

# We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

6,900

Open access books available

186,000

International authors and editors

200M

Downloads

Our authors are among the

154

Countries delivered to

TOP 1%

most cited scientists

12.2%

Contributors from top 500 universities



WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index  
in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?  
Contact [book.department@intechopen.com](mailto:book.department@intechopen.com)

Numbers displayed above are based on latest data collected.  
For more information visit [www.intechopen.com](http://www.intechopen.com)



# Queries Processing in Wireless Sensor Network

*Kamel Abbassi and Tahar Ezzedine*

## Abstract

For the super-excellence applications used to control the water level in rivers, temperature handles a very large volume of information and does not stop constantly changing. These spatio-temporal data collected by a network of sensors form a set of thematic, integrated, non-volatile and historical data organized to help decision-making. Usually this process is performed with temporal, spatial and spatiotemporal queries. This in turn increases the execution time of the query load. In the literatures, several techniques have been identified such as materialized views (MV), indexes, fragmentation, scheduling, and buffer management. These techniques do not consider the update of the request load and the modification at the database level. In this chapter, we propose an optimal dynamic selection solution based on indexes and VMs. the solution is optimal when it meets the entire workload with a reasonable response time. The proposed approach supports modification at the database level and at the workload level to ensure the validity of the optimal solution for this the knapsack algorithm was used.

**Keywords:** wireless sensor network, workload, optimized structure, NP-complete problem, knapsack, materialized view, index, multiple selection problem, monitoring

## 1. Introduction

A sensor network used to record physical conditions of the environment such as temperature, rainfall, pollution, humidity, wind, etc. These data are sent to a database server which will be processed later.

All this data collected by the sensors will be recorded in a database which is in turn queried by client applications, such as the supervisor, the security agent, or a third-party application.

This database will be queried by complex queries which require resources.

To decrease the response time, it is necessary to use optimization techniques such as materialized views (MV), indexes, fragmentation, and the caching system. All these techniques are proven in the case of relational databases.

A view is a virtual table representing the result of a query on the basis. As the name suggests and unlike a standard view, in a materialized view the data is duplicated.

The index placed on a table will provide quick access to records, depending on the value of one or more fields. In addition, allows to simplify and accelerate the operations of search, sorting, joining or aggregation.

In this work, an approach proposed for the multiple selection of indexes and materialized views with the knapsack algorithm. The work presents an

improvement of another approach based on the greedy algorithm. The rest of this work is organized like this: The first section deals with optimization techniques; the problem of multiple selection of indexes and materialized views will be presented in the second section. The contribution to the dynamic workload will be mentioned in Section 4. In Section 5, a discussion of our approach for the case where the database is dynamic will be described. Finally, a discussion on the experiment and the contribution will be discussed.

## 2. Optimization techniques

The use of optimization techniques is based on two approaches. The first is the sequential use of techniques such as indexes and fragmentations which have depleted physical structure, but the second is the simultaneous use of techniques which have similar physical structure such as materialized views and indexes.

In [1], authors proposed three approaches which are MVFirst, INDFirst and Joint enumeration. But the major drawback of this approach is the sequential and isolating use of these techniques, which does not make it possible to benefit from the advantage of the interactivity between these optimization structures.

The authors say that this last alternative is the best [2]. Bellatrech et al. [3] improve part of the multiple selection problem with storage space management.

The authors use two spies to manage the space shared between two structures, index and VM. If the optimal configuration needs more indexes, then the spy associated with the index will take up space from the VM spy and vice versa.

In [4] use the drop algorithm for the selection simulation of indexes and MV.

These works only deal with the case where the load of requests is static and does not change over time. In [5], authors proposed an approach to dynamically select materialized views. The approach is based on the PRQ predictor to predict the next request and materialize its corresponding views using the conditional probability. This approach uses a cost model based on cloud costs. This work shows a tremendous improvement in terms of cost, execution time, and processing, but the authors only used one optimization framework which is view materialized. Another dynamic approach called Dynamat refreshes the configuration of materialized views if their size exceeds the space allocated for it [6]. Several criteria are used, for example, delete rarely used views. A hybrid approach jointly exploits a static set of persistent views used by multiple request and maintenance sequences, and another dynamic set of aggregated and smaller sizes accessible and replaceable on the fly [7]. However, these approaches focus only on the refresh performance of materialized views and not on query workload. In addition, do not use other optimization structures. Karkad et al. [8] applied the buffer management and scheduling technique to three optimization structures (index, materialized views, and fragmentation). This approach requires caching, planning, and is not dynamic.

In our approach, the simultaneity between materialized and indexed views used to benefit from the interaction between these structures. In addition, a mathematical modulization of the problem has been proposed based on the backpack algorithm which proves their performance compared to the greedy algorithm used by N. Maiz et al. [5].

## 3. Materialized view and index selection problem

Simultaneous selection of indexes and materialized views is an NP-Hard type problem which gives several optimal solutions [9].

The Knapsack problem, also noted KP, is an optimization problem. Presents a situation which cannot support more than a certain weight, with all or part of a given set  $N \in O$  of objects  $O = \{o_1, o_2 \dots o_n\}$  each having a weight  $weight(o_i)$  and a value  $profit(o_i)$ . Items put in the backpack must maximize the total value and not exceed the maximum weight  $S$ . The problem is formalized as follows:

$$\sum_{o_i} weight(o_i) < S \quad (1)$$

$$\forall N \in O, \sum_{o_i \in N} profit(o_i) < \sum_{o_i \in S} profit(o_i) \quad (2)$$

On the other hand, the problem of selecting indexes and materialized views (PSIMV) consists in finding a set of indexes and materialized views constituting the final configuration to optimize the workload requests. This optimization can be in run time and storage space. The Workload requests, index and MV are presented as follows:

$$Q = \{q_1, q_2, \dots q_m\} \quad (3)$$

$$I = \{i_1, i_2, \dots i_n\} \quad (4)$$

$$V = \{v_1, v_2, \dots v_k\} \quad (5)$$

$Q$  presented the Workload queries. This set composed by  $m$  queries.  $I$  is the set of  $n$  indexes and  $V$  presented the  $k$  materialized views.

$S$  is the size allowed by the administrator to store indexes and MV. Then it is necessary to find a configuration without violating the following constraints:

- Minimize the cost of Workload, i.e.

$$C(Q, Config_{IV}) = Min(C_{IV}(Q)) \quad (6)$$

- The size of the configuration  $Config_{IV}$  does not exceed  $S$

$$\sum_{i \in Config_{IV}} size(iv) \leq S \quad (7)$$

The problem of selecting indexes and materialized views is adopted by the genetic algorithm. The starting population is the set of candidate indexes and MVs. The objective function to optimize is the cost of the workload. The next section shows the analogy between the problem of selecting indexes and materialized views and the knapsack algorithm.

### 3.1 Selection problem with index and MV vs. knapsack algorithm

In this work, we present the correspondence between the problem of the knapsack and that of the multiple selection of indexes and materialized views (**Table 1**).

### 3.2 Cost model

Typically, the number of indexes and candidate VMs is greater since the input load is significant. The creation of all these indexes and MVs is not possible due to the constraint on the allocated storage space. To solve the problem, we use a cost model which allows us to keep only the most advantageous indexes and MVs. This

Knapsack problem	Problem with selecting indexes and materialized views
Objects	The total set of Indexes and materialized views
Weight	The point shows the size of each object and the required execution time.
Profit	This is the profile to be won if these objects are used. Shows the gain in execution time and storage
Size	The number of bits needed to store the objects that form the optimal solution
Object set	The final configuration of indexes and materialized views

**Table 1.**  
*Selection of indexes and materialized vs. knapsack.*

model estimates the space in bytes occupied by indexes and VMs, the data access costs and the maintenance cost in terms of number of inputs and outputs.

Indexes and MVs are the objects in this optimization system. Cost of an object is the sum of storage size, access data cost both these indexes and MVs and maintenance cost.

$$Cost_{o_i} = Size_{o_i} + Cost\_Access_{o_i} + Cost\_Mat_{o_i} \quad (8)$$

The benefit provided by an  $O_i$  object is the difference the cost of the Workload before adding the object  $Cost\_Load\_Before(O_i)$  and after the addition of this object  $Cost\_Load\_After(O_i)$ . the following equation calculates the profit:

$$Profit(O_i) = Cost\_Load\_Before(O_i) - Cost\_Load\_After(O_i) \quad (9)$$

To add this object to the configuration list, we followed this equation

$$Profit(O_i) = \begin{cases} > 0, & Add O_i to Config \\ \leq 0, & donothing \end{cases} \quad (10)$$

If  $Profit(O_i) > 0$ , There is a benefit, so add the object  $O_i$  to the configuration, else not add  $O_i$  to the configuration.

This query system can be static or dynamic. When the Workload and the data stored in the database are invariable in this case, it is a static system which will be discussed in Section 4. On other hand, if the Workload and the database are modifiable, then it is a dynamic system which will be discussed in Section 5.

#### 4. Statistic workload

A request load is the set of requests that have arrived and are waiting for their turn to be executed. This section will discuss the case where the database does not change, and the requests have arrived successively in random order. Bellatrach et al. [3] proposed a static approach that does not support the changing of Workload. Authors apply the greedy algorithm which does not necessarily provide an optimal solution. In [6], authors show that dynamic programming and more optimal than the Greedy algorithm.

The Knapsack algorithm is an example of dynamic algorithms used for optimization problems. In the proposed approach, Artificial Intelligence uses this algorithm only in the learning phase and afterwards a model will be created to predict the final solution and avoid the execution of this algorithm on each new request.



An optimal configuration is the set of materialized views and indexes which extend to the workload in a reasonable time with the minimum of resources.

The following algorithm takes as input the list of indexes and Materialized views to create an optimal configuration on condition that this configuration extends to the entire load of requests and does not exceed the authorized storage size.

---

**Algorithm 1.** Static approach.

---

**Input:** Index  $I$ ,  $MV$

**Output:**  $Config$

**Initialization:**  $Config \rightarrow \{\emptyset\}$ ,  $Size_{max} = 0$ ,  $Profit_{max} = 0$

Start

1.  $O = I \cup MV$
2.  $S = \text{Space authorized by administrator}$
3. for  $(O_i \in O \setminus \{Config\} \text{ et } Size_{max} < S) \text{ then}$
4.     if  $Profit(O_i) > Profit_{max} \text{ then}$
5.          $Profit_{max} \leftarrow Profit(O_i)$
6.          $Config \leftarrow Config \cup \{O_i\}$
7.          $Size_{max} \leftarrow Size_{max} + Size(O_i)$
8.     End if
9. End for

End

---

$S$  is the size of the disk space allocated to store MVs and indexes, it is fixed by administrator.

Objective function  $Profit()$  calculates for each index or MV. It is the difference in cost between the workload run time with or without this object (Index or MV). If this object improves the system, it will be added to the entire configuration. At the end of this algorithm, the final configuration is made up of a set of indexes and MVs which represent the optimal solution. This technique considers the similarity between the two optimization structures index and MV.

These iterations will be repeated until there is no improvement in the  $Profit()$  function, or until all indexes and VMs have been selected, or until the limit storage space is exceeded.

Changes at the database level or in the workload require a new configuration to revert to the new Workload. Then you must rebuild new indexes and VMs. This operation is very time consuming.

In Dynamat [8] the authors have removed the least used VMs to free space for new creations. In this approach the authors limit themselves to use only the MV optimization structure.

To solve this inconsistency problem, the authors find three strategies. The first one is that all views are updated regularly at each time interval [10]. The second one is that all views are updated at the end of each transaction [11] and the last strategy is that the changes are propagated in a delayed manner. I.e. a VM is updated only when it is used by a request.

Our approach combines the two structures (Index and Materialized views) to benefit from the structural affinity between these two optimization techniques.

In a real-time survival system, query processing is important. To ensure optimal validation of the solution after the change in workload and database, two artificial intelligence techniques are used.

The arrival of requests is random and varied depending on the context and in this case. On the other hand, the database can be modified during the execution of the queries. In this part we will study the two cases.

We used artificial intelligence to create materialized views for the dynamic processing of the workload and to make requests as visible as possible. With automatic learning, we proposed an algorithm that allows to search for the logical link between the query load and the optimal configuration, then and after the learning phase will predict the final solution (Minimum configuration).

We started with a remodeling phase. Each request is presented by a factor which presents the list of attributes used. on the other side a matrix which presents all the possible solutions which are prepared in advance.

The Workload  $Q$  is formed by  $n$  queries, i.e.,  $Q = \{Q_1, \dots, Q_n\}$ . A query is composed by  $j$  attributes, where  $\epsilon \{a_1, \dots, a_k\}$ , and each query has the following form:  $Q_i = \{a_j^i\}, \forall i : 1..n, j : 1..k$ . The activation function used in this work is presented as follows:

$$f(a_j) = \begin{cases} 1 * a_j & \text{if } a_j \text{ used in } Q \\ 0 * a_j & \text{else} \end{cases} \quad (11)$$

The workload can be presented in form of matrix as follow:

$$MatA = \begin{pmatrix} a_1^1 * f(a_1) & \dots & a_k^1 * f(a_k) \\ \vdots & \ddots & \vdots \\ a_1^n * f(a_1) & \dots & a_k^n * f(a_k) \end{pmatrix} \quad (12)$$

A final solution is a set of structures such as Index and MV that guarantees the response to the entire query load with minimal cost. Based on,  $N$  attributes, we can find  $2^N - 1$  views and the final solution has a view between 1 and  $2^N - 1$ .

$$FS^f = \{v_e^f\}, \text{ where } e \in (1..2^N - 1), f = (1..2^{2^N - 1} - 1) \quad (13)$$

In order to verify if this materialized view  $v_e$  is included in the solution  $FS^f$  or not, the function  $h(v_e)$  having the following form should be used

$$h(v_e) = \begin{cases} 1, & v_e \text{ if } v_e \text{ used in } FS \\ 0, & v_e \text{ else} \end{cases} \quad (14)$$

The maximum number of final solutions is  $(2^N - 1)^{2^N - 1}$ , where  $N$  is the number of attributes in database tables.

The final solutions are presented as follows

$$FS = \begin{pmatrix} v_1^1 * h(v_1) & \dots & v_{2^N - 1}^{(2^N - 1)^{2^N - 1}} * h(v_{2^N - 1}) \\ \vdots & \ddots & \vdots \\ v_1^n * h(v_1) & \dots & v_{2^N - 1}^{(2^N - 1)^{2^N - 1}} * h(v_{2^N - 1}) \end{pmatrix} \quad (15)$$

The references of the final solutions are stored in a vector  $VS$  with the following form

$$VS = \{S_f\}, \text{ where } f = (1..2^{2^N - 1} - 1) \quad (16)$$

**Figure 1** shows the three layers of our modeling and the steps to create candidate solutions. First step is the extraction of the attributes used in all the tables of the database  $\{a_1, \dots, a_n\}$ , then create a vector containing all the possible materialized views, i.e. the possible combinations with the attributes  $\{v_1, \dots, v_{2^n-1}\}$ . A materialized view contains at least one attribute and at most all attributes. The number of VMs is  $2^n - 1$ .

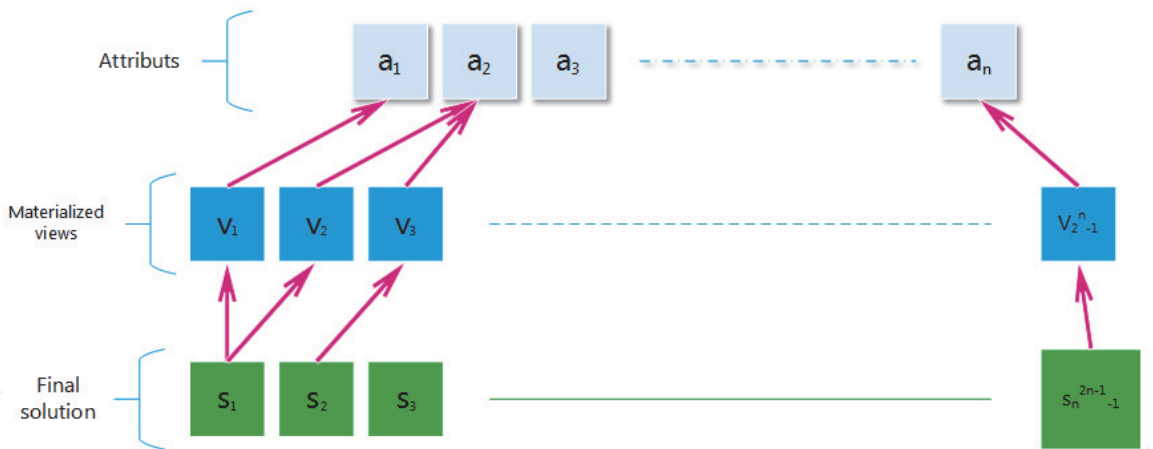
Then the candidate solutions, which presents all the possible combinations of the VMs. The maximum number of solutions is  $2^{2^n-1} - 1$ .

To apply the automatic learning, To apply machine learning, you have to start with the learning phase, this phase the algorithm will build a logical link between the attributes and the final solutions. The duration of this phase is set by the administrator (**Figure 2**).

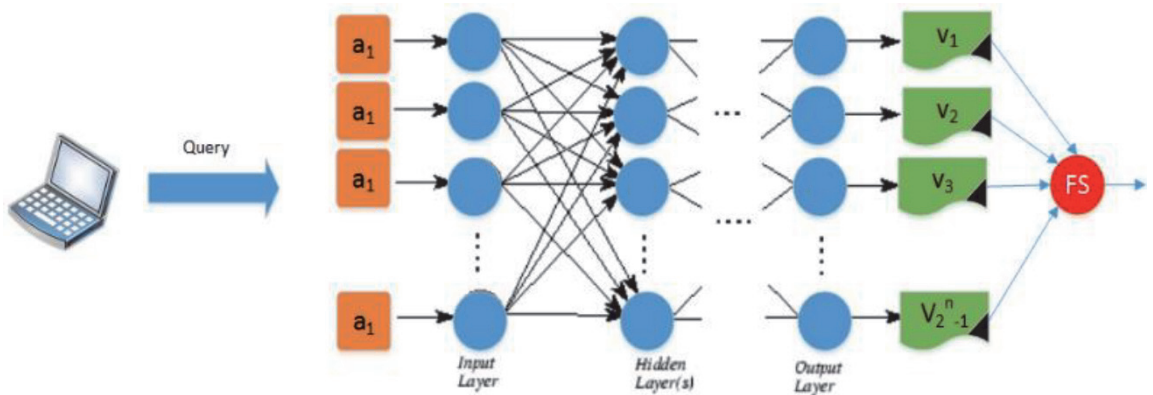
The algorithm is composed of two phases: The first phase is used for training. However, the second is used to predict materialized views.

**Figure 3** shows the architecture of our approach. The system administrator sets the period for learning the model. If this phase is in progress, each time a new request arrives the system will use the knapsack algorithm to find the right configuration and at the same time prepare the neural network model. At the end of the learning phase the system will use this module provided in the first phase to predict a new optimal configuration for each arrival query.

The final FS solution is the optimal configuration that extends to the workload with a reasonable execution time. With this approach, a logic established between the requests and the final solutions to avoid recalculating each time.

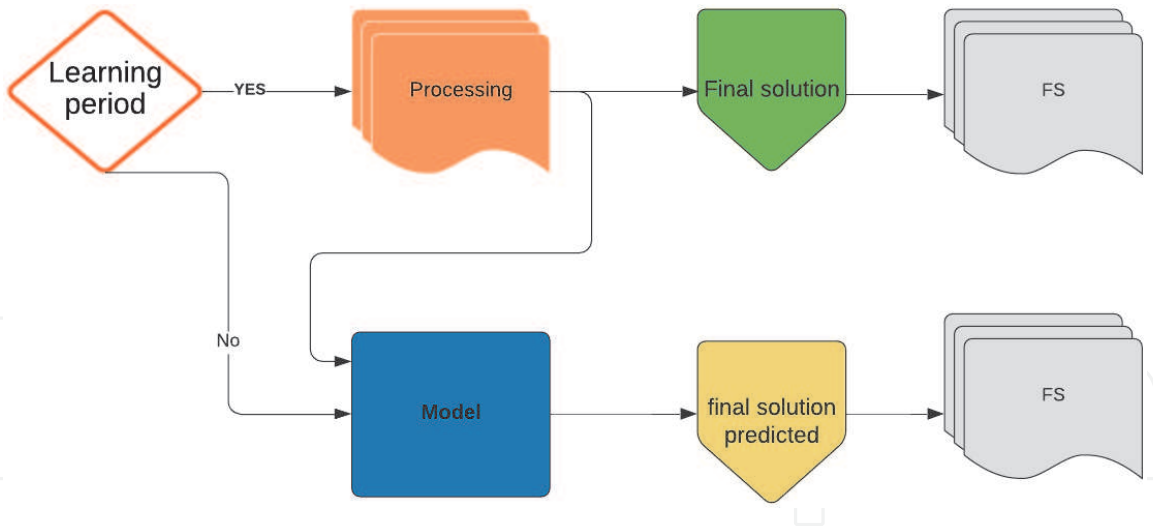


**Figure 1.**  
Final solutions tree.



**Figure 2.**  
Machine learning to create optimal solutions.





**Figure 3.**  
*Switching between the training phase and the prediction phase.*

In this experiment uses a workload containing 5 queries numbered from 1 to 5 and a database of 4 attribute differences that make 15 materialized views and 32,768 final solutions (**Table 2**).

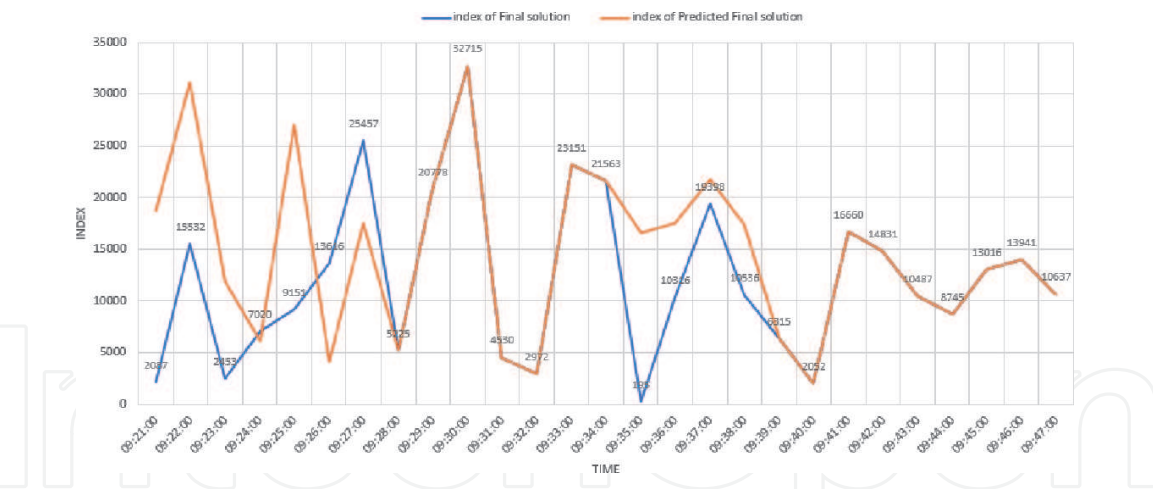
Between 09:21 am and 9:47 am the requests arrive randomly. At the start the Workload contains only the query Q5 and for this workload the final solution is 4523 on the other hand the predicted final solution is 25,531 which is our predicted solution is different from the real solution.

To test the approach, an implementation of the algorithm was carried out with Python 3 on a laptop computer equipped with a Windows 10 operating system, 64 bits and 8 GB of RAM. The experimental results are discussed in the following figure.

First, each query is executed with the greedy algorithm to see the final solution as shown by the blue dots in **Figure 4**. In the second step our algorithm will be compared with the first to see if possible, to predict the final solution (orange curve) without wasting the time to recalculate the configuration each time a request arrives.

Time	Query	Workload	Index of final solution	Index of predicted final solution
09:21:00	Q5	Q5	4523	25,531
09:22:00	Q4	Q5Q4	2660	18,747
09:23:00	Q3	Q5Q4Q3	29,366	21,896
09:24:00	Q4	Q5Q4Q3	16,468	24,525
09:25:00	Q5	Q5Q4Q3	29,845	5103
09:26:00	Q2	Q5Q4Q3Q2	3280	23,163
...	...	...	...	...
09:42:00	Q4	Q5Q4Q3Q2Q1	23,181	23,181
09:43:00	Q4	Q5Q4Q3Q2Q1	20,649	20,649
09:44:00	Q1	Q5Q4Q3Q2Q1	8366	8366
09:45:00	Q5	Q5Q4Q3Q2Q1	21,667	21,667
09:46:00	Q2	Q5Q4Q3Q2Q1	4942	4942
09:47:00	Q4	Q5Q4Q3Q2Q1	11,120	11,120

**Table 2.**  
*Dataset final solution.*



**Figure 4.**  
*Final solution vs. predicted final solution.*

This figure clearly shows that after a learning phase, the algorithm manages to predict the final solution and consequently a great gain in the execution time and the resources used.

5. Dynamic database

This section discusses the case where the database is dynamic, during the execution of the queries, an update on the data is in progress. Updating all optimization structures is very expensive, so it is a good idea to update only the affected optimization structures.

For this, two binary tables are proposed and stored in the database (**Table 3**). The Matrix  $IT[i, t]$  stores the link between the indexes and the tables of the database. If index number 3 is used by table number 5, then  $IT[3, 5] = 1$  otherwise equal to 0. Likewise, for the Matrix  $VT[v, t]$  which presents the materialized views linked to the tables. For example, if the materialized view number 5 (MV5) is linked with **Table 4** then  $VT[5,4] = 1$  otherwise equal to 0.

To understand, here is the following example: either Table T1 used by the indexes I1, I2, I4 and MVs V2, V4. Table T3 used by indexes I2, I4 and MVs V1, V4, so each time the database is updated, it is wise to modify only the structure concerned (index or MV). Each time the database tables are updated, a trigger searches for the index or Materialize view affected by this change. More details below (**Figure 5**).

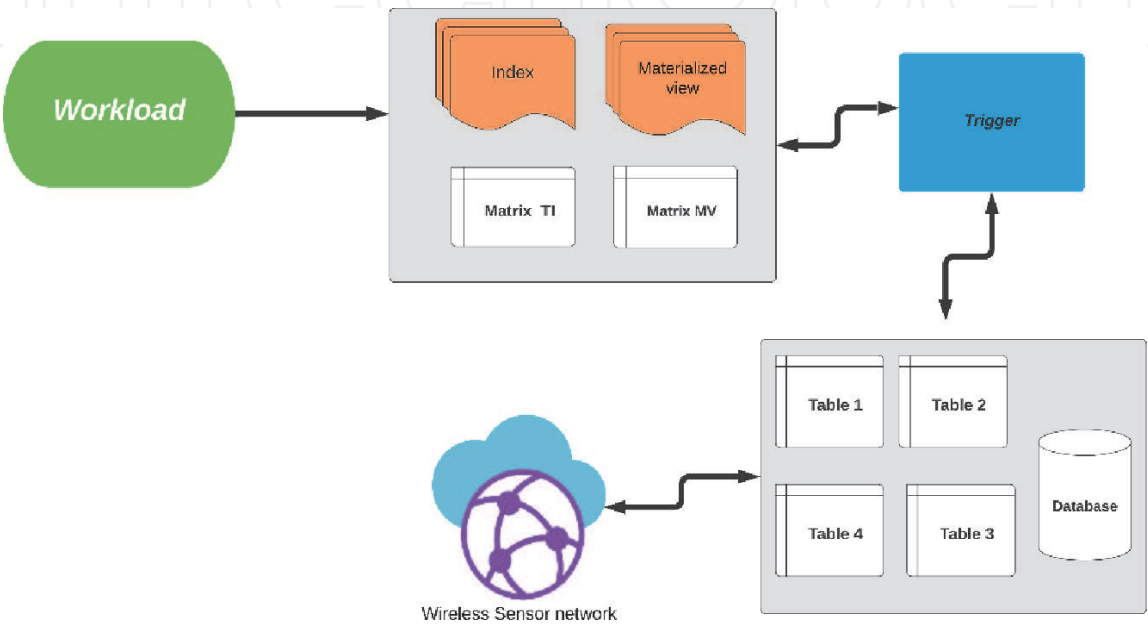
The trigger is an integrated solution in all DBMS. It is a program that launches a series of tasks with each change in the database. It identifies the objects to be modified in the configuration. At each update operation (insertion, update, or

	T1	T1	T3	T4	T5	T6
I1	1	0	0	1	1	0
I2	1	0	1	1	0	1
I3	0	1	0	1	1	1
I4	1	0	1	0	1	1
I5	0	1	0	1	0	1

**Table 3.**  
*Matrix IT.*

	T1	T1	T3	T4	T5	T6
v1	0	0	1	0	1	0
v2	1	1	0	1	0	1
v3	0	0	0	0	0	0
v4	1	0	1	1	1	0
v5	0	0	0	1	0	1

**Table 4.**  
*Matrix VT.*



**Figure 5.**  
*Algorithm of the dynamic approach.*

deletion) the trigger does the same operation on the object concerned (Index or MV). For example, if a new row is inserted in the Table  $T_i$ , the trigger inserts the same row in the index and the VM linked by the table  $T_i$ . After each iteration, if the size of the configuration exceeds  $S$  or if the solution has become non-optimal, Algorithm 1 must be restarted.

This architecture guarantees that all the indexes and MVs form the optimal configuration even after updating the Workload.

**Algorithm 2.** Dynamic database.

**Input:** Index  $I$ , MV, Tables, Workload  
**Output:** Config  
**Initialization:**  $task \rightarrow \{\emptyset\}$ ,  $lock = false$   
**Start**  
10. While (true) do  
11.  $task \leftarrow trigger()$   
12.  $lock \leftarrow false$   
13. if(task) then  
14.  $lock \leftarrow true$   
15.  $\{I, MV\} \leftarrow get\_structure(Tables)$   
16.  $Update(Config(I, MV))$

```

17.      lock ← false
18.      End if
19.      if (! lock) then
20.          Execute (Workload)
21.      End if
22.      End for
End
    
```

---

Algorithm 2 is still running, the *trigger* () function returns the list of tables infected by update if not returns null. Variable *lock* initialized to false to prevent the execution of the workload pending the configuration update. If there are updates, the *lock* variable takes true and *get\_structure*() function searches the structures infected with this modification. This function uses two matrices IT and VT. Then *Update*() function modifies the configuration to support the new updates in the tables. And at the end of this operation, the variable lock will be released to execute the Workload.

## 6. Conclusions

In this work, a similarity between the problem of selecting indexes and materialized views with the Knapsack algorithm was proposed. The contributions are: The first level, the use of the backpack algorithm to present this problem as well as a mathematical modeling, then the use of machine learning to reduce the execution time of the workload. For this, two tables were used to ensure that the optimal configuration remains reliable even after updating the database. To validate this approach, an algorithm developed in python.

## Acknowledgements


this work is the result of two works published as part of a Tunisian-South African research project (Grant Numbers: 113340, 120106) funded by the Ministry of Higher Education and Scientific Research and the National Research Foundation of South Africa presented by Prof Qing-Guo WANG.

## Author details

Kamel Abbassi\* and Tahar Ezzedine  
 Communication System Laboratory Sys'com, National Engineering School of Tunis,  
 University Tunis El Manar, PB 37, Belvedere, 1002, Tunis, Tunisia

\*Address all correspondence to: [kamel.abbassi@enit.utm.tn](mailto:kamel.abbassi@enit.utm.tn)

## IntechOpen

© 2021 The Author(s). Licensee IntechOpen. This chapter is distributed under the terms of the Creative Commons Attribution License (<http://creativecommons.org/licenses/by/3.0>), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited. 

## References

- [1] ORDONEZ-ANTE, Leandro, VAN SEGHBROECK, Gregory, WAUTERS, Tim, et al. a workload-driven approach for view selection in large dimensional datasets. *Journal of Network and Systems Management*, 2020, p. 1–26.
- [2] AOUICHE, K. Automatic selection of indexes in data warehouses. Research Report, Laboratory ERIC Lumière Lyon2University, 2005.
- [3] BAHACHE, Anwar Nour Eddine. A Metaheuristic Based Approach for Solving the Index Selection Problem in Data Warehouses. Doctoral thesis. In: FACULTY Mathematics and Computer Science DEPARTMENT of Computer Science. 2018
- [4] LETRACHE, Khadija, EL BEGGAR, Omar, et RAMDANI, Mohammed. OLAP cube partitioning based on association rules method. *Applied Intelligence*. 2019;**49**(2):420-434
- [5] TOUMI, Lyazid et UGUR, Ahmet. Static and incremental dynamic approaches for multi-objective bitmap join indexes selection in data warehouses. *The Journal of Supercomputing*. 2020:1-26
- [6] AL ETAWI, Namer Ali et ABUROMMAN. Fatima Thaher. 0/1 KNAPSACK PROBLEM: GREEDY VS. DYNAMIC-PROGRAMMING. *International Journal of Advanced Engineering and Management Research* Vol. 2020;**02**:5
- [7] GARAIN, Nilkantha, CHATTOPADHYAY, Samiran, MAHAPATRA, Gautam, et al. Design and Implementation of an Improved Data Warehouse on Clinical Data. In: *International Conference on Computational Intelligence, Communications, and Business Analytics*. Springer, Singapore, 2018. p. 278–290.
- [8] KERKAD, Amira, BELLATRECHE, Ladjel, RICHARD, Pascal, et al. A query beehive algorithm for data warehouse buffer management and query scheduling. *International Journal of Data Warehousing and Mining (IJDWM)*, 2014, vol. 10, no 3, p. 34-58.
- [9] Harinarayan V, Rajaraman A, Ullman a J. Implementing data cubes efficiently. *Proceedings of the ACM SIGMOD International Conference on Management of Data*. June 1996:205
- [10] COMER D. The difficulty of optimum index selection. *ACM Transactions on Database Systems (TODS)*. 1978:440-445
- [11] J. A. Blakeley, P. L. (June 1986). Efficiently updating materialized views. *Proceedings of the ACM SIGMOD International Conference on Management of Data*.