Automated performance tuning of database systems

Nan Noon Noon
University of Wollongong
Automated Performance Tuning of Database Systems

A thesis submitted in fulfillment of the requirements for the award of the degree

Master of Philosophy in Computer Science

from

UNIVERSITY OF WOLLONGONG

by

Nan Noon Noon

School of Computer Science and Software Engineering
May 2017
Dedicated to

My parents
Declaration

This is to certify that the work reported in this thesis was done by the author, unless specified otherwise, and that no part of it has been submitted in a thesis to any other university or similar institution.

Nan Noon Noon
May 2, 2017

One of the challenging tasks for database administrators is tuning database systems within a short period of time to make quick decisions. Automated performance tuning of database systems improves performance, reduces the cost of administrators, reduces high workload levels and decreases the number of tasks for database administrators, because database administrators need more help from database servers when they tune the performance of database systems to reduce the high workload level. To reduce high workload, tuning plans have to be implemented within the low workload time. Sometimes, low workload can be higher than high workload because of implementations of tuning plan within low workload time. To prevent that case, balancing the workload is the best idea.

This work considers how to automatically balance workloads to achieve the best performance out of database systems. This thesis presents insights on how to generate vertical partitions based on a predicted workload structure. In particular, we propose that the algorithms analyse the predicted workload and divide the workload length into smaller time frames. Next, the algorithms generate optimized vertical partitions for each time frame. After that, we use multi-layer transient and persistent storage devices to allocate the resources that require implementing the vertical partitions. Based on the vertical partitions, the algorithms distribute the resources required to implement the vertical partitions. After that, algorithms generate the best optimized plans and implement the plans within a low workload time to reduce high workload.

In this thesis, we propose index-only processing to implement vertical partition. There are some rules applied when algorithms implement the plans, and the plans
are tracked so that they do not go over the target workload level. The target work-
load level limits the level to achieve a balanced workload. Finally, we demonstrate
the results of experiments that show the best performance of database systems
by using index-only processing over multi-layer storage devices for internal query
processing.
The research work for this thesis was undertaken at the University of Wollongong. First, I would like to thank my thesis supervisor Dr Janusz R. Getta of the Faculty of Engineering and Information Sciences at the University Of Wollongong. The door to my supervisor office was always open whenever I ran into a trouble spot or had a question about my research or writing. He consistently allowed this thesis to be my own work but steered me in the right the direction whenever he thought I needed it. Without him, this thesis would not have been possible.

Also, I would like to thank the teams of Decision Systems Lab (DSL) who were supported in the conferences for this work and provided the idea for conferences presentation. Also, I wish to acknowledge the support I have received from all the stuff in the Faculty of Engineering and Information Sciences, University of Wollongong.

Finally, I must express my very profound gratitude to my parents for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Author
Nan Noon Noon
Contents

Publications v
Abstract vi
Acknowledgements viii

1 Introduction 1
  1.1 Performance tuning of database systems ....................... 1
    1.1.1 Manual performance tuning .............................. 1
    1.1.2 Automated performance tuning ............................ 2
    1.1.3 Predicting database workloads ........................... 2
  1.2 The problem ................................................... 4
  1.3 Motivating example ............................................. 5
    1.3.1 Implementation of synthetic workload ..................... 5
    1.3.2 Estimation of a target workload level .................... 8
    1.3.3 Experiment 1 .............................................. 8
    1.3.4 Experiment 2 ............................................. 11
    1.3.5 Experiment 3 ............................................. 12
    1.3.6 Experiment 4 ............................................. 13
    1.3.7 Conclusions from the experiments ......................... 15
  1.4 The objectives ................................................. 15
  1.5 The planned solutions .......................................... 16
  1.6 Thesis outline ................................................ 17

ix
# Literature review

2.1 Self-tune database systems .............................................. 19
2.2 Indexing, materialized view and partitioning ....................... 20
2.3 Fuzzy-based tuning .......................................................... 23
2.4 Cache-base optimization .................................................... 24
2.5 Summary and conclusion ..................................................... 24

# Automated tuning processes

3.1 Time frames ................................................................. 26
  3.1.1 Time units .............................................................. 26
  3.1.2 Workload estimations ................................................ 27
  3.1.3 Target workload level ................................................ 29
  3.1.4 Creating time frames ................................................ 29
  3.1.5 Assigning queries to time frames ................................. 30
3.2 Controller ........................................................................ 33

# Vertical partitioning

4.1 Basic concepts ............................................................... 41
4.2 Overview of vertical partitioning ........................................ 42
  4.2.1 Schemas of vertical partitions and relational tables .......... 42
  4.2.2 Optimisation of vertical partitions ................................. 44
  4.2.3 Optimisation of relational tables ................................. 45
  4.2.4 Estimation of storage requirements for vertical partition .. 46

# Storage management

5.1 Multilevel transient and persistent storage devices ............... 48
5.2 Basic concepts ............................................................... 49
5.3 Resources management ...................................................... 49
5.4 Resource allocation plans ................................................ 50
5.5 Implementation of resource allocation plans ......................... 52

# Experiments

6.1 Step 1: Estimation of predicted workload ............................ 53
6.2 Step 2: Estimation of target workload level . . . . . . . . . . . . . . 56
6.3 Step 3: Discovering time frames with queries . . . . . . . . . . . . 57
6.4 Step 4: Vertical partitions and relational tables . . . . . . . . . . 57
   6.4.1 Step 4.1: Finding tuning plans . . . . . . . . . . . . . . . . 57
   6.4.2 Step 4.2: Optimization of tuning plans . . . . . . . . . . . 59
6.5 Step 5: Implementation of experiment . . . . . . . . . . . . . . . 61
6.6 Step 6: Experiment results . . . . . . . . . . . . . . . . . . . . . . 63
6.7 Outcomes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63

7 Summary and future work 65
   7.1 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
   7.2 Future work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66

Appendices 67
   .1 Appendix 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
      .1.1 TPC-H benchmark database . . . . . . . . . . . . . . . . . . 68
      .1.2 Create table statements for TPC-H benchmark . . . . . . 68
      .1.3 Conceptual Schema for TPC-H . . . . . . . . . . . . . . . . 73

Bibliography 74
List of Tables

1  Rows and size of each table in TPC benchmark database  . . . . . .  68
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Discover the patterns based on events</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Predicted workload</td>
<td>7</td>
</tr>
<tr>
<td>1.3</td>
<td>Experiment 1 results</td>
<td>10</td>
</tr>
<tr>
<td>1.4</td>
<td>Experiment 2 results</td>
<td>12</td>
</tr>
<tr>
<td>1.5</td>
<td>Experiment 3 results</td>
<td>13</td>
</tr>
<tr>
<td>1.6</td>
<td>Experiment 4 results</td>
<td>14</td>
</tr>
<tr>
<td>3.1</td>
<td>Entire workload structure with time frames</td>
<td>26</td>
</tr>
<tr>
<td>3.2</td>
<td>An overview of the controller</td>
<td>35</td>
</tr>
<tr>
<td>3.3</td>
<td>An overview of the implementation processes</td>
<td>38</td>
</tr>
<tr>
<td>6.1</td>
<td>Predicted workload</td>
<td>56</td>
</tr>
<tr>
<td>6.2</td>
<td>Predicted workload with target workload level</td>
<td>57</td>
</tr>
<tr>
<td>6.3</td>
<td>Experiment result</td>
<td>63</td>
</tr>
<tr>
<td>1</td>
<td>Conceptual Schema for TPC-H</td>
<td>73</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Performance tuning of database systems

Performance tuning of database systems is an important task for all organisations that require efficient services from decision support systems, e-businesses, online stores, data warehousing and so on. The profitability of an organisation almost entirely depends on the satisfaction of its customers. A poor performance of a database system manifests itself as low response time when processing customers’ requests. Long response time discourages customers from using a system, and as a consequence, it contributes to low profits. Therefore, the business organisations, which use database systems, need expert database administrators to tune the performance of database systems. Database administrators must possess a vast amount of knowledge on how to configure the system parameters, and how quickly they should react to changing workloads. There are two main ways to tune the performance of database systems. These include manual tuning by database administrators [28] and automated tuning by a database system itself [5].

1.1.1 Manual performance tuning

Currently, database administrators manually tune database systems with his/her knowledge of the anticipated workload [27]. Tuning a database system manually is a tedious and a time-consuming job for database administrators. When the workloads suddenly increase, database administrators must respond very quickly to reduce the workload. If they are unable to do it on time, a database system
1.1. Performance tuning of database systems

suffers delays, and it could collapse at any time.

1.1.2 Automated performance tuning

Automated performance tuning [5] tries to replace human database administrators with system software. A system that automatically tunes performance should be able to create and drop persistent storage structures like indices and materialised views. It also should be able to dynamically re-allocate transient storage resources, such as data buffer cache, library cache and so on. In automated performance tuning, the system analyses the future workloads and depending on the results the system allocates or releases the resources which are no longer used. The final outcomes of the automatic performance tuning include the elimination of workload peaks and overall reduction of the workload level.

1.1.3 Predicting database workloads

The workloads can be predicted because, in the reality, very similar data processing tasks are repeated every year, every month, and every day, and every certain period of time. In a database system, we can discover the workload information from audit trials, from tracing user applications and from logs. We can analyse the result from audit trials and then, predict the workload like when the query can occur in the future. Furthermore, we can also predict the repetition of SQL statements in the future. Although, predicted workload might not be exactly the same as the actual workload. For example, certain workload for the university enrolment system occurred in 2016 in the first week of March but the same that predicted workload occurred in 2017 in the last week of February. That means, a predicted workload may occur earlier or later than what we expect. A work [29] proposes a new technique to solve the problem, through reorganization of the predicted workload based on the events. A solutions is to find a set of events \( \{e_1, \cdots, e_n\} \) for a certain period of time. Next, they associated period pattern with those events to determine the location of pattern depend on events like \( \langle e_1, p_1, e_2, e_3, p_2, p_4, \cdots, p_n, e_n \rangle \) where \( e_i \) is an event and \( p_i \) is a pattern that occurred
1.1. Performance tuning of database systems

between different events. The different period of time can be distributed events in
different time. After that, the events are mapped in the future period of time to
estimate when the similar pattern would occur in the future. For example, pattern
\( p_1 \) occurred between \( e_1 \) and \( e_2 \) in the year 2016 March, then find those \( e_1 \) and \( e_2 \)
in the year 2017 February and \( p_1 \) have to occur within this period of time. That
means the pattern \( p_1 \) occurred early in the current year, therefore, we have to
prepare performance tuning plan for \( p_1 \) before it is occurred in current time.

![Figure 1.1: Discover the patterns based on events](image)

The predictions of future database workloads allow for the automated perfor-
mance tuning of database systems. Analysis of the future workloads determines
the periods of high and low workload time. Therefore, it is possible to automate
the performance tuning of the database systems by re-organizing the persistent
data structures at low workload time in order to improve the performance during
heavy workload.

If in a predicted workload, a high workload follows a low workload, the tuning
techniques can be applied in the low workload time before the occurrence of a
heavy workload. Although a prediction can go wrong, for example, when an event
does not happen on time or if it never happens. For example, Jack is sick, and
he takes medical leave so the events do not happen, or Jack is late for the train;
in this case, the estimated events might happen later, whenever it is favourable.
When some queries from a predicted workload may come late or may not come at
all, the implementation of a preparation plan related to delayed queries may hold the resources for a long time until those queries appear in the system. Waiting a long time for delayed queries blocks the resources for the processing of other queries. Therefore, a quick decision in terms of what a system should do in such situation is required. It can either eliminate a preparation plan or wait until those queries come. If it waits for the delayed queries, fewer resources will be available for other queries. If it eliminates resources related to those queries, the workload associated with the queries will not change when the queries appear in a system.

1.2 The problem

A workload usually becomes heavy because a larger number of users process the resource demanding tasks in a more or less the same period of time. Before a period of high workload occurs, the expected data processing tasks can be examined, and the decisions on how to improve processing tasks that contribute to the delays and how high processing costs can be made. The best workload is a well-balanced workload and such that it does not exceed the physical capacities of the available hardware. It means that a well-balanced workload does not exceed the assumed target workload level and that it is close to the assumed target workload level. The target workload level is a level that is acceptable for the available hardware resources. The problem is how to automatically balance the workload such that the workload has no workload peaks, and the workload does not exceed a given target level. In this thesis, we show how to allocate the resources within the low workload time for the internal re-organization of the database structures to reduce the high workload to a level below a given target. Processing of the preparation tasks during low workload can increase the low workload level. Therefore, the processing of the preparation tasks must be controlled such that it does not exceed the given target workload level.
1.3 Motivating example

In this section, we describe four experiments to show that the problem can be solved. Experiments show that we can achieve the original workload level lower than the given target workload level and balance the workload level. As a motivating example, we create two workload peaks in a given period of time. We use off-the-shelf commercial database system running on a personal computer with a processor 2.50GHz Intel, 4.00 GB of main memory, and a 250 GB hard drive. As a sample database, we use TPC-H [23] benchmark database. The structure of TPC-H show in Appendix.1. We create three users that concurrently access the database and submit their applications in approximately the same period of time, creating a synthetic workload.

1.3.1 Implementation of synthetic workload

We set a time period of experiment for 20 minutes and we create three SQL scripts to simulate a behaviour of three users. First, we divide a time period into 20 smaller time units where the length of each time unit is one minute. After that, we run SQL scripts for each user in more or less the same period of time.

In each one of the SQL scripts, we set three minutes sleeping time from time unit 1 to time unit 3. Then, starting from time unit 4, each SQL script sequentially submits the following five queries.

$q_1$

SELECT L_ORDERKEY
FROM LINEITEM
WHERE L_QUANTITY > 40;

$q_2$

SELECT L_QUANTITY, O_TOTALPRICE, C_ADDRESS, N_NAME
FROM LINEITEM JOIN ORDERS
ON L_ORDERKEY = O_ORDERKEY
JOIN CUSTOMER ON C_CUSTKEY = O_CUSTKEY
1.3. Motivating example

JOIN NATION ON C_NATIONKEY = N_NATIONKEY
WHERE L_QUANTITY = 10;

$q_3$

SELECT O_ORDERKEY
FROM ORDERS
WHERE O_TOTALPRICE > 1000;

$q_4$

SELECT C_ADDRESS
FROM CUSTOMER
WHERE C_CUSTKEY = 18114;

$q_5$

SELECT L_ORDERKEY , O_TOTALPRICE
FROM LINEITEM JOIN ORDERS
ON L_ORDERKEY = O_ORDERKEY
WHERE L_QUANTITY = 10;

When the processing of queries is completed, we set the sleeping time to four minutes. After the sleeping time, another three queries are sequentially submitted by the three users.

$q_6$

SELECT L_ORDERKEY , O_TOTALPRICE
FROM LINEITEM JOIN ORDERS
ON L_ORDERKEY = O_ORDERKEY
WHERE L_QUANTITY < 10;

$q_7$

SELECT L_QUANTITY
FROM LINEITEM
WHERE L_LINENUMBER > 30;
I selected those 8 queries for the experiments 1 to 4, because most of the queries are operating on large relational tables in TPC-H a database. The complex queries over the largest tables can create a workload peak because access to such tables require the large number of read block operations and construction of indices over those tables also require the large number of write operations. An objective of the experiment is to pick the queries that create the workload with two very high peaks, see Figure 1.2. As planned, two peaks occur from time unit 5 to time unit 11 and from time unit 15 to time unit 20.

![Figure 1.2: Predicted workload](image)
1.3.2 Estimation of a target workload level

We examine the synthetic workload generated by the experiment, and we add it as target workload level. The target workload level is computed by the summation of a total number of read/write operations performed by a database system at a given period of time and divided by a total number of time units. In this example, the value of the total number of read operations is 17880 and the value of the total number of time units is 20. Therefore, we compute the average target workload level as $17880/20 = 894$. Based on the target workload level found, we identify two time frames. The first time frame from time unit 1 to time unit 11 and the second time frame from time unit 12 to 20. Every time frame starts with a low workload period and ends with a high workload period.

1.3.3 Experiment 1

We create a preparation script to reduce the high workloads. In our solution, we use the index-only processing technique to improve performance. We create indices such that each query is processed with an index only and access to the relational tables is not needed. It is commonly called as “index-only processing technique”. Without the index, we have to read all data blocks that contain all rows in the table. If we create indices, then in the worst case we have to only read data blocks at leaf level of indices instead of reading the entire tables. In the experiments, the indices are created within low workload time to eliminate the peaks at high workload time. However, the problem of creation index within low workload is such that creation of indices may significantly increase the predicted workload and may appear the new workload peaks. For example, we create indices to cut high peaks to become low workload and because of that creation, we do not want to make another peak in low workload time. Because the creation of indices may require a large number of write operations. In the experiments, we verify creation of indices does not create additional workload peaks in low workload time.

In this experiment, we take into account all the columns whose names are included in the queries. Then, we order the columns such that the columns included
1.3. Motivating example

in the `WHERE` clause of queries are followed by the columns in the `JOIN` clause and in the `SELECT` clause. Next, we create an index on those ordered columns. The indices are created in the following way.

```sql
CREATE INDEX L_IDX1 ON LINEITEM(L_QUANTITY,L_ORDERKEY);
CREATE INDEX O_IDX1 ON ORDERS(O_ORDERKEY, O_CUSTKEY, O_TOTALPRICE);
CREATE INDEX O_IDX2 ON ORDERS(O_TOTALPRICE, O_ORDERKEY);
CREATE INDEX C_IDX1 ON CUSTOMER(C_CUSTKEY, C_NATIONKEY, C_ADDRESS);
CREATE INDEX N_IDX1 ON NATION(N_NATIONKEY,N_NAME);
```

According to the predicted workload, an estimated execution time for the first high workload takes seven minutes. Therefore, we set the time to seven minutes in order for the system to sleep. Within that period of time, the user executes the first period of high workload. After that, we write the script for creation of indices that are supposed to eliminate the second workload peak as following.

```sql
CREATE INDEX L_IDX1 ON LINEITEM(L_QUANTITY, L_ORDERKEY);
CREATE INDEX O_IDX1 ON ORDERS(O_ORDERKEY, O_TOTALPRICE);
CREATE INDEX L_IDX2 ON LINEITEM(L_LINENUMBER, L_QUANTITY);
CREATE INDEX C_IDX1 ON CUSTOMER(C_ACCTBAL, C_ADDRESS);
```

In the optimisation processes, the duplicated and the unnecessary indices are removed. In this example, `L_IDX1` from both time frames are duplicated, and we do not need to execute again during the low workload of the second time frame. Next, we found that `O_IDX1` from the second time frame is overlapped with the first time frame’s `O_IDX1`; therefore, it is not necessary to generate this index again in the second time frame. The rest of the indices implemented in the first time frame need to be dropped because they are no longer used in the second time frame. We drop indices at this stage because we operate on multi-layer storage devices. It means the indices are always created at the highest level of multi-layer persistence storages to speed up the write operation when you create the indices and also to speed up the read operation on the indices in the future. Due to limited space, we need to remove unnecessary indices in this stage to free up the memory for future
use. The step for drop indices is necessary for all experiments to get more device spaces for future. The following indices are dropped.

```sql
DROP INDEX L_IDX1;
DROP INDEX O_IDX2;
DROP INDEX C_IDX1;
DROP INDEX N_IDX1;
```

After that, we update the creation indices for the second time frame in the following way.

```sql
CREATE INDEX L_IDX2 ON LINEITEM(L_LINENUMBER, L_QUANTITY);
CREATE INDEX C_IDX1 ON CUSTOMER(C_NAME, C_ADDRESS);
```

When the index preparation script is ready, we concurrently process it together with three user scripts. The results are visualised in figure 1.3. In this experiment, we schedule the creation of indices in the low workload time not to create another peak and reduce the high workload. The results confirm that there is no more workload peak in this experiment.

![Figure 1.3: Experiment 1 results](image)
1.3.4 Experiment 2

In this experiment, we take into account all the columns whose names are included in the queries. Then, we randomly order the columns and write the script for creating indices that require the first five queries to occur within the high workload of the first time frame as follows.

\begin{verbatim}
CREATE INDEX L_IDX1 ON LINEITEM(L_ORDERKEY, L_QUANTITY);
CREATE INDEX O_IDX1 ON ORDERS(O_CUSTKEY, O_TOTALPRICE, O_ORDERKEY);
CREATE INDEX C_IDX1 ON CUSTOMER(C_ADDRESS,C_CUSTKEY, C_NATIONKEY);
CREATE INDEX N_IDX1 ON NATION(N_NAME,N_NATIONKEY);
\end{verbatim}

According to the predicted workload, an estimated execution time for the first high workload takes seven minutes. Therefore, we set the time to seven minutes while making the system sleep. Within that period of time, the user will execute the first period of high workload. After that, we drop some indices which are unnecessary for the second time frame as follows.

\begin{verbatim}
DROP INDEX O_IDX2;
DROP INDEX C_IDX1;
DROP INDEX N_IDX1;
\end{verbatim}

Next, we write the script for the creation of indices that require the second three queries as follows.

\begin{verbatim}
CREATE INDEX L_IDX2 ON LINEITEM(L_QUANTITY, L_LINENUMBER);
CREATE INDEX C_IDX1 ON CUSTOMER(C_ADDRESS, C_ACCTBAL);
\end{verbatim}

When the index preparation script is ready, we run it together concurrently with three user scripts in more or less the same time. The results are visualised in figure 1.4.
1.3.5 Experiment 3

In this experiment, we assume that we do not have enough persistent storage at higher performance levels of multi-tier storages and time to create all the indices. So, we need to make the decision to remove some index instead of creating all indices. In this experiment, we decided to remove the indices related to the largest table in the database like the `LINEITEM` table and creating indices only for medium/smaller tables like `ORDERS`, `CUSTOMER` and `NATION`. The following indices are created for the first time frame.

```sql
CREATE INDEX O_IDX1 ON ORDERS(O_ORDERKEY, O_CUSTKEY, O_TOTALPRICE);
CREATE INDEX C_IDX1 ON CUSTOMER(C_CUSTKEY, C_NATIONKEY, C_ADDRESS);
CREATE INDEX N_IDX1 ON NATION(N_NATIONKEY, N_NAME);
```

According to the predicted workload, an estimated execution time for the first high workload takes seven minutes. Therefore, we set seven minutes for the system to sleep. We drop some indices which are unnecessary for the second time frame and create the indices for medium/small tables as follows.

```sql
DROP INDEX O_IDX1;
DROP INDEX C_IDX1;
```
1.3. Motivating example

When the index preparation script is ready, we run it together concurrently with three user scripts in more or less the same time. The results are visualised in figure 1.5.

![Figure 1.5: Experiment 3 results](image)

### 1.3.6 Experiment 4

We assume that we do not have enough space and time to create all the indices. So, we need to make the decision to remove some indices instead of creating all of them. In this experiment, we decide to remove the indices related to the smaller tables which are the `CUSTOMER` and the `NATION` tables. Indices for first time frame are created as follows.

```sql
CREATE INDEX L_IDX1 ON LINEITEM(L_QUANTITY, L_ORDERKEY);
CREATE INDEX O_IDX1 ON ORDERS(O_ORDERKEY, O_CUSTKEY, O_TOTALPRICE);
CREATE INDEX O_IDX2 ON ORDERS(O_TOTALPRICE, O_ORDERKEY);
```

According to the predicted workload, an estimated execution time for the first high workload takes seven minutes. Therefore, we set seven minutes for the system
1.3. Motivating example

to sleep. We drop some indices which are unnecessary for the second time frame and create the indices for the second time frame as follows.

```
DROP INDEX L_IDX1;
DROP INDEX O_IDX2;
CREATE INDEX L_IDX2 ON LINEITEM(L_QUANTITY, L_LINENUMBER);
```

When the index preparation script is ready, we run it together concurrently with three user scripts in more or less the same time. The results are visualised in figure 1.6.

![Figure 1.6: Experiment 4 results](image)
1.3.7 Conclusions from the experiments

Through the experiments described above, we demonstrate that the workload peak can be eliminated by using a technique of creating indices of low workload time and by using "index only query processing technique". We used the index-only processing technique of the queries in experiments to improve the performance of database systems. There is no need to rewrite queries by using index-only processing technique. For example, if we create the materialized view, we have to re-write the query to access the corresponding materialized view; on the other hand, if we create the index, we do not need to re-write the query.

By comparing the experiments, we can make decisions, for example, choosing the proper order of columns based on the structures of the queries. Moreover, a target workload level allows us to determine the low and the high workload level. That target workload level creates like boundaries, for example, the workload level of creating indices cannot be over the target workload level, and that indices are to be created within the low workload time only.

1.4 The objectives

A global objective of the thesis is to invent the algorithms for the automated performance tuning of database systems. The new algorithms proposed in the thesis allow for better balancing of the workload and for the elimination of workload peaks. Additionally, we use multilayer storage devices in the database systems when the automated performance tuning is processed. The specific research questions are as follows.

(1) How should the predicted workload be analysed in order to find the low and the high workload period?

(2) How should the entire time frame be divided into smaller time frames?

(3) How to find out what should be and what can be computed during the periods of low workload?
(4) What performance tuning techniques should be applied to reduce high workload?

(5) How should available multi-level persistence storage devices be used to improve the performance of the database system?

(6) What should be done when the predicted workload does not match the real workload?

(7) What should be done when the workload for the processing of preparation tasks is larger than a given target workload level?

(8) How should the data preparations tasks be balanced during low workload in order to avoid target limit from exceeding?

1.5 The planned solutions

To answer the questions listed in the previous section, we propose the following strategy.

(1) The target workload level is estimated from information about past behaviour of a system. Based on that target workload level, we determine the low and the high workloads.

(2) An entire time frame is divided into the smaller time frames. Each time frame follows the sequence of a low workload time and a high workload time.

(3) The tuning plans are generated by processing of each time frame. Then, the resources required to implement for those tuning plans are allocated during the low period of each time frame. First, we generate the best vertical partitions plans. Then, we execute indices on columns selected from vertical partitions within low periods of workload. When vertical partitions plans are more or less similar to original tables then, we move tables to faster devices.

(4) To reduce the high workload, the index-only processing and the multi-level persistence storage devices are used to implement the tuning plans.
(5) According to the priority level of tuning plans, the resources required to implement the tuning plans are allocated within the faster devices of multi-layer persistence storage devices.

(6) When the predicted queries do not appear on time or do not appear at all, the resources related to those queries must be removed when we move to the next time frame. When the predicted workload gets higher than we expected or predicted, i.e. higher than the target workload, we stop creating indices and wait until the workload goes down. When the predicted workload is lower than we expected or predicted, i.e. when the workload is lower than the target workload, we create indices for the oncoming high workload.

(7) When the workload for processing the preparation tasks is larger than a given target workload level, we stop the entire machinery and just wait until the workload becomes lower than a given target workload level.

(8) Before allocating the data preparations tasks within the low workload time, the workload level for data preparation tasks must be lower than the target workload. If it is lower than the target workload level, then we create the indices. If it is higher than the target workload level, then we stop the creation of indices.

1.6 Thesis outline

This thesis is separated into the following chapters:

- Chapter 2: Literature review
  This chapter presents a review of the literature that relates to the automatic performance tuning of database systems, including the solutions which are proposed by the previous researchers.

- Chapter 3: Automated tuning processes
  This chapter provides solutions in terms of how to create time frames over
the entire time frame. Then, we propose the processes of the controller that manages the entire procedure of the automated tuning processes.

- Chapter 4: Vertical partitioning
  This chapter discusses the overview of the vertical partitioning. That includes discussions about how to generate the best vertical partitions.

- Chapter 5: Resources management
  This chapter shows the processes of resources management systems that generate the allocation plans over the low workload time.

- Chapter 6: Experiments
  This chapter shows proofs that the algorithms proposed from the previous chapters are the best solutions for automated performance tuning processes.

- Chapter 7: Summary and conclusions
  This chapter summarises this thesis and describes the conclusions.
Chapter 2

Literature review

2.1 Self-tune database systems

In the last few decades, tuning database systems have been listed in the topic of research area for database systems [17]. Currently, the structures of database systems have become more complicated; therefore, a larger number of database administrators are needed to tune those complicated database systems. The manual tuning of database systems takes an enormous amount of effort and is a complicated job for database management administrators (DBAs) [8]. Therefore, researchers proposed different solutions to tune database systems automatically. The self-tuning techniques

- auto-generate indices before execution [18, 6, 25],
- analyse the workload and implement materializes indices based on predicted workload [18],
- automatically analyse the workload and generate the particular columns and rows for partitioning [6, 7],
- automatically select the data and push the data on cache by using cache-based optimization techniques [24],
- push the data on share memory by using shared memory multiprocessor database systems technique [16], and
• use fuzzy-based self-tuning techniques to tune the database system automatically [17].

2.2 Indexing, materialized view and partitioning

S. Chaudhuri et al. 2007 in [6] discussed the different techniques used over the past decade for self-tuning database systems. S. Chaudhuri et al. described index-based tuning techniques, materialized view and partitioning. Chaudhuri et al. highlighted different opinions from various papers. Finally, they analysed the pros and the cons of commercial Database Management System (DBMS) tools for tuning database systems like the Oracle Database 11g [3]. The tools generated execution plans depending on a set of SQL statements. Based on these plans, the tools allowed users to choose among the plans. After that, the tools start to improve the performance tuning processes by implementing the tuning plans that selected by users. This paper [6] shows that most of the researchers focused on indexing and materialized view techniques to solve the problem of the self-tuning database system.

Most tuning techniques generate processing plans based on the predicted workload. There are too many techniques to discover predicted workload, and we decided to choose the technique proposed in [30]; this study shows how to discover patterns in workloads. They got audit trail which contains information about the scopes of user applications and SQL statements. Then, they transformed SQL statements as syntax trees of extended relational expressions and created a syntax tree table. This table included tree number, operation, left and right arguments, workload name, and time-stamps.

Next, they got a reduced syntax tree table by removing common sub-trees from the syntax tree table and updated the frequencies. After that, they created a workload histogram of an audit trail by setting time units. Then, they discovered elementary period patterns by using both the syntax tree table and the workload histograms of individual syntax trees. Finally, they generated the periodic composite patterns by combining the elementary period patterns. This paper [30]
shows that it is easier to find elementary periodic patterns and to compose them into complex ones instead of directly searching for all the complex patterns.

According to the predicted workload, the commercial DBMS tools are being used in various techniques to tune the workload for database systems which are indexing, materialized view and partitioning [6]. In addition, some techniques like on-line index is tuned to the unexpected query which means that query is not included in the predicted workload. There are two techniques for automatic on-line index algorithm, proposed in [18]. They are automatic and semi-automatic tuning.

The automatic on-line indexing algorithm was COLT (Continuous On-Line Tuning) [19] and it generated the set of indexes based on the input set of the query. Those set of indexes were not over the limited budget. This algorithm mainly focused on minimal overhead based on predicted future workload.

The semi-automatic on-line indexing algorithm, partitioned the set of candidate indexes into smaller subsets and chose the best set of index based on the current query. According to the authors’ experiments, semi-automatic achieved better performance than COLT. In summary, COLT was less expensive, and semi-automatic could choose the best set of indexes. The final algorithm which was a semi-automatic on-line index provided the list of the index (recommends the set of indexes) and allowed database administrators to select whether they want to create or drop the index. Both on-line indexing tuning techniques generated the selection of indices.

A selection of index for workload was important to implement the index. Authors in [21] showed a problem in Microsoft’s SQL Server’s index selection tool. That tool generated the set index of the predicted workload submitted by DBAs. They proposed their algorithm of an index selection tool for the Microsoft SQL server. They proposed three novel techniques in [21], which are query-specific-best-configuration algorithm, reduced number of configuration algorithm, and controlled multi-column indexes by an iterative technique. Their techniques showed an increase in the overall efficiency and were qualified to choose indexes.

Furthermore, the authors proposed other techniques for the selection of indices
in [20]. They had been using index selection heuristics in the paper mentioned above. Similarly, authors in [25] proposed DB2 Adviser, which was used for a index-based query optimizer. That tool could suggest multi-column virtual indexes. There were too many tools like Oracle [9], which could suggest the best set of indices based on the input workload. Those works also showed how to choose the best set of the indices for determining the workload but not for automated performance tuning. The work presented in [11, 10] showed that the database system could be self-tuned by using index selection and drop. They proposed architectures for DBMS’s self-tuning and showed how to tune database systems by using the index technique.

One of the tuning techniques used for tuning the database systems is materialized view. The book [2] describes the advantages of materialized view and it argues that such method of performance tuning is one of the best techniques to speed up query processing in database systems. The materialized view can speed up query processing when the queries used joins multiple tables and aggregates like SUM. Materialized view stores the pre-computed summary information. Therefore, it can reduce the query processing cost (I/O, CPU, and memory costs) which are related to processing queries.

Another technique called database partitioning is used to tune the database systems. There are two methods under database partitioning: vertical partition [15] and horizontal partition [22]. The authors S. Agrawal et al. in [1] proposed both the partitioning techniques although they recommended vertical fragments and allowed database administrators to implement those partitions. The authors Z. Liu et al. in [15] present the dynamic vertical partitioning of relational tables. They apply automatic performance tuning techniques based on predicted workload. They detect the low workload time and implement vertical partitions in the current workload time. In other words, dynamic attribute clustering is also called dynamic vertical partitioning. The dynamic attribute clustering technique generates attribute clusters based on the predicted workload [13].

The horizontal partition also generates partitions by using two tasks, which are fragmentation selection and dimension table selection [12].
partition generates the number of partitions or fragments based on frequently-used queries (predicted workload). Authors L. Bellatreche et al. in [4] proposed selecting the horizontal fragmentation technique. In addition, there are several cases to consider while choosing the best dimension tables, which are frequently-used dimension, largest dimension, minimum share and so on [22]. Horizontal partitioning allows a better performance of database systems compared to single table partitioning.

### 2.3 Fuzzy-based tuning

The authors from [17] proposed self-tuning for the database system by using fuzzy-based tuning. Their approach mainly focuses on buffer cache and putting the resources over buffer cache. They used three main inputs in their approach:

- buffer cache size (BCS),
- buffer-hit-ration (BHR), and
- database size (DBSize).

Based on these three inputs, they used five rules to set the buffer cache (BCS). These rules are [17]:

- set BCS to low, when BHR is high, User load is low, and DBSize is high.
- set BCS to moderate, when BHR is high, user load is medium and DBSize is high.
- set BCS to high, when BHR is medium, the user load is high and DBSize is high.
- set BCS to very high, when BHR is poor, user load is high and DBSize is high.
- set BCS to low, when BHR is high, user load is less and DBSize is small.
2.4 Cache-base optimization

Cache-based optimization [24] is one of the techniques to tune the database systems. That technique moves the data from a slower hard drive to cache memory to achieve a better performance of database systems. In the cache-based optimization technique proposed in [24], no custom-designed hardware support is needed. Moreover, the authors M. C. Murphy et al. in [16] showed that a shared-memory multiprocessor database system also provides less execution time. They consider memory buffers, disk bandwidth, and general purpose processors while allocating the resources required to process the given queries.

2.5 Summary and conclusion

The literature review in the thesis highlights the different ways how a database system can be automatically tuned. Most of them are focusing on how to tune the predicted workload with the techniques like partitioning, buffer-caching and the others. Most of techniques proposed above are based on transformation of a submitted query before actual processing starts. For example, when user submits the query the processor re-writes that query to use a materialized view. The technique proposed in the thesis does not change any queries or re-write the queries submitted by a user. We create the tuning plans/processing plans based on the predicted workloads. Those predictions do not appear in fixed the period of time. The predicted workload can be changed according to the events. Based on the changes, the order of the plans can also change. Therefore, the query processing plans are not fixed in any period of time, they can change depending on the events of the current period of time.

According to my best knowledge, this is a completely new solution in automated performance tuning. This technique no need change the original application. Moreover, the algorithms proposed in the thesis automatically generate processing plans and according to the requirements of the current situation. That means, when the events and workloads are different from what we expected, then the algorithms are generate different plans based on changes.
Chapter 3

Automated tuning processes

This chapter presents the procedures that partition tuning time into the time frames and that manage the entire process within the time frames. The tuning time is a period of time when the future workload can be predicted for the analysis of the past workload. For example, if we would like to prepare tuning scripts for the coming month then the tuning time become a month. The tuning time is determined by information about predicted workload. To get time frames we split tuning time such that each time frame is determined by a pair of low and high workload times. The low and high workload time is determined by target workload level. A workload lower then target workload level called low workload time and a workload higher then target workload level called high workload time. The computation of target workload level shows in section 3.1.3. Next, the controller processes the time frame and it creates the preparation plans for the low workload periods. That plan may not over the target workload level.

The controller processes the time frames and the target workload level and it creates the preparation plans for the low workload periods. These preparation plans include the creation of indices on multi-layer storage devices. The plans provide the reduction of the processing time for predicted queries that occurred during high workload.

Furthermore, the controller decides whether the preparation plans are needed to implement or not. When the previous time unit’s workload level is greater than the target workload level, the controller does not execute the preparation plans for the next time unit. It is because the workload for the next time unit
3.1. Time frames

This section introduces the algorithms that partitions the tuning time into time frames. A predicted set of queries with the estimated time ranges is used to create time frames.

3.1.1 Time units

First, the automated performance tuning supposed to be partitioned into fixed size time units.
3.1. Time frames

Definition 3.1.1 Let $T_{aut} = (t_{start}, t_{end})$ be a **period of time** over which the automated tuning is supposed to be performed, where $t_{start}$ is the starting time-stamp and $t_{end}$ is the ending time-stamp of the period. We divide $T_{aut}$ into fixed sizes **time units** $u_1, \cdots, u_n$. Each time unit $u_i$ is a pair of time-stamps $(t_i, t_j)$, where $t_i$ is the starting time-stamp of the current time unit $u_i$, and $t_j$ is the ending time-stamp of the current time unit $u_i$ for $i = 1, \cdots, n$. $t_j$ is computed as $t_i + ((t_{end} - t_{start})/n)$, where $n$ is the **total number of time units**.

3.1.2 Workload estimations

Algorithm 3.1 uses a predicted set of queries to compute the estimated amounts of workload for time units. The amount of workload in a time unit is equal to the total number of estimated read/write operations processed for each time unit. The **EXPLAIN PLAN** statement of SQL is applied to find the total number of estimated read/write operations for each query.

Definition 3.1.2 A set of **predicted queries** for a period of automated tuning $T_{aut}$ is a set of pairs $\{(q_1, T_1), \cdots, (q_m, T_m)\}$, where $q_i$ is a query and the time range $T_i$ is a pair $(u_{start}, u_{end})$, where $u_{start}$ is the number of time unit when $q_i$ starts and $u_{end}$ is the number of time units when $q_i$ ends for $i = 1, \cdots, m$. The query can be occurring any time within a period of time that determine by the time units $u_{start}$ and $u_{end}$. For example, let the length of time range $T_1 = (u_{start}, u_{end})$ be 1 hour where $u_{start} = 9:00$ and $u_{end} = 10:00$. Let query $q_1$ occur within the time range of $T_1$. That mean, the predicted query $q_1$ will occur any time from 9:00 to 10:00.

Definition 3.1.3 The **workload value** $w_i$ represents the total number of read/write operations that occur in a time unit $u_i$ for $i=1, \cdots, n$. If $W = \langle w_1, \cdots, w_n \rangle$ is a sequence of workload values, then the total number of workload values $n$ is the same as the total number of time units $n$. 


Algorithm 3.1:

**Input:** A set of predicted queries with time ranges \{(q_1, T_1), \ldots, (q_m, T_m)\} and the total number of time units \(n\).

**Output:** A sequence of workload values \(W = \langle w_1, \ldots, w_n \rangle\).

(1) Create a sequence of workload values \(\langle w_1, \ldots, w_n \rangle\), where the total number of workload values, \(n\), is equal to the total number of time units \(n\).

(2) Set all the workload values to zero values like \(w_i = 0\) for all \(i = 1, \ldots, n\).

(3) Iterate over the set of pairs \{(q_i, T_i)\} to \{(q_m, T_m)\}. Let the current pair is \((q_i, T_i)\), where \(T_i = (u_{\text{start}}, u_{\text{end}})\).

(3.1) Apply EXPLAIN PLAN statement of SQL on \(q_i\) and get the total read/write operations \(p_i\).

(3.2) Compute the total number of time units \(num_{q_i}\) needed to process \(q_i\).

\[
num_{q_i} = (\text{indexOf}_{\text{u}_{\text{end}}} - \text{indexOf}_{\text{u}_{\text{start}}}) + 1.
\]

For example, let \(T_i = (u_1, u_4)\) then, \(num_{q_i} = (4 - 1) + 1 = 4\). The total time unit needed to process by \(q_i\) is 4.

(3.3) Compute the workload value that needs to be distributed over each time unit that \(q_i\) needs to process. The workload value \(w_{q_i} = p_i/num_{q_i}\).

(3.4) Iterate from \(t_{\text{start}}\) to \(t_{\text{end}}\). Let the current time unit is \(u_i\) and the workload value for \(u_i\) is \(w_i\).

(3.4.1) Update the workload value \(w_i = w_i + w_{q_i}\).

For example, let the time range of query \(q_i\) be \((u_{\text{start}}, u_{\text{end}}) = (u_4, u_7)\). The total number of read/write operations \(p_i = 100\). The number of time units needed to process \(q_i\) is \(u_{q_i} = (7-4)+1 = 4\). Then, we need to append \(w_{q_i} = p_i/u_{q_i} = 100/4=25\) into workload \(w_4, w_5, w_6,\) and \(w_7\). Therefore, we need to append 25 to each workload value, like \(w_4 = w_4 + 25, w_5 = w_5 + 25, w_6 = w_6+25\) and \(w_7 = w_7 + 25\).
3.1. Time frames

3.1.3 Target workload level

This section describes how to compute the target workload level that determines the low/high workload in each time unit. If the workload is greater than the target workload level, it is known as high workload. Otherwise, it is known as low workload.

Definition 3.1.4 Let $w_{\text{target}}$ be a target workload level. A target workload level is computed as $w_{\text{target}} = \frac{w_{\text{total}}}{n}$, where $w_{\text{total}} = \sum_{i=1}^{n} w_i$.

For example, let us have 5 queries $\{(q_1, (u_2, u_4)), (q_2, (u_3, u_4)), (q_3, (u_4, u_5)), (q_4, (u_3, u_6)), (q_5, (u_3, u_7))\}$. The total number of read/write operations from each of the queries are $p_1 = 30$, $p_2 = 10$, $p_3 = 40$, $p_4 = 20$, and $p_5 = 100$. The total number of time units $n = 7$. According to the formula $w_{\text{total}} = 30 + 10 + 40 + 20 + 100 = 200$ and $w_{\text{target}} = \frac{w_{\text{total}}}{n} = \frac{200}{7} = 28.57$.

3.1.4 Creating time frames

The algorithm 3.2 creates time frames. The time frames provide a smaller period of time instead of an entire period of time $T_{\text{aut}}$ in order to apply the automated tuning process.

Definition 3.1.5 High workload is a workload which is greater than $w_{\text{target}}$.

Definition 3.1.6 Low workload is a workload which is less than $w_{\text{target}}$.

Definition 3.1.7 Time frame $d$ is defined as triple $(u_{\text{start}}, u_{\text{high}}, u_{\text{end}})$. $u_{\text{start}}$ is a time unit for the beginning of a low workload. $u_{\text{high}}$ is the smallest number of time units where the workload is greater than $w_{\text{target}}$ and such that $u_{\text{high}} \geq u_{\text{start}}$. $u_{\text{end}}$ is the smallest number of time unit where the workload is less than $w_{\text{target}}$ and such that $u_{\text{end}} \geq u_{\text{high}}$. A sequence of time frames is denoted as $D = \langle d_1, \cdots, d_n \rangle$. For example, see figure 3.1.
Algorithm 3.2:

**Input:** A target workload level $w_{\text{target}}$, a sequence of workload values $\langle w_1, \cdots, w_n \rangle$, and the total number of time units $n$.

**Output:** A sequence of time frames $D = \langle d_1, \cdots, d_m \rangle$.

(1) Make a sequence of time frames $D$ empty.

(2) Iterate over the sequence of workload values $\langle w_1, \cdots, w_n \rangle$. Let the current workload is $w_i$. Let the current time frame is $d_i$ and the start time unit of $d_i$ be $u_{\text{start}} = \emptyset$, the end time unit of $d_i$ be $u_{\text{end}} = \emptyset$ and the start time unit of $d_i$ for high workload be $u_{\text{high}} = \emptyset$.

(2.1) If $w_i < w_{\text{target}}$ and $u_{\text{start}} = \emptyset$ then set the $u_{\text{start}} = u_i$.

(2.2) Else if $w_i > w_{\text{target}}$ and $u_{\text{start}} \neq \emptyset$ and $u_{\text{high}} = \emptyset$, then set the $u_{\text{high}} = u_i$.

(2.3) Else if $w_i > w_{\text{target}}$ and $u_{\text{start}} \neq \emptyset$ and $u_{\text{high}} \neq \emptyset$ and $w_{i+1} < w_{\text{target}}$ then $u_{\text{end}} = u_i$. Next, create a time frame like $d_i = (u_{\text{start}}, u_{\text{high}}, u_{\text{end}})$ and update $D = D \cup d_i$.

(3) Finally, we will get $D = \langle d_1, \cdots, d_m \rangle$.

### 3.1.5 Assigning queries to time frames

This section shows how to find a set of queries predicted for the high workload period of each time frame and creates a sequence of time frames, each one associated with a set of queries. By extracting a set of queries for each time frame, we can make the preparation plan only for those queries instead of applying tuning on all the queries that occur in each time frame. It not only saves the time but also reduces the high workload.

**Definition 3.1.8** The *acceptable overlap percentage* $O_p$ is a value of percentage which makes the decision in terms of whether query $(q_i, T_i)$ should be
3.1. Time frames

included in high workload time or not. It allows determining whether the overlapped percentage of a particular query \((q_i, T_i)\) over high workload time is small or large.

The \(O_p\) parameter needed because a query starts processing in the low workload time and continues in the high workload time or the query starts being processed in the high workload time and continues through the low workload time. Therefore, if the processing time of query partly overlaps the high workload time, the overlap percentage over high workload is needed to be compute. If the query’s overlap percentage is larger than \(O_p\), that query needs to be add into a set of queries to tune for the current time frame. If the overlap percentage of a query is less than \(O_p\), there is no need for the query to be tuned, and we can ignore that query.

**Algorithm 3.3:**

**Input:** A sequence of time frames \(D = \langle d_1, \cdots, d_n \rangle\), a set of predicted queries with time ranges \(\{ (q_1, T_1), \cdots, (q_m, T_m) \}\), and an acceptable overlap percentage \(O_p\).

**Output:** A sequence of time frames with queries \(D' = \langle (Q_1, d_1), \cdots, (Q_n, d_n) \rangle\).

1. Copy \(D\) to \(D'\) and replace each \(d_i \in D\) with a pair \((Q_i, d_i)\). Put \(Q_i\) is empty for \(i = 1, \cdots, n\).

2. Iterate over \(D'\) and let the current pair be \((Q_i, d_i)\). Let the time ranges for the time frame is \(d_i = (u_w, u_x, u_y)\) where \(u_w\) is starting time unit, \(u_x\) is high time unit, and \(u_y\) is ending time unit for \(d_i\).

2.1. Iterate over \(\{ (q_1, T_1), \cdots, (q_m, T_m) \}\) and let the current pair be \((q_i, T_i)\) and \(T_i := (u_i, u_j)\) where \(u_i\) is starting time unit and \(u_j\) is ending time unit for \(T_i\).

2.1.1. If \(u_i > u_x\) and \(u_i < u_y\) then append \(q_i\) into \(Q_i = Q_i \cup q_i\) and remove \(q_i\) from the original predicted set of queries, because the time range of \(q_i\) is totally in the high period of time.
3.1. Time frames

(2.1.2) Else, if \((u_i \langle u_x \text{ and } u_j \rangle u_x)\) or \((u_i \rangle u_x \text{ and } u_i \rangle u_y)\) or and \((u_i \langle u_x \text{ and } u_j \rangle u_y)\), then the queries partly overlap in the high workload period. Therefore, we need to compute the numbers of overlapped time units are followings.

(2.1.2.1) Iterate the time ranges from \(u_i\) to \(u_j\) and let the current time unit be \(u_z\). Let the number of time units overlapping in the high workload period be \(num_{ol} = 0\) and the number of time units in the high workload period be \(num_{high} = (\text{indexof}_y - \text{indexof}_x) + 1\).

- If \(u_z \geq u_x\) or \(u_z \leq u_y\) then the number of time units overlapped \(num_{ol} = num_{ol} + 1\).

(2.1.2.2) Compute the overlap percentage \(O_i = (100 / num_{high}) \times num_{ol}\).

(2.1.2.3) If \(O_i \geq O_p\) then append \(q_i\) into \(Q_i = Q_i \cup q_i\) and remove \(q_i\) from the original predicted set of queries.

(2.1.3) Else, if \(u_i\) and \(u_j\) are less than \(u_x\) or greater than \(u_y\) then, we do not need to add \(q_i\) into \(Q_i\) because \(q_i\) does not occur within the high workload period.

(3) Finally, we will get \(D' = ((Q_1, d_1), \cdots, (Q_n, d_n))\).

For example, we have a sequence of time frames \(D = ((u_1, u_3, u_6), (u_7, u_10, u_14))\), a set of predicted queries is \(\{(q_1, (u_3, u_4)), (q_2, (u_1, u_5)), (q_3, (u_7, u_9)), (q_4, (u_8, u_16)), (q_5, (u_8, u_14))\}\), and an acceptable overlap percentage be \(O_p = 75\%\). Algorithm 3.3 takes the first time frame \((u_1, u_3, u_6)\). We found that \(q_1\) occurs within the high workload period of the first window frame; therefore, we append it into \(Q_1 = \{q_1\}\).

Next, we found that \(q_2\) also occurs within the high workload period of the first window frame, but it also happened within the low workload period. Therefore, we need to compute the overlap percentage. The total number of time units for high workload is 4, and the total number of time units overlapping within the high workload is 3. Therefore, the overlap percentage is \((100/4) * 3 = 75\%\), and it is equal to the acceptable percentage and updates the set of queries as \(Q_1 = \{q_1, q_2\}\). We find the overlap percentages for \(q_3, q_4,\) and \(q_5\); following this, we get \(Q_2 =\)
\{q_4, q_5\}. Finally, we create a sequence of time frames with queries \(D' = \langle (d_1, Q_1), (d_2, Q_2) \rangle\).

## 3.2 Controller

The controller uses the target workload level and the time frames with the assigned queries created in the previous step to generate the best processing plans within low workload time. The processing plans include creation of vertical partitions, generation of resources allocation plans on multi-layer devices, and cleaning unneeded resources. The procedures regarding how to implement those plans by using indexing and storage allocation methods are represented in algorithms 4.1 and 5.1 respectively.

According to the plans, the controller implements those plans when the first time frame starts. If the actual workload level is lower than target workload level, controller implements the plans. When the actual workload level is higher than what we expected then, controller stop implements the plan. An overview of the controller is shown in figure 3.3.

**Definition 3.2.1** A linear sequence of **storage devices** is denoted as \(M = \langle m_1, \cdots, m_n \rangle\). Each device \(m_i\) is defined as triple \((r_i, w_i, c_i)\), where \(r_i\) is the read speed of the device \(m_i\), \(w_i\) is the write speed of the device \(m_i\), and \(c_i\) is the capacity of the device \(m_i\). The sequence of \(M\) is arranged from the highest to the lowest speed depending on an average speed of read/write operations measured as the total number of bytes read/write per second. For example, if we have \(n\) level devices, \(m_1\) is the highest speed and \(m_n\) is the lowest speed.

**Definition 3.2.2** A **schema of the vertical partition** of relational table \(r_i\) is denoted as \(v_i\), and it is defined as a pair \(v_i = (r_i, \langle \{c_1, \cdots, c_n\}, \cdots, \{c_i, \cdots, c_j\} \rangle)\), where \(r_i\) is the relational table and \(\langle \{c_1, \cdots, c_n\}, \cdots, \{c_i, \cdots, c_j\} \rangle\) is a sequence of the set of columns required to process the particular predicted queries. A set of vertical partitions is denoted as \(V = \{v_1, \cdots, v_n\}\).
3.2. Controller

Algorithm 3.4 generates the plans that determine what resources are to be allocated at the low workload time and in what order. It accepts the sequence of time frames with queries $D'$ and passes one sequence of pairs by one to algorithm 4.1 in order to get the tuning plans. After that, the controller passes those plans and implemented plans from previous time frames to algorithm 5.1 and it processes the cleaning tasks. Next, the controller gets the resources allocation plans from algorithm 5.2 by passing the information of optimized tuning plans and storage devices. Finally, controller makes decision to implement those plans by sending signal "Start" and "Stop" to implementation processes at algorithm 3.5. The algorithm 3.5 implements the plans according to signal send by controller. The figure 3.2 shows the overview of the controller and the figure 3.4 shows the overview of the implementation processes.
3.2. Controller

Figure 3.2: An overview of the controller
Algorithm 3.4:

**Input:** A sequence of pairs of time frames \( D' = \langle (Q_1, d_1), \cdots, (Q_n, d_n) \rangle \) and a sequence of devices \( M = \langle m_1, \cdots, m_j \rangle \).

1. The controller listening the incoming a sequence of time frames \( D' \). Before the first time frame of \( D' \) starts, the controller generates the tuning plans for each time frame.

2. Iterate over a sequence of pairs of time frames \( D' = \langle (Q_1, d_1), \cdots, (Q_n, d_n) \rangle \). Let current time frame be \((Q_i, d_i) \in D'\).

   2.1) Pass the \( Q_i \) to algorithm 4.1 and get the schemas of vertical partitions \( V_{Q_i} \) and the schemas of relational tables \( T_{Q_i} \) (see algorithms 4.3 and 4.4).

   2.2) Optimize the plans \( V_{Q_i} \) and \( T_{Q_i} \) (see algorithm 5.1).

   2.3) Pass updated plans \( V_{Q_i}, T_{Q_i}, \) and \( M \) to algorithm 5.2 and get the result of the resource allocation plan \( A_{Q_i} = \langle (a_i, m_k), \cdots, (a_n, m_j) \rangle \), where \( a_i \) is either a schema of vertical partition or a schema of relational table and \( m_j \) is a location of device where \( a_i \) should be located.

3. At the implementation point, the controller is activated at the beginning of each time unit and record the workload level after each time unit passed. Depend on the workload level, the controller makes decision weather system needs to implement plans for the current time unit or not.

4. Wait until the time arrives at starting time unit of current time frame \((Q_i, d_i)\). If current time unit \( u_c = u_{start} \) starting time unit of current time frame, then the current time frame begins. When the beginning of the time frame \((Q_i, d_i)\), we get a sequence of resource allocation plans \( A_{Q_i} \) for the current time frame and begin to implement the plans.

5. Let next time unit be \( u_n \) and previous time unit be \( u_p \). The workload of time unit \( u_p \) is \( w_p \). If \( u_c = u_{start} \) then, \( u_p = \text{null} \) and \( w_p = 0 \). Because there are
3.2. Controller

no previous time unit and workload when the current time unit is beginning
of tuning processes.

(6) If \( u_c = u_{\text{start}} \) (current time unit equal to beginning of tuning processes) or
\( u_c \neq u_{\text{end}} \) (current time unit not equal to the ending of tuning processes
that means we are still in the current time frame) and \( w_p < w_{\text{target}} \) (the
previous workload is lower than target workload level means that we are
in the low workload time) then, the controller sends the signal ”Start” to
the implementation processes when the implementation processes are not
started.

(7) If \( u_c = u_{\text{end}} \) (at the end of current time frame) then, check whether the
implementation processes are not stopped. If they are not stopped then, the
controller sends the signal ”Stop” to the implementation processes and
releases the resources.

(7.1) If the current time frame is the last time frame and current time unit
is the ending of current time frame then, we de-allocate all plans that
we implemented in previous time frames.

(7.2) Else, the controller releases the resources when predicted queries never
occur within high workload time. At the end of current time frame \((u_c
= u_{\text{end}})\), the controller finds the predicted queries did not occur within
the high workload time. Then, the controller obtains the resources
required to create those queries, and then release the resources related
to those queries. Next, the controller waiting for next time frame and
back to the point 4.

(8) Else, if \( w_p > w_{\text{target}} \) (previous workload level is higher than target workload
level) then, check whether the implementation processes are not stopped. If
they are not stopped then, the controller sends signal ”Stop” to the imple-
mentation processes for current time unit and back to point 6. Because, if
the workload of previous time unit is high, the workload for current time
unit can be high too. Therefore, the controller waits for a low workload time
to implement the plans.

![Diagram of implementation processes]

Figure 3.3: An overview of the implementation processes

**Algorithm 3.5:**

**Input:** A sequence of resources allocation plan \( A_Q = ((a_1, m_j), \ldots, (a_n, m_k)) \) and the signal from the controller.

1. Get a sequence of plans \( A_Q \) for current time frame.

2. Wait the signal from the controller.

   (2.1) If the signal is "Start" then, check \( A_Q \) is empty or not.

   (2.1.1) If \( A_Q \) not empty, then iterate over \( A_Q \) and let current plan is \((a_i, m_k)\).
(2.1.1.1) Implementation process takes action by algorithm 5.3 by using $(a_i, m_k)$.

(2.1.1.2) Remove implemented plan from a sequence of plans.

(2.1.1.3) If the signal receive ”Stop” then, exit the iteration point 1.1.1 and go to point 1.2.

(2.1.2) Else, if $A_{Q_i}$ empty, then there are no more plan to implement for current time frame and end implementation for current time frame.

(2.2) If the signal is ”Stop” then, algorithm stop submitting the plans for implementation and move to point 2.
Chapter 4

Vertical partitioning

This chapter presents the algorithms that find the best schemas of the vertical partitions to be created at a low workload time. The algorithms reduce the costs of creating vertical partitions, which is supposed to improve the performance during high workload time. Performance tuning through vertical partitioning involves the selection of columns from the relational tables used by the queries during high workload.

According to Definition 3.2.2, a schema of vertical partition of relational table $r_i$ is the pair $v_i = (r_i, \langle\{c_1, \cdots, c_n\}, \cdots, \{c_i, \cdots, c_j\}\rangle)$. The first element of a pair is relational table $r_i$ and the second element is a sequence of sets of columns required to process the predicted query.

In our approach, vertical partitions are implemented as indices. Typically, vertical partitioning splits a table into multiple tables linked with primary and foreign keys linking the partitions. In this thesis, we use a different technique to create the vertical partitions. We do not split original table into new tables. Instead, we create indices over the most frequently used columns. The implementation through indexing are not a logical grouping of columns, no need to create a new tables and no need to use join to extract the information from tables. Indexing is one of the techniques, which allow creation of the vertical partition stimulation of the vertical partition. After that database system will automatically pick traversal of leaf level of an index. Then, the query processor is able to detect such situation. Therefore, it is the important technique because it is actually implemented vertical partitioning. Moreover, it is no need to re-write the queries to extract the
4.1 Basic concepts

Definition 4.1.1 Let \( freq(v_i) \) be a frequency of how many times a vertical partition \( v_i \) occurs in a set of queries. It is contrary to how many times a vertical partition \( v_i \) is used within a current time frame.

Definition 4.1.2 Let \( cost(v_i) \) be the cost of creating a vertical partition \( v_i \). The cost is computed according to the cost model presented in [15]. The total cost is expressed as the total number of physical read/write block operations which are required to implement a set of queries.

Definition 4.1.3 Let \( storage(v_i) \) be a storage requirement for a vertical partition \( v_i \). The computations of storage requirements are included in algorithm 4.3.

Definition 4.1.4 A schema of relational table is denoted as \( t_i \), and it is defined as a pair \( t_i = (t_i, \{c_1, \ldots, c_n\}) \), where \( t_i \) is a relational table and \( \{c_1, \ldots, c_n\} \) is a set of columns included in relation to table \( r_i \). A set of tables is denoted as \( T = \{t_1, \ldots, t_n\} \).
4.2 Overview of vertical partitioning

This section includes the algorithms that find the schemas of vertical partitions. The algorithm processes queries that occur during high workload. The algorithms find the smallest set of columns for each vertical partition that are to be created during low workload.

4.2.1 Schemas of vertical partitions and relational tables

Algorithm 4.1 finds the smallest set of columns for each vertical partition required to process the queries in $Q$. Furthermore, to create the index on an almost entire schema of a table is not a good idea. Therefore, we plan to keep those tables on a set of relational table $T$ instead of the vertical partition. Improving access to relational tables is described in chapter 5.

Algorithm 4.1:

**Input:** A set of queries $Q$.

**Output:** A set of schemas of vertical partitions $V = \{v_1, \cdots, v_n\}$ that improve performance of query processing for the queries in $Q$. A set of relational tables $T = \{t_1, \cdots, t_m\}$ that does not have any vertical partitions.

1. Make a set of schemas of vertical partitions $V$ and schemas of relational tables $T$ empty.

2. Iterate from $q_1$ to $q_n \in Q$. Let the current query be $q_i$. Let a set of pairs is $\{(r_1, \{c_1, \cdots, c_n\}), \cdots, (r_n, \{c_i, \cdots, c_j\})\}$ where each pair include a relational table with a set of columns processed by $q_i$.

2.1 Iterate over a set of pairs of the relational tables with columns processed by $q_i$.

- Let the current pair be $(r_i, \{c_1, \cdots, c_n\})$.

- Let the current table be $r_i$. 
4.2. Overview of vertical partitioning

Let a set of columns processed by \( q_i \) be \( X_i = \{c_1, \cdots, c_n\} \) and the total size of all columns be \( \text{total}_x \).

Let a set of columns included in original table be \( Y_i = \{c_1, \cdots, c_m\} \) and the total size of all the columns is \( \text{total}_y \).

(2.1.1) If \( \frac{\text{total}_x}{\text{total}_y} = 0.9 \), then check whether that table is in a set of \( T \). If it is not, append the relational table \( t_i = (r_i, Y_i) \) into \( T \). The 90% of columns in the original table are processed by \( q_i \). If we create a vertical partition, then it will be less benefit. Therefore, we just append that table to a set of relational tables \( T \).

(2.1.2) Else, create one vertical partition like \( v_i = (r_i, \{\cdots\}) \).

(2.1.2.1) If the \textbf{WHERE} clause from \( q_i \) is found, and the disjunction is not found, or the \textbf{ORDER BY} clause from \( q_i \) is found, then get the columns used in the \textbf{WHERE} clause and the \textbf{ORDER BY} clause. Let the columns found in those clauses be \( W_i = \{c_i, \cdots, c_j\} \). Append the set of columns \( W_i \) to \( v_i = (r_i, \{W_i\}) \). Next, remove those columns from the current set of columns \( X_i = X_i \cap W_i \).

(2.1.2.2) If the \textbf{JOIN} clause from \( q_i \) is found, get the columns used in that clause and create a set of columns \( J_i \). Append \( J_i \) into \( W_i = W_i \cup J_i \). After that, remove those columns from the current set \( X_i = X_i \cap J_i \).

(2.1.2.3) If the current set \( X_i \) is not empty, append it into a sequence of sets in \( v_i = (r_i, \{W_i, X_i\}) \).

(3) Repeat until all the queries are processed.

\textbf{Example 4.2.1} As a simple example, assume that we have the queries \( q_1 : \text{SELECT a FROM x WHERE b \geq 10 and c \geq 10} \); and \( q_2 : \text{SELECT x.a, y.b FROM x JOIN y ON x.a=y.b} \); and \( q_3 : \text{SELECT a FROM x WHERE b = 20} \). First, the algorithm analyse \( q_1 \) and generates the table with the smallest set of columns \( x \) and \( \{a, b, c\} \). Next, the algorithm founds the \textbf{WHERE} clause is used in conjunction operator. Then, algorithm gets the columns which include in that condition like \( \{b, c\} \). Then, the algorithm adds those columns into a set \( \{b, c\} \) and append into...
4.2. Overview of vertical partitioning

a sequences of sets like \( v_1 = (x, \{\{b, c\}\}) \) and removes \( b \) and \( c \) from a smallest set of columns. After that, the algorithm adds the remaining columns from smallest set of columns into a set of columns like \( v_1 = (x, \{\{b\}, \{a\}\}) \). Repeat the same processes for \( q_2 \) and we found two vertical partitions for \( q_2 \) which are \( v_2 = (x, \{\{a\}\}) \) and \( v_3 = (y, \{\{b\}\}) \). After processing \( q_3 \), we found one vertical partition \( v_4 = (x, \{\{a\}\}) \). Finally, we get the vertical partitions \( V = \{ (x, \{\{b, c\}, \{a\}\}), (x, \{\{a\}\}), (y, \{\{b\}\}), (x, \{\{b\}, \{a\}\}) \} \).

4.2.2 Optimisation of vertical partitions

Algorithm 4.2 analyses a set of vertical partitions \( V = \{ v_1, \cdots, v_n \} \) created by an algorithm 4.1 and create a set of vertical partitions \( V' \) with costs and frequencies \( V' = \{ (v_1, \text{cost}(v_1), \text{freq}(v_1)), \cdots, (v_n, \text{cost}(v_n), \text{freq}(v_n)) \} \).

When a relation table is too small and appears only once in a set of queries, there is no need to create any indices. Indexing of small tables is not profitable because the query optimizer traversing the index searches for data rather than performing a simple table scan. For example, in the TPC-H database, the \textsc{Nation} and the \textsc{Region} tables are too small, and we do not need to create indices on them. Algorithm 4.2 requires inputting a size \( t_{\text{size}} \) to determine how large a table support be to have a vertical partition. If the table is smaller than \( t_{\text{size}} \), we can say that the table does not need to be constructed for vertical partition.

Algorithm 4.2:

\textbf{Input:} A set of schemas of vertical partitions \( V \) and a size \( t_{\text{size}} \) to determine whether a table is small or large. 

\textbf{Output:} A set of vertical partitions with costs and frequencies \( V' = \{ (v_1, \text{cost}(v_1), \text{freq}(v_1)), \cdots, (v_n, \text{cost}(v_n), \text{freq}(v_n)) \} \).

1. Make a set of schemas of vertical partitions with the costs and the frequencies \( V' \) empty.

2. Iterate over \( V \) and let the current vertical partition be \( v_i = (r_i, \{\{c_1, \cdots, c_n\}, \cdots, \{c_j, \cdots, c_k\}\}) \). Let the sequence of sets of columns be \( C_i = \{\{c_1, \cdots, c_n\}, \cdots, \{c_j, \cdots, c_k\}\} \).
4.2. Overview of vertical partitioning

\[ \cdots, c_n \}, \cdots, \{c_j, \cdots, c_k \} \}, \text{where the first set of columns are } W_i = \{c_1, \cdots, c_n \}. \text{ Let the frequency for } v_i \text{ is } freq(v_i) = 1. \]

(2.1) Iterate over \( V - v_i \) and let current vertical partition be \( v_j \) and its sequence of sets of columns be \( C_j \), and the first set of columns be \( W_j \).

(2.1.1) If tables from both \( v_i \) and \( v_j \) are the same and \( C_i = C_j \) both sequences of sets of columns from \( v_i \) and \( v_j \) are the same, then remove \( v_j \) from \( V \) and updated \( freq(v_i) = freq(v_i) + 1. \)

(2.1.2) Else, if tables from both \( v_i \) and \( v_j \) are the same and all elements from \( W_j \) include in \( W_i \), then update the \( W_i \) into two sets like \( W_j \), \( W_k \) where \( W_k = W_i \cap W_j \). For example, \( W_i = a, b, c \) and \( W_j = a, b \) then \( W_i \) into two sets like \( a, b, c \). Then update the \( C_i = \langle W_j, W_k, \cdots, \{c_j, \cdots, c_k \} \rangle \).

- Remove all columns from \( C_j \) which are included in \( C_i \). Append the rest of the columns from \( C_j \) to \( C_i \) and remove \( v_j \) from \( V \) and updated \( freq(v_i) = freq(v_i) + 1. \)

(2.2) If \( freq(v_i) = 1 \) and size of table \( r_i \) is less than \( t_{size} \) (\( r_i \) include in smaller tables of database and appears only one time) then remove \( v_i \) form \( V \).

(2.3) Else compute the \( cost(v_i) \) then, create triple like \( (v_i, cost(v_i), freq(v_i)) \) and append it into \( V' \).

**Example 4.2.2** Algorithm 4.2 gets the input from the output of algorithm 4.1. Then, it computes the cost for each schema of vertical partitions. Then, take one schema of vertical partition from \( V \) and compare with other. We found that \((x, \langle \{b\}, \{a\} \rangle)\) is included in \((x, \langle \{b, c\}, \{a\} \rangle)\), then, merge them become \((x, \langle \{b\}, \{c\}, \{a\} \rangle)\). Next, we found that the others vertical are not related with each other. Therefore, the final result of algorithm 4.2 is \( V' = \{(x, \langle \{b\}, \{c\}, \{a\} \rangle), cost(v_1), 2), (x, \langle \{a\} \rangle), cost(v_2), 1), (y, \langle \{b\} \rangle), cost(v_3), 1) \).

4.2.3 Optimisation of relational tables

Algorithm 4.3 analyses a set of relational tables \( T = \{t_1, \cdots, t_n\} \) created by algorithm 4.1 and creates a set of relational tables with costs, frequencies, and
storage \( T' = \{(t_1, cost(t_1), freq(t_1), storage(t_1)), \ldots, (t_n, cost(t_n), freq(t_n), storage(t_n))\} \), where \( freq(t_i) \) is a frequency of how many times \( t_i \) appears within a set of queries, \( cost(t_i) \) is a cost of moving the relational table \( t_i \) from a slower device to a faster device, and \( storage(t_i) \) is the storage required for relational table \( t_i \).

Algorithm 4.3:

Input: A set of relational tables \( T \) and a set of queries \( Q \).
Output: A set of relational tables with cost, frequency and storage \( T' = \{(t_1, cost(t_1), freq(t_1), storage(t_1)), \ldots, (t_n, cost(t_n), freq(t_n), storage(t_n))\} \) of vertical partitions created with the costs and frequencies.

(1) Make a set of schemas of relational tables with the costs, frequencies and storages \( T' \) empty.

(2) Iterate over \( T \) and let current relational table be \( t_i = (r_i, \{c_1, \ldots, c_n\}) \).

(3.1) Count how many time \( t_i \) appears within a set of \( Q \). Then, update a frequency \( freq(t_i) \).

(3.2) If \( freq(t_i) = 1 \) then remove \( t_i \) from \( T \).

(3.3) Else compute the \( cost(v_i) \) and get the storage require for table \( storage(t_i) \) from data dictionary then, create quadruple like \( (t_i, cost(t_i), freq(t_i), storage(t_i)) \) and append it into \( T' \).

(3) Send the optimized a set of relational tables \( T'' \) to the controller.

4.2.4 Estimation of storage requirements for vertical partition

Algorithm 4.4 computes the storage required for implementing vertical partitions in \( V' \). In this work, vertical partitions are implemented by the index-only processing technique. Therefore, the algorithm estimates the storage requirements for each vertical partition by using both storage requirement for using leaf level index and
the storage requirements for the non-leaf level index. Finally, it sums both storages required for the leaf level and the non-leaf level index to get the total storage requirement $storage(v_i)$.

**Algorithm 4.4:**

**Input:** A set of schemas of vertical partitions $V'$.

**Output:** A set of schemas of vertical partitions $V''$ with storage requirements for each vertical partition.

1. Make $V''$ empty and copy \{(v_1, cost(v_1), freq(v_1)), \ldots, (v_n, cost(v_n), freq(v_n))\} $\in V'$ to $V''$. Then, set all storage sizes to zero like \{(v_1, cost(v_1), freq(v_1), storage(v_1) = 0), \ldots, (v_n, cost(v_n), freq(v_n), storage(v_n) = 0)\} $\in V''$.

2. Iterate over \{(v_1, cost(v_1), freq(v_1), storage(v_1)), \ldots, (v_n, cost(v_n), freq(v_n), storage(v_n))\} $\in V''$. Let current schema of vertical partition be \((v_i, cost(v_i), freq(v_i), storage(v_i))\). Let the block size is $size_b$, the length of columns in $v_i$ is $length_c$, the space to record the index pointer is $space_i$ and the number of rows in $v_i$ is $num_r$.

   2.1) Estimate the number of rows in a fully packed block $rows = size_b / (length_c + space_i)$.

   2.2) Estimate the storage required to keep the leaf level of index for $v_i$ is $leaf = num_r / rows$.

   2.3) Estimate the storage required to keep the intermediate and root levels of index for $v_i$ is $non-leaf = leaf / rows$.

   2.4) Calculate the storage require for vertical partitions $storage(v_i) = leaf + non-leaf$.

3. Send the optimized a set of vertical partitions $V''$ to the controller.
Chapter 5

Storage management

Nowadays, with advanced technologies of electronic storage systems being widely available, it is easy to install many different types and capacities of transient and persistent storages. Availability of the various types and capabilities of transient and persistent storages allows for more sophisticated implementations of database query processing systems. For example, it is possible to allocate more frequently used resources into faster storage devices. This chapter shows how to use multilevel transient and persistent storage devices to obtain better performance of a database system.

5.1 Multilevel transient and persistent storage devices

A collection of transient and persistent storage devices is a special hardware organisation of persistent and transient storage modules that can be visualised as a tiered storage pyramid structure [14]. The slowest, the cheapest, and the largest modules of persistent storage allocated at the bottom tier of the storage pyramid include slow hard drives. The smaller number of more expensive and faster storage modules like fast hard drives and solid state drives occupy the upper tiers of the pyramid. The fastest and the most expensive modules of transient storage devices are from the topmost and the smallest tiers of the storage pyramid.

The relational tables that are larger and less frequently used are initially kept at lower levels of persistent storage pyramid. More frequently used and smaller
tables are kept at the higher tiers of the persistent storage pyramid. Algorithms 5.1 and 5.2 determine a strategy for the dynamic allocation of storage resources for a given database workload.

Storage pyramid \( M = \langle m_1, \cdots, m_n \rangle \) is a sequence of storage devices where each device \( m_i \) \( (i = 1, \cdots, n) \) has a certain read speed \( r_i \), write speed \( w_i \) and capacity \( c_i \). The sequence of devices \( M \) is ordered by the total speed of the read and the write operations.

5.2 Basic concepts

Definition 5.2.1 A plan of resource allocation is a sequence \( A = \langle (a_1, m_k), \cdots, (a_n, m_j) \rangle \), where \( a_i \) is either a vertical partition \( v_i \) in \( V \) or a relational table \( t_i \) in \( T \), and \( m_k \) is the device location, where \( a_i \) is allocated.

5.3 Resources management

At the beginning of every time frame except the first time frame, the system needs to move the storage resources which are no longer used in the current time frame to lower tiers of the storage pyramid. When a vertical partition \( v_i \) or a relational table \( t_i \) is no longer used for the current time frame, we look for the index or the table that implemented according to \( v_i \) or \( t_i \). After that, we remove an index or move a table from higher level devices to original hard drive.

In algorithm 5.1, we use the set of vertical partitions \( V \), relational tables \( T \), and resource allocation plan \( A_{previous} \) to remove the unnecessary resources, where \( A_{previous} \) is a set of resource allocation plans that have already been implemented in previous time frames.

Algorithm 5.1

1. Iterate over \( V \cup T \) and let current step of elements be \( e_i \) where \( e_i \) is either a vertical partition \( v_i \) or a relational table \( t_i \).
Input: A set of vertical partitions $V$ and a set of relation tables $T$ ($V \cup T$) for current time frame. A sequence of resource allocation plans from previous time frames $A_{\text{previous}}$ and a sequence of storage pyramid $M = (m_1, \ldots, m_n)$ where $m_i$ is a storage device.

Output: An updated set of vertical partitions $V$ and a set of relational tables $T$.

(1.1) Iterate over $A_{\text{previous}}$ and let current step of resource allocation plan be $a_i$.

(1.1.1) If $a_i = e_i$ ($a_i$ already exists) then, remove $a_i$ from $A_{\text{previous}}$.

(1.2) If $a_i$ is not equal to all the vertical partitions from $V$ and all relational tables from $T$ that means $a_i$ is no longer needed and continue to 1.2.1.

(1.2.1) If $a_i$ is vertical partition $v_i$ then, looking for index related with that vertical partition and drop that index. Update the capacity of device $c_i = c_i + \text{storage}(v_i)$ where $c_i$ is capacity of device that dropped index created on it and $\text{storage}(v_i)$ is storage require to create that dropped index.

(2.2.2) Else if $a_i$ is relational table $t_i$ then, drop table related with $t_i$ from faster storage device and create $t_i$ on the lowest storage device $m_n$. Then increase the capacity of faster device where table created on it $m_i = m_i + \text{storage}(t_i)$. Next, reduce the lowest level device $m_n = m_n - \text{storage}(t_i)$ where $\text{storage}(t_i)$ is storage require to implement $t_i$.

5.4 Resource allocation plans

Algorithm 5.2 shows how to process the schemas of vertical partitions $V$ and the schemas of relational tables $T$. A set of schemas of vertical partitions $V = \{(v_1, \text{cost}(v_1), \text{freq}(v_1), \text{storage}(v_1)), \ldots, (v_n, \text{cost}(v_n), \text{freq}(v_n), \text{storage}(v_n))\}$ where $v_i$ is vertical partition, $\text{cost}(v_i)$ is the cost require for $v_i$, $\text{freq}(v_i)$ is how many times $v_i$ appears within current time frame, and $\text{storage}(v_i)$ is size require to create $v_i$. A set of schemas of relational tables $T = \{(t_1, \text{cost}(t_1), \text{freq}(t_1), \text{storage}(t_1)), \ldots,$
5.4. Resource allocation plans

\[(t_m, \text{cost}(t_n), \text{freq}(t_n), \text{storage}(t_n))\] where \(t_i\) is relational table, \(\text{cost}(t_i)\) is the cost require for \(t_i\), \(\text{freq}(t_i)\) is how many times \(t_i\) appears within current time frame, and \(\text{storage}(t_i)\) is size require to move \(t_i\) from slower device to faster device. Next, \(T \cup V\) is ordered by \(\text{cost}(v_i) \times \text{freq}(v_i)\) or \(\text{cost}(t_i) \times \text{freq}(t_i)\). After that, allocate them into a different layer of the device \(M\). Next, generate the allocation plan \(A\).

Algorithm 5.2:

**Input:** A set of schemas of vertical partitions \(V = \{(v_1, \text{cost}(v_1), \text{freq}(v_1), \text{storage}(v_1)), \cdots, (v_n, \text{cost}(v_n), \text{freq}(v_n), \text{storage}(v_n))\}\) and a set of schemas of relational tables \(T = \{(t_1, \text{cost}(t_1), \text{freq}(t_1), \text{storage}(t_1)), \cdots, (t_m, \text{cost}(t_n), \text{freq}(t_n), \text{storage}(t_n))\}\).

**Output:** A sequence of resource allocation plans for vertical partitions \(A = ((a_1, m_i), \cdots, (a_n, m_j))\).

1. Initially, make a sequence \(A\) empty.
2. Calculate the priority for every schemas of vertical partitions in \(V\) and \(T\) by using \(\text{cost}(v_i) \times \text{freq}(v_i)\) or \(\text{cost}(t_i) \times \text{freq}(t_i)\). Vertical partitions \(V\) ordered by \(\text{cost}(v_i) \times \text{freq}(v_i)\) and relational table \(T\) order by \(\text{cost}(t_i) \times \text{freq}(t_i)\).
3. Iterate over a sequence of storage device \(M\). Let current device be \(m_j\) and its capacity be \(c_j\).
   3.1. Iterate over \(V \cup T\) and current element be \(e_i\) where \(e_i\) is either a vertical partition \(v_i\) or a relational table \(t_i\) and storage requirement to implement \(e_i\) is \(\text{storage}(e_i)\).
      3.1.1. If \(\text{storage}(e_i) \leq c_j\) then, assign \(a_i = e_i\) and \(c_j = c_j - \text{storage}(e_i)\).
      Next, create a pair \((a_i, m_j)\) and append into \(A\).
4. Finally, we will get the sequence of resource allocation plans \(A = ((v_1, m_i), \cdots, (v_n, m_k))\).
5. Send optimized resource allocation plans \(A\) to controller.
(6) Update all capacity to actual free size. That means, all capacity in $M$ are updated to actual free size.

5.5 Implementation of resource allocation plans

After we get the optimized resource allocation plans, we can start implementing the plan. The input of this algorithm is a resource allocation plan $(a_i, m_j)$. Then, the algorithm will decide whether we need to create a table or an index. The input is only one resource allocation plan because the controller will pass a plan to implement in the current time unit. The controller will make the decision of creation time. Therefore, this algorithm does not implement the entire resource allocation plan.

Algorithm 5.3

**Input:** A resource allocation plan $(a_i, m_k)$ and a sequence of resource allocation plans $A_{\text{previous}}$ which included all resource allocation plans from previous time frames.

(1) Get input resource allocation plan $(a_i, m_k)$.

(1.1) If $a_i$ is vertical partition then, create the index on those columns in the storage device $m_k$.
- Update actual free space of $m_k = m_k - s_i$ where $s_i$ is the size for create particular index.
- Append $(a_i, m_k)$ to $A_{\text{previous}}$.

(1.2) If $a_i$ is relational table then, create table on the storage device $m_k$.
- Update actual free space of $m_k = m_k - s_i$ where $s_i$ is the size for create particular table.
- Append $(a_i, m_k)$ to $A_{\text{previous}}$. 
Our experiments used a synthetic TPC-H benchmark relational database [23] implemented on an "off-a-shelf" commercial database software. In the experiments, we used three storage level devices which included certain amounts of transient storage in a data buffer cache ($m_1$), a solid state drive ($m_2$), and a hard drive ($m_3$). We set up five users and let those users execute ten SQL statements concurrently. We used ten complex queries for this experiment, which are taken from TPC-H’s Template set [23]. The experiments are repeated several times to get more statistically reliable results.

### 6.1 Step 1: Estimation of predicted workload

First, we estimate the predicted workload according to algorithm 3.1 in chapter 3 as follows:

1. Execute the first five queries $q_1, q_2, q_3, q_4$ and $q_4$ from five users concurrently.
2. Sleep progresses for 3 minutes.
3. Execute another five queries $q_6, q_7, q_8, q_9$ and $q_{10}$ from five users concurrently.
4. Record the total time needed to create the entire time frame.
5. Divided the time frame into fixed time units.
6. Compute the total read and write operations for each time unit.
7. Pilot into the graph.

The ten queries are following

\(q_1\)

\[
\text{SELECT L\_ORDERKEY} \\
\text{FROM LINEITEM} \\
\text{WHERE L\_QUANTITY > 40};
\]

\(q_2\)

\[
\text{SELECT L\_ORDERKEY, SUM(L\_QUANTITY)} \\
\text{FROM LINEITEM} \\
\text{GROUP BY L\_ORDERKEY} \\
\text{HAVING SUM(L\_QUANTITY) > 100};
\]

\(q_3\)

\[
\text{SELECT L\_QUANTITY, O\_TOTALPRICE, C\_ADDRESS, N\_NAME} \\
\text{FROM LINEITEM} \\
\text{JOIN ORDERS ON L\_ORDERKEY = O\_ORDERKEY} \\
\text{JOIN CUSTOMER ON C\_CUSTKEY = O\_CUSTKEY} \\
\text{JOIN NATION ON C\_NATIONKEY = N\_NATIONKEY} \\
\text{WHERE L\_QUANTITY = 10};
\]

\(q_4\)

\[
\text{SELECT \ast FROM ORDERS;}
\]

\(q_4\)

\[
\text{SELECT \ast FROM ORDERS;}
\]

\(q_6\)

\[
\text{SELECT L\_ORDERKEY} \\
\text{FROM LINEITEM} \\
\text{WHERE L\_QUANTITY = 10;}
\]
6.1. Step 1: Estimation of predicted workload

$q_7$

SELECT L_QUANTITY, O_TOTALPRICE, C_ADDRESS, N_NAME
FROM LINEITEM JOIN ORDERS
ON L_ORDERKEY = O_ORDERKEY
JOIN CUSTOMER ON C_CUSTKEY = O_CUSTKEY
JOIN NATION ON C_NATIONKEY = N_NATIONKEY
WHERE L_QUANTITY > 40;

$q_8$

SELECT O_ORDERKEY
FROM ORDERS
WHERE O_TOTALPRICE > 1000;

$q_9$

SELECT C_ADDRESS
FROM CUSTOMER
WHERE C_CUSTKEY = 18114;

$q_{10}$

SELECT L_ORDERKEY, O_TOTALPRICE
FROM LINEITEM JOIN ORDERS
ON L_ORDERKEY = O_ORDERKEY
WHERE L_QUANTITY = 10;
6.2 Step 2: Estimation of target workload level

In this step, we estimate the target workload level to determine the low and the high workload. This workload level limited the workload of creation indices below the target level.

1. Get the total number of read/write operations.
2. Get the total number of time units.
3. Divided them to get an average workload called target workload level \( w_{target} \).

According step 2, the total number of read/write operation level is 123456 and the total number of time units is 20. The target workload level is \( \frac{24697}{20} = 1234.85 \).
6.3 Step 3: Discovering time frames with queries

According to the previous steps, we use algorithm 3.2 and identify two time frames with its queries $D = \langle (Q_1, d_1), (Q_2, d_2) \rangle$, where $Q_1 = \{q_1, q_2, q_3, q_4, q_5\}$, $d_1 = (\text{timeunit}_1, \text{timeunit}_5, \text{timeunit}_11)$, $Q_2 = \{q_6, q_7, q_8, q_9, q_{10}\}$ and $d_2 = (\text{timeunit}_12, \text{timeunit}_15, \text{timeunit}_20)$.

6.4 Step 4: Vertical partitions and relational tables

In this section, we create vertical partitions and relational tables for each query by using algorithm 4.1 in chapter 4.

6.4.1 Step 4.1: Finding tuning plans

The set of columns for query $q_1$ is as follows

$v_1 = (\text{LINEITEM}, <\text{LQUANTITY, LORDERKEY}>)$

The set of columns for query $q_2$ is as follows
The set of columns for query \( q_3 \) is as follows
\[
v_3 = (\text{LINEITEM}, <\text{L\_QUANTITY}, \text{L\_ORDERKEY}>)
\]

The set of columns for query \( q_4 \) is as follows
\[
v_4 = (\text{ORDERS}, <\text{O\_ORDERKEY}, \text{O\_CUSTKEY}, \text{O\_TOTALPRICE}>)
\]

The set of columns for query \( q_5 \) is as follows
\[
v_5 = (\text{CUSTOMER}, <\text{C\_CUSTKEY}, \text{C\_NATIONKEY}, \text{C\_ADDRESS}>)
\]

The set of columns for query \( q_6 \) is as follows
\[
v_6 = (\text{NATION}, <\text{N\_NATIONKEY}, \text{N\_NAME}>)
\]

Therefore, we are created relational table \( t \) instead of vertical partition \( v \) is as follows
\[
t_1 = (\text{ORDERS}, \text{O\_ORDERKEY}, \text{O\_CUSTKEY}, \text{O\_ORDERSTATUS}, \text{O\_TOTALPRICE}, \\
    \text{O\_ORDERDATE}, \text{O\_ORDER\_PRIORITY}, \text{O\_CLERK}, \text{O\_SHIP\_PRIORITY}, \text{O\_COMMENT})
\]

The set of columns for query \( q_7 \) is as follows
\[
v_7 = (\text{LINEITEM}, <\text{L\_QUANTITY}, \text{L\_ORDERKEY}>)
\]

The set of columns for query \( q_8 \) is as follows
\[
v_8 = (\text{LINEITEM}, <\text{L\_QUANTITY}, \text{L\_ORDERKEY}>)
\]

The set of columns for query \( q_9 \) is as follows
\[
v_9 = (\text{ORDERS}, <\text{O\_CUSTKEY}, \text{O\_ORDERKEY}, \text{O\_TOTALPRICE}>)
\]

The set of columns for query \( q_{10} \) is as follows
\[
v_{10} = (\text{CUSTOMER}, <\text{C\_CUSTKEY}, \text{C\_NATIONKEY}, \text{C\_ADDRESS}>)
\]

The set of columns for query \( q_{11} \) is as follows
\[
v_{11} = (\text{NATION}, <\text{N\_NATIONKEY}, \text{N\_NAME}>)
\]

The set of columns for query \( q_{12} \) is as follows
\[
v_{12} = (\text{ORDERS}, <\text{O\_TOTALPRICE}, \text{O\_ORDERKEY}>)
\]

The set of columns for query \( q_{13} \) is as follows
\[
v_{13} = (\text{CUSTOMER}, <\text{C\_CUSTKEY}, \text{C\_ADDRESS}>)
\]

The set of columns for query \( q_{14} \) is as follows
\[
v_{14} = (\text{NATION}, <\text{N\_NATIONKEY}, \text{N\_NAME}>)
\]

Vertical partitions for first time frame is \( V_1 = \{v_1, \ldots, v_6\} \) and relational tables for first time frame is \( T_1 = \{t_1, t_2\} \).
6.4. Step 4: Vertical partitions and relational tables

\[ v_{14} = (\text{LINEITEM, } <\text{L\_QUANTITY, L\_ORDERKEY}>)) \]
\[ v_{15} = (\text{ORDERS, } <\text{O\_ORDERKEY, O\_TOTALPRICE}>)) \]

Vertical partitions for second time frame is \( V_2 = \{v_7, \ldots, v_{15}\} \).

6.4.2 Step 4.2: Optimization of tuning plans

In this step, we create the quadruples \((v_i, \text{cost}(v_i), \text{freq}(v_i), \text{storage}(v_i))\), where \(\text{cost}(v_i)\) is the cost of creation of vertical partition \(v_i\), \(\text{freq}(v_i)\) is the frequency which denotes the number of times \(v_i\) partition is used by queries, and \(\text{storage}(v_i)\) are the storage requirements for \(v_i\). Then, we append these information to \(V'\).

Next, we create the quadruples \((t_i, \text{cost}(t_i), \text{freq}(t_i), \text{storage}(t_i))\), where \(\text{cost}(t_i)\) is the cost of moving a relational table \(t_i\) from a slower device to a faster device, \(\text{freq}(t_i)\) is the frequency of the number of times \(t_i\) is used by queries, and \(\text{storage}(t_i)\) are the storage requirements for \(t_i\). Then, we append these information to \(T'\).

The following are optimization procedure for a set of vertical partitions \(V_1\) by using algorithms 4.2 and 4.4 in chapter 4.

1. \(v_1, v_2\) and \(v_3\) are exactly the same therefore, we eliminate \(v_2\) and \(v_3\) then increase the frequency then create a quadruple like \((v_1, \text{cost}(v_1), 3, \text{storage}(v_1))\). Next, append it into \(V'_1\).

2. \(v_4\) cannot find similar vertical partitions then, create a quadruple like \((v_4, \text{cost}(v_4), 1, \text{storage}(v_4))\). Next, append it into \(V'_1\).

3. \(v_5\) cannot find similar vertical partitions then, create a quadruple like \((v_5, \text{cost}(v_5), 1, \text{storage}(v_5))\). Next, append it into \(V'_1\).

4. \(v_6\) partition over the small table called \text{NATION} therefore, we eliminate the \(v_6\) from \(V_1\).

5. The updated \(V'_1 = \{(v_1, \text{cost}(v_1), 3, \text{storage}(v_1)), (v_4, \text{cost}(v_4), 1, \text{storage}(v_4)), (v_5, \text{cost}(v_5), 1, \text{storage}(v_5))\}\).
6.4. Step 4: Vertical partitions and relational tables

The following are optimization procedure for a set of relational tables $T_1$ by using algorithm 4.3 in chapter 4.

1. $t_1$ and $t_2$ are exactly the same therefore, we eliminate $t_2$ then increase the frequency of $t_1$. Then, create a quadruple like $(t_1, \text{cost}(t_1), 1, \text{storage}(t_1))$. Next, append it into $T_1'$.

2. The updated $T_1' = \{(t_1, \text{cost}(t_1), 1, \text{storage}(t_1))\}$.

The following are optimization procedure for a set of vertical partitions $V_2$ by using algorithms 4.2 and 4.4 in chapter 4.

1. $v_7$, $v_8$ and $v_{14}$ are the same therefore, we eliminate $v_8$ and $v_{14}$ from $V_2$ then increase the frequency of $v_7$. Next we create a quadruple like $(v_7, \text{cost}(v_7), 3, \text{storage}(v_7))$. After that append it into $V_2'$.

2. $v_9$ and $v_{15}$ can be combine because first set of columns from $v_{15}$ includes in the fist set of columns from $v_9$. The combined vertical partition is $v_9 = (\text{ORDERS}, \langle\text{O_ORDERKEY}, \text{O_CUSTKEY}, \text{O_TOTALPRICE}\rangle)$. Then we create a quadruple like $(v_9, \text{cost}(v_9), 2, \text{storage}(v_9))$. Next, we append it into $V_2'$.

3. $v_{10}$ and $v_{13}$ can be combine too. The combined vertical partition is $v_{10} = (\text{CUSTOMER}, \langle\text{C_CUSTKEY}, \text{C_NATIONKEY}, \text{C_ADDRESS}\rangle)$. Then, we create a quadruple like $(v_{10}, \text{cost}(v_{10}), 2, \text{storage}(v_{10}))$. Next we append it into $V_2'$.

4. $v_{11}$ is partition over small table \textsc{Nation}, therefore, we no need to implement this vertical partition. Then, we just eliminate that $v_{11}$ from $V_2$.

5. $v_{12}$ no need any changes and create a quadruple like $(v_{12}, \text{cost}(v_{12}), 1, \text{storage}(v_{12}))$. Next, we append it into $V_2'$.

6. The updated $V_2' = \{(v_7, \text{cost}(v_7), 3, \text{storage}(v_7)), (v_9, \text{cost}(v_9), 2, \text{storage}(v_9)), (v_{10}, \text{cost}(v_{10}), 2, \text{storage}(v_{10})), (v_{12}, \text{cost}(v_{12}), 1, \text{storage}(v_{12}))\}$. 
6.5 Step 5: Implementation of experiment

The first step is to create a tablespace on $m_2$ to allocate resources on it. The script for creating tablespace is as follows:

```sql
CREATE TABLESPACE M2
  DATAFILE 'm_2/ts2.dbf'
  SIZE 4000M
  EXTENT MANAGEMENT LOCAL
  SEGMENT SPACE MANAGEMENT AUTO;
```

Next, we create a tuning script (script1.sql) for a low workload time as follows.

```sql
CREATE TABLE t1 AS
  (SELECT * FROM ORDERS)
  TABLESPACE m2;

DROP TABLE ORDERS;

RENAME TABLE t1 TO ORDERS;

CREATE INDEX v1 ON LINEITEM(L_QUANTITY, L_ORDERKEY);
ALTER INDEX v1 STORAGE (BUFFER_POOL KEEP);

CREATE INDEX v4 ON ORDERS(O_ORDERKEY, O_CUSTKEY, O_TOTALPRICE);
ALTER INDEX v4 STORAGE (BUFFER_POOL KEEP);

CREATE INDEX v5 ON CUSTOMER(C_CUSTKEY, C_NATIONKEY, C_ADDRESS);
ALTER INDEX v5 STORAGE (BUFFER_POOL KEEP);

\BEGIN
  DBMS_LOCK.sleep(300);
\END;
```
//v7 and v1 is equal therefore we no need to create v7 again.
//v9 and v4 is equal therefore we no need to create v9 again.
//v10 and v5 is equal therefore we no need to create v10 again.

CREATE INDEX v12 ON ORDERS(O_TOTALPRICE, O_ORDERKEY);
ALTER INDEX v12 STORAGE (BUFFER_POOL KEEP);

Next, we create a script (script2.sql) for high workload time as follows.

\BEGIN
DBMS_LOCK.sleep(240);
\END;
/
@q1.sql
@q2.sql
@q3.sql
@q4.sql
@q5.sql
\BEGIN
DBMS_LOCK.sleep(180);
\END;
/
@q6.sql
@q7.sql
@q8.sql
@q9.sql
@q10.sql
6.6 Step 6: Experiment results

The execution procedure as following

1. We execute script1.sql by admin account and script2.sql by 5 users concurrently.

2. The results shown in figure below.

![Figure 6.3: Experiment result]

6.7 Outcomes

This chapter has shown the solution of implementation vertical partitions and relational tables on pyramid storages for an automated performance tuning of the database system. We implement vertical partitions as an index only processing on the faster device and push the relational table up on the faster device. The experiments in this chapter shows:

1. Implementation vertical partitions as indices can be transparent to applications.

2. Execute indices over faster devices can extract data faster.
3. The larger vertical partitions can be implemented as tables on faster device of pyramid storages.

4. Achieve better cost and reduce execution time.

5. The workload has no workload peaks.

6. The workload does not exceed a given target level.
Chapter 7

Summary and future work

7.1 Summary

This thesis addresses the problem of automated performance tuning of database systems. The problem solved in the thesis is related to automated workload tuning and balancing. This thesis shows how to allocate the tuning tasks within the low workload time in order to reduce and balance the high workload properly. The techniques used in this thesis apply index-based vertical partitioning and multi-layer system of persistent storage. A motivating example of experiments show that it is possible to implement and to achieve lower and reduced workload by using both pyramid storage tiers and index-only processing.

We propose how to divide the given time frame into time units and smaller time frames. Next, we discuss the techniques of computing the target workload level for the entire time frame in order to control the level of low workload and high workload when the tuning is being processed. After that, we pass each time frame to the controller to manage the automated performance tuning processes.

We present the algorithms for the best selection of vertical partitions and relational tables based on the input set of queries from the controller. We explain how to pick the best procedure of vertical partitions and relational tables along with the estimation cost, frequencies, and storage for each partition and table.

We also propose the usage of storage management processes. First, we propose an algorithm to clean unnecessary resources which are no longer used in the current time frame and recycle the unwanted plans performed at the beginning of every
time frame, except the first one. Next, we allocate the vertical partitions and the
relational tables over the storage pyramid according to the priority level of each
partition and table. When the plan is a vertical partition, we implement vertical
partition as an index. When the plan is a relational table, we move that table to
a faster device.

Finally, we investigate an experiment several times to achieve accurate results.
In the experiment, we compare the original workload without tuning processes
and tuned workload by using multi-layer devices and indices. As a result, the
original workload has a high workload, which can make systems slower. Also, tuned
workload has no peak workload, and the entire workload is lower than the target
workload level. Furthermore, we show that our techniques can be implemented
and the results demonstrate this fact.

7.2 Future work

This thesis implements index-only processing based on vertical partitioning and
caching in multi-layer persistent storage devices. Other tuning methods, such as
materialized views [6] and clusters [26, 13] are expected to be adapted to our meth-
ods. The methods proposed in the thesis would be implemented as a commercial
database management system for automated performance tuning and eliminating
the need for manual tuning.

In this thesis, the algorithms showed how to allocate the resources when the
predicted workload might occur early or might never occur within the predicted
workload time. In the future, we are expected to add on-line tuning plans based
on the actual workload level to our algorithms. Because the predicted workload
might not be the same with a real workload, for example, within the predicted low
workload time, the actual workload can go higher, or within the predicted high
workload time, the actual workload can go lower. Therefore, we need to add the
algorithm which includes how to allocate the resources for that situation.
Appendices
.1 Appendix 1

.1.1 TPC-H benchmark database

In this thesis, we use 4 GB database size of TPC-H. The LINEITEM and ORDERS tables take 80% of database size.

<table>
<thead>
<tr>
<th>Table name</th>
<th>Rows</th>
<th>Size of the table</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINEITEM</td>
<td>16,725,145</td>
<td>2.16 GB</td>
</tr>
<tr>
<td>ORDERS</td>
<td>6,000,000</td>
<td>701.35 MB</td>
</tr>
<tr>
<td>CUSTOMER</td>
<td>600,000</td>
<td>98.34 MB</td>
</tr>
<tr>
<td>NATION</td>
<td>25</td>
<td>2.13 KB</td>
</tr>
<tr>
<td>PART</td>
<td>800,000</td>
<td>97.96 MB</td>
</tr>
<tr>
<td>PARTSUPP</td>
<td>3,200,000</td>
<td>482.57 MB</td>
</tr>
<tr>
<td>REGION</td>
<td>5</td>
<td>401 Bytes</td>
</tr>
<tr>
<td>SUPPLIER</td>
<td>40,000</td>
<td>5.71 MB</td>
</tr>
</tbody>
</table>

Table 1: Rows and size of each table in TPC benchmark database

.1.2 Create table statements for TPC-H benchmark

CREATE TABLE REGION(
    R_REGIONKEY NUMBER(12) NOT NULL,
    R_NAME CHAR(25) NOT NULL,
    R_COMMENT VARCHAR(152) NOT NULL,
    CONSTRAINT REGION_PKEY PRIMARY KEY(R_REGIONKEY),
    CONSTRAINT REGION_CHECK1 CHECK(R_REGIONKEY >= 0) );

CREATE TABLE NATION(
    N_NATIONKEY NUMBER(12) NOT NULL,
    N_NAME CHAR(25) NOT NULL,
    N_REGIONKEY NUMBER(12) NOT NULL,
    N_COMMENT VARCHAR(152) NOT NULL,
CONSTRAINT NATION_PKEY PRIMARY KEY (N_NATIONKEY),
CONSTRAINT NATION_FKEY1 FOREIGN KEY (N_REGIONKEY)
REFERENCES REGION(R_REGIONKEY),
CONSTRAINT NATION_CHECK1 CHECK(N_NATIONKEY >= 0) );

CREATE TABLE PART(
P_PARTKEY NUMBER(12) NOT NULL,
P_NAME VARCHAR(55) NOT NULL,
P_MFGR VARCHAR(25) NOT NULL,
P_BRAND CHAR(10) NOT NULL,
P_TYPE VARCHAR(25) NOT NULL,
P_SIZE NUMBER(12) NOT NULL,
P_CONTAINER CHAR(10) NOT NULL,
P_RETAILPRICE NUMBER(12,2) NOT NULL,
P_COMMENT VARCHAR(23) NOT NULL,
CONSTRAINT PART_PEKEY PRIMARY KEY (P_PARTKEY),
CONSTRAINT PART_CHECK1 CHECK(P_PARTKEY >= 0),
CONSTRAINT PART_CHECK2 CHECK(P_SIZE >= 0),
CONSTRAINT PART_CHECK3 CHECK(P_RETAILPRICE >= 0) );

CREATE TABLE SUPPLIER(
S_SUPPKEY NUMBER(12) NOT NULL,
S_NAME CHAR(25) NOT NULL,
S_ADDRESS VARCHAR(40) NOT NULL,
S_NATIONKEY NUMBER(12) NOT NULL,
S_PHONE CHAR(15) NOT NULL,
S_ACCTBAL NUMBER(12,2) NOT NULL,
S_COMMENT VARCHAR(101) NOT NULL,
CONSTRAINT SUPPLIER_PKEY PRIMARY KEY (S_SUPPKEY),
CONSTRAINT SUPPLIER_FKEY1 FOREIGN KEY (S_NATIONKEY)
REFERENCES NATION(N_NATIONKEY),
CONSTRAINT SUPPLIER_CHECK1 CHECK(S_SUPPKEY >= 0) );

CREATE TABLE PARTSUPP(
  PS_PARTKEY  NUMBER(12) NOT NULL,
  PS_SUPPKEY  NUMBER(12) NOT NULL,
  PS_AVAILQTY NUMBER(12) NOT NULL,
  PS_SUPPLYCOST NUMBER(12,2) NOT NULL,
  PS_COMMENT VARCHAR(199) NOT NULL,
  CONSTRAINT PARTSUPP_PKEY PRIMARY KEY (PS_PARTKEY, PS_SUPPKEY),
  CONSTRAINT PARTSUPP_FKEY1 FOREIGN KEY (PS_PARTKEY)
    REFERENCES PART(P_PARTKEY),
  CONSTRAINT PARTSUPP_FKEY2 FOREIGN KEY (PS_SUPPKEY)
    REFERENCES SUPPLIER(S_SUPPKEY),
  CONSTRAINT PARTSUPP_CHECK1 CHECK(PS_PARTKEY >= 0),
  CONSTRAINT PARTSUPP_CHECK2 CHECK(PS_AVAILQTY >= 0),
  CONSTRAINT PARTSUPP_CHECK3 CHECK(PS_SUPPLYCOST >= 0) );

CREATE TABLE CUSTOMER(
  C_CUSTKEY NUMBER(12) NOT NULL,
  C_NAME VARCHAR(25) NOT NULL,
  C_ADDRESS VARCHAR(40) NOT NULL,
  C_NATIONKEY NUMBER(12) NOT NULL,
  C_PHONE CHAR(15) NOT NULL,
  C_ACCTBAL NUMBER(12,2) NOT NULL,
  C_MKTSEGMENT CHAR(10) NOT NULL,
  C_COMMENT VARCHAR(117) NOT NULL,
  CONSTRAINT CUSTOMER_PKEY PRIMARY KEY(C_CUSTKEY),
  CONSTRAINT CUSTOMER_FKEY1 FOREIGN KEY (C_NATIONKEY)
    REFERENCES NATION(N_NATIONKEY),
  CONSTRAINT CUSTOMER_CHECK1 CHECK(C_CUSTKEY >= 0) );
CREATE TABLE ORDERS(
  O_ORDERKEY NUMBER(12) NOT NULL,
  O_CUSTKEY NUMBER(12) NOT NULL,
  O_ORDERSTATUS CHAR(1) NOT NULL,
  O_TOTALPRICE NUMBER(12,2) NOT NULL,
  O_ORDERDATE DATE NOT NULL,
  O_ORDERPRIORITY CHAR(15) NOT NULL,
  O_CLERK CHAR(15) NOT NULL,
  O_SHIPPRIORITY NUMBER(12) NOT NULL,
  O_COMMENT VARCHAR(79) NOT NULL,
  CONSTRAINT ORDERS_PKEY PRIMARY KEY (O_ORDERKEY),
  CONSTRAINT ORDERS_FKEY1 FOREIGN KEY (O_CUSTKEY)
    REFERENCES CUSTOMER(C_CUSTKEY),
  CONSTRAINT ORDER_CHECK1 CHECK( O_TOTALPRICE >= 0 )
);

CREATE TABLE LINEITEM(
  L_ORDERKEY   NUMBER(12) NOT NULL,
  L_PARTKEY    NUMBER(12) NOT NULL,
  L_SUPPKEY    NUMBER(12) NOT NULL,
  L_LINENUMBER NUMBER(12) NOT NULL,
  L_QUANTITY   NUMBER(12,2) NOT NULL,
  L_EXTENDEDPRICE NUMBER(12,2) NOT NULL,
  L_DISCOUNT   NUMBER(12,2) NOT NULL,
  L_TAX        NUMBER(12,2) NOT NULL,
  L_RETURNFLAG CHAR(1) NOT NULL,
  L_LINESTATUS CHAR(1) NOT NULL,
  L_SHIPDATE   DATE NOT NULL,
  L_COMMITDATE DATE NOT NULL,
  L_RECEIPTDATE DATE NOT NULL,
  L_SHIPINSTRUCT CHAR(25) NOT NULL,
  L_SHIPMODE   CHAR(10) NOT NULL,
L_COMMENT VARCHAR(44) NOT NULL,
CONSTRAINT LINEITEM_PKEY PRIMARY KEY (L_ORDERKEY, L_LINENUMBER),
CONSTRAINT LINEITEM_FKEY1 FOREIGN KEY (L_ORDERKEY) REFERENCES ORDERS(O_ORDERKEY),
CONSTRAINT LINEITEM_FKEY2 FOREIGN KEY (L_PARTKEY) REFERENCES PART(P_PARTKEY),
CONSTRAINT LINEITEM_FKEY3 FOREIGN KEY (L_PARTKEY, L_SUPPKEY) REFERENCES PARTSUPP(PS_PARTKEY, PS_SUPPKEY),
CONSTRAINT LINEITEM_FKEY4 FOREIGN KEY (L_SUPPKEY) REFERENCES SUPPLIER(S_SUPPKEY),
CONSTRAINT LINEITEM_CHECK1 CHECK (L_QUANTITY >= 0),
CONSTRAINT LINEITEM_CHECK2 CHECK (L_EXTENDEDPRICE >= 0),
CONSTRAINT LINEITEM_CHECK3 CHECK (L_TAX >= 0),
CONSTRAINT LINEITEM_CHECK4 CHECK (L_DISCOUNT BETWEEN 0.00 AND 1.00) );
.1.3 Conceptual Schema for TPC-H

Figure 1: Conceptual Schema for TPC-H
Bibliography


