Partially Materialized Partitioned Views

Satyanarayana R Valluri
satya@iiit.net
Centre for Data Engineering
International Institute of Information Technology
Gachibowli, Hyderabad, INDIA

ABSTRACT

Selection of materialized views and maintaining them is an important problem studied in literature. In this paper, we develop the notion of *Partially Materialized Partitioned Views (PMPV)* in which a materialized view is first partitioned and only a part of the view is materialized. The partitioning is done based on the workload of the database: the read-only queries and the update queries. The PMPV approach offers many advantages. The notion of PMPV is independent of the view selection algorithm and the algorithm used for answering the queries using the materialized views. The space occupied by the PMPVs will be less than the normal materialized views and hence saving the storage space. Since the sizes of PMPVs will be less than the normal materialized views, the cost of processing queries using PMPVs might be less than that of using the normal materialized views. The number of updates that need to be considered while maintaining PMPVs will be less than that of the normal materialized views. We discuss the algorithms for selecting the PMPVs and maintaining them. The experimental results show that the PMPV approach offers advantage in terms of saving the storage space and decrease in the query processing cost at the expense of increase in the maintenance cost of the PMPVs.

1. INTRODUCTION

A data warehouse provides integrated access to multiple, distributed, heterogeneous databases and other information sources [38]. Pre-computation of query results and storing them as materialized views is a very popular technique used in data warehousing environments. Given the query workload, the problem of determining the materialized views that are to be maintained at the warehouse is called the "view selection problem" [13]. Since the underlying data sources get updated, the process of updating the materialized views to keep them consistent with the data sources is called "view maintenance" [11]. Storing materialized views incurs two kinds of costs: storage cost and view maintenance cost. Various algorithms are proposed to select that set of materialized views which minimize the total query response time and the cost of maintaining the selected views, in the presence of storage space/maintenance cost constraint [17, 13, 15, 3, 39].

The cost to be paid for having materialized views is the view maintenance cost. Since it is wasteful to compute the entire view for every update, "incremental view maintenance" becomes very attractive to reduce the maintenance cost. Algorithms that compute changes to a view in response to changes to the base relations are called "incremental view maintenance algorithms" [11]. Many incremental view maintenance algorithms are proposed in literature [6, 12, 11, 14, 35, 10, 1, 26]. In the presence of extra information, some views can be maintained without accessing the base relations. Such views are called self-maintainable views and the extra information required is called auxiliary view. The problem of deriving the axillary views and thereby making the materialized views self-maintainable is studied in [29, 19, 32].

Traditional method of processing OLTP queries is to extract the information from the sources only when the queries are posed. This approach is called a *lazy* or *on-demand* approach [38] . The other alternative is an *eager* or *in-advance* approach wherein the information is extracted from the sources in advance and stored in a repository, and when the queries are posed, they are answered using the repository without accessing the original information sources. The lazy and the eager approaches represent two ends of vast spectrum of possibilities [18]. In this paper, we explore an alternative which lies between the two ends. Instead of materializing the full view, the view is partitioned such that only those partitions which are required by the workload are materialized. We assume that the workload of the database, the read-only and the update queries, are known in advance and the partitioning of the views is done based on them. We identify the overlap between the queries conservatively, we do not consider explicitly how conditions relate to each other as in [4].

1.1 Motivating Example

When a view V is materialized, it may contain many tuples which are not useful to any of the queries in the workload. The PMPV technique proposed determines the useful set of tuples and materializes only them. We first present a

ADVANCES IN DATA MANAGEMENT 2005
Jayant Haritsa, T.M. Vijayaraman (Editors)
© CSI 2005

simple example to illustrate the idea of the PMPV method.

Example 1. Consider a database which has four relations A, B, C and D. Let $(A1, A2, \ldots)$, $(B1, B2, \ldots)$, $(C1, C2, \ldots)$ and $(D1, D2, \ldots)$ be the attributes of A, B, C and D respectively. Consider two read-only queries:

$$\text{Query 1:}((\sigma_{A3>10}A) \bowtie_{A2=B2} B) \bowtie_{B3=C3} C$$

$$\text{Query 2:}((\sigma_{A1\leq 35}A) \bowtie_{A2=B2} B) \bowtie_{B4=D4} D$$

Consider a materialized view V which has definition:

$$V = A \bowtie_{A2=B2} B$$

Both the queries can be answered by using the view V. If there is no other query which uses V for processing, then instead of materializing the whole view, materializing those tuples of V which join either with C or D on the given join conditions would reduce the size of V. Consider a partition of V, V' which has the definition[1]:

$$\begin{aligned} V' \;=\; & (\sigma_{(A1\leq 35)\vee(A3>10)}A) \bowtie_{A2=B2} B \text{ and } (\forall t \in V' \\ & \text{either } \exists\, t_c \in C \text{ such that } t \bowtie_{B3=C3} t_c \text{ is true} \\ & \text{or } \exists\, t_d \in D \text{ such that } t \bowtie_{B4=D4} t_d \text{ is true}) \end{aligned}$$

The select condition on A: $(A1 \leq 35) \vee (A3 > 10)$ selects only those tuples of A which are useful to either Query1 or Query2 or both. The condition "$\exists\, t_c \in C$ such that $t \bowtie_{B3=C3} t_c$ is true" selects only those tuples of V which are useful in answering Query 1. Similarly, the condition "$\exists\, t_d \in D$ such that $t \bowtie_{B4=D4} t_d$ is true" selects only those tuples of V which are useful in answering Query 2. Thus, V' denotes only those tuples of V which are useful in answering either Query 1 or Query 2 or both.

Materializing V' instead of V will still allow both the queries to be answered. At the same time, it saves the space because the size of V' will be less than V. If the differences in sizes of V and V' is large, then the cost of answering the queries using V' will be less than that by using V. Now consider the following update queries on the database:

```
Update 1: UPDATE A SET   A4 = 40
                   WHERE A5 > 10
Update 2: UPDATE A SET   A4 = 10
                   WHERE A1 > 50 AND A3 < 50
Update 3: UPDATE C SET   C3 = 20
                   WHERE C1 < 20
```

As discussed earlier, storing V' instead of V would be beneficial. Now the update query 1 updates only some tuples of V'. The update query 2 does not updates any of the tuples in V' because the 'where' condition of the query ($A1 > 50$ AND $A3 < 50$) contradicts with the condition of V' ($A1 \leq 35$ OR $A3 > 10$). The update query 3 updates V' although C is not a base relation of V' because it modifies the attribute $C3$ and the definition of V' depends on the values of the attribute $C3$. There might be an overlap between the tuples which are accessed by Query 1 and the tuples which are updated by either Update 1 or Update 2 or both. Thus, we now define two partitions of the view V, V^{qu} and V^q which are defined as:

$$\begin{aligned} V^{qu} \;=\; & (\sigma_{(A1\leq 35)\vee(A3>10)}A) \bowtie_{A2=B2} B \text{ and } \forall t \in V^{qu} \\ & (\text{either } \exists\, t_c \in C \text{ such that } t \bowtie_{B3=C3} t_c \text{ is true}) \text{ or} \\ & ((t \text{ satisfies } A5 > 10) \text{ or} \\ & (\exists\, t_c \in C \text{ such that } t \bowtie_{B3=C3} t_c \text{ is true} \\ & \text{and } t_c \text{ satisfies } C1 < 20)) \end{aligned}$$

$$\begin{aligned} V^{q} \;=\; & ((\sigma_{(A1\leq 35)\vee(A3>10)}A) \bowtie_{A2=B2} B \text{ and } \forall t \in V^{q} \\ & (\exists\, t_d \in D \text{ such that } t \bowtie_{B4=D4} t_d \text{ is true}) \end{aligned}$$

Now, Query 1 can be answered using V^{qu} and Query 2 can be answered using V^q. The view V^{qu} needs to be maintained due to the updates of the queries Update 1 and Update 3. The update query Update 2 need not be maintained because V^{qu} does not contain any tuples which are updated by it. The sizes of V^{qu} and V^q will be less than of V. Hence, materializing V^{qu} and V^q instead of V reduces the storage cost. Also, it does not reduce the number of queries that can be answered since every read-only query can still be answered using V^{qu} or V^q. Also, the cost of answering the queries using V^{qu} and V^q will be less than that by using V because the sizes of V^{qu} and V^q will be less than that of V. Thus, it gives two advantages: the saving in the storage cost and the saving in the query answering cost.
If V were to be materialized, all the update queries Update 1, Update 2 and Update 3 incur maintenance on V. But, if V^{qu} and V^q were to be materialized, then only Update 1 and Update 3 queries incur maintenance on V^{qu} as illustrated above. Thus, the number of updates that need to maintained will decrease if V^{qu} and V^q are materialized. But, the cost of maintaining updates of an update query on V^{qu} will be much more than the cost of maintaining on V because extra relations need to be scanned for maintaining V^{qu}. For example, if updates due to Update 3 are to be maintained, in addition to scanning V^{qu}, relation C also needs to be scanned to determined the tuples that belong to/does not belong to V^{qu} after the update.
Thus, at the expense of the increase in the maintenance cost, materializing V^{qu} and V^q instead of V offer advantages in terms of decrease in the storage cost and the query processing cost. □

The partitions of V, V^{qu} and V^q defined in example 1 are called the Partially Materialized Partitioned Views (PMPVs) of V. We now present a more complicated example which illustrates the usefulness of the PMPV approach. We use this as a running example through out the paper.

Example 2. Consider schema of a company database as shown in table 1. All the primary keys are underlined.

- The relation **Employee** stores the information about employee which is the ID of the employee (EID), the name of the employee (EName), the age of the employee (EAge), the salary of the employee (ESal) in thousands and the ID of the department in which he works (DID which is a foreign key of DID of Department).
- The relation **Department** stores the information about the department which is the ID of the department

[1]We used relational algebra along with logical conditions to define the view definitions for ease of specifying constraints.

(DID), the name of the department (DName), the location of the department (DepLoc) and the budget of the department (DBudget) in thousands.

- The relation **Project** stores the project details which are the ID of the project (PID), the name of the project (PName), the duration of the project (PDur) and the location of the project (PLoc).
- The relation **Controls** stores the details of which department (DID which is a foreign key of DID of Department) controls which project (PID which is a foreign key of PID of Project). StartDate denotes date on which the department takes charge of the project.
- The relation **Has_Dependent** stores the information about the dependents of each employee. It has ID of the employee (EID which is a foreign key of EID of Employee), name of the dependent (Dep_Name) and the relationship the employee has with the dependent (Relnship).

Employee(EID, EName, EAge, ESal, DID)
Department(DID, DName, DepLoc, DBudget)
Project(PID, PName, PDur, PLoc)
Controls(DID, PID, StartDate)
Has_Dependent(EID, Dep_Name, Relnship)

Table 1: Example Schema

The read-only queries of the workload are shown in table 2 (queries Q1-Q5) and table 3 (queries Q6-Q10). The update queries of the workload are shown in the table 4 (queries U1-U8).

Q_1:	**SELECT** EName, ESal, DepLoc
	FROM Employee, Department
	WHERE Employee.DID = Department.DID
	AND ESal > 80

Q_2:	**SELECT** EID, EName, DName
	FROM Employee, Department
	WHERE Employee.DID = Department.DID
	AND DepLoc = "Hyderabad"

Q_3:	**SELECT** EName, PName, StartDate
	FROM Employee, Department, Project, Controls
	WHERE Employee.DID = Department.DID
	AND Department.DID = Controls.DID
	AND Controls.PID = Project.PID
	AND StartDate > "01-01-04"

Q_4:	**SELECT** EName, DBudget, PDur
	FROM Employee, Department, Project, Controls
	WHERE Employee.DID = Department.DID
	AND Department.DID = Controls.DID
	AND Controls.PID = Project.PID
	AND StartDate < "03-05-04" **AND** DBudget > 75

Q_5:	**SELECT** EName, Dep_Name
	FROM Employee, Has_Dependent
	WHERE Employee.EID = Has_Dependent.EID
	AND ESal < 40

Table 2: Read-only queries: Queries Q1-Q5

Now, let us assume that a materialized view V is defined whose definition is: $V = Employee \bowtie_{DID=DID} Department$. The SQL definition for V is as follows:

Q_6:	**SELECT** DName, PDur, EName
	FROM Employee, Department, Project, Controls
	WHERE Employee.DID = Department.DID
	AND Department.DID = Controls.DID
	AND Controls.PID = Project.PID
	AND PDur > 30 **AND** DBudget > 40

Q_7:	**SELECT** EName, DName
	FROM Employee, Department
	WHERE Employee.DID = Department.DID
	AND DBudget > 40 **AND** DBudget < 50
	AND ESal < 10

Q_8:	**SELECT** DName, PName, PLoc
	FROM Department, Project, Controls
	WHERE Department.DID = Controls.DID
	AND Controls.DID = Project.DID
	AND DBudget > 80

Q_9:	**SELECT** EName, DBudget **FROM** Employee, Department, Project, Controls **WHERE** Employee.DID = Department.DID **AND** Department.DID = Controls.DID **AND** Controls.PID = Project.PID **AND** PLoc = "Chennai" **AND** DBudget > 40 **AND** DBudget < 50
Q_{10}:	**SELECT** EName, DBudget **FROM** Employee, Department, Project, Controls **WHERE** Employee.DID = Department.DID **AND** Department.DID = Controls.DID **AND** Controls.PID = Project.PID **AND** ESal > 50 **AND** DBudget > 40 **AND** EAge > 25

Table 3: Read-only queries: Queries Q6-Q10

```
SELECT *
FROM Employee, Department
WHERE Employee.DID = Department.DID
```

Then using the traditional materialized view approach, the queries $\{Q_1, Q_2, Q_3, Q_4, Q_6, Q_7, Q_9, Q_{10}\}$ can be answered using V. And the update queries $\{U_1, U_2, U_3, U_4, U_7, U_8\}$ need to be maintained, i.e, whenever one of the above update queries occurs, V has to be maintained.

We show later in the paper that using the PMPV approach V will be partitioned into two partitions. Also, the queries $\{ Q_1, Q_4, Q_6, Q_{10} \}$ can be answered using one partition and the queries $\{ Q_2, Q_3, Q_7, Q_9 \}$ can be answered using the another. Thus, all the queries can still answered using the PMPV approach. The sizes of the partitions of V will be less than that of V and hence the cost of storing the partitions of V will be less than that of storing V and the cost of answering queries using the partitions will be less than the cost using V.

Only the update queries $\{U_2, U_3, U_5\}$ need to be maintained, thus decreasing the number of update queries that incur maintenance. But the cost maintaining the updates of $\{U_2, U_3, U_5\}$ on the partitions of V will be more than the cost of maintaining the updates of $\{U_1, U_2, U_3, U_4, U_7, U_8\}$ on V. Thus, the cost of maintenance may increase although the number of update queries that incur maintenance may decrease.

Thus, using the PMPV approach, the storage cost and the query answering cost can be decreased at the expense of

U_1:	**UPDATE** Employee
	SET ESal = 40
	WHERE ESal > 15
	AND ESal < 30

U_2:	**UPDATE** Employee
	SET ESal = 60
	WHERE ESal > 60
	AND EAge < 30

U_3:	**UPDATE** Department
	SET DBudget = DBudget*0.5
	WHERE DBudget > 60

U_4:	**UPDATE** Department
	SET DBudget = 60
	WHERE DBudget < 30

U_5:	**UPDATE** Project
	SET PDur = PDur - 10 **AND** PLoc = "Delhi"
	WHERE PDur > 20

U_6:	**UPDATE** Project
	SET PLoc = "Mumbai"
	WHERE PLoc = "Delhi"

U_7:	**UPDATE** Department
	SET DLoc = "Delhi"
	WHERE PLoc = "Mumbai"

U_8:	**UPDATE** Employee
	SET ESal = ESal*2
	WHERE ESal > 11
	AND EAge < 20

Table 4: Update queries of the workload

increase in the maintenance cost. □

1.2 Advantages of PMPV approach

We develop the notion of partially materialized partitioned view (PMPV) wherein each view is partitioned based on the workload of the database and only some of the partitions of each view are materialized (hence the name "partially materialized"). The notion of PMPV has many-fold advantages:

- The idea of PMPV is **independent** of the view selection algorithm and the algorithm used for answering the queries using the materialized views (sections 3.2 and 3.3).
- Since only a portion of the view is materialized, the space occupied by the view will be less. Hence, under a storage space constraint more views can be materialized.
- Since the sizes of the PMPVs will be less than the traditional views, the cost of processing queries using PMPVs will be less than the cost of using the traditional views.
- Not all updates on the base relations incur updates on the PMPVs. Based on the workload and the PMPVs, it can be easily determined which of the updates queries incur updates on the PMPVs. But the cost of maintaining a PMPV will be greater than cost of maintaining its corresponding normal materialized view.

1.3 Contributions

In this paper, we

- developed the notion of *Partially Materialized Partitioned Views* and showed its advantages over the normal materialized views,
- developed algorithm for selecting and maintaining PMPVs and
- experimentally evaluated the performance of PMPVs over the traditional approach and showed the usefulness of the approach.

The organization of the paper is as follows. Section 2 describes the notions of partitioning the materialized views and partially materializing them. Section 3 describes algorithm to select PMPVs. Section 4 discusses method to maintain PMPVs. Section 5 presents the experimental results. Section 6 presents the related work. Section 7 describes the future work and finally section 8 concludes the paper.

2. PARTIALLY MATERIALIZED PARTITIONED VIEWS

We assume that the workload of the database is known in advance. The workload consists of the "read-only queries" and "update queries". In this paper, we assume that the read-only queries will only be SPJ queries and the update queries will be on single relations (i.e. both the read sets and write sets of any update query belong to a single relation).

2.1 Notation

Let $\mathbb{R} = \{R_1, R_2, \ldots, R_n\}$ be the set of n base relations. $\mathbb{A}(R_i)$ denotes the set of attributes of relation R_i.

Let $\mathbb{Q} = \{Q_1, Q_2, \ldots, Q_p\}$ be the set of p read-only queries. For each query Q_i:

- $\mathbb{R}(Q_i)$ denotes the set of base relations which participate in the query Q_i.
- $\mathbb{A}(Q_i)$ denotes the set of attributes which belong to the *WHERE clause or SELECT clause* of Q_i.
- $C(\mathbb{A}(Q_i))$ denotes the *where clause* of Q_i.
- f_i denotes the frequency of occurrence of the query Q_i.

Example 3. For query Q_1 of the example,

- $\mathbb{R}(Q_1)$ = {Employee, Department}
- $\mathbb{A}(Q_1)$ = {Employee.DID, EName, ESal, Department.DID, DepLoc} and
- $C(\mathbb{A}(Q_1))$ = "Employee.DID = Department.DID AND ESal > 80"

Similarly, for query Q_9,

- $\mathbb{R}(Q_9)$ = {Employee, Department, Project, Controls},
- $\mathbb{A}(Q_9)$ = {Employee.DID, EName, Department.DID, DBudget, Project.PID, PLoc, Controls.PID, Controls.DID} and
- $C(\mathbb{A}(Q_9))$ = "Employee.DID = Department.DID AND Department.DID = Controls.DID AND Controls.PID = Project.PID AND PLoc = "Chennai" AND DBudget > 40 AND DBudget < 50". □

Let $\mathbb{U} = \{U_1, U_2, \ldots, U_r\}$ be the set of r update queries. For each update query U_i:

- $R(U_i)$ denotes the base relation on which U_i performs the update. Since we assume only single table updates, $R(U_i)$ will be a single relation.
- $\mathbb{A}_S(U_i)$ denotes the set of attributes that belong to the *set clause* of the update query U_i.
- $\mathbb{A}_W(U_i)$ denotes the set of attributes that belong to the *where clause* of the update query U_i.
- $C(\mathbb{A}_S(U_i))$ and $C(\mathbb{A}_W(U_i))$ denotes the set condition and where condition of the update query U_i respectively.
- g_i denotes the frequency of occurrence of the query U_i.

Example 4. For the update query U_2 in the example,

- $R(U_2)$ = Employee
- $\mathbb{A}_S(U_2)$ = {ESal}
- $\mathbb{A}_W(U_2)$ = {ESal, EAge}
- $C(\mathbb{A}_S(U_2))$ = "ESal = 60"
- $C(\mathbb{A}_W(U_2))$ = "ESal > 60 AND EAge < 30"

Similarly, for the update query U_5 in the example,

- $R(U_5)$ = Project
- $\mathbb{A}_S(U_5)$ = {PDur, PLoc}
- $\mathbb{A}_W(U_5)$ = {PDur, PLoc}
- $C(\mathbb{A}_S(U_5))$ = "PDur = PDur-10 AND PLoc="Delhi""
- $C(\mathbb{A}_W(U_5))$ = "PDur > 20" □

Note that $\mathbb{A}_S(U_i), \mathbb{A}_W(U_i) \in \mathbb{A}(R(U_i))$. Also, the set $\mathbb{A}_S(U_i) \cap \mathbb{A}_W(U_i)$ *may not* be empty.

Let V be a view that is selected to be materialized. Based on V, we define the notion of PMPV.

- Let $\mathbb{R}(V)$ denote the set of base relations that participate in the view V.
- Let $\mathbb{A}(V)$ be the set of attributes that are present in the definition of V.
- Let $C(\mathbb{A}(V))$ be the condition on which V is defined.

Let $\mathbb{Q}(V)$ be the set of read-only queries which can be answered using the view V, i.e., each query in it can be re-written using V. Trivially, $\mathbb{Q}(V) \subseteq \mathbb{Q}$. The set $\mathbb{Q}(V)$ is called *the set of answerable queries of V*

The set of answerable queries of V is further divided into two parts: the set of answerable queries of V with updates $\mathbb{Q}_{qu}(V)$ and the set of answerable queries of V without updates $\mathbb{Q}_q(V)$. They are defined as follows:

- **The set of answerable queries of V with updates, $\mathbb{Q}_{qu}(V)$:** It is the subset of the set of answerable queries of V, such that for every query Q_i that belongs to it, there exists at least one update query which updates the attributes of Q_i, $\mathbb{A}(Q_i)$.

$$\begin{aligned}\mathbb{Q}_{qu}(V) = \; & \{Q_i \in \mathbb{Q}(V) \mid \exists U_i \in \mathbb{U} \text{ such that} \\ & \mathbb{A}(Q_i) \cap \mathbb{A}_S(U_i) \neq \phi \text{ and} \\ & C(\mathbb{A}(Q_i)) \wedge C(\mathbb{A}_W(U_i))\}\end{aligned}$$

- **The set of answerable queries of V without updates, $\mathbb{Q}_q(V)$:** It is the subset of the set of answerable queries of V, such that for every query Q_i that belongs to it, there **does not** exists even one update query which updates the attributes of Q_i, $\mathbb{A}(Q_i)$.

$$\begin{aligned}\mathbb{Q}_q(V) = \; & \{Q_i \in \mathbb{Q}(V) \mid \forall U_i \in \mathbb{U} \text{ either} \\ & \mathbb{A}(Q_i) \cap \mathbb{A}_S(U_i) = \phi \text{ or} \\ & C(\mathbb{A}(Q_i)) \wedge C(\mathbb{A}_W(U_i)) \text{ is false} \\ & \text{or both}\}\end{aligned}$$

Any query which can be answered using V should belong to either $\mathbb{Q}_{qu}(V)$ or $\mathbb{Q}_q(V)$. That is, $\mathbb{Q}_{qu}(V) \cup \mathbb{Q}_q(V) = \mathbb{Q}(V)$.

Based on the definition of $\mathbb{Q}_{qu}(V)$, the update queries which affect the PMPV is defined which is called the set of required updates of V, $\mathbb{U}_{qu}(V)$.

The set of required updates of V, $\mathbb{U}_{qu}(V)$: It is defined as the set of update queries which update the attributes of at least one query in the set of answerable queries of V with updates.

$$\begin{aligned}\mathbb{U}_{qu}(V) = \; & \{U_i \in \mathbb{U} \mid \exists Q_i \in \mathbb{Q}_{qu}(V) \text{ such that} \\ & \mathbb{A}(Q_i) \cap \mathbb{A}_S(U_i) \neq \phi \text{ and} \\ & C(\mathbb{A}(Q_i)) \wedge C(\mathbb{A}_W(U_i))\}\end{aligned}$$

The disjunction of the conditions of all the queries in $\mathbb{Q}_{qu}(V)$ is denoted as $\mathbb{C}_\vee(\mathbb{Q}_{qu}(V))$ and is called as *disjunction of* $\mathbb{Q}_{qu}(V)$.

$$\mathbb{C}_\vee(\mathbb{Q}_{qu}(V)) = \bigvee_{Q_i \in \mathbb{Q}_{qu}(V)} C(\mathbb{A}(Q_i))$$

Similarly, the disjunction of $\mathbb{Q}_q(V)$ is defined and denoted as $\mathbb{C}_\vee(\mathbb{Q}_q(V))$.

The disjunction of $\mathbb{U}_{qu}(V)$ is defined as the disjunction of the where conditions of all its update queries and is denoted as $\mathbb{C}_\vee(\mathbb{Q}_{qu}(V))$.

$$\mathbb{C}_\vee(\mathbb{U}_{qu}(V)) = \bigvee_{U_i \in \mathbb{U}_{qu}(V)} C(\mathbb{A}_W(U_i))$$

Example 5. Let V be the materialized view as defined in the example: $V = Employee \bowtie_{DID=DID} Department$. Then,

- $\mathbb{R}(V)$ = {Employee, Department}
- $\mathbb{A}(V)$ = {Employee.DID, Department.DID}
- $C(\mathbb{A}(V))$ = "Employee.DID = Department.DID"
- $\mathbb{Q}(V) = \{Q_1, Q_2, Q_3, Q_4, Q_6, Q_7, Q_9, Q_{10}\}$
- $\mathbb{Q}_{qu}(V) = \{Q_1, Q_4, Q_6, Q_{10}\}$
- $\mathbb{Q}_q(V) = \{Q_2, Q_3, Q_7, Q_9\}$
- $\mathbb{U}_{qu}(V) = \{U_2, U_3, U_5\}$

For each query in $\mathbb{Q}_{qu}(V)$, there exists at least one update query which updates the attributes of it. For Q_1, the update query U_2 updates the tuples which are accessed by Q_1. Similarly, for Q_4 it is U_3, for Q_6 it is U_3 and U_5 and for Q_{10} it is U_2 and U_3.

For each query in $\mathbb{Q}_q(V)$, there does not exist at least one update query which updates the tuples which are accessed by it.

For each update query in $\mathbb{U}_{qu}(V)$, there is at least one query in $\mathbb{Q}_{qu}(V)$ whose tuples it updates.

- $\mathbb{C}_V(\mathbb{Q}_{qu}(V))$ will be the disjunction of the *where clauses* of $\{Q_1, Q_4, Q_6, Q_{10}\}$.
- $\mathbb{C}_V(\mathbb{Q}_q(V))$ will be the disjunction of the *where clauses* of $\{Q_2, Q_3, Q_7, Q_9\}$.
- $\mathbb{C}_V(\mathbb{U}_{qu}(V))$ will be the disjunction of the *where clauses* of $\{U_2, U_3, U_5\}$. □

2.2 Partitions of Views

Given a view V, based on the set of answerable set of queries with updates and without updates $\mathbb{Q}_{qu}(V)$ and $\mathbb{Q}_q(V)$ and the set of required update queries $\mathbb{U}_{qu}(V)$, the view V will be partitioned into three partitions:

- **Read and Update Partition of V, V^{qu}:** It contains all the tuples of V which are accessed by at least one of the queries in the set of answerable queries of V with updates or updated by at least one of the queries in the set of required update queries.

$$\begin{aligned} V^{qu} &= \{t \in V \mid \exists Q_i \in \mathbb{Q}_{qu}(V) \text{ such that } \\ &\qquad Q_i \textbf{ accesses } t \\ &\qquad \textbf{or } \exists U_i \in \mathbb{U}_{qu}(V) \text{ such that } \\ &\qquad U_i \textbf{ accesses } t\} \\ &= \{t \in V \mid t \text{ satisfies } \mathbb{C}_V(\mathbb{Q}_{qu}(V)) \vee \mathbb{C}_V(\mathbb{U}_{qu}(V))\} \end{aligned}$$

- **Read-Only Partition of V, V^q:** It contains all the tuples of V which are accessed by at least one of the query in the set of answerable queries of V without updates. Since each tuple in it will be accessed by one of the query in $\mathbb{Q}_q(V)$, none of the update queries in the set of required update queries of V update it.

$$\begin{aligned} V^q &= \{t \in V \mid \exists Q_i \in \mathbb{Q}_q(V) \text{ such that } Q_i \textbf{ accesses } t\} \\ &= \{t \in V \mid t \text{ satisfies } \mathbb{C}_V(\mathbb{Q}_q(V))\} \end{aligned}$$

- **Remaining Partition of V, V^{rem}:** It contains all the tuples of V which belong to neither V^{qu} nor V^q.

$$V^{rem} = V - (V^{qu} \cup V^q)$$

Thus, each tuple in V^{qu} satisfies the condition $C(\mathbb{A}(V)) \wedge (\mathbb{C}_V(\mathbb{Q}_{qu}(V)) \vee \mathbb{C}_V(\mathbb{U}_{qu}(V)))$. And, each tuple in V^q satisfies the condition $C(\mathbb{A}(V)) \wedge \mathbb{C}_V(\mathbb{Q}_q(V))$. Also, the partitions V^{qu} and V^q may be overlapping, i.e., $V^{qu} \cap V^q$ may not be empty.

2.3 Partial Materialization

Given a view V, the read and update partition of V, V^{qu} and the read-only partition of V, V^q constitute the Partially Materialized Partition View of V. Thus, instead of materializing the entire V, only V^{qu} and V^q will be materialized. The remaining partition V^{rem} need not be materialized. All the queries which belong to $\mathbb{Q}_{qu}(V)$ can be rewritten using V^{qu} instead of V. Similarly, all the queries which belong to $\mathbb{Q}_q(V)$ can be rewritten using V^q instead of V. Since $\mathbb{Q}_{qu}(V) \cup \mathbb{Q}_q(V) = \mathbb{Q}(V)$, all the queries which can be answered using V can now be answered using either V^{qu} or V^q using the PMPV approach. Thus, not materializing the remaining partition of V, V^{rem} does not affect the set of queries which can be answered using the materialized view. Only the update queries which belong to $\mathbb{U}_{qu}(V)$ incur maintenance cost for PMPV. Also, each update query in $\mathbb{U}_{qu}(V)$ updates only the tuples of V^{qu}.

Example 6. If V is defined as before: $V = Employee \bowtie_{DID=DID} Department$, if the PMPV V^{qu} and V^q are materialized then:

- the queries $\mathbb{Q}_{qu}(V) = \{Q_1, Q_4, Q_6, Q_{10}\}$ can be answered using V^{qu}
- the queries $\mathbb{Q}_q(V) = \{Q_2, Q_3, Q_7, Q_9\}$ can be answered using V^q
- only the update queries in $\mathbb{U}_{qu}(V) = \{U_2, U_3, U_5\}$ incur maintenance of V^{qu}. □

3. SELECTION OF PMPV

Since the workload that is to be supported is a fixed set of "read-only queries", a multi-query optimizer [7, 33, 31, 30] can be run and an execution plan will be generated. The execution plan will contain intermediate results of the queries and each such intermediate result is a potential candidate for materialization since it may reduce the query processing cost. Once an intermediate node is materialized, the updates on the base relations will incur updates on the materialized view in order to maintain the consistency.

The existing techniques to select the "best" set of materialized views is to associate a benefit value to each of the intermediate results and select the most beneficial set of views to be materialized based on a cost model [13, 15]. [31] discusses three heuristics to select the materialized views based on the Volcano architecture [9]. [3, 34] discuss the selection of materialized views in a multidimensional database. [39] proposes a heuristic approach called MVPP to select the most beneficial set of views. *The notion of PMPV is independent of the cost model and the view selection algorithm used to select the views.*

Intuitively, the technique of PMPV works well, if the selectivity factors of the read-only queries and the update queries is low and if there is a large overlap between the tuples that are accessed by the various read-only queries as well as between the tuples that are accessed by the read-only queries and the update queries. On the other hand, if the above conditions are not met, it will be beneficial to materialize V itself, instead of partitioning it and partially materializing it. We now define the notion of benefit of a PMPV.

3.1 Benefit of PMPV of a view

Let V be the materialized view and V^{qu}, V^q be its corresponding PMPVs. Let $\mathbb{Q}(V)$ and $\mathbb{U}(V)$ denote the set of read-only and the set of update queries that access/update the view V respectively. Let $\mathbb{Q}_{qu}(V)$ and $\mathbb{Q}_q(V)$ denote the set of read-only queries that access V^{qu} and V^q respectively. Let $\mathbb{U}_{qu}(V)$ denote the set of update queries that update V^{qu}. Let $QC(Q_i, V)$ be the cost of processing the query Q_i using the materialized V and let $UC(U_i, V)$ be the cost of updating the materialized view V because of the update query U_i. Then the benefit of PMPV of a materialized view V, $B_{PMPV}(V)$ is given by:

$$\begin{aligned} B_{PMPV}(V) &= [\Sigma_{Q_i \in \mathbb{Q}(V)} f_i * QC(Q_i, V) \\ &\quad + \Sigma_{U_i \in \mathbb{U}(V)} g_i * UC(U_i, V)] \\ &- \quad [\Sigma_{Q_i \in \mathbb{Q}_{qu}(V)} f_i * QC(Q_i, V^{qu}) \\ &\quad + \Sigma_{Q_i \in \mathbb{Q}_q(V)} f_i * QC(Q_i, V^q) \\ &\quad + \Sigma_{U_i \in \mathbb{U}_{qu}(V)} g_i * UC(U_i, V^{qu})] \end{aligned}$$

The first part of the benefit function denotes the sum of the cost of processing the read-only queries using V and the cost of maintaining V due to the update queries. The second part of the query denotes the sum of answering the read-only queries using V^{qu} and V^q and the cost of maintaining V^{qu} due to the update queries. The difference between the two denotes the benefit of materializing the PMPV of V, V^{qu} and V^q instead of materializing V.

If $B_{PMPV}(V)$ is a positive quantity, then it implies that it is beneficial to materialize V^{qu} and V^q instead of V.

If $M = \{V_1, V_2, \ldots, V_m\}$ is the set of materialized views and PMPVs already selected, then the benefit of PMPV of V w.r.t. M is denoted by $B_{PMPV}(V, M)$. The views in M may be the full complete views in the traditional sense or some of them might be PMPVs. The benefit function for $B_{PMPV}(V, M)$ can be written as:

$$\begin{aligned} B_{PMPV}(V, M) &= [\Sigma_{Q_i \in \mathbb{Q}(V)} f_i * QC(Q_i, M \cup V) \\ &\quad + \Sigma_{U_i \in \mathbb{U}(V)} g_i * UC(U_i, M \cup V)] \\ &- [\Sigma_{Q_i \in \mathbb{Q}_{qu}(V)} f_i * QC(Q_i, M \cup V^{qu}) \\ &\quad + \Sigma_{Q_i \in \mathbb{Q}_q(V)} f_i * QC(Q_i, M \cup V^q) \\ &\quad + \Sigma_{U_i \in \mathbb{U}_{qu}(V)} g_i * UC(U_i, M \cup V^{qu})] \end{aligned}$$

3.2 PMPV Selection Algorithm

If M is the set of materialized views and PMPVs already selected and if a view selection algorithm selects V to be materialized, then if the benefit of PMPV of V w.r.t. M, $B_{PMPV}(V, M)$ is a positive quantity then V^{qu} and V^q will be materialized. Otherwise, V will be materialized. Thus, the selection of PMPV is independent of the algorithm used to select the traditional materialized views.

3.3 Answering queries using PMPV

If PMPVs of a view V are materialized, then the queries $\mathbb{Q}_{qu}(V)$ can be answered using V^{qu} and the queries $\mathbb{Q}_q(V)$ can be answered using V^q. Also, $\mathbb{Q}_{qu}(V) \cup \mathbb{Q}_q(V) = \mathbb{Q}(V)$. Thus, all the queries that can be answered using V can as well be answered using the PMPVs of V.

Corollary 1. *If a query Q can be answered by a view V, it can also be answered by the PMPVs of V, V^{qu} and V^q.*

Therefore, the standard algorithms that are used to answer queries using materialized views [21, 28, 16], can be used for PMPVs also without any changes.

4. MAINTENANCE OF PMPV

Let $U_i \in \mathbb{U}_{qu}(V)$ be an update query which updates the base relation $R_j \in \mathbb{R}$. Let $t_b \in R_j$ be a tuple which is getting updated due to U_i. Hence, t_b satisfies $C(A_W(U_i))$. Let t_a be the resultant tuple obtained after updating t_b.

Let $t'_b \in V$ be the corresponding tuple in V to which t contributes. Let t'_a be the resultant tuple obtained after updating t'_b.

The various cases for maintaining PMPVs are shown in the table 5. insert(t, V) means inserting the tuple t into the view V and delete(t, V) means deleting the tuple t from the view V.

For cases 1 to 16, we assume that $t'_b \in V$ and $t'_a \in V$.

• **Case 1:** $t'_b \in V^{qu}$ and $t'_b \in V^q$, $t'_a \in V^{qu}$ and $t'_a \in V^q$. Since t'_b and t'_a both belong to V^{qu} and V^q, no action is required.

• **Case 2:** $t'_b \in V^{qu}$ and $t'_b \in V^q$, $t'_a \in V^{qu}$ and $t'_a \notin V^q$. Since t'_a does not belongs to V^q, delete the tuple t'_b from V^q.

• **Case 3:** $t'_b \in V^{qu}$ and $t'_b \in V^q$, $t'_a \notin V^{qu}$ and $t'_a \in V^q$. Since t'_a does not belongs to V^{qu}, delete the tuple t'_b from V^{qu}.

• **Case 4:** $t'_b \in V^{qu}$ and $t'_b \in V^q$, $t'_a \notin V^{qu}$ and $t'_a \notin V^q$. Since, t'_a does not belongs to V^{qu} and V^q, delete the tuple t'_b from V^{qu} and V^q.

• **Case 5:** $t'_b \in V^{qu}$ and $t'_b \notin V^q$, $t'_a \in V^{qu}$ and $t'_a \in V^q$. Since t'_a belongs to V^q though t'_b does not belongs to V^q, insert t'_a into V^q.

• **Case 6:** $t'_b \in V^{qu}$ and $t'_b \notin V^q$, $t'_a \in V^{qu}$ and $t'_a \notin V^q$. Since there are no changes, no action is required.

• **Case 7:** $t'_b \in V^{qu}$ and $t'_b \notin V^q$, $t'_a \notin V^{qu}$ and $t'_a \in V^q$. Since t'_b belongs to V^{qu} but t'_a does not belongs to V^{qu}, delete the tuple t'_b from V^{qu}. Since t'_b does not belongs to V^q but t'_a belongs to V^q, insert the tuple t'_a into V^q.

• **Case 8:** $t'_b \in V^{qu}$ and $t'_b \notin V^q$, $t'_a \notin V^{qu}$ and $t'_a \notin V^q$. Since t'_b belongs to V^{qu} but t'_a does not belongs to V^{qu}, delete the tuple t'_b from V^{qu}.

• **Case 9:** $t'_b \notin V^{qu}$ and $t'_b \in V^q$, $t'_a \in V^{qu}$ and $t'_a \in V^q$. Since t'_b does not belongs to V^{qu} but t'_a not belongs to V^{qu}, insert the tuple t'_a into V^{qu}.

• **Case 10:** $t'_b \notin V^{qu}$ and $t'_b \in V^q$, $t'_a \in V^{qu}$ and $t'_a \notin V^q$. Since t'_b does not belongs to V^{qu} but t'_a not belongs to V^{qu}, insert the tuple t'_a into V^{qu}. Since t'_b belongs to V^q but t'_a does not belongs to V^q, delete the tuple t'_a from V^q.

• **Case 11:** $t'_b \notin V^{qu}$ and $t'_b \in V^q$, $t'_a \notin V^{qu}$ and $t'_a \in V^q$. Since there are no changes, no action is required.

• **Case 12:** $t'_b \notin V^{qu}$ and $t'_b \in V^q$, $t'_a \notin V^{qu}$ and $t'_a \notin V^q$. Since t'_b belongs to V^q but t'_a does not belongs to V^q, delete the tuple t'_a from V^q.

• **Case 13:** $t'_b \notin V^{qu}$ and $t'_b \notin V^q$, $t'_a \in V^{qu}$ and $t'_a \in V^q$. Since t'_b does not belongs to V^{qu} but t'_a not belongs to V^{qu}, insert the tuple t'_a into V^{qu}. Since t'_a belongs to V^q though t'_b does not belongs to V^q, insert t'_a into V^q.

• **Case 14:** $t'_b \notin V^{qu}$ and $t'_b \notin V^q$, $t'_a \in V^{qu}$ and $t'_a \notin V^q$. Since t'_b does not belongs to V^{qu} but t'_a not belongs to V^{qu}, insert the tuple t'_a into V^{qu}.

• **Case 15:** $t'_b \notin V^{qu}$ and $t'_b \notin V^q$, $t'_a \notin V^{qu}$ and $t'_a \in V^q$. Since t'_a belongs to V^q though t'_b does not belongs to V^q, insert t'_a into V^q.

• **Case 16:** $t'_b \notin V^{qu}$ and $t'_b \notin V^q$, $t'_a \notin V^{qu}$ and $t'_a \notin V^q$. Since there are no changes, no action is required.

For cases 17 to 20, we assume that $t'_b \in V$ and $t'_a \notin V$.

• **Case 17:** $t'_b \in V^{qu}$ and $t'_b \in V^q$. Since t'_a does not belongs to V, delete the tuple t'_b from V^{qu} and V^q.

• **Case 18:** $t'_b \in V^{qu}$ and $t'_b \notin V^q$. Since t'_b belongs to V^{qu} but t'_a does not belongs to V, delete the tuple t'_b from V^{qu}.

• **Case 19:** $t'_b \notin V^{qu}$ and $t'_b \in V^q$. Since t'_b belongs to V^q but t'_a does not belongs to V, delete the tuple t'_b from V^q.

• **Case 20:** $t'_b \notin V^{qu}$ and $t'_b \notin V^q$. Since there are no changes, no action is required.

For cases 21 to 24, we assume that $t'_b \notin V$ and $t'_a \in V$.

• **Case 21:** $t'_a \in V^{qu}$ and $t'_a \in V^q$. Since t'_b does not belongs to V but t'_a belongs to both V^{qu} and V^q, insert t'_a into V^{qu} and V^q.

• **Case 22:** $t'_a \in V^{qu}$ and $t'_a \notin V^q$. Since t'_b does not belongs to V but t'_a belongs to V^{qu}, insert t'_a into V^{qu}.

• **Case 23:** $t'_a \notin V^{qu}$ and $t'_a \in V^q$. Since t'_b does not belongs

		$t'_a \in V$				$t'_a \notin V$
		$t'_a \in V^{qu} \wedge$ $t'_a \in V^q$	$t'_a \in V^{qu} \wedge$ $t'_a \notin V^q$	$t'_a \notin V^{qu} \wedge$ $t'_a \in V^q$	$t'_a \notin V^{qu} \wedge$ $t'_a \notin V^q$	
$t'_b \in V$	$t'_b \in V^{qu} \wedge$ $t'_b \in V^q$	**1.**No action	**2.** delete(t'_b, V^q)	**3.** delete(t'_b, V^{qu})	**4.** delete(t'_b, V^{qu}) delete(t'_b, V^q)	**17.** delete(t'_b, V^{qu}) delete(t'_b, V^q)
	$t'_b \in V^{qu} \wedge$ $t'_b \notin V^q$	**5.** insert(t'_a, V^q)	**6.**No action	**7.** delete(t'_b, V^{qu}) insert(t'_a, V^q)	**8.** delete(t'_b, V^{qu})	**18.** delete(t'_b, V^{qu})
	$t'_b \notin V^{qu} \wedge$ $t'_b \in V^q$	**9.** insert(t'_a, V^{qu})	**10.** insert(t'_a, V^{qu}) delete(t'_b, V^q)	**11.**No action	**12.** delete(t'_b, V^q)	**19.** delete(t'_b, V^q)
	$t'_b \notin V^{qu} \wedge$ $t'_b \notin V^q$	**13.** insert(t'_a, V^{qu}) insert(t'_a, V^q)	**14.** insert(t'_a, V^{qu})	**15.** insert(t'_a, V^q)	**16.**No action	**20.**No action
$t'_b \notin V$		**21.** insert(t'_a, V^{qu}) insert(t'_a, V^q)	**22.** insert(t'_a, V^{qu})	**23.** insert(t'_a, V^q)	**24.**No action	**25.**No action

Table 5: Maintenance of PMPV

to V but t'_a belongs to V^q, insert t'_a into V^q.
• **Case 24:** $t'_a \notin V^{qu}$ and $t'_a \notin V^q$. Since, there are no changes, no action is required.
For case 25, we assume that $t'_b \notin V$ and $t'_a \notin V$.
• **Case 25:** $t'_b \notin V$ and $t'_a \notin V$. Since, there are no changes, no action is required.

Depending on into which case each tuple falls into, the corresponding action will be taken.

5. EXPERIMENTAL RESULTS

We implemented the notion of PMPV and compared the performance of the results with that of the normal materialized views.

We considered ten read-only queries and five update queries (we omit their SQL statements due to lack of space) on the schema described in example 2 for our experiments. The selectivity factors of the conditions in the queries and the frequencies of the queries are varied as described later. We do not assume presence of indexes on any of the relations. The workload of the database is analyzed and a global plan for answering the queries is constructed. A greedy version of the algorithm proposed in [31, 30] is implemented to obtain the global plan. We implemented the greedy algorithm (Gupta algorithm) proposed in [13] to select the views to be materialized based on the global plan constructed. Using the method proposed in the paper, we find the corresponding PMPVs for each of the views selected by Gupta algorithm. Let the views selected by Gupta algorithm be called as Gupta views and the views selected by the PMPV algorithm be called as PMPVs.

Let S_g and S_p be the space occupied by the views selected by Gupta algorithm and the PMPV algorithm respectively. Let QC_g and QC_p be the cost of answering the queries using the Gupta views and using the PMPV respectively. Let UC_g and UC_p be the cost of maintaining the Gupta views and PMPVs respectively.

To compare the performance of the PMPVs with that the Gupta views, we considered three measures. The constraint is put on the number of views selected. If the same number of views are allowed to be selected by both Gupta algorithm and PMPV algorithm, then the following measures are calculated from them.

• **Percentage Space Saved:** It gives the percentage of the space saved by storing the PMPVs with respect to storing the Gupta views. It denotes the percentage of the space occupied by Gupta views that is saved by storing PMPV instead of Gupta views. The more its value, the better is the efficiency of the PMPV approach. It is given by:

$$\text{Percentage Space Saved} = \frac{S_g - S_p}{S_g} * 100$$

• **Percentage Query Cost Saved:** It gives the percentage of the saving made on the cost of answering the queries using PMPVs with respect to the cost using the Gupta views. It denotes the percentage of cost of answering queries using Gupta views that is saved if the queries are answered using PMPVs. The more its value, the better is the efficiency of the PMPV approach. It is given by:

$$\text{Percentage Query Cost Saved} = \frac{QC_g - QC_p}{QC_g} * 100$$

• **Percentage Maintenance Cost:** It gives the percentage of the extra maintenance cost incurred due to storing the PMPVs with respect to the maintenance cost the Gupta Views. It denotes the percentage of the extra maintenance cost that is incurred due to storage of PMPVs instead of normal materialized views. The less its value, the better is the efficiency of the PMPV approach. It is given by:

$$\text{Percentage Maintenance Cost} = \frac{UC_p - UC_g}{UC_p} * 100$$

We studied the effect of the selectivity factors of the queries and the frequency of the queries on the above three measures. The selectivity factors and frequency of the queries in the workload are varied as follows.

The selectivity factor of a query is defined as the average of the selectivity factors of all the conditions present in the query. The selectivity factors of the queries are varied at three values: 0.1 (low), 0.5 (medium) and 0.9 (high). The average frequency of the queries in the workload is varied from 5 to 100 in steps of 5. The number of views to be selected is varied between 1 and 10.

5.1 Storage Space Saved

Figure 1 shows the percentage of the space saved when: the average selectivity factor is 0.1, the number of views selected varies from 1 to 10 and the average frequency of the queries increases from 25 to 100. Since the selectivity factors of the queries is very low (average is 0.1), the sizes of

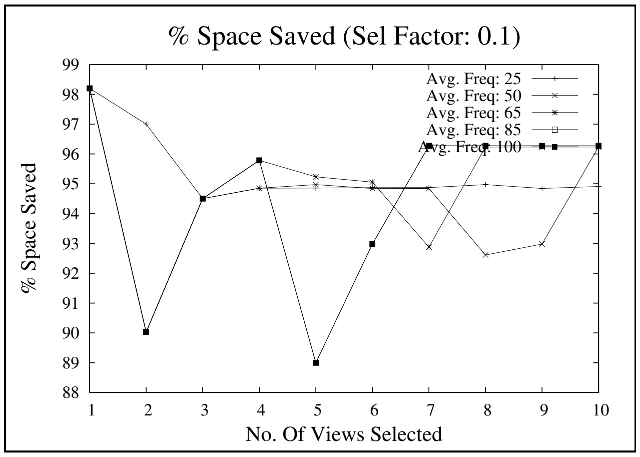

Figure 1: % Space Saved (sel. factor: 0.1)

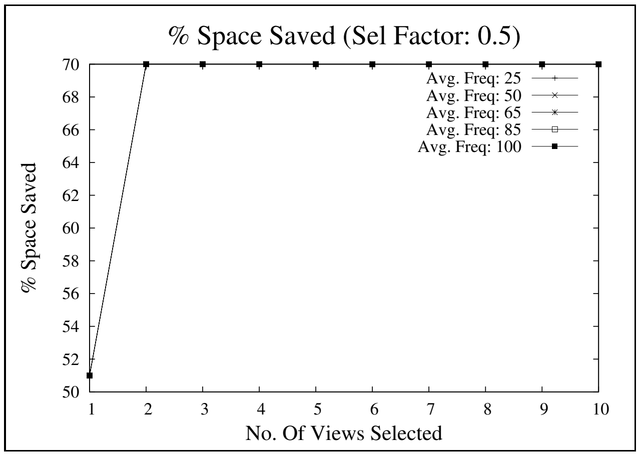

Figure 2: % Space Saved (sel. factor: 0.5)

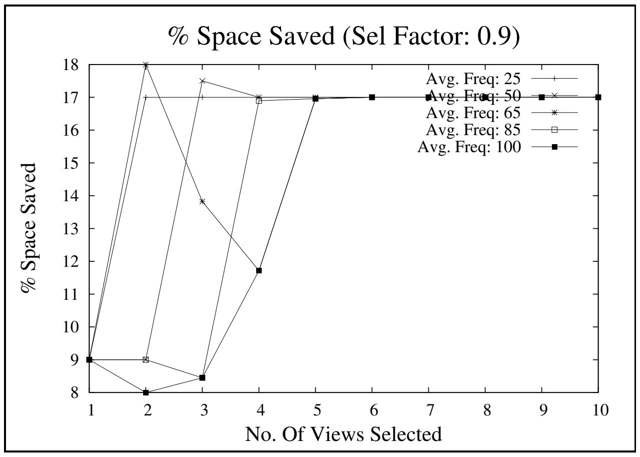

Figure 3: % Space Saved (sel. factor: 0.9)

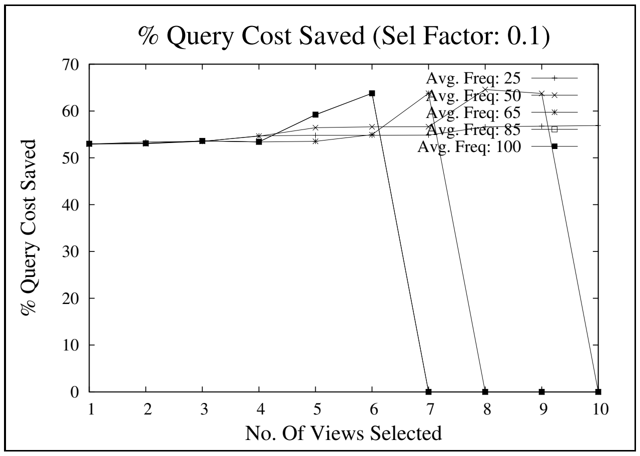

Figure 4: % Query Cost Saved (sel. factor: 0.1)

the PMPVs compared to the sizes of the Gupta views will be very less. Hence, the percentage of the space saved will be high and it varies around 95 %.

Figure 2 shows the percentage of the space saved when the selectivity factor is increased to 0.5. Since the selectivity factor of the queries has increased, the sizes of the PMPVs also increase when compared to the previous case of low selectivity factor. Hence, the percentage of the space saved decreases and it varies between 50 and 70 %.

Figure 3 shows the percentage of the space saved when the selectivity factor is further increased to 0.9. Since the selectivity factor is very high, the sizes of the PMPVs selected will be of the order of the Gupta views. Hence, the percentage of the space saved will decrease further and it varies between 8 and 18 %.

5.2 Query Processing Cost Saved

Figure 4 shows the percentage of the query cost saved when: the selectivity factor is taken to be 0.1, the number of views selected is varied between 1 and 10 and the average frequency increases from 25 to 100. At a lower value of the average frequencies (around 25), the query cost saved varies around 50. But it decreases from 50 to 0 as the number of views selected increases from 1 to 10. When the average frequency increases to 100, as the number of views selected increases to 7 and thereafter, the query processing cost with Gupta views increases and hence the percentage of the query cost saved tends to 0.

Figure 5 shows the percentage of the query cost saved when the selectivity factor increases to 0.5. The behavior of the graph is similar to the above case where the query cost saved decreases as the number of views selected increases from 1 to 10.

Figure 6 shows the percentage of the query cost saved when the selectivity factor further increases to 0.9. Just like the above cases, even here the query cost saved decreases as the number of views selected increases from 1 to 10.

Thus, as the selectivity factor of the queries increases, the sizes of the PMPVs increase and hence the percentage of the query cost saved will decrease.

5.3 Maintenance Cost

Figure 7 shows the percentage of the increase in the maintenance cost when: the selectivity factor is taken to be 0.1, the number of views selected varies between 2 and 10 and the average frequency of the queries increases from 25 to

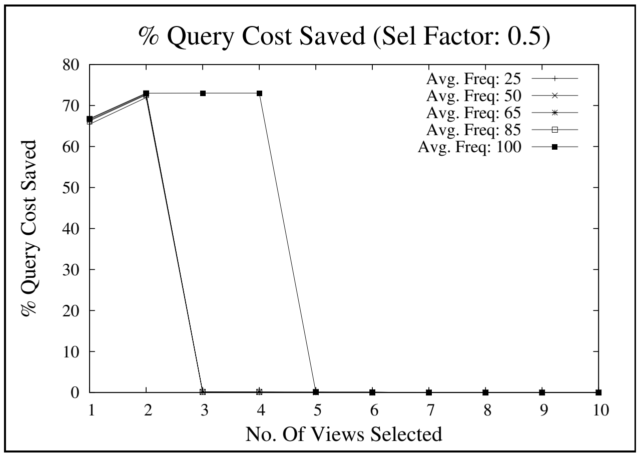

Figure 5: % Query Cost Saved (sel. factor: 0.5)

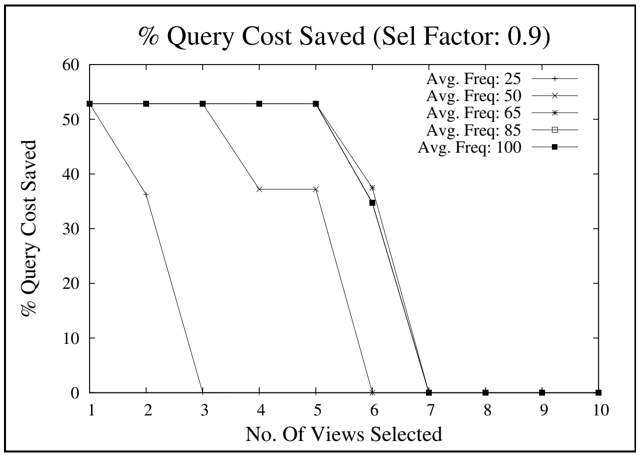

Figure 6: % Query Cost Saved (sel. factor: 0.9)

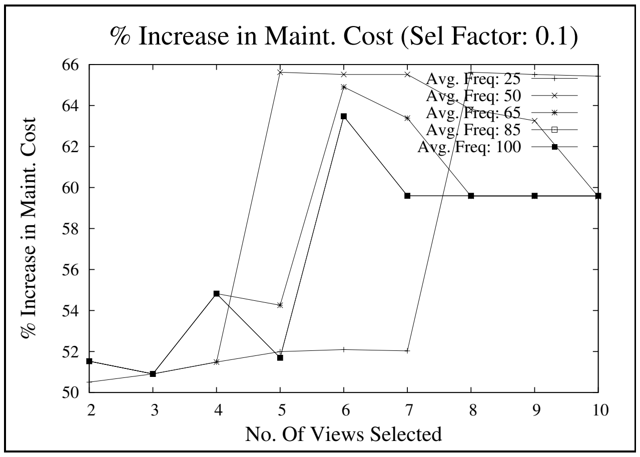

Figure 7: % Increase in Maint. Cost (sel. factor: 0.1)

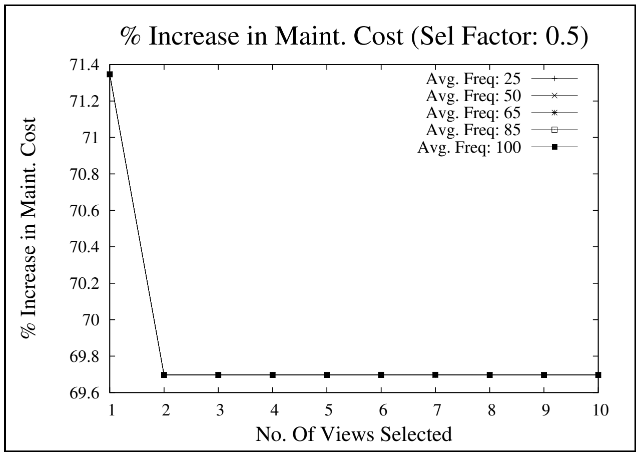

Figure 8: % Increase in Maint. Cost (sel. factor: 0.5)

100. As the number of views selected varies between 2 and 10, the increase in the maintenance cost also increases.

When the selectivity factor is taken to be 0.5, the percentage maintenance cost varies between 69 and 71 as shown in figure 8. When the selectivity factor is taken at 0.9, it varies between 73 and 75 as shown in figure 9.

As the selectivity factor of the query increases, the sizes of the PMPVs also increase and hence the cost of maintaining them also increase. The cost of maintenance of the PMPVs will also depend on the commonality between the expressions that are present in the 'where' clause of read-only queries and the 'where' clause of the update queries.

Thus, the experimental results show that PMPV approach decreases the cost of storing the materialized views and the cost of answering the queries at the expense of increase in the maintenance cost.

6. RELATED WORK

The problem of materialized view selection and maintenance has received wide attention in research community. In this section, we cite the major papers on the materialized views. The list is not exhaustive due to the space limitation.

The algorithms to select the set of materialized views to be maintained based on the workload of the databases are discussed in [13, 15, 39, 36, 3, 34, 2, 8, 40, 23]. [13, 15] select the views using a greedy approach by defining a cost model which assigns benefit to each of the candidate views under space constraint and maintenance cost constraint. [39] discusses a heuristic approach to select the set of candidate views based on the notion of Multiple View Processing Plan (MVPP) and defines a cost model to select the materialized views based on MVPP. [36] discusses selecting the materialized views based on the notion of view relevance wherein the relevance of a view in the presence of already selected set of materialized views is considered. [3, 34] discuss the problem of selecting the materialized views for multidimensional databases. [40] presents a genetic algorithm for selection of materialized views based on multiple processing plans. [2] presents an architecture for automated selection of indexes and materialized views. [8] discusses materialized views for nested Generalized Projection, Selection and Join queries. [23] discusses the problem of view selection by exploiting the common sub expressions between different view maintenance expressions.

The problem of view maintenance is studied in [11, 12, 1,

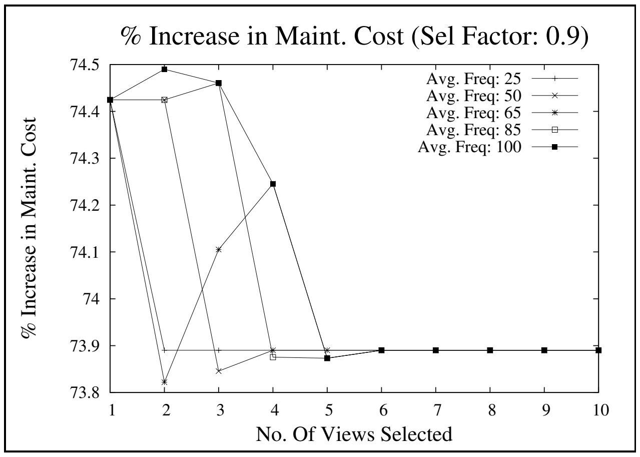

Figure 9: % Increase in Maint. Cost (sel. factor: 0.9)

6, 14, 35, 10, 26, 41, 37, 23, 22, 25]. Self-maintenance of the materialized views, views that can be maintained without accessing the source data, has also been studied well. [29, 19, 32, 20, 24, 22] discuss the problem of self-maintenance.

The work in literature that is closest to the PMPV approach, to the best of our knowledge, are [18] and [5]. In [18], three view materialization strategies are described: fully materialized, fully virtual and partially materialized. The notion of partially materialized means some of the attributes of the views will be materialized while others remain virtual. The paper describes a framework for supporting hybrid materialized/virtual integrated views based on a special class of mediators called Squirrel integration mediators. The PMPV approach is substantially different from the hybrid approach. In PMPV approach, the query workload is analyzed and a view is partitioned such that only those partitions which are accessed by the read-only and the update queries are materialized instead of the whole view.

[5] discusses strategies to fragment the tables of a star schema to reduce the query execution time and facilitate parallel execution of queries. Thus, the problem addressed is the issue of designing a star schema by partitioning its tables and using the partitioned warehouse for efficiently answering the queries. On the other hand, the problem addressed by the PMPV approach is completely different which is partitioning the materialized views of a warehouse so as to reduce the storage cost and the query answering cost. [5] does not considers the workload of the database in terms of the read-only and the update queries whereas the PMPV approach finds the partitions based on the workload.

7. FUTURE WORK

The work presented in this paper on the PMPV approach is only preliminary and there is lot of scope for doing future work in this direction.

In this paper, we assumed that the queries will be simple SPJ queries. Extending the PMPV approach for supporting aggregate queries with group by and having clauses along with the nested queries will be an interesting direction for future work.

In this paper, we assume that for a view V, the PMPVs of V, V^{qu} and V^{q} will be materialized, if their benefit is positive. But, this restriction can be removed and the benefit of V^{qu} and V^{q} can be calculated independently and a decision to materialize either V^{qu} or V^{q} or both can be taken. Also, our approach for selecting the PMPVs is based on the standard view selection algorithm. Although this approach might have the advantage of using the existing view selection algorithms, it may not be the optimal way of selecting the PMPVs. Designing an optimal PMPV selection algorithm is challenging.

In the PMPV approach, presented in the paper, the materialized views are partitioned horizontally, i.e., based on the conditions in the 'where' clause of the queries. The approach can be generalized where a materialized view can be partitioned vertically also using the vertical fragmentation algorithms [27].

Developing a formal framework for supporting PMPVs is an important future work. Based on the workload of the database, the frequencies of the queries and the storage space/maintenance cost constraint, determining the optimal set of PMPVs to be selected is very interesting.

Making the PMPVs self-maintainable by selecting some extra auxiliary views may decrease the cost of maintaining the PMPVs.

8. CONCLUSIONS

Instead of materializing a complete view, materializing only a portion of the view which is accessed by the queries offers many advantages as discussed in the paper. To do this, the view is first partitioned and only some of the partitions are materialized. The PMPV selection algorithms and the method for maintaining PMPVs are naive and more sophisticated methods can be developed. The experimental results show the utility of the PMPV approach in terms of decrease in the storage space and the query processing cost at the expense of increase in the maintenance cost. The maintenance cost of the PMPVs can be brought down by making them self-maintainable which is part of the future work.

Acknowledgments: I thank Kamal Karlapalem for his numerous comments and suggestions during the preparation of the paper. I thank Jhansi Rani Vennam, Ravindranath Jampani, Risi V. Thonangi and Soujanya Vadapalli for proofreading the paper. I thank the anonymous reviewers for their helpful comments.

9. REFERENCES

[1] D. Agrawal, A. E. Abbadi, A. K. Singh, and T. Yurek. Efficient view maintenance at data warehouses. pages 417–427. SIGMOD, 1997.

[2] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in sql databases. pages 496–505. VLDB, 2000.

[3] E. Baralis, S. Paraboschi, and E. Teniente. Materialized view selection in a multidimensional database. pages 156–165. VLDB, 1997.

[4] E. Baralis and J. Widom. An algebraic approach to rule analysis in expert database systems. pages 475–486. VLDB, 1994.

[5] L. Bellatreche, K. Karlapalem, M. K. Mohania, and M. Schneider. What can partitioning do for your data warehouses and data marts? pages 437–446. IDEAS, 2000.

[6] J. Blakeley, P. N. Larson, and F. Tompa. Efficiently updating materialized views. pages 61–71. SIGMOD, 1986.

[7] U. S. Chakravarthy and J. Minker. Processing multiple queries in database systems. volume 5(3), pages 38–44. Database Engineering, 1982.

[8] M. Golfarelli and S. Rizzi. View materialization for nested gpsj queries. page 10. DMDW, 2000.

[9] G. Graefe and W. J. McKenna. Extensibility and search efficiency in the volcano optimizer generator. Technical Report CU-CS-91-563, University of Colorado at Boulder, 1991.

[10] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. pages 328–339. SIGMOD, 1995.

[11] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. volume 18(2), pages 3–18. Data Engineering Bulletin, 1995.

[12] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. pages 157–166. SIGMOD, 1993.

[13] H. Gupta. Selection of views to materialize in a data warehouse. pages 98–112. ICDT, 1997.

[14] H. Gupta and I. S. Mumick. Incremental maintenance of aggregate and outerjoin expressions. Technical report, Stanford University, 1999.

[15] H. Gupta and I. S. Mumick. Selection of views to materialize under a maintenance cost constraint. pages 453–470. ICDT, 1999.

[16] A. Y. Halevy. Answering queries using views: A survey. volume 10(4), pages 270–294. VLDB J., 2001.

[17] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes efficiently. pages 205–216. SIGMOD, 1996.

[18] R. Hull and G. Zhou. A framework for supporting data integration using the materialized and virtual approaches. pages 481–492. SIGMOD, 1996.

[19] N. Huyn. Efficient view self-maintenance. pages 17–25. VIEWS, 1996.

[20] N. Huyn. Multiple-view self-maintenance in data warehousing environments. pages 26–35. VLDB, 1997.

[21] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. pages 95–104. PODS, 1995.

[22] J. Liu, M. W. Vincent, and M. K. Mohania. Maintaining views in object-relational databases. volume 5(1), pages 50–82. Knowl. Inf. Syst., 2003.

[23] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. SIGMOD, 2001.

[24] M. K. Mohania and Y. Kambayashi. Making aggregate views self-maintainable. volume 32(1), pages 87–109. Data Knowl. Eng., 2000.

[25] M. K. Mohania, K. Karlapalem, and Y. Kambayashi. Data warehouse design and maintenance through view normalization. pages 747–750. DEXA, 1999.

[26] M. K. Mohania, S. Konomi, and Y. Kambayashi. Incremental maintenance of materialized views. pages 551–560. DEXA, 1997.

[27] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, second edition, 1999.

[28] R. Pottinger and A. Y. Halevy. Minicon: A scalable algorithm for answering queries using views. volume 10(2-3), pages 182–198. VLDB J., 2001.

[29] D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making views self-maintainable for data warehousing. pages 158–169. PDIS, 1996.

[30] P. Roy. *Multi-Query Optimization and Applications.* PhD thesis, 2000.

[31] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. pages 249–260. SIGMOD, 2000.

[32] S. Samtani, V. Kumar, and M. K. Mohania. Self maintenance of multiple views in data warehousing. pages 292–299. CIKM, 1999.

[33] T. K. Sellis. Multiple query optimization. volume 13(1), pages 23–52. ACM TODS, 1988.

[34] A. Shukla, P. Deshpande, and J. Naughton. Materialized view selection for multidimensional datasets. pages 488–499. VLDB, 1998.

[35] M. Staudt and M. Jarke. Incremental maintenance of externally materialized views. pages 75–86. VLDB, 1996.

[36] S. R. Valluri, S. Vadapalli, and K. Karlapalem. View relevance driven materialized view selection in data warehousing environment. Australasian Database Conference, 2002.

[37] D. Vista. *Optimizing Incremental View Maintenance Expressions in Relational Databases.* PhD thesis, 1997.

[38] J. Widom. Research problems in data warehousing. pages 25–30. CIKM, 1995.

[39] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in data warehousing environment. pages 136–145. VLDB, 1997.

[40] C. Zhang, X. Yao, and J. Yang. An evolutionary approach to materialized views selection in a data warehouse environment. volume 31(3), pages 282–294. IEEE Transactions on Systems, Man, and Cybernetics, Part C, 2001.

[41] Y. Zhuge, H. G. Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. pages 316–327. SIGMOD, 1995.