world has a high utilization which interferes with the second assumption named above. In these cases the metric cannot be used to improve energy efficiency since the idle time after finishing a query is often used to actually execute the next query and can therefore not be traded for a lower overall energy consumption. Nevertheless, the average server has a utilization of 10 - 50% which means that even if the response time for queries doubles or sometimes even triples they can usually be executed with the predefined time constraints. To understand the applicability and usage of the metric consider the following scenario: A system executes a query periodically with an execution time of 10 seconds and a period of 30 seconds (which is also the deadline). Therefore the system is idling for 20 seconds per period. Now we assume that there is an energy-efficient plan for the same query which takes 10 seconds longer. The execution process for both plans is illustrated in figure 3 where the first row shows the periodic
3 Energy-efficient query planning

execution of the standard plan and the second row the periodic execution of the energy-efficient version. The metric would charge the standard plan the cost for executing the query (in 10 seconds) and an additional cost of idling for 10 seconds while the other plan is still executed. Now assume that the query planer finds an energy-efficient plan which takes 40 seconds instead. The execution process is illustrated in figure 4. In this case

the second assumption from above would be violated since the query can no longer be scheduled to finish within its deadline. The metric might still return a better energy efficiency for the slower plan but the idle time charged for executing the standard plan (in particular the last 10 seconds of the additional 30 seconds used for the energy-efficient plan) would actually be time used for executing the query when the next period starts. This invalidates the metric’s results. We will discuss later how the planning process can be modified to prevent situations like this one.

3.2 System power costs

To see a change in energy efficiency between two plans it is self-evident that one of the parameters for the formulas explained above needs to change. Earlier we explained that the power consumption of the overall system is calculated by adding up the consumptions of its different components and that the consumption of a single component varies based on the state of the system. In general, we are only interested in the power cost in idle mode, i.e. when no query is executed, and in active mode, i.e. when a query is executed.
To identify the power cost of different components the following experiment was executed for each relevant component:

- A Voltcraft VC-530 clamp meter was attached to the internal power line(s) leading to this component only. Disk drives like CD/DVD drives, or hard disks have separate serial ATA power connectors at which currents associated with these drives can be measured. CPU power is fetched from the +12V lines leading to pins 11 & 12 of the 24-pin ATX12V 2.0 power supply connector. The lines leading to the remaining pins of the power supply connector (except ground and negative lines) power other components like main memory, graphics card, or fans. Figure 5 depicts the basic power supply architecture of the system where rounded rectangles describe power consumers or supplier, rectangles describe connectors, and arrow lines stand for power lines directed from the supplier to the consumer.

- The current in idle mode was measured.

- The current in load mode was measured. The load was produced by creating a CPU utilization of 100% on one or both of the cores and constantly writing and reading data from the HDD.

Figure 5: Basic power supply architecture of the system used in the test environment

The first aspect of the experiment proved to be an obstacle since many internal components draw their power from the mainboard and it is impossible to extrapolate the
current running through the connectors between the mainboard and these components. However, the mainboard power lines (except the power lines leading to the CPU) showed no significant difference in the current value in idle and load mode and consequently a further distinction between the components on the mainboard was obsolete in this test environment. Table 3 shows the measured currents and resulting power consumptions for the system’s components (the voltage is fixed for each power line and the HDD and other components draw their power from multiple power lines exhibiting different voltages)\textsuperscript{9}.

<table>
<thead>
<tr>
<th>Component</th>
<th>( V )</th>
<th>( I_{\text{idle}} )</th>
<th>( P_{\text{idle}} )</th>
<th>( I_{\text{load}} )</th>
<th>( P_{\text{load}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU 1</td>
<td>12V</td>
<td>0.34A</td>
<td>4W</td>
<td>1.83A</td>
<td>22W</td>
</tr>
<tr>
<td>CPU 2</td>
<td>12V</td>
<td>0.34A</td>
<td>4W</td>
<td>1.33A</td>
<td>16W</td>
</tr>
<tr>
<td>HDD</td>
<td>3.3-12V</td>
<td>0.18-0.55A</td>
<td>11W</td>
<td>0.26-0.79A</td>
<td>16W</td>
</tr>
<tr>
<td>Other</td>
<td>3.3-12V</td>
<td>1.66-3.55A</td>
<td>50W</td>
<td>1.73-3.69A</td>
<td>52W</td>
</tr>
</tbody>
</table>

Table 3: Power costs of system components in idle and load mode

All values were divided by the performance coefficient of the power supply unit to get the total power drawn from the outlet. In the test environment a Dell L305P-03 power supply unit with an efficiency of 65\% was used.

Note that the second CPU consumes slightly less power when active compared to the first CPU\textsuperscript{10}. Principally this is caused by the decreased power required to drive signals external to the chip. Furthermore, some circuitry is shared between the cores, like the L2 cache and the interface to the front side bus (FSB)\textsuperscript{11}. Figure 6 illustrates how two CPU cores work together. To simplify later experiments and due to the fact that PostgreSQL is currently not capable of processing a single query using multiple cores, only one core will be incorporated for this study with a load power cost of 22 Watts.

Using the values from table 3 we can extract active power costs for each component, i.e. the power which is consumed when executing commands in addition to the power which is already consumed in the idle state. Table 4 shows the active power costs for the CPU, HDD, and the remaining components (omitting the second CPU core) as well as the resulting active power factor where a factor of 1 denotes the active power consumed by the HDD.

\textsuperscript{9} Additional components like a DVD drive or external hardware are not included in this discussion since they do not offer any relevance for this study. A DVD drive might be employed when using a DBMS, for example when loading data into a table, but it is up to the user to decide whether to use it or not.

\textsuperscript{10} The second CPU denotes in this case the CPU which is activated when the other one is already active. This is not necessarily a particular one in all cases. Consequently both CPUs on the chip can take the spot of the first or second CPU active depending on the activation order.

\textsuperscript{11} The shared components can differ between technologies. [ABC\textsuperscript{+}06] gives an overview about multicore architectures.
3 Energy-efficient query planning

![Diagram of a dual-core processor](image)

Figure 6: Diagram of a dual-core processor, with CPU-local level 1 caches, and a shared, on-die level 2 cache

<table>
<thead>
<tr>
<th>Component</th>
<th>$P_{active}$</th>
<th>Power factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>16W</td>
<td>3.2</td>
</tr>
<tr>
<td>HDD</td>
<td>5W</td>
<td>1.0</td>
</tr>
<tr>
<td>Other</td>
<td>2W</td>
<td>0.4</td>
</tr>
</tbody>
</table>

Table 4: Active power and power factors of the system components in the test environment

To match the calculated power consumptions of the individual components with the actual power drawn from the outlet a power meter is interposed between the outlet and the system’s power supply unit. Table 5 shows the measurements of the idle power, active power, and the full load power, which is the sum of the previous two values.

<table>
<thead>
<tr>
<th>$P_{idle}$</th>
<th>$P_{active}$</th>
<th>$P_{load}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>68.5W</td>
<td>21W</td>
<td>89.5W</td>
</tr>
</tbody>
</table>

Table 5: Idle, active, and full load power consumed by the system in the test environment (only one active CPU core)

Nevertheless, the values for active and full load power consumed in table 5 are somewhat diffuse for application scenarios. The full load power already indicates that this value is only applicable for full load situations, i.e. where the CPU is used at full capacity and data is constantly written or read from the HDD. The following list names the most important aspects influencing the power drawn from the outlet:
1. The power consumption of the CPU increases linear with its utilization. For example, in the used test environment the CPU consumes 0 Watts of active power when no commands are executed, 8 Watts of active power when the utilization is 50% and 16 Watts at full load\textsuperscript{12}.

2. The HDD consumes active power when the disk is spinning and data written or read from it. The cost for spin-up and spin-down is not included here since the system’s state is not considered in the query planning process and it is therefore not possible to know when the disk actually starts or stops to spin. However, the cost for changing the spinning speed can influence the power consumption of the disk to execute a query. For example, a query plan which reads from the disk sporadically may force multiple spin-ups and spin-downs to do so while a plan for the same query reading the same amount of data does this in a single read operation. In this case the latter plan would consume less energy since it only causes a single spin-up and spin-down. Other influences can be the access order of the data and the spinning speed at the moment where the query execution starts. We will later discuss the concept of energy-efficient query scheduling (see chapter ??) where spinning costs are a potential savings entrypoint.

3. Other components draw a varying amount of power depending on the system’s condition. For example, fans draw more power when they are spinning at a high speed. This can be caused by a highly cpu-bound query plan but it is also possible that high outside temperatures or work-intensive previous query executions caused the heat-up. As seen in table 4 these components contribute only a very small fraction to the active power consumption and improving energy efficiency for them is not discussed any further at this point.

3.3 Possible energy savings

The computation formula for energy efficiency and the power costs observed in the system used lead to the question how we can save energy or in other words improve energy efficiency. One answer to this question lies in the power factors of the different computer components (shown in table 4). The disk has a power factor of 1 whereas the CPU has a power factor of 3.2. Consequently the CPU consumes 320% of the power the HDD requires in the used test system. Rephrased this means that using the CPU is much more expensive than using the HDD which leads to the following statement:

\textsuperscript{12}Note that even when no commands are executed halt instructions (HLT) are issued to suspend CPU operation until an interrupt is received [Int10b]. The cost for executing HLT instructions is part of the idle power cost and is lower than the cost for executing any other instruction. Usually a higher CPU frequency is correlated with a higher number of HLT instructions which increases idle power at that frequency. However, the differences are not significant and therefore only one fixed value for the idle power is used throughout this study.
Theorem 1. If the CPU shows a higher power consumption than the HDD, a memory-bound query plan which needs a large time fraction for disk operations and a small time fraction for processor operations will provide a better energy efficiency than a CPU-bound query plan which executes in the same time and needs a small time fraction for disk operations and a large time fraction for processor operations.

Proof. According to formula 5 the energy efficiency for a memory-bound plan $p_1$ is higher than for a CPU-bound plan $p_2$ if the energy consumption for the former plan is lower than the energy consumption for the latter plan. It can be computed by multiplying the component costs with the respective usage times as defined by formula 3. For a plan $p_i$ we get the following formula for computing the energy consumption (a proof for this follows in conjunction with theorem 2):

$$E_i = E_{cpu} \cdot t_{i,cpu} + E_{hdd} \cdot t_{i,hdd}$$

Furthermore we have the following knowledge base:

- $E_{cpu} > E_{hdd}$
- $t_{1,cpu} < t_{2,cpu}$
- $t_{1,hdd} > t_{2,hdd}$
- $t_{1,cpu} + t_{1,hdd} = t_{2,cpu} + t_{2,hdd}$

The last formula can be rewritten as:

$$t_{1,hdd} - t_{2,hdd} = t_{2,cpu} - t_{1,cpu}$$

We can now prove that the assumption $E_1 < E_2$ is correct by solving for one of the formulae from the knowledge base which we know to be true:

$$E_{cpu} \cdot t_{1,cpu} + E_{hdd} \cdot t_{1,hdd} < E_{cpu} \cdot t_{2,cpu} + E_{hdd} \cdot t_{2,hdd}$$

$$E_{hdd} \cdot (t_{1,hdd} - t_{2,hdd}) < E_{cpu} \cdot (t_{2,cpu} - t_{1,cpu})$$

$$E_{hdd} \cdot (t_{2,cpu} - t_{1,cpu}) < E_{cpu} \cdot (t_{2,cpu} - t_{1,cpu})$$

$$E_{hdd} < E_{cpu}$$

The last formula is part of the knowledge base.

The theorem has the restriction that both query plans have to be executed within the same time but it is possible to soften this condition by including the definition of the energy efficiency metric introduced in formula 6. This means that a memory-bound query plan is still more efficient than CPU-bound plan as long as both plans can be executed within a particular time frame and the prior plan results in a lower energy consumption (including possible idle power) in this time frame. The following example is used to explain how savings can be achieved by choosing a plan with more disk operations and less processor operations.
Assume we have a query on a relation with 500,000 tuples which are stored in 265,000 pages on the disk and a query selectivity of 0.2 which means that we expect 100,000 tuples to be returned. The query contains a column condition which can be processed by using an existing index on the same unique column stored in 3,300 pages on the disk. To simplify the following discussion the table and index ordering are perfectly correlated\textsuperscript{13}.

The following statement which was already mentioned earlier could be a query matching the definition:

\begin{verbatim}
SELECT * FROM customer WHERE c_custkey > 1000
\end{verbatim}

p\textsc{sql} offers different access methods leading to different plans to process the query. Among them are the following two which will be inspected now (the remaining plan types are explained briefly in chapter 2.1):

**Sequential Scan** All tuples in the relation are read from the disk and processed sequentially by checking the condition for each tuple.

**Bitmap Heap Scan** The existing index is accessed and a bitmap is created encoding the tuple locations which satisfy the condition. Multiple conditions can be encoded by applying a bitwise AND/OR to the bitmaps produced for every single condition. The final bitmap will encode the tuple locations satisfying all conditions. These are then fetched from the disk and for all fetched tuples the conditions are rechecked.

Using \textsc{sql}'s standard parameters for the cost to process a tuple on the CPU (which is 0.01), to process an indexed tuple on the CPU (which is 0.005) and the cost to fetch a page from the disk sequentially (which is 1.0) the cost items shown in table 6 arise for both scan modes\textsuperscript{14}.

It is recognizable that the sequential scan is cheaper than the bitmap heap scan after looking at the total cost produced with the standard cost model. Since \textsc{sql} (like every other DBMS) is trimmed towards producing plans with the highest performance the total cost can be seen as measure of time consumed. Although the value cannot be simply mapped to seconds or any other time unit the values can be used to compare the time required to execute a plan. In this case we can conclude that the sequential

\textsuperscript{13}In this case all pages can be accessed sequentially to get the required tuples. The number of pages fetched is computed by using a formula proposed by Mackert \& Lohman [ML89] and is here equal to the number table pages. With a smaller selectivity the number of fetched pages may decrease. Normally the calculation of the access cost depends on the correlation between the index ordering and the table ordering and assumes random page accesses. The total cost is then determined by interpolating between the minimum and maximum access cost.

\textsuperscript{14}The cost evaluation process is much more complex than the results in the table suggest. To simplify the example, costs for evaluating restrictions, modifying the bitmaps, etc. are not included here. They would slightly change the overall cost but have only little influence on the energy efficiency comparison.
3 Energy-efficient query planning

<table>
<thead>
<tr>
<th>Item</th>
<th>Sequential scan</th>
<th>Bitmap heap scan</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$n$</td>
<td>$f_T$</td>
</tr>
<tr>
<td>Fetch index pages</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Process index pages</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Fetch tuple pages</td>
<td>26500</td>
<td>1</td>
</tr>
<tr>
<td>Process tuple pages</td>
<td>500000</td>
<td>0.01</td>
</tr>
<tr>
<td>Total</td>
<td>31500</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 6: Cost items for a sequential scan or bitmap heap scan to answer the query specified above using the standard cost model (simplified) showing the number of operations ($n$), the processing cost/time factor ($f_T$), and the performance cost ($C_T$).

scan executes faster than the bitmap heap scan. But this does not automatically mean that the sequential scan is also more energy-efficient than the bitmap heap scan. Table 7 shows the estimated number of CPU and disk operations used to execute the query.

<table>
<thead>
<tr>
<th></th>
<th>Sequential scan</th>
<th>Bitmap heap scan</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU operations</td>
<td>500000</td>
<td>350000</td>
</tr>
<tr>
<td>Disk operations</td>
<td>26500</td>
<td>29800</td>
</tr>
</tbody>
</table>

Table 7: Estimated number of CPU and disk operations to execute a sequential scan or bitmap heap scan to answer the query specified above.

Since the bitmap heap scan uses significantly less CPU operations but only a slightly higher number of disk operations, according to theorem 1 it is likely that the bitmap heap scan is in fact more energy-efficient than the sequential scan. But disk operations take significantly longer than CPU operations and need to be multiplied by a matching time factor ($f_T$) to compute the actual execution cost. In the standard cost model this factor equals the processing cost for a tuple, an indexed tuple, or an internal operation, or the cost for accessing a page in sequential or random order. As mentioned earlier, the standard model returns a relative measure of time ($C_T$) for the cost evaluation of each query plan:

\[
C_T = \sum_{type} n_{type} \cdot f_{T,type}
\]

(7)

\[type \in \{\text{tup, idx, op, seq, rnd}\}\]

Based on this formula (see chapter 2.2 for a detailed explanation of the formula) and formula 3 which was introduced earlier a theorem for computing an energy consumption cost in a DBMS can be compiled:

15Processing indexed tuples on the CPU are only counted half since the cost for processing them is half the cost for processing regular tuples.
**Theorem 2.** A relative measure for the energy consumption of a query plan can be established by multiplying the time measure for each component used with the power factor of the respective component.

*Proof.* This theorem is directly derived from the general formula for computing the energy consumption (defined by formula 3). $P$ is mapped on the power factors and $t$ on the time measures for the respective components. \(\square\)

Using the power factors ($f_P$) of the system components this means that CPU operations are multiplied by the processing time for the operation and the power factor of the CPU ($f_{P,CPU}$), and disk operations are multiplied by the access time for the operation and the power factor of the HDD ($f_{P,HDD}$). The power factors (where the second index of the factor variable stands for the cost type) and the resulting cost function computing the energy cost $C_E$ are defined as follows:

\[
\begin{align*}
    f_{P,tup} &= f_{P,idx} = f_{P,op} = f_{P,CPU} \\
    f_{P,seq} &= f_{P,rnd} = f_{P,HDD} \\
    C_E &= \sum_{type} n_{type} \cdot f_{T,type} \cdot f_{P,type}
\end{align*}
\]

(8) (9) (10)

Theorem 2 and formula 10 are the integral building blocks of a cost model for energy efficiency. Applying the formula with the values from table 4 to the values in table 6 the energy efficiency between the sequential scan and the bitmap heap scan from the earlier example can be determined. Table 8 shows the results.

<table>
<thead>
<tr>
<th>Item</th>
<th>Sequential scan</th>
<th>Bitmap heap scan</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$n$</td>
<td>$f_{T+P}$</td>
</tr>
<tr>
<td>Fetch index pages</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Process index pages</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Fetch tuple pages</td>
<td>26500</td>
<td>1</td>
</tr>
<tr>
<td>Process tuple pages</td>
<td>500000</td>
<td>0.032</td>
</tr>
<tr>
<td>Total</td>
<td>42500</td>
<td>41000</td>
</tr>
</tbody>
</table>

Table 8: Cost items for a sequential scan or bitmap heap scan to answer the query specified above using the energy-efficient cost model (simplified) showing the number of operations ($n$), the product of the time factor and power factor ($f_{T+P}$), and the energy cost ($C_E$).

It can be seen that the bitmap heap scan provides a lower energy cost (and therefore also a higher energy efficiency) than the sequential scan. Figure 7 shows a comparison of the cost development for both scan methods using the performance cost function and the energy cost function. What we get out of this figure is that while the cost function for the sequential scan is a constant because the processing cost does not depend on the
selectivity here, the cost function for the bitmap heap scan is steeper for the energy cost than for the performance cost (due to the power factors). As a result the bitmap heap scan is cheaper for a low selectivity.

Figure 7: Comparison of the cost development depending on the selectivity of the query and the usage of (a) the performance cost function or (b) the energy cost function

Nevertheless, we have to keep in mind that the lower energy cost is not correlated with a lower performance cost. The sequential scan would in this case still be the faster option.

Before we turn to the discussion of the implementation of the energy-efficient cost model we want to present two scenarios where a higher energy efficiency is actually viable:

- Overnight query processing introduces the possibility for trading performance for energy efficiency. Batch processes can be started in the evening and results are ready in the morning with the optimization potential of choosing a plan which executes within specified time constraints, i.e. that the results are available in the morning.

- Depending on the cooling requirements for the CPU or HDD processing can be relocated to either of the components if necessary. On desktop machines the CPU usually requires the highest cooling effort whereas servers can require more fan activity to dissipate the heating caused by disk operations. Altogether, a reduced need for cooling would also lead to a higher energy efficiency.

\[\text{Currently overnight batch processing is usually used to execute UPDATE transactions to prevent interference with daytime query executions [Gra96].}\]
In general a higher energy efficiency can be achieved by employing a memory-bound query plan instead of a CPU-bound query plan as long as theorem 1 is not violated.

3.4 Implementation

The cost model for energy efficiency changes the methods for calculating the cost of a query plan as well as the decision process in the plan evaluator. But instead of completely replacing the cost functions used for computing performance costs, the startup and total energy cost for a plan is computed and stored separately. On the one hand this allows the evaluator to weight between performance and energy costs and not strictly use either the performance or energy cost model. On the other hand it gives the opportunity to compare the most performant with the most energy-efficient plan (this aspect will be discussed later). Figure 8 shows a modified version of the workflow of the standard query processor (see figure 1) incorporating the cost model for energy efficiency.

Like the performance factors described in chapter 2 which are implemented in PostgreSQL as double variables, the power factors were added to the classes responsible for calculating the cost of a plan. The cost functions were modified such that they compute the startup and total performance cost as well as the startup and total energy cost. This was achieved by incorporating formula 10 into the cost functions. Based on additional parameters which can be set by the user the plan evaluator weights all cost values and compares the overall cost values \( C \) to extract the optimal plan.

\[
C = w_T \cdot C_T + w_E \cdot C_E \tag{11}
\]
Energy-efficient query planning

<table>
<thead>
<tr>
<th>Parameter</th>
<th>postgresQL variable</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_{P,CPU}$</td>
<td>cpu_power_cost</td>
<td>3.2</td>
</tr>
<tr>
<td>$f_{P,HDD}$</td>
<td>disk_power_cost</td>
<td>1</td>
</tr>
<tr>
<td>$w_T$</td>
<td>performance_cost_weight</td>
<td>1</td>
</tr>
<tr>
<td>$w_E$</td>
<td>energy_cost_weight</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 9: Parameters introduced with the cost model for energy efficiency, their complements in the postgresQL extensions, and the default values

$w_T$ and $w_E$ denote the weights for the time and the energy cost respectively\(^ {17}\). The standard query processor can be simulated by using $w_T = 1$ and $w_E = 0$.

Table 9 lists the additional parameters, their complements in the extended postgresQL implementation, and the default values. The default values for the power factors are drawn from table 4 and energy costs are initially ignored by the evaluator.

To make these variables accessible by the user they were added to the grand unified configuration (GUC) tables. The elements in these tables are visible to the user and connected with external C variables used in the program definition. The default values are specified in the C code but the user can change and look up these values at any time by using the \texttt{SET} and \texttt{SHOW} commands respectively. Their syntax is defined as follows:

\begin{verbatim}
SET <postgresql variable> TO <value>;
SHOW <postgresql variable>;
\end{verbatim}

However, these parameters are set separately for each connection to the DBMS. Consequently the \texttt{SET} commands have to be issued every time a new connection is employed to answer a query if the default values should not be used. In addition to that it is possible to change the default values by including GUC variable assignments in postgresQL’s configuration file\(^ {18}\).

It is easy to forget to set the parameters once in a while for which reason an alternative approach for modifying them has been implemented. Instead of setting the parameters in advance of a query statement the configuration can be coupled with the query itself by defining a block containing the settings. For postgresQL this was realized by introducing an \texttt{OPTIONS} block in which cost parameters (and additional planning variables which will

\(^{17}\)Like all weights they should be non-negative numbers and sum up to 1 although this constraint is not a necessary condition. Theoretically negative weights could be assigned which would mean that a plan with a higher cost is preferable.

\(^{18}\)This file should already include the default assignments in a commented version. Thus, they are not used during the initialization (the default values defined in the C source are used which should be the same) and are simply there to give the user an overview of the accessible variables and their default values.
be introduced later) can be specified\textsuperscript{19}. The syntax for a \texttt{SELECT} statement is extended as follows:

\begin{verbatim}
SELECT [...] OPTIONS <option> [AND <option> ...];
\end{verbatim}

The \texttt{OPTIONS} block contains one or more of the following clauses separated by the \texttt{AND} keyword:

- \texttt{CPU POWER COST = <value>}
- \texttt{DISK POWER COST = <value>}
- \texttt{PERFORMANCE WEIGHT = <value>}
- \texttt{ENERGY WEIGHT = <value>}

To handle the syntax modifications when executing the query a pre-parser was implemented which extracted the optional \texttt{OPTIONS} block, processed all settings and forwarded the remainder of the query to the actual query processor. To avoid major modifications of the PostgreSQL source hooks were used to reroute control from the query processing routine to a plugin, called the energy plugin, and back. This plugin contains among other functionalities the pre-parser. A hook is a code stub left in the source code to support a future enhancement. In the case of PostgreSQL hooks were placed at different points of interest, for example in the routine which is called when a query plan is executed, where developers sensed that additional contributions in the form of plugins could enhance the system. This is done by defining function pointers which can be modified externally. By default, these are null pointers and as a consequence the default function is executed. If the pointer actually points to a function (called the hook function), control is diverted to the hook function. The hook function can either completely replace the behavior of the default function or execute commands and call the default function afterwards in which case the hook function acts as a preprocessor. For the pre-parser an additional hook was placed in the routine processing all incoming commands and if the pre-parser is installed control is redirected to it when a query has to be processed. Once the options are extracted from the query control is forwarded to the original query parsing routine which evaluates the query syntax and organized further processing steps. Figure 9 depicts the initial steps of processing a query depending on the availability of the pre-parser. The plugin control is defined by the components on the right side of the dashed line.

As soon as the query execution is finished all settings are set back to their previous values. This behavior is achieved by using another hook which is placed in the routine which is called as soon as query execution ends.

\textsuperscript{19}The \texttt{OPTIONS} keyword is already used in some DBMS implementations. However it is not necessary for the purpose of this thesis to define a syntax which can be employed without interfering with any SQL syntax implemented in any DBMS.
Figure 9: Petri net model describing the initial steps of processing a query depending on the availability of a pre-parser.

As mentioned earlier plan trees including statistics like cardinalities and performance costs can be examined by using the `EXPLAIN` statement. Since the output included only performance costs but no data about energy costs (because they did not exist before modifications were made) these cost statistics were added to the textual representation. A sample output (for the query used in chapter 2.1) is shown below.
3 Energy-efficient query planning

HashAggregate (cost=124271.53..124860.04 energy=309268.10..311622.14
   rows=58851 width=4)
   -> Merge Join (cost=1.98..119786.77 energy=4.34..291329.08
      rows=1793902 width=4)
      Merge Cond: (orders.o_orderkey = lineitem.l_orderkey)
      -> Index Scan using orders_pkey on orders
         (cost=0.00..18583.79 energy=0.00..38767.04 rows=448500
            width=4)
      -> Index Scan using l_orderkey_idx on lineitem
         (cost=0.00..77659.94 energy=0.00..158386.28 rows=1793902
            width=8)

3.5 System configuration

To test the effects on system performance and energy efficiency when employing the energy cost model a test environment was set up which was also used for further extensions explained in the upcoming chapters. The main component of the system is a desktop machine. The abundance of different system configurations makes it impossible to choose one particular system which depicts the typical machine a DBMS is operating on but there is at least the distinction between desktop and server machines which was mentioned earlier. However, this thesis only employs a desktop machine and did not use a server environment for testing due to funding, space and time constraints but we will discuss possible impacts of the energy cost model in server environments in conjunction with the results for the desktop environment.

The desktop machine has the following properties:

- The machine used is a Dell Optiplex 780 with an Intel Core 2 Duo processor E7500 with a maximum clock speed of 2.9GHz for each core, 3MB cache, and a front side bus frequency of 1066MHz\(^{20}\). The system is capable of using the Intel SpeedStep technology which allows the modification of the chip’s frequency and input voltage. This aspect is required for the dynamic frequency / voltage scaling approach which is discussed in chapter 4.

- The storage component used is a Western Digital Caviar Blue hard disk drive with a storage capacity of 160GB, a cache size of 8MB, a rotational speed of 7200RPM, and a SATA interface capable of transferring up to 3GB per second\(^{21}\).

A description of other system components is omitted at this point since it poses no or only

\(^{20}\) For further details see: http://ark.intel.com/Product.aspx?id=36503

\(^{21}\) For further details see: http://www.wdc.com/en/products/products.asp?driveid=254
minor relevance for the experiments conducted. For recording the power consumption of the machine and its components two different devices were used:

- A Voltcraft VC-531 clamp meter was used to record the currents on separate power lines sustaining the components of importance. In all cases the conductors with the positive currents were used to get readings for the currents to avoid that different current flows cancel out each other. Due to the extremely short conductors used in the machine it was not possible to wind the conductor around the clamp several times which would have provided a more accurate reading for the low currents at work.

- A watts up? PRO power meter was installed between the power outlet and the power supply unit of the machine to record the power consumption of the whole system. A new value is generated every second and can be read via a USB interface. In all test cases a script was used to establish a connection to the power meter before the test is executed, to read the power consumption values in one-second intervals, and to write them to a log file. After the test is finished, the runtime of the test is used to compute the energy consumption and the average power consumption.

The database environment is defined by two different TPC-H databases populated with generated datasets of 1GB and 10GB total size respectively. These datasets were generated automatically by the DBGEN tool provided by the TPC. The schema and the indices created are predefined and both, the database creation and population, was done by a shell script. To avoid caching effects, i.e. lower response times when executing the same query in succession since the data is (partly) still in main memory, each database was replicated multiple times.

### 3.6 Effects on the TPC-H benchmark

To test the effect of using the energy cost model the following experiment was conducted:

- The whole TPC-H benchmark consisting of 22 queries was executed on both, the 1GB and 10GB database, using either the performance or energy cost model.

- Each configuration (combination of the database and cost model used) is used in a different test series consisting of 10 runs each. A test series using the performance cost model will subsequently be tagged as a performance test series and a test series using the energy cost model as an energy test series.

- Each test was executed using a different database replication to avoid caching effects. After all replications were used once the server was restarted to invalidate
3 Energy-efficient query planning

cached pages.

- The power consumption was recorded in one-second intervals as well as the total runtime of the whole benchmark. The resulting energy consumption, average power consumption of a run, and runtime were averaged among all runs in a test series.

Table 10 shows the results for the two test series which used the 1GB database. Columns 2 to 4 provide the average runtime, the average total and active power consumption. The last two columns specify the total and active energy savings when using the energy cost model instead of the performance cost model.

<table>
<thead>
<tr>
<th>Cost model</th>
<th>\langle t \rangle</th>
<th>\langle P_{\text{total}} \rangle</th>
<th>\langle P_{\text{active}} \rangle</th>
<th>\langle E_{\text{total}} \rangle</th>
<th>\langle E_{\text{active}} \rangle</th>
<th>\langle \text{Sav}_{\text{total}} \rangle</th>
<th>\langle \text{Sav}_{\text{active}} \rangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>214s</td>
<td>84.5W</td>
<td>16.0W</td>
<td>19658J</td>
<td>3424J</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Energy</td>
<td>237s</td>
<td>81.0W</td>
<td>12.5W</td>
<td>19197J</td>
<td>2962J</td>
<td>2.3%</td>
<td>13.5%</td>
</tr>
</tbody>
</table>

Table 10: Average runtime, total and active power consumption results and energy savings for the two test series which used the 1GB database

Note that the runs in the performance test series are charged idle power for the differential between the average runtime of the performance and energy test series. This concurs with the definition of energy efficiency described in section 3.1. The results in the table indicate that the average total power consumption is only marginally different between the performance and energy test series. Due to the relatively large idle power consumed this is expected and we have to take a closer look at the active power consumptions. The absolute difference between the values is logically the same but the relative difference is much larger. When using the energy cost model an average 13.5% of the active energy consumed was saved in contrast to the test series employing the performance cost model. In the same instance the total time consumed for finishing the benchmark increased by only 10.7%. This means that even if we would weight the results of the performance and energy cost model equally, i.e. we want to choose the optimal plan in terms of equal shares between performance and energy efficiency, improvements of the default query processor are possible.

To get a better impression of the development of the power consumption during a benchmark, the curves for two sample runs from the test series with the configurations using the 1GB database are shown in figure 10. The x-axis denotes the progressing runtime of the benchmark and the y-axis the total power consumptions at particular moments in time. Since the power meter provides discrete observations in one-second intervals interpolation between neighboring values was used to draw the curve. Note that the range of the y-axis starts at 65W (which will be the case in all figures depicting total power consumptions during benchmark runs). Since the idle power is already 68.5W the total power consumption will never drop below that value and since the differences between the energy and performance test series are the important aspects it is suitable to omit the lower range.
The run using the performance cost model is represented by the solid line and the run using the energy cost model by the dashed line. Both runs show fluctuations in the power they consume which is caused by disk activity but even more importantly by changes in the CPU utilizations. As mentioned earlier, the CPU power consumption increases linear with its utilization. As a consequence queries which require significant CPU processing cause the consumption curve to peak while queries which are memory-bound and only need limited CPU resources will effect the power consumption to drop.

The results for the runs using the 10GB database are similar and show no noticeable differences. For the remainder of this thesis we will therefore only use the 1GB database in the experiments.

Until now we only looked at the results for the whole benchmark without analyzing the processing for each query separately. But plan changes are ultimately done for single queries without looking at the bigger picture (in this case without looking at the other queries which have to be processed as well). For that reason the next step is to discuss the effects on the query level but to do this a means for making the comparison easier and more visible will be introduced first in the next section.
3 Energy-efficient query planning

3.7 Visualization of query plan changes

PostgreSQL provides the facility to evaluate a query plan by using the `EXPLAIN` statement which produces a textual output of the plan tree including cost, cardinality, and output tuple size for each node (see chapter 2.2 for an example). However, it is complicated to compare two plans since the plan tree often becomes very intricate for complex queries (which is the case with the TPC-H benchmark queries). A remedy has been provided by Reddy and Haritsa [RH05] who introduced the Picasso tool in 2005. This tool is capable of comparing query plans which depend on selectivity parameters, i.e., column conditions which cause a variable output cardinality of queries or subqueries. As an output they produce plan diagrams where each color denotes a unique plan and in addition to that plans can be inspected and compared. Unfortunately, it was not possible to utilize the software for the purpose of this thesis for several reasons:

- The selectivity parameters have to be associated with table columns. This means that parameters used by the cost model like the power costs or weights cannot be used as variables.

- The user can only define resolutions and relative selectivity ranges (between 0 and 100 percent) but no absolute ranges which is necessary for the cost model parameters. The relative ranges are translated to absolute values by using histograms of the respective table columns.

- Picasso uses the textual outputs produced by `EXPLAIN` statements to create internal and graphical representations of the plan trees. Logically, the software is not able to parse the output once energy costs are also returned by `EXPLAIN`.

The third problem is solvable with a relatively small effort, but the other two issues require major changes of the Picasso architecture. For that reason, we implemented a visualization tool from scratch incorporating some of the concepts used in Picasso. Self-evidently, this tool does not have the same variety of functions as Picasso but it includes all components necessary for comparing query plans with different cost model parameters. This makes it possible to compare plans produced by the performance and energy cost model. The basic use case of the tool is designed as follows:

1. The client connects to the database using the settings provided by the user.

2. The SQL query is specified by either entering it via the keyboard or by loading it from a plain text file.

3. The variable parameters are defined by modifying the query. This is done by assigning either ":x" or ":y" as the value for a parameter constraint or definition. The different values will later be projected on the x- or y-axis of the plan diagram.
4. Both variable ranges and resolutions (interval between neighbor values) are defined. By default the variable values will range from 1 to 10 with a resolution of 1, i.e. all natural numbers between 1 and 10 will be used as values. A screenshot of the tool’s query view where the first four steps are performed is shown in figure 11.

5. The query is executed for all value combinations of the two variables. Unique plans will be assigned an ID and a plan array will be created where each cell corresponds to a value combination and contains the ID of the plan returned by the query optimizer for this value combination.

6. The plan diagram is defined as a graphical representation of the plan array which is done by using a grid where each cell corresponds to a cell of the plan array and the color is determined by the ID stored in the respective plan array cell\textsuperscript{22}. As a result value combinations which induce the same query plans will have the same cell color in the grid. A sample plan diagram is shown in the left part of figure 12.

7. The unique plans can be further analyzed by inspecting the plan tree. A sample plan tree visualization is shown in the right part of figure 12. For producing the graph representation the prefuse visualization toolkit [HCL05] was used\textsuperscript{23}.

8. Plans can be compared in a separate view. For both plans the corresponding plan tree will be outlined where red nodes represent differences between both plans. In addition to that statistics and a cost comparison is provided. The view for comparing plans including a sample comparison is shown in figure 13.

A plan diagram as shown in the top left part of figure 12 signals immediately whether plans for a query change under varying conditions. In this case the CPU power cost ($x$-axis) and disk power cost ($y$-axis) were used as variables. The values were in the range between 1 and 10. After trying all value combinations the query optimizer produced a total of three different plans, each one identified by a different color in the plan diagram. The first plan (red) is returned for the majority of value combinations, the second plan (dark red) for a low CPU power cost and a high disk power cost, and the third plan (green) in the cases where the CPU power cost is significantly higher than the disk power cost.

\textsuperscript{22}The colors do not characterize the query plans. The first plan returned by query optimizer will always have the ID 0 and as such assigned a red color. Consequently each run with a possibly different query will produce at least one red cell which is not related to red cells created in previous runs.

\textsuperscript{23}See \url{http://www.prefuse.org} for further details.
3 Energy-efficient query planning

Figure 11: Screenshot of the query view of the visualization tool

Figure 12: Screenshot of the view showing the plan diagram and details
Figure 13: Screenshot of the view for comparing plan trees
3.8 Effects on query plans

Now that we established a means to graphically compare query plans under varying conditions the effects of the energy cost model on the queries in the TPC-H benchmark can be analyzed. Initially the following experiment was conducted:

- The plan diagrams for each TPC-H query were created by using the previously introduced tool. Query 15 was not included in this or any of the following experiments since it actually contains three statements for creating a view, executing the actual query, and deleting the view afterwards. The visualization tool was only conceived for SELECT statements and does not allow the execution of view creation or deletion statements.

- The energy awareness parameter was projected on the $x$-axis. No variable was projected on the $y$-axis\(^24\).

- The variable takes either the value 0 or 1 which translates to using the performance or energy cost model. Consequently the plan diagrams in this experiment consist of two cells only.

- The CPU and disk power cost where set to the power factors calculated for the test system (see table 4).

Figure 14 shows the simple plan diagrams for the TPC-H queries under the experiment conditions. The captions identify the queries associated with the diagrams. Logically there are only two possibilities how the diagrams can look like:

1. The cells have the same color (red). This means that the same plans are produced when using either, the performance or energy cost model.

2. The cells have different colors (red and dark red). This means that the performance and energy cost model come up with different optimal plans.

It can be seen that the query optimizer produces a different plan in six cases only, in particular for queries 2, 3, 5, 7, 10, and 17. The differences manifest itself in alternative subplans, different join orders, the usage of index or bitmap scans instead of sequential scans, and the use of other join types (here nested loops or hash joins). Since these aspects were already discussed earlier in this chapter a detailed discussion of the plan changes is omitted here. Instead we turn to the more important aspect of the cost development. Table 11 shows a comparison of the performance costs ($C_T$) and energy costs ($C_E$) for

\(^{24}\)The tool actually requires a variable setting for both dimensions. To fulfill this requirement a dummy variable was placed in the query with a range of only a single value which does not influence the query processing.
the plans chosen for the queries named above. We compared the performance plan which is the plan chosen when optimizing for performance cost with the energy-efficient plan which is the plan chosen when optimizing for energy cost. In addition to that the table includes the percentual differences between the cost values. Positive numbers indicate additional costs when using the energy-efficient plan whereas negative numbers indicate savings.

<table>
<thead>
<tr>
<th>Query</th>
<th>Performance plan</th>
<th>Energy-efficient plan</th>
<th>Diff$_T$</th>
<th>Diff$_E$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$C_T$</td>
<td>$C_E$</td>
<td>$C_T$</td>
<td>$C_E$</td>
</tr>
<tr>
<td>2</td>
<td>116324</td>
<td>185898</td>
<td>116613</td>
<td>185204</td>
</tr>
<tr>
<td>3</td>
<td>677999</td>
<td>1447163</td>
<td>680708</td>
<td>1442297</td>
</tr>
<tr>
<td>5</td>
<td>323942</td>
<td>594227</td>
<td>453274</td>
<td>588757</td>
</tr>
<tr>
<td>7</td>
<td>276754</td>
<td>444028</td>
<td>278632</td>
<td>442269</td>
</tr>
<tr>
<td>10</td>
<td>417951</td>
<td>737877</td>
<td>419290</td>
<td>722768</td>
</tr>
<tr>
<td>17</td>
<td>805464</td>
<td>823726</td>
<td>814025</td>
<td>831256</td>
</tr>
</tbody>
</table>

Table 11: Comparison of the predicted performance and energy costs for selected query plans produced when using either of the cost values for determining the optimal plan

For all but the last query in the table the query optimizer produces more energy-efficient plans when using the energy cost as the optimization criterion. On first sight it is counterintuitive that the energy-efficient plan for the last query is actually less energy-efficient. A closer look at the plan trees for this query reveals that this is caused by using a subplan.
which is more energy-efficient than the subplan used in the default case but slightly increases the overall energy cost. As noted earlier, the plan search is not fully exhaustive in PostgreSQL and minor discrepancies between the cost of the optimal plan and the plan chosen are not checked at all times. The next piece of information which can be extracted from the table is that the predicted savings are smaller than the actual savings recorded when executing the benchmark (see table 10). This could have been triggered by several causes. First, the cost parameters including the performance cost parameters like the cost for accessing a page on the disk or a CPU operation might be inaccurate. It is not trivial to find the optimal settings for the parameters and the process requires intensive fine tuning. Second, the cost functions themselves as provided in PostgreSQL might be inaccurate. A specific scenario will be discussed later in this chapter. A final insight gained from the table is that performance and energy cost are not necessarily correlated. The energy-efficient plan for query 5 provides mediocre energy savings of only 0.9% but causes a 39% increase in performance cost. Deductively limits should be defined to prevent choosing a plan which has a much higher performance cost for a small energy cost gain. Basically this can be achieved by weighting between performance and energy cost.

An evaluation of the actual savings on the query level will follow in chapter 5 where the optimizations of the query planner and executor will be combined. We omit a discussion of feasible experiments at this point since this thesis is not targeted at evaluating differences between cost predictions and actual costs. In the optimal case we would expect that the predicted and actual costs are roughly the same.

We have seen how the query planning process for the TPC-H benchmark is affected by the energy cost model. However this was only done for the power cost factors specific to the test system. To get a better impression if and how plans change depending on the power consumption of the CPU and HDD another experiment was designed:

- The plan diagrams for each TPC-H query were created by using the previously introduced tool.
- The CPU power cost was projected on the x-axis, the disk power cost on the y-axis.
- Both variable values ranged from 1 to 10 with an interval of 1.

As a result we get plan diagrams which indicate if the plans differ for a query when either the power cost of the CPU or HDD (or both) is changed. Figure 15 shows the plan diagrams for the TPC-H queries under the experiment conditions. The captions identify the queries associated with the diagrams.
Figure 15: Plan diagrams for the TPC-H queries with variable CPU and disk power costs
In all diagrams the plan along the main diagonal starting at the origin in the bottom left is the plan which can be produced by using the performance cost model only since the CPU and disk power cost are equal in these cases. This plan is the default plan for a query and can always be recognized by red-colored cells. All other colors in a plan diagram stand for different plans which are returned by the query planner when the changed cost parameters make them become the optimal ones.

At the first glance it becomes obvious that for a significant fraction of the queries the plans do not change at all if the cost parameters are varied in the specified range. In particular 12 queries elicit different plans in the experiment. The corresponding plan diagrams signal one or both of the following two characteristics:

1. The query optimizer produces at least one plan which is different from the default plan for low CPU and high disk power costs. This is depicted by multiple cell colors above the main diagonal. This property applies to the plan diagrams for queries 2, 4, 5, 7, 8, 10, and 18.

2. The query optimizer produces at least one plan which is different from the default plan for high CPU and low disk power costs. This is depicted by multiple cell colors below the main diagonal. This property applies to the plan diagrams for queries 2, 3, 5, 7, 9, 10, 12, 17, and 21.

These results show that the claim that query processing has a potential of improving energy efficiency is justified. In chapter 3.2 we described how the cost of system components can differ. In one part of the application scenarios the disk induces a high energy consumption which brings the potential of improving processing of queries eliciting the first characteristic. Likewise, in the other part of the scenarios where the CPU induces a high energy consumption optimizations are possible by looking at the queries eliciting the second characteristic. The latter situation was evaluated in this chapter.

At this point we want to mention that the cost model for energy efficiency does not guarantee that the plan chosen will actually provide a better energy efficiency than the plan chosen when the performance cost model is employed. This is the result of using predictive cost values which may differ from the actual cost when the plan is executed. It is the responsibility of the underlying cost function to provide accurate estimates. A particular case where the energy efficiency decreases when energy-efficient query planning is used is characterized by the following sample query:
SELECT *
FROM lineitem
WHERE l_orderkey < 1300000
    AND o_partkey < 30000
OPTIONS energy awareness = on
    AND cpu power cost = 4.0
    AND disk power cost = 1.0

The standard model proposes the use of an index scan to satisfy the query while the energy model prefers employing a bitmap heap scan to reduce the energy consumption. Figure 16 shows the power consumption curves for the two plans produced. It can be seen that the plan chosen in energy mode consumes significantly more power due to a high CPU utilization. However this is not a flaw in the implementation of the energy cost model but rather a deficiency in the cost function for the bitmap heap scan which miscalculates the number of CPU operations required. To solve this issue one can either correct the associated cost function or extend the energy cost model by using power meter readings. This aspect will be further discussed in the outlook in chapter 6.

![Figure 16: Power consumption curves when executing the query mentioned above using either the performance or energy cost model for plan selection](image-url)
4 Dynamic Voltage / Frequency Scaling

In the previous section we looked at opportunities for improving the query optimizer. In particular we introduced an energy cost model which can be used to produce energy-efficient query plans. In this section we want to take a closer look at the subsequent step in the processing workflow. Figure 8 on page 28 outlines the components involved in the query processing job. Until now we discussed modifications of the optimizer and will now introduce a technique for saving energy when the query is actually executed.

Recalling the assumptions made in chapter 1.1 we know that the system is not required to work at peak performance to complete incoming queries within specified time constraints. However, the system is not aware of this and will always try to complete a task as fast as possible by operating at the peak clock rate. As a result the system will operate under a high load (when a query comes in) for a short period and afterwards idle until a new request has to be processed. In chapter 3.2 we mentioned that the power consumption of the CPU increases linearly with its utilization. This means that if we assume the clock rate and input voltage to be fixed that it makes no difference in terms of total energy consumption whether the query is executed as fast as possible utilizing the CPU to the full extent or implementing a maximum CPU utilization (smaller than 100%) which would require more processing time\textsuperscript{25}. If we consider the clock rate and input voltage to be alterable we do not have a linear correlation anymore which becomes obvious when we look at the computation formula for the CPU power consumption or more accurately for the switching power\textsuperscript{26}:

\[ P(F, V) = C \cdot f \cdot V^2 \]  

(12)

\( C \) is the capacitance of the CPU transistors which cannot be changed by the user and is therefore a constant. \( f \) is the clock rate or frequency the CPU is operating with and \( V \) is the input voltage. From the formula we can derive that the power consumption grows linearly with the frequency but quadratic with the voltage. This is the crucial aspect which will be exploited in this chapter and can be phrased as follows:

**Theorem 3.** A system will become more energy-efficient when the input voltage for the CPU is reduced.

**Proof.** The theorem is directly derived from formulae 5 and 12. We know that the energy efficiency increases if the energy consumption decreases. Assuming two system settings

\textsuperscript{25}When looking at the energy consumption of the whole system it would actually make a difference since a high utilization would increase heat dissipation. This would require the CPU fans to spin at a higher speed which would draw more power from the outlet to compensate for the additional heat emitted.

\textsuperscript{26}As mentioned by Rabaey et al. [RCN04] this formula assumes that the chip uses CMOS gates only. Since modern chips also uses pseudo-nMOS gates the formula is no longer exact. Moreover, decreasing feature sizes of the chip increase the static leakage current of the CPU which was studied by Jejurikar et al. [JPG04]. However, the formula provides a sufficient accuracy for this study.
$S_1$ and $S_2$ using different input voltages $V_1$ and $V_2$ with $V_1 < V_2$ we can show that in fact $S_1$ yields a lower energy consumption than $S_2$ for a constant CPU capacitance and a constant clock rate, i.e. we show that $E_1 < E_2$. Note that the runtime for any operation is not influenced as well since a lower input voltage alone does not influence processing speed. For setting $S_i$ the energy consumption is computed as follows:

$$E_i = C \cdot f \cdot (V_i)^2 \cdot t$$

Now we use $E_1 < E_2$ and show that the assumption $V_1 < V_2$ is correct.

$$C \cdot f \cdot (V_1)^2 \cdot t < C \cdot f \cdot (V_2)^2 \cdot t$$

By dividing by all constants (assuming that they are non-zero) we get:

$$(V_1)^2 < (V_2)^2$$

Since the input voltages have positive values we can derive that the equation upholds for the root values.

The straightforward approach to maximize energy efficiency is now to set the input voltage to the lowest value possible but this value depends on the clock rate. A lower frequency increases the reduction potential of the input voltage. Among other reasons this is the case because a lower input voltage results in a longer response time of the transistors on the chip. If a switch operation cannot be finished before the next clock tick the system can stall or crash. If the frequency is lowered the transistors have more time to switch which in turn reduces the lower bound of the input voltage value. As a consequence a consistent approach for minimizing the input voltage is to interlink it with the frequency used which leads to the following statement:

**Theorem 4.** A system will become more energy-efficient when the CPU clock rate is reduced. The reduction is coupled with a reduction of the input voltage for the CPU.

**Proof.** Theorem 3 already states that a reduction of the input voltage yields a higher energy efficiency.

The process of reducing the CPU clock rate is called *underclocking*. The exact opposite is *overclocking* which is usually practiced by enthusiasts seeking an increase in the performance of their computers [WR03].

Before we turn to the description of an algorithm which uses this technique to improve the energy efficiency during query execution we briefly discuss the constraints imposed by the computational unit, in particular by looking at the constraints of the test machine.
4.1 Internal power states

The clock speed and input voltage of a CPU chip cannot be changed arbitrarily. Instead chip vendors provide means to modify these parameters within relatively safe limits. In particular this is done by defining system states which have specific properties. The state categories are organized as layers which means that the system will be in a particular state on each layer. In general the following state categories or layers are of importance: performance states (P-states), processor states (C-states), and power or sleep states (G/S-states). The state definitions follow the specification of the Advanced Configuration and Power Interface (ACPI) introduced by the Compaq Computer Corporation et al. [Com00]. Since we only want to look at modifications while the machine is processing a query the system must be awake which means that it has to be in state G0 which is the non-sleep state. In addition to that the system is active which means that it is in state C0 which is the non-idle state. Consequently we only deal with different P-states which affect the CPU frequency and input voltage. Figure 17 gives an overview about the state layers and the characteristics of the states. The states which are of interest in this section are cornered red and it becomes clear that the processor and power layers are not affected by the underclocking approach.

The number of states per layer is not predefined in the ACPI and can be different on each layer and on each machine. Additionally, the ACPI is momentarily not fully supported on every system but the layer we focus on, the performance layer, can be found within dynamic frequency scaling technologies invented by the main CPU vendors. Our test machine supports the Intel SpeedStep technology [Int04]. Besides that AMD invented the PowerNow! and Cool’n’Quiet technologies for the same purpose27. The architecture of the test machine defines frequency identifier (FIDs) and voltage identifier (VIDs) which can be used to modify both parameters. The identifiers are then mapped on actually clock rates and voltage values considering restrictions integrated by the vendor using the following formula with BCLK being the constant Front Side Bus clock rate, $V_{step}$ being the step size of voltage changes between neighboring VID values, and $V_{min}$ being the minimum voltage allowed:

$$f = \text{FID} \cdot \text{BCLK}$$

$$V = \text{VID} \cdot V_{step} + V_{min}$$

According to the Intel Core 2 Duo Processor E7000 Series Datasheet [Int09] the test system has a Front Side Bus clock rate of 266MHz, a voltage step size of 12.5mV, and a minimum voltage of 850mV.

The underlying test system provides six different predefined performance states which are described in table 12. The FIDs can be extracted from the clock specification in the processor datasheet [Int09]. As opposed to this, the VIDs associated with the FIDs are

---

Figure 17: Overview about the state layers and states in a system as defined in the ACPI (the \( x \) identifies the state with the highest index on each layer, red-cornered states are of interest to the underclocking algorithm)

not defined in this document. Instead it is necessary to use a mainboard sensor which is capable of identifying the current input voltage of the CPU or a program which can extract the VIDs from the Model Specific Register (MSR)\(^{28}\). In this case we had to choose the latter option since no interface for accessing the internal sensor was available. The tool used was \texttt{linux-PHC} (PHC stands for Processor Hardware Control) which provides additional data including the default VIDs in the \texttt{/sys} interface\(^{29}\).

The state definitions show that a modification of the frequency is coupled with a change

\(^{28}\)MSRs are control registers provided by processor implementations which define features that are provided on specific processor implementations and settings which vary among different processors. This includes the definition of performance states [Int10a].

\(^{29}\)Further information about the tool can be found at \url{http://www.linux-phc.org/}. 
Table 12: Details about the performance states available on the test machine

<table>
<thead>
<tr>
<th>State</th>
<th>FID</th>
<th>$f$</th>
<th>VID</th>
<th>$V$</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>11</td>
<td>2.93GHz</td>
<td>37</td>
<td>1.3125V</td>
</tr>
<tr>
<td>P1</td>
<td>10</td>
<td>2.67GHz</td>
<td>34</td>
<td>1.2750V</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>2.40GHz</td>
<td>31</td>
<td>1.2375V</td>
</tr>
<tr>
<td>P3</td>
<td>8</td>
<td>2.13GHz</td>
<td>28</td>
<td>1.2000V</td>
</tr>
<tr>
<td>P4</td>
<td>7</td>
<td>1.87GHz</td>
<td>25</td>
<td>1.1625V</td>
</tr>
<tr>
<td>P5</td>
<td>6</td>
<td>1.60GHz</td>
<td>22</td>
<td>1.1250V</td>
</tr>
</tbody>
</table>

of the input voltage. For that reason, the technique for dynamically setting the CPU speed is called Dynamic Voltage / Frequency Scaling (DVFS). If we use the values from table 12 for calculating the energy consumption we can easily derive that theorem 4 is correct. Exemplarily let us assume that we have a query which can be executed in one second using the highest frequency. This means that the same query would need approximately 831 milliseconds longer if we use the lowest frequency. Using these values for the time consumed we get the following energy consumption for the high and low frequency scenario ($E_{\text{high}}$ and $E_{\text{low}}$) with $C$ being the unknown constant capacitance:

$$E_{\text{high}} = 2.93 \cdot 10^9 \text{Hz} \cdot (1.3125V)^2 \cdot C \cdot 1s$$
$$\approx 5.05 \cdot 10^9 V^2 \cdot C$$

$$E_{\text{low}} = 1.60 \cdot 10^9 \text{Hz} \cdot (1.1125V)^2 \cdot C \cdot 1.83125s$$
$$\approx 3.63 \cdot 10^9 V^2 \cdot C$$

It can be seen that the theoretical energy consumption in the low frequency scenario is approximately 28% smaller than the consumption in the high frequency scenario which provides a large potential of increasing the energy efficiency during query execution.

4.2 DVFS and Linux

To use DVFS one can either use an already implemented interface for modifying voltage and frequency or write a kernel module itself. For the purpose of this thesis we used the `cpufreq` module for Unix which provides all features required for regulating the clock rate at query execution time. In particular, the module allows to set the frequency directly or to employ a scaling policy which switches between performance states depending on the rules set in the policy definition. The behavior is associated with a so-called governor which represents the control unit for the DVFS procedure. The following governors are available:

Performance The frequency is always set to its maximum, i.e. the CPU is always in the P0 state, to achieve maximum performance.
4 Dynamic Voltage / Frequency Scaling

**Powersave** The frequency is always set to its minimum, i.e. the CPU is always in the $P_x$ state (where $x$ is the highest index available), to achieve maximum power savings. Note that this is only true if the system is running all times. Otherwise it is likely that the lowest power consumption is achieved if the job is finished as fast as possible (and therefore the performance governor is used).

**On-demand** This governor changes the frequency dynamically depending on the utilization. If the CPU is not fully utilized the frequency can be reduced to save energy. If the workload increases and the CPU reaches its capacity the frequency can be increased to accelerate processing.

**Conservative** The conservative governor works in principal like the on-demand governor but allows only gradual state changes, for example from $P_0$ to $P_1$, $P_1$ to $P_2$, etc. This prevents that the frequency jumps between high and low values in short time spans.

**Userspace** This governor releases the DVFS control to a module defined in user space. Alternatively, the CPU frequency can be specified explicitly by the user.

A /sys file interface is provided to obtain and modify the DVFS settings. A comprehensive introduction to `cpufreq` can be found in IBM’s developerWorks technical library [Hop09].

The governor to be used can be specified within the `OPTIONS` block at the end of a query specification. The corresponding clause looks as follows:

```
GOVERNOR = [PERFORMANCE|POWERSAVE|ONDEMAND|CONSERVATIVE]
```

The governor will be activated once the query plan execution starts and reset to its previous setting after the processing is finished.

4.3 Experimental results with DVFS

To determine the effect of DVFS the following experiment was conducted:

- The whole TPC-H benchmark consisting of 22 queries was executed on the 1GB database using the performance cost model.

- Four different test series were constructed each using a different DVFS governor (performance, powersave, on-demand and conservative).

- A test series consists of 10 runs (this proved to be sufficient since different runs
showed only minor deviations).

- Each test was executed using a different database replication to avoid caching effects. After all replications were used once the server was restarted to invalidate cached pages.

- The power consumption was recorded in one-second intervals as well as the total runtime of the whole benchmark. The resulting energy consumption, average power consumption of a run, and runtime were averaged among all runs in a test series.

Table 13 lists the experimental results including the average runtime \( t \), the average total and active power consumption per run \( P_{\text{total}} \) and \( P_{\text{active}} \) respectively, the total and active energy consumption per run \( E_{\text{total}} \) and \( E_{\text{active}} \) respectively, and the savings compared to the results of the performance test series for the last two values.

<table>
<thead>
<tr>
<th>Governor</th>
<th>( \langle t \rangle )</th>
<th>( \langle P_{\text{total}} \rangle )</th>
<th>( \langle P_{\text{active}} \rangle )</th>
<th>( \langle E_{\text{total}} \rangle )</th>
<th>( \langle E_{\text{active}} \rangle )</th>
<th>( \langle \text{Sav}_{\text{total}} \rangle )</th>
<th>( \langle \text{Sav}_{\text{active}} \rangle )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>214s</td>
<td>84.5W</td>
<td>16.0W</td>
<td>25001J</td>
<td>3424J</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Powersave</td>
<td>315s</td>
<td>74.5W</td>
<td>6.0W</td>
<td>23467J</td>
<td>1890J</td>
<td>6.1%</td>
<td>44.8%</td>
</tr>
<tr>
<td>On-demand</td>
<td>233s</td>
<td>80.8W</td>
<td>12.3W</td>
<td>24443J</td>
<td>2865J</td>
<td>2.2%</td>
<td>16.3%</td>
</tr>
<tr>
<td>Conservative</td>
<td>228s</td>
<td>79.6W</td>
<td>11.1W</td>
<td>24108J</td>
<td>2530J</td>
<td>3.5%</td>
<td>26.1%</td>
</tr>
</tbody>
</table>

Table 13: Experimental results for executing the TPC-H benchmark using different DVFS governors

As described in section 3.1 the total and active energy consumptions are consumed by taking the maximum of the runtimes (in this case the runtime of the powersave test series \( t = 315s \)) and charging the faster test series the idle energy cost for the differential. The results in the table clearly show that significant active energy savings of up to almost 45% can be achieved by employing lower CPU clock rates and voltages and even the total power consumption can be reduced by as much as 6%. The powersave governor induces the lowest average power consumption and also the lowest maximum power consumption. This is not surprising since the frequency stays fixed at its minimum value in the test series. All other governors exploit the whole range of performance states available which is indicated by the difference between the average and maximum power consumption of the associated test runs and becomes even clearer when looking at the power meter readings while running the benchmark. Figures 18 to 21 show the power consumption curves associated with sample runs from the different test series. For every run only the time interval where the benchmark is actually running is depicted (the post-execution idle time span is not shown).

Figure 18 shows that the power consumption during a run with the performance governor usually ranges above 85 Watts. Nevertheless, the power consumption drops significantly at several times. However, this is not caused by frequency modifications but rather by a low CPU utilization which reduces the number of switching operations and consequently
the switching power. Usually this is caused by disk accesses which have relatively high latencies and leave the CPU without instructions to be executed. In contrast, the power consumption during a run employing the powersave governor stays at an almost perfectly steady level. The low frequency improves the probability that the CPU is highly utilized and therefore the switching power fluctuates rarely. Highly disk-bound queries can still result in bottlenecks which increases CPU idle time which is in this case happening at the end of the benchmark run. The remaining two figures 20 and 21 show large similarities to the curve from the performance run but include a smaller number of sudden power consumption spikes. This is the case because the governor policy does not react to an increasing workload immediately. Instead the utilization history, for example the utilization development in the last second, is used to determine whether the frequency should be changed or not. As a result, short utilization spikes get smoothed out. Differences between the on-demand and conservative run are barely noticeable due to the fact that gradual frequency adjustments can happen within short time spans which exceed the resolution of the diagram.
Figure 19: Power consumption curve for a sample run using the powersave governor

Figure 20: Power consumption curve for a sample run using the on-demand governor
Figure 21: Power consumption curve for a sample run using the conservative governor
5 Combining planning and scaling

In the previous two sections we introduced a means to select energy-efficient query plans for execution and a way for reducing the energy consumption while actually executing the query. Both techniques were tested independently and in this section we will merge them together to form a whole query planning and execution process optimized for energy efficiency. To achieve this goal we have to add a deadline parameter to the query definition which allows the user to set a maximum latency until the query execution should be finished. During query plan execution DVFS functionalities can be employed to minimize energy consumption as long as the deadline constraint is not violated. The whole query processing can be summarized by the following workflow:

1. Extract and set options and constraints used for energy-efficient query processing (including component costs, cost model weights, and deadlines)

2. Select the optimal query plan using the performance and energy cost model

3. Execute the plan and use DVFS as long as the execution is expected to finish within the deadline

4. Reset the options and constraints used for energy-efficient query processing

The following subsection describes how the deadline concept works and how DVFS is enabled during query execution.

5.1 Deadlines

To include the semantics of deadlines during query processing an additional query clause similar to the ones described in section 3.4 was introduced:

DEADLINE = <value> [SECONDS|MINUTES|HOURS]

This clause can be included in the OPTIONS block at the end of a query specification. In addition to that the plugin contains an additional variable, the deadline threshold fraction, which is used to decide when the system switches into performance mode and no longer tries to conserve energy by operating in powersave mode. When the execution of the optimal query is initialized a separate thread, the monitoring thread is started which computes the threshold time, sleeps until the threshold time is reached and activates the performance mode afterwards. Nevertheless it is possible that the query execution finishes before the sleep instruction in the monitoring thread terminates. In this case the governor settings are not affected.
The thread can be finished in two possible ways. Either the thread exits itself once the threshold time is exceeded and the performance mode activated or the thread is cancelled by the parent process as soon as query execution terminates. The actual implementation uses POSIX threads which can be cancelled if the appropriate settings are made in advance [But97]. As a result the monitoring thread will only be active as long as the query is executed. This is guaranteed by defining the necessary commands in the hook functions which are called when the execution starts and ends. Figure 22 shows a model for the concurrent processing when the query is executed. The dashed line distinguishes between the control flow in the main and monitoring thread. Note that the thread is cancelled and subsequently joined with even when it exits itself after activating the performance governor. This is done since no explicit communication between both threads is implemented. In the model it is suggested that the cancel point, i.e. the node after the sleep transition, is only reached after the sleep instructions finishes. However, the sleep instruction itself defines a cancel point which means that it can be cancelled while it is running. In the model this concurs with an early termination of the sleep transition triggered by the cancellation signal.

At this point one might ask the question why the performance cost computed by the query optimizer for the given plan is not used for predicting the runtime behavior and change the governor settings accordingly. This approach was considered in the early development stage but not implemented for the following reason. Test runs of the TPC-H benchmark have shown that the predicted performance cost can show large deviations from the real runtime cost. The ratio between both cost values ranged from 0.2 to 100. This makes it impossible to correlate the computed performance cost with suitable governor settings at runtime. Nevertheless this does not mean that the optimizer produces unreliable predictions at all times but rather that it cannot be guaranteed that the cost ratio remains within a suitable predefined range. One might use the predicted cost to prevent the usage of the powersave mode for queries which are very unlikely to finish within the deadline but we will show later that the deadline threshold can be adjusted to the user’s needs such that a low maximum delay can be guaranteed.

To get a better impression of the effect of the deadline threshold we can calculate the possible delay for a query which is capable of executing within its deadline in the performance mode (the execution time in this mode is specified by the variable \( t \)). Hereby, the delay is defined as the additional time required to finish execution after the deadline has passed. Consequently, the formula for the delay is defined by adding the time spent in performance mode (\( t_{\text{perf}} \)) and powersave mode (\( t_{\text{pow}} \)) together and subtracting the allowed runtime represented by the deadline \( d \):

\[
\text{Delay} = t_{\text{perf}} + t_{\text{pow}} - d
\]  

(15)

As defined earlier, the machine will operate in the powersave mode until the deadline threshold is exceeded and afterwards operate in performance mode until query processing is completed (the absolute threshold value is computed by multiplying the percentual
representation $t^*$ with the deadline). Note that the execution time in the performance mode depends on the speed factor $s$ (or performance gain) which is derived by taking the quotient of the frequency used in performance and powersave mode.

\[
s = \frac{f_{perf}}{f_{pow}} \tag{16}
\]

\[
t_{pow} = t^* \cdot d \tag{17}
\]

\[
t_{perf} = t - \frac{t^* \cdot d}{s} \tag{18}
\]
Using these variable definitions we can compute the delay as follows:

\[
\text{Delay} = t - \frac{t^* \cdot d}{s} + t^* \cdot d - d
\]

(19)

\[
= t - d + t^* \cdot d \cdot \left(1 - \frac{1}{s}\right)
\]

(20)

\[
(21)
\]

The product defines the additional time consumed when operating in powersave mode and it becomes clear that this time fraction increases proportionally to the speed factor. Furthermore we can see that the worst case or the maximum delay eventuates when the deadline is equal to the runtime in the performance mode, i.e. \( t - d = 0 \). In this case any reduction of the CPU frequency would cause a delay. Figure 23 shows the development of the maximum delay depending on the deadline threshold fraction on the testing machine. We have to bear in mind that the delay can be larger if a deadline is chosen which cannot be satisfied under any conditions. It can be seen that the maximum delay can be approximately 45% if the system operates in the powersave mode until the deadline is exceeded and that the delay increases proportional regarding the deadline threshold. We can also take a different vantage point and compute the maximum fraction of the allowed time used \((U_{\text{max}})\) in the performance mode to guarantee that the query can be executed without any delay. This can be done by deriving the minimum deadline value required to satisfy the deadline at all times (by introducing the constraint \(\text{Delay} \leq 0\))
and dividing it from the time consumed in performance mode.

\[ 0 \geq t - d + t^* \cdot d \cdot \left( 1 - \frac{1}{s} \right) \]  
\[ (22) \]

\[ 0 \geq t - d \cdot \left( 1 - t^* \cdot \left( 1 - \frac{1}{s} \right) \right) \]  
\[ (23) \]

\[ d \geq \frac{t}{1 - t^* \cdot \left( 1 - \frac{1}{s} \right)} \]  
\[ (24) \]

\[ U_{max} \leq 1 - t^* \cdot \left( 1 - \frac{1}{s} \right) \]  
\[ (25) \]

Formula 25 shows that the maximum utilization\(^{30}\) is 100% minus the possible fraction of time spent in the powersave mode. For our test system this means that with a deadline threshold of 100% we can guarantee delay-free processing if every query consumes at most 55% of the time available for finishing processing within the deadline. This is exactly the performance of the CPU in the powersave mode since we can guarantee processing within the deadline constraints as long as the query can be executed in time when the lowest clock rate is used. However, a lower deadline threshold should be used to reduce the gap between the maximum utilization and the specified deadline (for example for a deadline threshold of 50% the maximum utilization can be as high as 77% to guarantee processing without a delay) which reduces the risk of causing a delay.

The analysis points out that assumptions about the deadlines have to be introduced to guarantee in-time processing. On the contrary, maximum delays can be predicted as long as a query satisfies its deadline in performance mode. Altogether no hard deadlines can be implemented and consequently this approach is not suitable for real-time database systems\(^{31}\). Since this type of processing is currently not supported by most DBMS including PostgreSQL there is no need to preserve the real-time capabilities at this moment.

To access the DVFS functionality provided by the `cpufreq` module the `/sys` interface is used. In particular the `scaling_governor` file contains the identifier of the governor that should be used and to change the governor the respective identifier has to be written to that file. The file is then accessed by the kernel module and governor settings are changed accordingly. The governor is changed for each CPU separately which means that in the test system two governor files exist and have to be modified to use the same scaling policy for the whole system. Vice versa it possible to get the current governor settings by reading the data from the governor files.

\(^{30}\)We want to emphasize at this point that the utilization keyword refers to the quotient of the time consumed and the time allowed to finish processing (defined by the deadline). A utilization of 100% means that the complete time span until the deadline is needed to finish processing on time.

\(^{31}\)Further insight about real-time database systems can be found in the book by Lam & Kuo [LK01], or the technical report by Kao & Garcia-Molina [KGM93].
5.2 Experiment design & results

To test the effects of the combinatorial approach of employing both energy-efficient query planning and execution the following experiment was conducted:

- The whole TPC-H benchmark consisting of 22 queries was executed on the 1GB database.
- Three different test series were constructed with the first using the performance cost model and no deadlines, the second using the energy cost model, a deadline $d = 15s$ and a deadline threshold fraction $t^* = 0.5$, and the third using the same settings as the second test series except for a deadline $d = 60s$.
- The properties 3 to 5 from the DVFS experiment in section 4.3 concerning possible caching effects and power consumption evaluation were used.

Table 14 lists the average runtimes for the runs within each test series for all TPC-H queries.

It can be seen that almost all deadlines can be satisfied if a deadline of 60 seconds is used. For the first query which needs almost all of the available time in performance mode a delay of 10.1 seconds (or 18%) is caused. If a time limit of 15 seconds is used a total of four deadlines are missed. This is self-evident since the deadlines are either missed in performance mode as well or the utilization is above the value which guarantees on-time processing (77% for $t^* = 0.5$). For a significant portion of the queries no changes in runtime behavior can be perceived between the two test series using deadlines since the deadline threshold is not reached in either case and the queries can be completely executed in powersave mode.

Table 15 shows a comparison of the average runtime, power, and energy consumption for the whole benchmark.

Significant active energy savings of up to almost 24% can be achieved when employing the energy cost model as well as DVFS with a deadline of 60 seconds during query plan execution. Logically this savings potential comes with an increased runtime for the benchmark. The reason for this is that the usage of the powersave mode causes longer response times if the CPU utilization reaches the maximum when the minimum frequency is used.

To get a better view on the impact of using the energy cost model in conjunction with DVFS and deadlines figures 24 to 26 show the power consumption curves for sample runs of each test series. Figure 24 shows the curve for a sample run using the performance cost model which we have already seen earlier. The power consumption is high at most
5 Combining planning and scaling

\[
\langle t_P \rangle (\text{in s}) \quad \langle t_E \rangle (\text{in s}) \text{ using } t^* = 0.5
\]

<table>
<thead>
<tr>
<th>Query</th>
<th>\langle t_P \rangle</th>
<th>\langle t_E \rangle (d = 15s)</th>
<th>\langle t_E \rangle (d = 60s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>56.3</td>
<td>59.7</td>
<td>70.1</td>
</tr>
<tr>
<td>2</td>
<td>3.0</td>
<td>3.8</td>
<td>3.8</td>
</tr>
<tr>
<td>3</td>
<td>14.5</td>
<td>16.1</td>
<td>15.5</td>
</tr>
<tr>
<td>4</td>
<td>1.6</td>
<td>5.6</td>
<td>5.6</td>
</tr>
<tr>
<td>5</td>
<td>4.6</td>
<td>9.0</td>
<td>9.0</td>
</tr>
<tr>
<td>6</td>
<td>4.2</td>
<td>13.9</td>
<td>15.2</td>
</tr>
<tr>
<td>7</td>
<td>5.9</td>
<td>14.6</td>
<td>15.8</td>
</tr>
<tr>
<td>8</td>
<td>3.1</td>
<td>5.6</td>
<td>5.6</td>
</tr>
<tr>
<td>9</td>
<td>31.3</td>
<td>34.3</td>
<td>39.9</td>
</tr>
<tr>
<td>10</td>
<td>7.7</td>
<td>10.8</td>
<td>13.1</td>
</tr>
<tr>
<td>11</td>
<td>0.6</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>12</td>
<td>4.7</td>
<td>7.7</td>
<td>8.1</td>
</tr>
<tr>
<td>13</td>
<td>4.4</td>
<td>7.6</td>
<td>8.0</td>
</tr>
<tr>
<td>14</td>
<td>1.4</td>
<td>2.4</td>
<td>2.4</td>
</tr>
<tr>
<td>15</td>
<td>5.4</td>
<td>8.2</td>
<td>8.9</td>
</tr>
<tr>
<td>16</td>
<td>5.8</td>
<td>8.9</td>
<td>10.5</td>
</tr>
<tr>
<td>17</td>
<td>1.1</td>
<td>1.5</td>
<td>1.4</td>
</tr>
<tr>
<td>18</td>
<td>6.3</td>
<td>10.1</td>
<td>12.4</td>
</tr>
<tr>
<td>19</td>
<td>3.0</td>
<td>5.4</td>
<td>5.4</td>
</tr>
<tr>
<td>20</td>
<td>46.4</td>
<td>46.1</td>
<td>54.9</td>
</tr>
<tr>
<td>21</td>
<td>6.0</td>
<td>12.1</td>
<td>15.2</td>
</tr>
<tr>
<td>22</td>
<td>0.7</td>
<td>1.3</td>
<td>1.3</td>
</tr>
</tbody>
</table>

Table 14: Average query runtimes (in seconds) when operating in performance mode, or energy mode (with deadlines of 15 or 60 seconds) and a deadline threshold fraction of 50%.

times and only short intervals with a lower CPU utilization cause the consumption to drop.

Figure 26 misses most of the spikes from the previous figure since most deadline thresholds are not exceeded. In particular, only query 1, 9, and 20 cause a transition to the maximum frequency after 30 seconds. As a result additional power can be saved by using the lowest clock rate for a longer time span. If we increase the deadlines even further the remaining spikes can be erased as well and the output would be same as with using the powersave governor by default without any deadlines.

Figure 25 shows the curve for a sample run using the energy cost model, a deadline \( d = 15s \), and a deadline threshold fraction \( t^* = 0.5 \) which means that the monitoring thread will cause a switch into performance mode (using the performance governor) after 7.5 seconds. The curve stays at a lower power consumption level much longer than the
5 Combining planning and scaling

<table>
<thead>
<tr>
<th>Mode</th>
<th>$\langle t \rangle$</th>
<th>$\langle P_{\text{total}} \rangle$</th>
<th>$\langle P_{\text{active}} \rangle$</th>
<th>$\langle E_{\text{total}} \rangle$</th>
<th>$\langle E_{\text{active}} \rangle$</th>
<th>$\langle \text{Sav}_{\text{total}} \rangle$</th>
<th>$\langle \text{Sav}_{\text{active}} \rangle$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>219s</td>
<td>81.5W</td>
<td>13.1W</td>
<td>24914J</td>
<td>2858J</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Energy ($d = 15s$)</td>
<td>287s</td>
<td>77.2W</td>
<td>8.7W</td>
<td>24562J</td>
<td>2505J</td>
<td>1.4%</td>
<td>12.4%</td>
</tr>
<tr>
<td>Energy ($d = 60s$)</td>
<td>322s</td>
<td>75.2W</td>
<td>6.7W</td>
<td>24223J</td>
<td>2176J</td>
<td>2.7%</td>
<td>23.9%</td>
</tr>
</tbody>
</table>

Table 15: Comparison of the average runtime, power, and energy consumption for executing the TPC-H benchmark when operating in performance mode, or energy mode (with deadlines of 15 and 60 seconds respectively) and a deadline threshold fraction of 50%.

previous curve which is the result of using the powersave governor. Note that there are still spikes which are caused by switching into performance mode after the deadline threshold is exceeded. For the first query which executes in approximately 46 seconds, this is the case after 7.5 seconds which means that the system will operate at the highest frequency for the remaining 38.5 seconds. As a result the curve is characterized by a high function value for this time frame.

![Power consumption curve of a sample run executing the whole TPC-H benchmark using the performance cost model](image)

Figure 24: Power consumption curve of a sample run executing the whole TPC-H benchmark using the performance cost model.
Figure 25: Power consumption curve of a sample run executing the whole TPC-H benchmark using the energy cost model and DVFS with a deadline of 15 seconds and a deadline threshold fraction of 50%.

Figure 26: Power consumption curve of a sample run executing the whole TPC-H benchmark using the energy cost model and DVFS with a deadline of 60 seconds and a deadline threshold fraction of 50%.
After analyzing the results for the whole TPC-H benchmark we now want to look at the results for three selected queries. The experimental design is principally the same except for the following modifications:

- Only queries 1, 3, and 9 are used and all runs are executed using the performance cost model only.

- Two test series are constructed by using either no deadline or a deadline $d = 30s$.

As a result query 1 and 9 cannot be executed within the deadline and are guaranteed to cause delays. During query execution the CPU will operate in powersave mode for 15 seconds each and afterwards switch into performance mode. Query 3 is likely to finish execution before the threshold is passed. Table 16 shows the experimental results for the three queries with the rows where the query index is appended by a (d) indicating the runs using the deadline. The savings for each query when using a deadline are shown as well.

<table>
<thead>
<tr>
<th>Query</th>
<th>$\langle t \rangle$</th>
<th>$\langle P_{total} \rangle$</th>
<th>$\langle P_{active} \rangle$</th>
<th>$\langle E_{total} \rangle$</th>
<th>$\langle E_{active} \rangle$</th>
<th>$\langle Sav_{total} \rangle$</th>
<th>$\langle Sav_{active} \rangle$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (d)</td>
<td>63s</td>
<td>82.1W</td>
<td>13.6W</td>
<td>5177J</td>
<td>856J</td>
<td>1.9%</td>
<td>11.0%</td>
</tr>
<tr>
<td>3 (d)</td>
<td>16s</td>
<td>71.9W</td>
<td>3.4W</td>
<td>1150J</td>
<td>54J</td>
<td>4.7%</td>
<td>51.3%</td>
</tr>
<tr>
<td>9 (d)</td>
<td>34s</td>
<td>80.1W</td>
<td>11.6W</td>
<td>2723J</td>
<td>394J</td>
<td>3.1%</td>
<td>18.3%</td>
</tr>
</tbody>
</table>

Table 16: Comparison of the average runtime, power, and energy consumption for executing selected queries, either without a deadline, or with a deadline $d = 30s$ and an deadline threshold fraction of 50% (denoted by a (d) appended to the query index)

All runs using a deadline show only small increases in runtime but achieve significant active energy savings. For query 3 up to 51% of the active energy can be saved when operating in powersave mode. This becomes even more apparent when looking at the power consumption curves for sample runs of the queries which are shown in figures 27 to 29. The curve for query 1 marks clearly where the system switched from the lowest to the highest frequency\(^{32}\) which is after approximately 15 seconds. The performance run finishes earlier than the powersave run and consequently idle power will be consumed for the differential in time.

\(^{32}\)The switch does not happen exactly after 15 seconds since monitoring starts when the execution of the optimal query plan is initiated. Since the query optimization process consumes some time the switch is logically delayed for a short time. Further delays are caused by the fact that the power meter records only in one-second intervals and the execution of the query and the point in time where the first reading is made cannot be synchronized perfectly.
Combining planning and scaling

Figure 27: Power consumption curves for sample runs of TPC-H query 1 using either the performance governor at all times or employing the powersave governor until the deadline threshold is reached.

Figure 28 shows the power consumption curve for query 3. In both test series the average power consumption is relatively low compared to the values observed for the whole TPC-H benchmark as well as the selected queries 1 and 9. This is caused by the fact that the query is memory-bound and requires a lower CPU utilization (since the disk accesses are actually the bottleneck here). The whole query can be executed before reaching the deadline threshold but notice that the power consumption is constantly increasing. This is the result of an increasing power consumption over time induced by an increasing CPU utilization.

The power consumption for query 9 shown in figure 29 shows an interesting effect. Both curves look relatively similar to the curve for the run using a deadline which has the spike shifted to the right. In this case the query plan has the characteristic of being memory-bound in the first phase (until approximately $t = 12s$) and CPU-bound afterwards. For memory-bound operations no high energy savings can be achieved since the CPU utilization is already low. Since we used a deadline of 30 seconds in the experiment the potential savings in the second phase cannot be exhausted because the deadline threshold is reached and the CPUs switch to the maximum clock rate.
Figure 28: Power consumption curves for sample runs of TPC-H query 3 using either the performance governor at all times or employing the powersave governor until the deadline threshold is reached.

Figure 29: Power consumption curves for sample runs of TPC-H query 9 using either the performance governor at all times or employing the powersave governor until the deadline threshold is reached.
5.3 Complexity analysis

After discussing the design and effects of the energy-efficient query optimizer and executor one aspect has to be reviewed: the additional cost of the algorithms added to the PostgreSQL system. At the beginning of this chapter we mentioned the workflow for processing queries with focus on energy efficiency. The associated components which produce costs are the following:

Pre-parser The pre-parser is responsible for extracting and setting the options required to do energy-efficient query processing. It is accessed once for every query at the beginning of the processing.

Energy cost model The energy costs are computed in conjunction with the performance costs. This is done for every plan which has to be evaluated.

Setting runtime parameters At the moment this involves only accessing the /sys interface and setting the governor to be used. This is done if the user specifies a governor or deadline and is done only once right before the plan execution starts.

Monitoring thread If a deadline is specified a monitoring thread is started which will activate the performance governor if the deadline threshold is exceeded. The thread is created before the query plan execution starts and cancelled / joined after the processing finishes.

All other aspects are handled by algorithms provided within the standard PostgreSQL edition. We will now look at the time complexity using the $O$-notation regarding the following parameters:

- Number of queries denoted by the variable $n$
- Query complexity or number of alternative plans for a query\(^{33}\) denoted by the variable $q$
- Query runtime denoted by the variable $r$

All parameters are independent from each other and therefore time complexity is computed separately regarding each of the parameters. To fit the $O$-notation we compute the cost or complexity for processing queries using either the standard model denoted by the function $f_{\text{std}}(n, q, r)$, or the energy-efficient model described earlier denoted by the function $f_{\text{eff}}(n, q, r)$. When using the standard model the processing complexity can

\(^{33}\)The number of alternative plans is usually correlated with the query complexity since a complex query provides more opportunities to change processing steps like join orders. However it is possible that a complex query does not leave multiple choices for processing it.
be computed by taking the sum of the runtime cost and the plan evaluation costs and multiplying it with the number of queries:

\[ f_{std}(n, q, r) = n \cdot (q \cdot c_q + r) \] (26)

\( c_q \) represents the cost for evaluating a single plan and depends on the query complexity \( q \) because the cost is evaluated for each plan node and a higher query complexity results in a higher number of plan nodes. Consequently we get a linear time complexity with regard to the number of queries and the query runtimes and a quadratic time complexity with regard to the query complexity.

\[ f_{std} \in O(n, q^2, r) \] (27)

We now look at the components required for energy-efficient processing and how they influence the time complexity of the query processing task. The pre-parser is called once at the begin of the query planning process. The number of options to extract is limited which yields an upper bound for the cost or in terms of the \( O \)-notation a constant cost for the pre-parsing job (\( c_{pp} \)). The next component, the energy cost model, is used in conjunction with the performance cost model. All cost computations are pessimistically said done twice, once for computing the performance cost and once for computing the energy cost. As a result this step is represented by the following term: \( 2 \cdot q \cdot c_q \). The runtime parameters are once more limited in diversity and only set once. Again this yields a constant cost (\( c_{set} \)). Finally the monitoring thread is activated which consumes a startup and shutdown cost (\( c_{start} \) and \( c_{stop} \)) as well as a cost for changing the governor settings once the threshold time is reached (\( c_{gov} \)). All the costs are with regard to the complexity variables constant. Overall the processing complexity when employing the energy-efficient model can be computed and transformed as follows:

\[ f_{eff}(n, q, r) = n \cdot (2 \cdot q \cdot c_q + r + c_{pp} + c_{set} + c_{start} + c_{stop} + c_{gov}) \] (28)

\[ f_{eff}(n, q, r) = n \cdot (2 \cdot q \cdot c_q + r + c) \] (29)

Constants have no relevance in the \( O \)-notation and consequently the time complexity does not change at all.

\[ f_{eff} \in O(n, q^2, r) \] (30)

A detailed analysis of the space complexity is omitted at this point but it can be said that the energy-efficient model requires only additional storage for saving the user-defined properties like the component energy costs as well as two additional fields per query plan node to store the startup and overall energy cost.

Regarding actual runtimes when using either the standard or energy-efficient processing model no measurable runtime differences for employing the additional components were observed. However this does not mean that for example starting and cancelling the monitoring thread does not consume any additional time at all. Instead of that the additional costs pose no relevance when put in relation to the overall runtime costs. This applies specifically to OLAP applications since they usually have long runtimes [CD97].
With OLTP this might not be the case and the constant costs which were discarded in the $O$-notation can become significant if query execution times are very short. A detailed analysis for OLTP workloads (for example provided by the TPC-C benchmark) should be a part of upcoming developments in this area.
6 Outlook

In the previous section we showed that manifold extensions to a DBMS including the definition of a cost model targeting energy efficiency as well as the usage of DVFS can lead to significant active power savings in desktop environments. It remains to be proven if the same savings potential applies to server environments. Besides we have explained that energy efficiency depends on the energy proportionality of the system components and we expect to achieve a higher efficiency with an increasing energy proportionality in the near future. In addition to that we focussed on improvements on the software side, in particular on the query optimizer and executor. Since the components involved in processing a query including all hardware and software components can barely be examined in a single study we will only discuss a few potential extensions as well as the concept of query scheduling which is yet to be implemented in DBMS like PostgreSQL but provides promising theoretical improvements on the energy efficiency side.

6.1 Potential extensions

In chapter 3 we introduced the usage of power cost parameters to compute the energy consumption of a query plan. At the moment these costs are set by the user and in the test environment they were determined by basic tests creating no load or maximum load on the particular component. A possible extension could include an automated process to set up these parameters by executing standardized operations which create the required load situations and taking the readings from a power meter automatically.

Another useful extension would be a database which stores information about the power consumption of all operations (or nodes) provided in the DBMS. The data can be recorded during the execution of queries involving the respective operations and then be used to optimize the energy cost model to provide a better estimate for the energy consumption of a query plan. Xu et al. [XTW10] also used a calibration mechanism to find good cost estimates. However their approach was completely manual.

A final improvement involves the visualization tool introduced in chapter 3.7. At the moment the only statistics provided are the performance and energy cost fetched from the result set of the EXPLAIN statement. If we interface the tool with the power meter we could obtain the sensor readings and provide diagrams showing power consumption curves and making them comparable between plans for the same query. This would be especially useful to evaluate the utilization and possible bottlenecks during query execution and provide in general a higher transparency [BK01].
6.2 Query scheduling

As noted earlier, energy efficiency can either be achieved by reducing the time consumption while the power consumption stays fixed or by reducing the power consumption while the time consumption stays fixed. In this section, we want to introduce a concept called query scheduling, which can reduce the overall time consumption for executing multiple queries while the latency for single queries might increase.

The concept for query scheduling can be compared with the algorithms used for scheduling processes and threads in an operating system [Sta08]. Instead of treating queries sequentially and isolated on a first in - first out (FIFO) basis we introduce a scheduler which is responsible for manifesting the execution order. To do this, the scheduler is capable of identifying overlaps between plans, i.e., parts of plans which include the same operations on the same data. To reduce redundancies, these operations need only be executed once as long as the output data stays available until depending operations are executed as well and consequently queries and nodes involved in an identified overlap should be executed in succession or at the same time. As a result, the overall time consumption would decrease while a different execution order of operations may increase the latencies for some queries. The easiest way to illustrate the behavior is by looking at a set of queries involving the same operation. Here we focus on the hash join which contains a build phase where a hash table is created for the smaller relation, and a probe phase where the larger relation is scanned and the hash table is probed for matches with the current tuple [SD89]. If a match is found, the merged data is written as output. Now suppose we have four relations (A, B, C, and D) and three queries which contain basic join operations and arrive in the order as follows: A ⨉ B, B ⨉ C, D, and A ⨉ C. Using the hash algorithm explained above to process the queries without changing the order, the following steps have to be executed:

1. Create hash table for relation A
2. Scan relation B and probe for matches in the hash table (join A and B)
3. Create hash table for relation C
4. Scan relation D and probe for matches in the hash table (join C and D)
5. Create hash table for relation A
6. Scan relation C and probe for matches in the hash table (join A and C)

It is self-evident that there is an overlap among the queries. For example, for the first and the third query, a hash table is created for the same relation A. In the standard case, the query processor is not aware about overlaps between multiple queries since they are...
processed isolated. With the introduction of a scheduler the isolation would be removed and overlaps can be detected and used for optimizing the query processing. For the three join queries the scheduler could for example propose the following execution order:

1. Create hash table for relation A

2. Scan relation B and probe for matches in the hash table (join A and B)

3. Scan relation C and probe for matches in the hash table (join A and C)

4. Create hash table for relation C

5. Scan relation D and probe for matches in the hash table (join C and D)

In this case we omit the second construction of the hash table for relation A which can save a significant amount of time depending on the size of the input. If multiple processors can be used for joining we could additionally execute steps 2 and 3 in parallel [MSD93]. The overall processing time would consequently drop as well but the changed processing order of the queries (in this case the third query is processed earlier than the second query) can increase the latency for one query and reduce it for a different one. To maintain a balance between latency changes a maximum shift can be introduced which defines how many queries which arrived after a particular query can be executed before processing of the particular query cannot be deferred anymore. Alternatively the queue size for the scheduler can be limited to minimize latency fluctuations.

If we take another look at the example we used one might recognize another optimization potential. Suppose we have only the second and the third query, which are $C \bowtie D$ and $A \bowtie C$. The default plans create hash tables for relations C and A. However in this case it is more efficient if we only create a hash table for relation C and use it to process both queries. Since the typical scheduler is only dealing with finalized plans this behavior cannot be achieved there. Instead a higher degree of interlocking is required to evaluate the overall processing pipeline and influence cost parameters for single operations. In this case the optimizer would be required to perceive that a hash table on relation C should be preferred since it can be used for probing in both queries. To integrate this ability into the cost model the optimizer could award a reduced cost to the node representing the construction of the hash table for relation C.

Query scheduling can also be discussed in different contexts. Among others multi-site and parallel database systems require a scheduling policies to organize the workflow between multiple nodes. Further insight from this vantage point is provided by Mehta et al. [MSD93], Frieder & Baru [FB94], and Garofalakis & Ioannidis [GI97].
References


[CAB+81] Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, James N. Gray, W. Frank King, Bruce G. Lindsay, Raymond Lorie, James W. Mehl, Thomas G. Price, Franco Putzolu, Patricia Griffiths Selinger, Mario Schkol-


References


[Kum10] Rakesh Kumar. Data center power, cooling and space: A worrisome outlook for the next two years. Gartner RAS Core Research Note G00174420, 2010.


[ML89] Lothar F. Mackert and Guy M. Lohman. Index scans using a finite


References


Declaration

I hereby declare in lieu of oath that I composed this thesis independently and without inadmissible help from outside. The sources used are quoted in full and parts that are direct quotes or paraphrases are identified as such.

Potsdam, August 31, 2010

Tobias Flach