Automated Generation of Materialized Views in Oracle

Rafi Ahmed
Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065, U.S.A.
rafi.ahmed@oracle.com

Randall Bello
Oracle Corporation
2300 Cloud Way
Austin, TX 78741, U.S.A.
randall.bello@oracle.com

Andrew Witkowski
Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065, U.S.A.
andrew.witkowski@oracle.com

Praveen Kumar
Oracle Corporation
1 Oracle Drive
Nashua, NH 03062, U.S.A.
praveen.kumar@oracle.com

ABSTRACT

Automated generation of a right set of materialized views is a challenging task. It is a highly desirable feature for autonomous databases. The selection of materialized views must be based on cost and verifiable in the actual database environment. This paper describes an automated system that generates, selects, verifies, and maintains materialized views in the Oracle RDBMS; it presents a novel technique, called the extended covering sub-expression algorithm, for the automated generation of materialized views. An extensive set of experiments is described that demonstrates the feasibility and efficiency of this approach. This system has been fully implemented and is going to be deployed on the Oracle Autonomous Database on the Cloud.

PVLDB Reference Format:
Rafi Ahmed, Randall Bello, Andrew Witkowski, Praveen Kumar. Automated Generation of Materialized Views in Oracle. *PVLDB*, 13(12): 3046-3058, 2020.
DOI: https://doi.org/10.14778/3415478.3415533

1. INTRODUCTION

Current relational database systems process complex SQL queries involving multiple fact and dimension tables and containing several nested sub-query blocks. Such queries are becoming increasingly important in Decision-Support Systems (DSS). Generating optimal execution plans for such queries has become critical for commercial database systems. Materialized view rewrite [8, 14] is a well-known technique that is used for optimizing such queries. The richness of structure of materialized views makes the task of selecting the right set of materialized views generally a daunting task for the DBA.

Automated generation of materialized views, on the other hand, poses a variety of challenges [1, 17, 29] of its own. We could consider generating every syntactically relevant materialized view for workload queries based on all possible subsets of tables in workload queries, but it would explode the search space even with some heuristic-based pruning; arbitrary table subsets may introduce Cartesian Products in the materialized view definitions. At the other extreme, we could generate for each query, when syntactically possible, one candidate materialized view that exactly matches the text of the query; this would generally violate the storage constraints and result in making materialized view refresh an intractable task. The idealized objective is to generate a small number of materialized views, which are of reasonable size, contain large pre-computations of joins and grouping, and can rewrite a substantial number of current and future workload queries. This presents conflicting demands on the system. A materialized view that contains large pre-computations is more beneficial to the queries it rewrites, but it would normally rewrite fewer queries. Further, a materialized view that rewrites many queries tends to have large sets of grouping columns and very few or no selection predicates, which tend to increase materialized view size in terms of the number of rows it contains.

In this paper, we discuss a novel technique, called the extended covering sub-expression (ECSE) algorithm, for automated generation of materialized views. The ECSE algorithm seeks to achieve a compromise between the conflicting demands and to find a balance between the two extremes.

Automated materialized view project is a crucial component of a wider effort called Oracle Autonomous Databases. Other components of this project include task management, ML-based automatic refresh of materialized views, etc.

The rest of the paper is organized as follows. We first give an overview of the system architecture for the automated generation of materialized views in Section 2. We describe the basic concepts, the ECSE algorithm, and the technique for cost-based selection of materialized views in Section 3. Section 4 describes the verification module. Materialized view maintenance is outlined in Section 5. In Section 6, we describe an extensive set of experiments we performed on several customer workloads. Finally, in Section 7 we survey the related work and provide our conclusion in Section 8.

Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415533

2. ARCHITECTURE FOR AUTOMATED MATERIALIZED VIEWS

In this paper, we focus on a class of single query-block materialized views that contain join of multiple tables, grouping, aggregation, and – in rare cases – filter (i.e., selection) predicates; these materialized views are based on queries containing one or more query blocks (a query block contains SELECT, FROM, WHERE, and, optionally, GROUP-BY clauses), which have filter predicates, join of multiple tables, grouping, and aggregation. The workload may contain arbitrarily complex SQL statements.

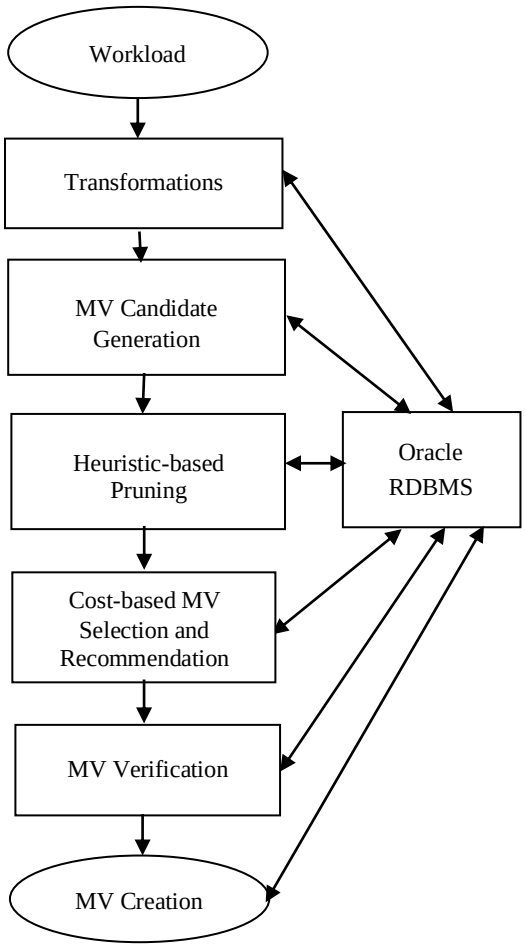

Figure 1. Architecture for Automated Materialized Views

Figure 1 presents an architectural overview of automated generation of materialized views (MVs) in Oracle. Our starting point is a workload containing a set of queries, for which a set of materialized views capable of rewriting a substantial number of current and future queries are to be generated. The key components of the architecture are: (i) query transformations, such as simple subquery unnesting and select-project-join view merging, to reduce the number of query blocks in a query (Oracle RDBMS performs complex transformations [4] in a cost-based manner after materialized view generation and rewrite modules have been invoked), (ii) candidate materialized view generation using the ECSE algorithm, (iii) heuristic-based pruning of candidate materialized views, (iv) enumeration of mapping between queries and materialized view groups and cost-based recommendation of materialized views, (v) verification of recommended materialized views by executing relevant workload queries with and without materialized view rewrite, and (vi) creation of materialized views that pass verification; the partitioning scheme, if any, of the underlying fact table is used to partition the materialized views.

3. MATERIALIZED VIEW SELECTION

In the following, we highlight the distinguishing aspects of our approach.

We consider queries containing multiple query blocks. Each query block may be based on a star, snowflake, or snowstorm schema [3]. The materialized views generated by our system may require joins with other tables [6] for rewrite; this strategy is more versatile than the view-lattice approach [14, 28], which assumes that all workload queries have the same join pattern. In our scheme, a materialized view is generally anchored on a large fact table and can rewrite multiple queries; it can contain pre-computations (i.e., joins and grouping) that do not appear in the queries eligible for rewrite, as it may be based on referential integrity constraints.

The ECSE algorithm, using pair-wise comparisons, considers *all* possible relationships – equivalence, superset, subset, intersection, and union – that can exist among join graphs of given queries and exploits invariant join property, when applicable, for extracting covering sub-expressions, which are then used to generate candidate materialized views. The novel ECSE strategy is more efficient than generating arbitrary subsets or subplans of every query [1, 14, 19], and it is different from the reported works on sub-expression selection, multi-query optimization, and materialized view selection.

Since the problem of the selection of materialized views or indexes has been shown to be NP-Hard [11, 26], it is critical that the proposed solutions recommend high quality materialized views in a scalable manner. Under the worst-case scenario, the time complexity of the ECSE algorithm (Section 3.4) is $O(N^2)$, as the ECSE algorithm does pair-wise comparisons of join graphs in a given workload, which contains N query blocks. In order to further restrict the search space, we apply several heuristics within and after the ECSE algorithm (Sections 3.3 and 3.4).

3.1 Basic Concepts

Join Graph. The join graph of a query block may have undirected edges resulting from inner or full outer joins and directed edges resulting from left outer, anti, and semi joins. In our scheme, a join graph is considered *connected*, if there exists at least one vertex from which all other vertices are reachable by traversing directed and undirected edges. For example, join graph {T1 ─ T2, T2 → T3} is connected but {T1 → T3, T2 → T3} is not. We consider only connected join graphs, as they do not produce Cartesian product. In our scheme, a join graph may have cycles; a join graph that has a cycle containing *only* directed edges is considered illegal in Oracle.

Classification of join graphs of a given query workload is an important aspect of our candidate generation algorithm. We examine the shape of each join graph, table cardinalities, and number of distinct values (NDVs) of joining columns and identify

fact, dimension, and branch tables (i.e., the tables that join with dimension tables in snowflake and snowstorm schemas [3]). We then divide the join graphs into classes, where join graphs in each class reference a single common fact table. Each class of join graphs is considered separately by the ECSE algorithm for generating candidate materialized views, which contain the common fact table and various dimension and branch tables. The ECSE algorithm, however, will work correctly even if fact, dimension, and branch tables cannot be identified and thus the join graphs cannot be divided into classes.

Join set is an abstraction of a connected join graph that is based on a query block. A *join set*, which is essentially a set of join edges, allows us to apply set operations on the underlying join graph. A join set has an associated field called *QB set*, which represents a set of query blocks (QBs) that can be potentially re-written in terms of a materialized view based on the join set. For the sake of brevity, in this paper a join set is represented as a set of *simplified* join edges that do not show columns or relational operator: e.g., {F1 – D1, D1 – B1}, where F1 – D1 stands for a join edge between tables F1 and D1 and D1 – B1 stands for a join edge between tables D1 and B1.

Set operations (i.e., equivalence, subset, superset, union, and intersection) are performed on join sets at various steps of the ECSE algorithm. Two join edges, which may originate from two different query blocks, are considered equivalent, if they are defined in terms of the same pair of Table.Column, relational operator (e.g., =, >, ≤, etc.) and join type (e.g., inner, outer, anti, etc.). Two join sets are considered *equivalent*, if they contain equivalent sets of join edges. The standard definitions of the set operations subset, superset, union, and intersection on join sets follow directly from the definition of equivalence of join edges. The join sets that result from these operations must be connected.

Invariant joins can be derived from table and join properties. A table T1 is invariant in a join with table T2, if the following five conditions are satisfied: (i) The join is specified by a simple equality, inner join predicate T1.fk = T2.pk; (ii) there is a referential integrity constraint from T1.fk to T2.pk; that is, T1.fk is a foreign key that references the primary key T2.pk; (iii) the column T1.fk has a non-null constraint; (iv) T2 does not have any filter or subquery predicates; (v) T2 is invariant in joins with tables other than T1 (if any); e.g., dimensions that join with branch tables in snowflake schema. The conditions (iv) and (v) can be circumvented in materialized view creation by excluding from the materialized view definitions filter predicates and tables that violate the invariance property. The invariance of a table or join set J with respect to its join with table T is denoted by *Invariant (J, T)*, which implies that table T joins with table(s) in J without affecting the resulting rows of J. The presence of invariant joins is used to identify a join set that is a union or superset of underlying join sets thereby allowing materialized views to contain larger pre-computations; i.e., a query block can be rewritten with a materialized view that has more tables and joins than the query block.

Partition of Join Set. In a snowstorm schema [3], a join set may contain multiple fact tables, where each fact table has its own dimension and branch tables. Consider an example where there are two fact tables, F1 and F2, in a join set; JS1: {B1 – D1, D1 – F1, F1 – F2, F2 – D2}. In such a case, JS1 will be partitioned into two join sets JS2 and JS3, each containing a single fact table and its dimension and branch tables; JS2: {F1 – D1, D1 – B1}; JS3: {F2 – D2}. The join sets JS2 and JS3 inherit the QB set of JS1.

Reduction of Join Set. A join set may contain anti-joined or semi-joined tables, which result from subquery unnesting [4]. However, materialized views in Oracle may not contain anti-joined or semi-joined tables. A join set is, therefore, reduced by removing tables that are anti-/semi-joined.

Filter Predicates. Most workload queries are repeatedly issued over time where they differ only in the constant values of filter predicates. Therefore, most materialized view definitions in our system do not include filter predicates. This allows them to rewrite current as well as future queries with the same signature.

Left Outer-Join. Oracle allows left outer-join in materialized view definitions. Consider a query: SELECT * FROM T1 left outer join T2 on T1.x = T2.y and T2.z = 5; in case of a *many-to-many* join between T1 and T2, rows of both tables can be duplicated. In our scheme, a materialized view definition with left outer-join contains an indicator column, which has value 1 indicating inner-joined (i.e., matching) rows and 0 indicating anti-joined (i.e., non-matching) rows. The rewrite of the above query, which contains a filter predicate on the outer-joined table T2, with a materialized view containing the left outer-join between T1 and T2 and no filter predicate is non-trivial, if the materialized view must appear only once in the rewrite. We use a technique that involves the LEAD window function and the indicator column to rewrite this type of queries by referencing the materialized view only once. The details of the technique are beyond the scope of this paper.

Scope of Materialized Views. We support nested subqueries, views, standard aggregate functions in materialized view definitions, and distinct aggregates based on a bitmap technique.

3.2 Operations on Join Sets

In this section, we present functions for five basic set operations, which identify or create join sets for defining materialized views. We use a list of items, referred to here as *JQLST*, for each class of join graphs (Section 3.1). Every item in JQLST contains a join set and its associated QB set. Initially, the join set is based on a single query block; the QB set is initialized to the query block where the join set originates. We define a function *Tables ()* that takes a join set and returns a set of tables that appear in the join set.

Every set operation involves pair-wise comparisons of items from the list, where a union operation on QB sets shows the join set that can rewrite all query blocks belonging to both the operands. After the first step of the ECSE algorithm, a join set remains static, whereas its QB set may dynamically grow.

Each set operation is illustrated by two single-block queries and a materialized view definition derived from the two queries. In the examples, every query is followed by the join set and QB set it produces, whereas every materialized view definition is preceded by join set and QB set, which are derived from the join and QB sets of the two query blocks by performing one of the set operations. We use the common notations () for a list, {} for a set, and [] for a structure; the join sets (i.e., join graphs) and QB sets are shown in bold.

The derived join and QB sets are used to generate materialized view definitions. A derived join set is used to form the FROM and WHERE clauses of the materialized view definition. The

SELECT and GROUP-BY lists of the query blocks in the derived QB set are merged to construct its SELECT and GROUP-BY lists. The columns in the filter predicates of the query blocks in the derived QB sets are added to the GROUP-BY and SELECT lists to enable materialized views rewrite of query blocks with similar signatures. The joining columns used in join predicates with tables not included in the materialized view are also added to its GROUP-BY and SELECT lists.

This materialized view definition can rewrite all query blocks in the derived QB set; such query blocks are considered *eligible* for the materialized view, and *vice versa*. In most of the following examples, materialized view rewrite would involve re-computation of grouping and aggregation.

3.2.1 Equivalence

If the join sets of two items are found to be equivalent, one of them is removed and the QB set of the retained join set is augmented by the QB set of the removed item indicating that the join set of the retained item can be used to rewrite all the query blocks in its QB set.

```
Function JS-Equivalence (JQLST)
{
  // Prune join sets based on equivalence
  For each item X in JQLST do
     For each item Y in JQLST do
        If (X != Y ∧ X.joinset = Y.joinset)
        {
           X.qbset = X.qbset ∪ Y.qbset;
           Remove Y from JQLST;
        }
}
```

Figure 2. JS-Equivalence

Consider two query blocks Q1 and Q2, whose join sets are equivalent; and therefore, one of the join sets can be discarded.

```
SELECT F.n, F.g, SUM(F.m1), COUNT(F.m3), D2.z, D7.y
FROM  F, D7, D2
WHERE F.f7 = D7.k and F.f2 = D2.k and
        F.x IN (4,6) and D7.c =  25
GROUP BY F.n, F.g, D7.y, D2.z;
[{F — D7, F — D2}, {Q1}]

SELECT F.n, MAX(F.m2), D7.p, D2.y
FROM F, D7, D2
WHERE F.f7 = D7.k and F.f2 = D2.k and
        F.x = 9 and D7.c =  5
GROUP BY F.n, D7.p, D2.y;
[{F — D7, F — D2}, {Q2}]

[{F — D7, F —  D2}, {Q1, Q2}]
Create materialized view MV0 AS
SELECT  F.n, F.g, D7.y, D2.z, D7.p, D2.y, F.x, D7.c,
         MAX(F.m2), SUM(F.m1), COUNT(F.m3)
FROM F, D7, D2
WHERE F.f7 = D7.k and F.f2 = D2.k and
        F.x IN (4, 6, 9) and D7.c IN (5, 25)
GROUP BY F.n, F.g, D7.y, D2.z, D7.p,  D2.y, F.x, D7.c;
```

In materialized view MV0, merging of SELECT and GROUP-BY lists have taken place. For the purpose of illustration only, we also show, in the definition of MV0, a unification of filter predicates that originate from Q1 and Q2.

3.2.2 Subset

```
Function JS-Subset (X, Y)
{
  // Identify join set based on subset.
  If (X.joinset ⊂ Y.joinset)
    X.qbset = X.qbset ∪ Y.qbset;
}
```

Figure 3. JS-Subset

Consider two query blocks Q3 and Q4. The join set of Q4 is a subset of that of Q3.

```
SELECT F.x, D1.y, D2.z, SUM(F.m1)
FROM  F, D1, D2, B2
WHERE  F.f1 = D1.k and F.f2 = D2.k and D2.c = B2.r and
        F.y = 5 and D1.c =  9 and D2.s < 25
GROUP BY F.x, D1.y, D2.z;
[{F — D1, F — D2, D2 — B2}, {Q3}]

SELECT  F.x, D1.h, COUNT(F.m2),
FROM  F, D1
WHERE F.f1 = D1.k and F.y = 7 and D1.g = 7 and D1.c = 9
GROUP BY F.x, D1.h;
[{F — D1}, {Q4}]

[{F — D1}, {Q3, Q4}]
Create materialized view MV1 AS
SELECT  F.x, D1.y, D1.h, D1.c, D1.g, F.y,
         F.f2, COUNT(F.m2) , SUM(F.m1)
FROM F, D1
WHERE F.f1 = D1.k
GROUP BY F.x, F.y, D1.y, D1.h, D1.c, D1.g, F.f2;
```

In MV1, merging of the SELECT and GROUP-BY have taken place. The SELECT and GROUP-BY have been augmented with the columns in filter predicates and with F.f2, the column used in join with D2. This enables the rewrites with MV1 of Q4 without requiring any join and of Q3 by joining MV1 with D2.

3.2.3 Intersection

```
Function JS-Intersection (JQLST)
{
   // Generate a new join set based on intersection.
   For each item X in JQLST do
      For each item Y in JQLST do
        If (X != Y ∧ Y.joinset ⊈ X.joinset ∧
           X.joinset ⊈ Y.joinset ∧
           X.joinset ∩ Y.joinset ≠ ∅)
        {
          Z.joinset = X.joinset ∩ Y.joinset;
          Z.qbset = X.qbset ∪ Y.qbset;
          Insert Z into NLST;
        }
   Append NLST to JQLST;
}
```

Figure 4. JS-Intersect

The function JS-Intersection in Figure 4 generates a new join set from the given join sets. In order to maximize computations

contained in derived join sets, we do not generate intersection closure (i.e., derivation of intersection join sets from other intersection join sets). Consider queries Q5 and Q6, whose join sets overlap, and therefore intersection can be applied to them.

```
SELECT  F.n, MIN(F.m1), D7.y, D2.z
FROM  F, D7, D2
WHERE  F.f7 = D7.k and F.f2 = D2.k and
          F.x IN (4,6) and D7.c =  25
GROUP BY  F.n, D7.y, D2.z;
```

[{F — D7, F — D2}, {Q5}]

```
SELECT F.y, SUM(F.m2), D7.h, D3.x
FROM F, D7, D3
WHERE F.f7 = D7.k and F.f3 = D3.k and D7.y = 5
         F.x = 11 and D3.w > 15
GROUP BY F.y, D7.h, D3.x;
```

[{F — D7, F — D3}, {Q6}]

[{F — D7}, {Q5, Q6}]

```
Create materialized view MV2 AS
SELECT F.n, F.y, D7.y, D7.h, D7.c, F.x, F.f2, F.f3,
          MIN(F.m1) mn, SUM(F.m2) sm
FROM F, D7
WHERE F.f7 = D7.k
GROUP BY F.n, F.y, F.x, D7.y, D7.h, D7.c, F.f2, F.f3;
```

Here a new join set is generated. In MV2, merging of SELECT and GROUP-BY lists have taken place; the SELECT and GROUP-BY lists have also been augmented with the joining columns of tables not included in the materialized view MV2. Note that the rewrite of Q5 will require a join with D2 using F.f2 and the rewrite of Q6 will require a join with D3 using F.f3. In the following, we show query Q6, which has been rewritten with the materialized view MV2.

```
SELECT M.y, M.h, D3.x, SUM(M.sm) sm
FROM  MV2  M,  D3
WHERE  M.f3 = D3.k and M.x = 11 and D3.w > 15 and
          M.y = 5
GROUP BY M.y, M.h, D3.x;
```

3.2.4 Superset

Figure 5 presents the derivation of a superset join set. Superset join sets can be derived only if the invariance property is satisfied for relevant joins. The condition in Figures 5 checks if the join set of Y is invariant in join with *all* tables in the difference of join sets of X and Y.

Consider two query blocks Q7 and Q8. The join set of Q7 is a subset of that of Q8; the superset operation can be applied to them, if the relevant join is invariant.

```
SELECT F.n, SUM(F.m1), D1.m
FROM F, D1
WHERE F.f1 = D1.k and F.x = 6 and D1.y = 25
GROUP BY F.n, D1.m;
```

[{F — D1}, {Q7}]

```
SELECT F.y, MIN(F.m2), D1.h, D5.z
FROM F, D1, D5
WHERE F.f1 = D1.k and F.fk5 = D5.pk and
          F.x = 11 and D1.y = 33 and D5.g > 6
GROUP BY F.y, D1.h, D5.z;
```

[{F — D1, F — D5}, {Q8}]

[{F — D1, F — D5}, {Q7, Q8}]

```
Create materialized view MV3 AS
SELECT  F.n F.y, D1.m, D1.h, D5.z, D1.y,
           D5.g, F.x, MIN(F.m2), SUM(F.m1)
FROM F, D1, D5
WHERE F.f1 = D1.k and F.fk5 = D5.pk
GROUP BY F.n, F.y, F.x, D1.y, D1.m, D1.h, D5.z, D5.g;
```

```
Function JS-Superset (X, Y)
{
  // Identify invariance-based superset join set.
   If (Y.joinset ⊂ X.joinset ∧
      ∀ T ∈ Tables (X.joinset – Y.joinset),
      Invariant (Y.joinset, T))
   {
       X.qbset = X.qbset ∪ Y.qbset;
       Return True;
   }
   Else
       Return False;
}
```

Figure 5. JS-Superset

In Q8, F.fk5 must be a non-null foreign key (F.K.) that references primary key (P.K.) D5.pk, which indicates that the join between F and D5 is invariant (Section 3.1). Augmentation of SELECT and GROUP-BY with joining columns is not required. No join is needed to rewrite either Q7 or Q8 with materialized view MV3.

3.2.5 Union

Figure 6 presents the derivation of a new join set based on the union operation, which applies to overlapping join sets X and Y. The invariance condition is checked for *all* join edges that are either in X or Y but not in both. A union join set can be derived only if the invariance property is satisfied for relevant joins.

For the sake of brevity, in Figure 6 we do not consider the case where two join sets have only a single (fact) table in common and no common join edges.

```
Function JS-Union (JQLST)
{
  // Generate invariance-based union join sets.
  For each item X in JQLST do
     For each item Y in JQLST do
       If (X != Y ∧ Y.joinset ⊄ X.joinset ∧
          X.joinset ⊄ Y.joinset ∧
          X.joinset ∩ Y.joinset ≠ ∅ ∧
          ∀ T ∈ Tables ((X.joinset ∪ Y.joinset) –
                        (X.joinset ∩ Y.joinset)),
          Invariant (X.joinset ∩ Y.joinset, T))
       {
          Z.joinset = X.joinset ∪ Y.joinset;
          Z.qbset = X.qbset ∪ Y.qbset;
          Insert Z into JQLST;
       }
}
```

Figure 6. JS-Union

A materialized view produced by the successive application of the union operation may ultimately be not very useful, as it tends to have a large group-by list and thus high cardinality. Therefore, the operand (parent) join sets are also retained by the ECSE algorithm in JS-Union (Figure 6) along with resulting (child) join sets. Both parent and child join sets compete in the final selection of recommended materialized views (Section 3.5).

Consider two query blocks Q9 and Q10, whose join sets overlap; the union operation can be applied, if the relevant joins are invariant.

```
SELECT F.n, D1.m, D5.x, SUM(F.m1),
FROM F, D1, D5
WHERE F.fk1 = D1.pk and F.fk5 = D5.pk and F.x = 6
        and D1.z =  25
GROUP BY F.n, D5.x, D1.m;
[{F — D1, F — D5}, {Q9}]

SELECT  F.y, D2.w, D5.z, AVG(F.m2)
FROM  F, D2, D5
WHERE  F.fk2 = D2.pk and F.x = 12 and  D2.g > 7 and
          F.fk5 = D5.pk
GROUP BY F.y,  D5.z, D2.w;
[{F — D2, F — D5}, {Q10}]

[{F — D1, F — D2, F — D5}, {Q9, Q10}]
Create materialized view MV4 AS
SELECT  F.n, F.y, D1.m, D2.w, D1.z, D2.g, F.x, D5.x, D5.z,
          SUM(F.m2), COUNT(F.m2), SUM(F.m1)
FROM F, D1, D2
WHERE F.fk1 = D1.pk and F.fk2 = D2.pk and F.fk5 = D5.pk
GROUP BY F.n, F.y, F.x, D1.m, D2.w, D1.z, D2.g, D5.x,
            D5.z;
```

In Q9, F.fk1 must be a non-null foreign key (F.K.) that references primary key (P.K.) D1.pk. In Q10, F.fk2 must be a non-null F.K. that references P.K. D2.pk. A new join set is generated here. The filter predicates on the tables D1 and D2 cannot be included in the definition of materialized view MV4. Augmentation of SELECT and GROUP-BY lists with joining columns is not required, since no join is needed to rewrite either Q9 or Q10 with MV4. The rewrite module will validate the invariance property before query rewrite with MV4.

3.3 Heuristics for Pruning Join Sets

In this section, we describe five heuristics for pruning join sets based on (A) join set reduction, (B) join set size, (C) QB set size, (D) maximal join and QB sets, and (E) the cardinality ratio of a candidate materialized view and the fact table it references.

These heuristics use four threshold values α, β, λ, and ρ, which are configurable**.** Precise determination of the threshold values depends upon many factors, such as the count and complexity of queries in the workload, cardinalities of fact tables, storage requirement, etc. Automated derivation of these threshold values for a given workload requires further research.

3.3.1 Heuristic A: Join Set Reduction

To prevent an explosion of materialized view size, it may become necessary to reduce a join set by removing a dimension or branch table that causes a many-to-many join.

We identify a many-to-many equi-join by examining its join predicates to determine if neither of its operand columns has a unique constraint/index or number of distinct values (NDVs) close to the cardinality of the table. A join set is reduced by removing the table that causes a many-to-many join; all branch tables of the removed (dimension or branch) table are also recursively removed from the join set.

3.3.2 Heuristic B: Join Set Size

A join set is pruned, if the number of tables it contains is below a given threshold value α, which may be computed as half of the average number of tables in all query blocks in all queries of a given workload. In our experiments (Section 6), we set the value of α to 2.

This heuristic ensures that only those materialized views are recommended which have a certain number of join computations.

3.3.3 Heuristic C: QB Set Size

A join set is pruned, if the cardinality of its QB set is below a threshold value, β (e.g., 2). This ensures that only those materialized views are recommended which can rewrite at least β query blocks. In our experiments (Section 6), the value of β is set to 2; that is, we always prune a join set if it can rewrite only a single query block.

3.3.4 Heuristic D: Maximal Join and QB Sets

Prune a join set J_k, if there exists a maximal join set J_i. J_i is considered maximal in relation to J_k, if J_k is a subset (not necessarily proper) of J_i and J_k's QB set is a subset (not necessarily proper) of J_i's QB set.

```
For each item X in JQLST do
  For each item Y in JQLST do
    If (X != Y ∧ Y.joinset ⊆ X.joinset ∧ Y.qbset ⊆ X.qbset)
      Remove Y;
```

Here, a join set is pruned, if there exists another join set that contains larger pre-computations and it can rewrite more query blocks.

3.3.5 Heuristic E: Cardinality Ratio

In this scheme, every join set – or the materialized view based on the join set – contains one fact table and one or more dimension and branch tables. Experience has shown that in most cases if the cardinality of a materialized view is not significantly smaller than that of the fact table, then query rewrite based on the materialized view does not prove to be beneficial. Since in our scheme, materialized view definitions rarely contain any filter predicates, pruning out materialized view candidates based on their cardinality is crucial.

We define *cardinality ratio* as the number of rows of the materialized view's fact table divided by the number of rows of the materialized view. We prune a materialized view, if its cardinality ratio is smaller than a given threshold value, λ (e.g., 3).

The accurate cardinalities of fact tables are found in the database dictionary tables. We do not use optimizer estimates of materialized view's cardinalities, as cardinality estimation error in query optimizers remains a pervasive and persistent problem [16, 23, 24]. Instead, we issue a query based on the materialized view

definition using a sample block clause to estimate the number of rows of a materialized view; alternative methods include approximate count distinct and the method described in [9]. For a given percentage, ρ, of block sampling, the linearly scaled cardinality is obtained by multiplying the number of rows returned by the query with 100/ρ. Block sampling with a small percentage gives a reasonably accurate estimate, since these materialized view definitions contain no filter predicates and all joins are many-to-one (Section 3.3.1).

Consider, for example, the following materialized view definition for the TPC-DS schema.

```
Create materialized view MV10 AS
SELECT hd_vehicle_count, hd_dep_count,
          s_store_name, t_minute, t_hour,
          count (*), sum (ss_ext_sales_price)
FROM store_sales, household_demographics, store,
        time_dim
WHERE hd_demo_sk = ss_hdemo_sk and
          s_store_sk = ss_store_sk and
          t_time_sk = ss_sold_time_sk
GROUP BY hd_vehicle_count, hd_dep_count,
             t_minute, t_hour;
```

The following shows a query that returns the cardinality of the materialized view MV10 using 1% block sampling of the fact table, stores_sales.

```
SELECT count (*)
FROM (SELECT 1
        FROM  store_sales SAMPLE BLOCK (1),
                household_demographics, store, time_dim
        WHERE hd_demo_sk = ss_hdemo_sk and
                  s_store_sk = ss_store_sk and
                  t_time_sk = ss_sold_time_sk
        GROUP BY hd_vehicle_count, hd_dep_count,
                     s_store_name, t_minute, t_hour);
```

The estimation of cardinalities of candidate materialized views serves a dual purpose. First, it is used for pruning out unpromising materialized views. Second, the sampled cardinalities are injected into the database dictionary tables replacing optimizer estimated cardinalities; this enables the optimizer cost model to use more accurate cardinalities thereby significantly improving the count and quality of recommended materialized views (Section 3.5).

3.4 The Extended Covering Sub-Expression (ECSE) Algorithm

We describe the ECSE algorithm in Figure 7. As can be seen, the algorithm has in-built heuristics to weed out unpromising MV candidates.

The input to automated materialized view candidate generation is the SQL Tuning Set [15], which is an Oracle tool for storing, managing, and tuning workloads. A SQL Tuning Set persistently stores the texts of SQL statements, their SQL-IDs, execution plans, optimizer estimated costs, execution statistics (e.g., CPU time, elapsed time, buffer-gets, rows processed), execution contexts, etc.

As described in Section 3.1, we analyze the join graph of each query block in the given workload and classify the join graphs based on their fact tables. The ECSE algorithm is invoked for each class of join sets. The input to this algorithm is a list of items containing join sets and QB sets.

```
Algorithm ECSE (JQLST)
{
   // Prune join sets based on equivalence
   JS-Equivalence (JQLST);

   // Generate intersection join sets.
   JS-Intersection (JQLST);

   // Generate invariance-based union join sets.
   JS-Union(JQLST);

   // Prune join sets based on equivalence.
   JS-Equivalence (JQLST);

   For each item X in JQLST do
      For each item Y in JQLST do
      {
         // Identify invariance-based superset.
         Valid := JS-Superset (X, Y));

         // Identify join sets based on subset.
         If (! Valid)
            JS-Subset (X, Y);
      }
   // Apply the heuristics in the given order.
   Prune the join sets based on heuristics A, B, C,
   D, and E;

   Form candidate materialized view definitions
   based on JQLST;
```

Figure 7. Algorithm ECSE

3.4.1 ECSE Example

We present a simple example that demonstrates the workings of the ECSE algorithm. Consider a workload of 8 single-block SQL statements: Q1, Q2, Q3, Q4, Q5, Q6, Q7, and Q8. In this example, there are no referential integrity constraints and thus no invariant joins. As every query block here references a single fact table, the partitioning of join sets is not required at Step 2 below.

1. Generate join graphs and identify fact, dimension, and branch tables
2. Divide join graphs into classes (Section 3.1) based on fact table: {Q2, Q3, Q5, Q6, Q8}, which references fact table F1, and {Q1, Q4, Q7}, which references fact table F3.
3. Start with the initial JQLST for the class containing fact table F1.

   JQLST: ([{F1 – D4}, {Q2}],
   [{F1 – D4, F1 – D3}, {Q3}],
   [{F1 – D1, F1 – D5, F1 – D6}, {Q5}],
   [{F1 – D1, F1 – D6, F1 – D7}, {Q6}],
   [{F1 – D1, F1 – D5, F1 – D6}, {Q8}])

   3.1 Apply JS-Equivalence.

   [{F1 – D1, F1 – D5, F1 – D6}, {Q5, Q8}] ←
   [{F1 – D1, F1 – D5, F1 – D6}, {Q5}],
   [{F1 – D1, F1 – D5, F1 – D6}, {Q8}]

   JQLST: ([{F1 – D4}, {Q2}],

[{F1 – D4, F1 – D3}, {Q3}],
[{F1 – D1, F1 – D5, F1 – D6}, {Q5, Q8}],
[{F1 – D1, F1 – D6, F1 – D7}, {Q6}])

3.2 Apply JS-Intersection.

[{F1 – D1, F1 – D6}, {Q5, Q6, Q8}] ←
[{F1 – D1, F1 – D5, F1 – D6}, {Q5, Q8}],
[{F1 – D1, F1 – D6, F1 – D7}, {Q6}]

JQLST: ([{F1 – D4}, {Q2}],
[{F1 – D4, F1 – D3}, {Q3}],
[{F1 – D1, F1 – D5, F1 – D6}, {Q5, Q8}],
[{F1 – D1, F1 – D6, F1 – D7}, {Q6}],
[{F1 – D1, F1 – D6}, {Q5, Q6, Q8}])

3.3 JS-Equivalence, JS-Superset, and JS-Union are not applicable here.

3.4 Apply JS-Subset.

[{F1 – D4}, {Q2, Q3}] ←
[{F1 – D4}, {Q2}],
[{F1 – D4, F1 – D3}, {Q3}]

JQLST: ([{F1 – D4}, {Q2, Q3}],
[{F1 – D4, F1 – D3}, {Q3}],
[{F1 – D1, F1 – D5, F1–D6}, {Q5, Q8}],
[{F1 – D1, F1 – D6, F1 – D7}, {Q6}],
[{F1 – D1, F1 – D6}, {Q5, Q6, Q8}])

3.5 Apply heuristics A, B, C, D, and E, where $\alpha = 2$, $\beta = 2$, $\lambda = 2$ and $\rho = 10$.

JQLST: ([{F1 – D1, F1 – D6}, {Q5, Q6, Q8}],
[{F1 – D1, F1 – D5, F1 – D6}, {Q5, Q8}],
[{F1 – D4}, {Q2, Q3}])

Note that for the class containing fact table F3 steps similar to the above take place.

3.5 Cost-Based Recommendation

In this section, we first discuss some basic concepts for the algorithm used for the cost-based selection and recommendation of materialized views.

Figure 8 shows an example of 6 queries and 5 materialized view candidates. A query here is followed by the query blocks it contains and the set of materialized views that can *individually* rewrite the query block.

Q1. QB11 {MV1, MV2}, QB12 {MV3}
Q2. QB2 {MV2, MV4}
Q3. QB3 {MV5, MV4}
Q4. QB4 {MV3, MV5}
Q5. QB5 {MV2, MV4}
Q6. QB6 {MV5, MV1}

Figure 8. Eligible Queries and MVs

The relationships between candidate materialized views and eligible query blocks can be many-to-many; i.e., a materialized view can rewrite many query blocks and a query block can be rewritten individually with multiple materialized views. Several materialized view candidates can *simultaneously* rewrite a multi-block or snowstorm query. These materialized view candidates are collected together to form a unique set of materialized views called *MV-group*. Figure 9 shows the enumeration of MV-groups for the example in Figure 8. It represents many-to-many relationships between MV-groups and eligible queries. All materialized views in an MV-group simultaneously rewrite the query.

The *estimated benefit* of an MV-group for an eligible query is defined as the difference between the optimizer estimated costs of the query without rewrite and with rewrite. The *cumulative estimated benefit* of an MV-group is simply a summation over the estimated benefits for all its eligible queries; this is the performance metric used for the GGR algorithm below.

[MV1, MV3] ↔ {Q1}
[MV2, MV3] ↔ {Q1}
[MV1] ↔ {Q1, Q6}
[MV2] ↔ {Q1, Q2, Q5}
[MV3] ↔ {Q1, Q4}
[MV4] ↔ {Q2, Q3, Q5}
[MV5] ↔ {Q3, Q4, Q6}

Figure 9. MV-Groups and Queries

The *reduction factor* of a materialized view MV_i is defined as the sum of cardinalities of all tables referenced in MV_i divided by the cardinality of MV_i.

The *global greedy recommendation* (GGR) algorithm, shown in Figure 10, takes as an input a workload (a SQL Tuning Set), a set of candidate materialized view groups $\boldsymbol{M}$ and their eligible query blocks (Section 3.4), and a storage space constraint $\boldsymbol{C}$. The objective of the GGR algorithm is to select a set of materialized view groups $\boldsymbol{R}$ ($\boldsymbol{\subseteq M}$) such that $\boldsymbol{R}$ maximizes the cumulative estimated benefit under the storage space constraint $\boldsymbol{C}$.

Although the GGR algorithm provides an efficient and effective solution, it does not guarantee a globally optimal solution [14], since the ECSE algorithm does not generate *all* possible materialized view candidates (Section 3.4) and the GGR algorithm uses heuristics to enumerate MV groups.

Currently, the GGR algorithm considers only one type of constraint – the size of available storage space – but, in the future, it can be extended to also include materialized view maintenance cost (Section 5).

Algorithm GGR {

1. For each candidate materialized view definition, parse its text and create a *virtual* materialized view with only statistics and meta-data by invoking optimizer cost functions on the parsed structure to generate estimated statistics.
2. Modify the statistics of virtual materialized views with sampling-based cardinalities (Section 3.3.5).
3. For each query block QB_i of the workload queries do:
   3.1. Sort all eligible materialized views of QB_i in the descending order of their reduction factors.
   3.2. Retain only the top κ (e.g., 5) materialized views for QB_i.
4. Enumerate M, the set of MV-groups, for all workload queries using a greedy technique.
5. For each MV-group G in M do:
   5.1. Rewrite all eligible workload queries with G (without considering other MV-groups) and compute its cumulative estimated benefit.
   5.2. Discard G, if its cumulative estimated benefit is not positive.

6. For each query Q_j in the workload do:
   6.1. From all the MV-groups eligible for Q_j, pick the MV-group that has the highest cumulative estimated benefit and mark it.
7. Discard MV-groups that have not been marked at Step 6.1 for any query.
8. Let C be the specified storage space constraint. Sort all the MV-groups in the descending order of their (cumulative estimated benefit / estimated storage size).
9. T := 0. For each materialized view group G, in the order generated at Step 8, do:
   9.1. S := Estimated storage size of G;
   9.2. If (S + T) > C, then discard G; else, T := T + S;
10. Recommend R, the set of all remaining candidate materialized views. **}**

Figure 10. Algorithm GGR

The optimizer rewrite strategy used in recommending materialized views at Step 5.1 in the GGR algorithm (Figure 10) is different from the optimizer rewrite strategy applied in a user environment. The objective of the former is to recommend the right set of materialized view candidates for multiple workload queries, whereas the objective of the latter is to optimize a single query at a time given one or more materialized views. The latter strategy is used in Section 4 for verification.

At Step 3.2 in Figure 10, the heuristic simply selects the best κ materialized views for that query block without affecting any eligible materialized view of other query blocks.

At Step 7 in Figure 10, the reason for discarding an MV-group that is not the best for any of its eligible queries is that there exist other MV-groups which are eligible for these queries and have higher cumulative estimated benefits than that of the MV-group being discarded.

4. MATERIALIZED VIEW VERIFICATION

At the final step, we verify the performance of recommended materialized views using an Oracle tool called SQL Performance Analyzer [15], which accepts a SQL workload and allows us to measure the impact of recommended materialized views on the execution of workload queries using various performance metrics.

For the verification phase, the optimizer rewrite module decides in a cost-based manner which recommended materialized view(s) will be the most beneficial for the rewrite of each workload query.

A stratified sample of the workload queries that can be rewritten with the recommended materialized views is used to verify their performance. Stratification partitions a set of eligible queries into non-empty disjoint strata such that every query appears in exactly one stratum. Here the criterion used for forming a stratum is that all queries in a stratum can be rewritten by the *same* set of recommended materialized views. The idea behind stratification is that it puts queries with structural similarity into a single stratum and thus provides a more representative sample. A random selection is used to choose a certain percentage of queries from each stratum; this forms a sample that is used for verification.

Figure 11 shows an example of a workload of 19 queries Q1-Q19, for which 6 strata, S1-S6, are formed based on 5 recommended materialized views, MV1-MV5. Queries Q16 and Q18 do not appear in Figure 11, since they are not eligible for cost-based materialized view rewrite. Other subsets – e.g., {MV1, MV5} – belonging to the power set of the recommended materialized view set are not shown, since no queries are rewritten by the optimizer using those subsets. Each stratum below shows a set of queries and their eligible materialized view(s). The stratum S3, for example, contains queries Q3, Q8, and Q12, which are rewritten using materialized views MV1 *and* MV2 together. Unlike the groupings in Figure 9, stratification, shown in Figure 11, puts an eligible query in exactly *one* stratum.

S1. {Q2, Q9, Q11} → {MV1}
S2. {Q1, Q4, Q5, Q6} → {MV2}
S3. {Q3, Q8, Q12} → {MV1, MV2}
S4. {Q7, Q10} → {MV3}
S5. {Q14, Q15, Q17} → {MV4}
S6. {Q13, Q19} → {MV4, MV5}

Figure 11. Stratification of Queries

A percentage improvement (or regression) of a query Q_i with materialized view rewrite is called *execution benefit* (EB) and is given by the following formula, where MVR and PM refer to materialized view rewrite and performance metric (e.g., elapsed time, CPU time, buffer-gets, etc.) respectively.

$$EB = [PM(Q_i) - PM(MVR(Q_i))] \times 100 / PM(Q_i).$$

The baseline queries against which execution benefit is measured may involve pre-existing access structures.

If multiple MVs are used to rewrite Q_i, then the execution benefit of Q_i is divided equally by the count of materialized views used in Q_i. This provides a rough estimate of *partial impact* each materialized view has on execution benefit.

The materialized views are first created with data in an invisible mode such that they are not accessible to the user. Once a materialized view is created for verification, the optimizer collects real statistics for it. The sampled queries are executed with materialized view rewrite to determine its performance; the non-rewritten performance numbers found in the given SQL Tuning Set are used.

A materialized view is considered to have passed verification, if its average execution benefit is more than a certain percentage. The materialized views that pass verification are made visible to the user; this step is called *publication* of materialized views.

The materialized views that do not pass verification are registered in a feedback table before they are discarded. Subsequent runs of the selection module will proactively discard recommended materialized views that have a match in the feedback table.

5. MATERALIZED VIEW MAINTENANCE

In this section, we briefly describe the maintenance [14] of automated materialized views (auto-MVs). This topic deserves a paper of its own.

5.1 Tracking of DML and MV Usage

Oracle provides Object Activity Tracking Subsystem (OATS) that tracks data manipulation language (DML) operations, partition

maintenance operations, materialized view query rewrites, and materialized view refreshes.

Tracking, which is cumulative, is done for every 15-minute interval. For each table, OATS tracks the number of inserts, deletes, updates as well as the number of rows affected. For each materialized view, it tracks the number of query rewrites, type of rewrites (e.g., full, partial, etc.), method of refresh (e.g., incremental, complete, etc.), refresh times, and the number of missed rewrites due to its staleness.

5.2 Materialized View Refresh

Auto-MV refresh is performed by a background job that executes periodically every 15 minutes for a duration of one hour with pre-defined resource limits.

Auto-MV maintenance uses a neural-net-based machine learning algorithm [21] available in Oracle data mining package. The goal is to schedule the refresh of all stale auto-MVs so that the number of future query rewrites is maximized. For every 24-hour period, we build a new neural net model to predict future DML operations and future auto-MV usage. The data – number of inserted, deleted, and updated rows, and the number of auto-MV rewrites – for building the neural net model comes from OATS. Once a model is built, it is validated using a five-fold cross-validation technique [7], which divides the data from OATS into five equal chunks. To ensure the accuracy of the model, the neural net algorithm is run five times, each time using a different chunk as test set and the remaining four chunks as training set.

The model, if it passes cross-validation for an auto-MV, provides its expected rewrite count and its next quiet window (i.e., the time period where the defining tables of the auto-MV's are not modified and thus it can be used for rewrite). For each auto-MV, we determine its estimated refresh time using a generalized linear regression algorithm [5], whose input includes the size of the auto-MV, method of its refresh, affected number of rows in its defining tables, and the average CPU time of its previous refreshes. The stale auto-MVs are scheduled for refresh in the descending order of their effective net impact, which is computed from an auto-MV's cumulative expected rewrite count, quiet window, estimated refresh time, and execution benefit supplied by the verification module (Section 4).

However, if the model fails cross-validation, we use a simpler algorithm called change events. It first excludes those auto-MVs whose defining tables have undergone modifications within the last four time-intervals thereby avoiding the auto-MVs that are likely to become stale in the near future. The remaining stale auto-MVs are then scheduled for refresh in the descending order of their execution benefit.

6. EXPERIMENTS

An implementation of the system of automated generation of materialized views was used to perform extensive experiments on several customer workloads. In this section, we provide a summary of these experiments for three customer workloads. The experiments were done on an Exadata X2-8 machine with 2 compute nodes, each with 8x8-core Intel X7560 processors. Our performance reports show results in terms of three performance metrics buffer-gets, CPU time, and elapsed time, but we show the results of our experiments only in elapsed times.

6.1 Customer Workload-P

A customer workload, referred to here as Workload-P, has a star schema and contains 91 queries. These queries reference over 200 base tables. The number of tables in the queries ranges from 1 to 5. Seven of these tables are fact tables; the 3 largest fact tables contain about 2.4 B rows.

In one experiment with Workload-P, we used the following threshold values: $\alpha = 2$, $\beta = 2$, and $\lambda = 2$. There were only 2 recommended materialized views, which contained 2 and 3 tables. These materialized views rewrote 5 queries, whose elapsed times showed an average improvement of over 250%. The reasons for the small numbers of recommendations are manifold: the query blocks have very few tables in common, as 91 queries reference over 200 tables; 26 of these queries contain only a single table; and our heuristics require that join sets must contain at least 2 tables ($\alpha = 2$) and the final derived join sets must rewrite at least 2 query blocks ($\beta = 2$). If the restrictions on α and β were relaxed by setting them to 1, then many materialized views could be recommended.

6.2 Customer Workload-G

One customer workload, referred to here as Workload-G, has a snowstorm [2] schema and contains about 650 queries, which were used for this experiment. These queries reference over 30 base tables. The number of tables in the queries ranges from 1 to 19; four of these tables are fact tables. The average number of tables per query is 11. The largest fact table contains about 791 M rows.

In an experiment with Workload-G, we used the following threshold values: $\alpha = 2$, $\beta = 2$, $\lambda = 2$, and $\rho = 0.1$. There were 29 recommended materialized views. The verification module discarded 12 materialized views, because the benefits of 7 materialized views were below the required percentage and 5 materialized views were not selected by the optimizer to rewrite any queries. It published 17 materialized views that rewrote a total of 83 queries.

The scatter graph in Figure 12 compares the elapsed times of these queries before and after rewrite with the published materialized views. In the graph, every data point under the diagonal represents a query with improvement and a data point that appears above the diagonal represents a query with regression.

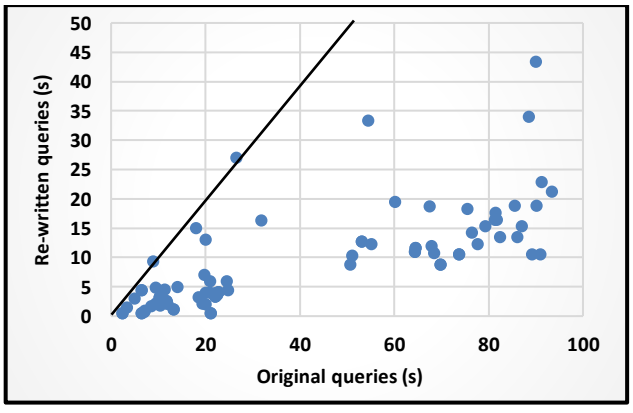

Figure 12. Elapsed Times for Workload-G

The 17 published materialized views, which rewrote a total of 83 queries, provided an average performance improvement of more than 440% in terms of elapsed time.

6.3 Customer Workload-H

Another customer workload, referred to here as Workload-H, has a snowflake schema and contains 64 queries and 12 base tables. The average number of tables per query is 7. There is only one fact table, which contains 3.6 B rows.

In experiments with Workload-H, all 64 queries were selected. We used the following threshold values: $\alpha = 2$, $\beta = 2$, $\lambda = 2$, and ρ (the block sampling percentage) was varied from 1 to 25. The results of recommendation and verification modules are summarized in Table 1.

Since we inject sampled materialized view cardinalities into the dictionary tables, which are used by the optimizer cost model (Section 3.3.5), we observed increasingly better materialized view selection with larger sampling percentage, although the primary purpose of sampled cardinalities is to prune out unpromising materialized views.

Table 1. Impact of Block Sampling Percentage

Sampling % (ρ)	Count of Recommended MVs	Count of Published MVs	Count of Rewritten Queries
1	4	4	15
2	4	4	15
5	4	4	15
15	5	5	21
20	6	6	23
25	8	7	32

The scatter graph in Figure 13 compares the elapsed times of 32 queries with $\rho = 25$ before and after rewrite with the published materialized views.

The 7 materialized views, which rewrote a total of 32 queries, provided a performance improvement of more than 400% in terms of elapsed time.

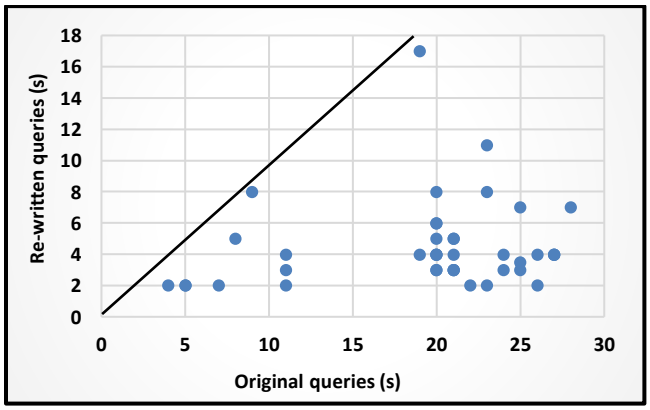

Figure 13. Elapsed Times for Workload-H

7. RELATED WORK

The literature on materialized view selection uses many ideas developed by multi-query optimization and common sub-expression selection research [17, 18, 19, 20, 22, 25, 27, 29], as they share similar strategies, though not necessarily the same goals. The problem of materialized view selection is much more general than that of sub-expression selection, as the former can consider computations that do not appear in the workload queries; this increases the space of possible solutions and complicates query containment and materialized view rewrites.

In [19], the authors use an ILP-based formulation and focus on the problem of sub-expression selection for large workloads by selecting common parts of logical plans of queries and materializing them to speed-up the evaluation of subsequent jobs; they consider one optimizer generated logical plan at a time and consider all its sub-plans. This technique has been integrated with Microsoft SCOPE. Another approach was previously taken in [27] for utilizing common sub-expressions for cloud query processing; this work has also been prototyped in SCOPE. Both these works present formal treatments of their techniques.

There are currently several automated physical design tools [2, 15, 31] offered by commercial database vendors and by third-party tool developers. These tools support tuning of different aspects of physical design.

IBM's DB2 Advisor [30, 31] recommends materialized views and indexes; this tool uses the query optimizer itself to both suggest and evaluate candidate MV's and indexes; the algorithm, which is based on the Knapsack problem, trades off the cost of MV or index storage against its benefits of workload queries, builds a new 'explain plan mode' to build hypothetical configuration and exploits multi-query optimization techniques developed in [22] to construct candidate MVs. DB2 design Advisor is architected to have independent advisors for each physical design structure; the search step that produces the final integrated recommendation iteratively invokes each advisor for a physical structure in a staged manner.

Oracle 10g shipped the SQL Access Advisor [15], which takes a workload and provides index and materialized view recommendations for the overall workload. The current work described in this paper is very different from the existing SQL Access Advisor.

The Database Tuning Advisor (DTA) from Microsoft SQL Server 2005 [2] is a tool that provides fully integrated recommendations for indexes, materialized views, and horizontal range partitioning. DTA builds upon the Index Tuning Wizard and improves it in several aspects. The basis of DTA's recommendations is the "what-if" analysis of MS SQL Server [10] extended to support simulation of materialized views; it uses a three-step process of candidate recommendation, which is described in detail in [1]. The idea of workload compression [2, 12] as a technique to improve the scalability of workload was adapted into DTA. A workload is partitioned based on the signature of each query; two queries have the same signature, if they are identical in all respects except the literals. Workload compression chooses a subset from each partition using a clustering-based method.

Given a workload of queries, [1] describes a technique for recommending materialized views. It uses table cardinalities and optimizer estimated costs of workload queries for exploring

arbitrary subsets of all tables in the database schema to come up with interesting table subsets, which can be used to generate hypothetical candidate materialized views. It then applies the Greedy (m, k) algorithm [13] to enumerate configurations for each query, at a time, and to choose the lowest cost configuration using estimated cost of the rewritten query. Lastly, it merges two or more materialized view definitions using heuristics involving their optimizer estimated cardinalities. This work has been implemented for the MS Tuning Wizard.

8. CONCLUSION

In this paper, we described a novel extended covering sub-expression (ECSE) algorithm for the automated generation of candidate and recommended materialized views on various set-based relationships among queries in a given workload. As searching the space of all possible materialized views in a scalable manner is of paramount importance, we apply the in-built heuristics in the ECSE algorithm, a set of external heuristics, and the optimizer-estimated cost-based selection for recommending effective and efficient materialized views, which are then verified by creating the recommended materialized views and comparing performance of a sample of workload queries with and without rewrite. These experiments show that our techniques provide significant performance gains for various customer workload queries. This system has been fully implemented and will be deployed on the Oracle Autonomous Database on the Cloud.

Our future research may involve determining the threshold values, α, β, λ, ρ, and κ, based on factors such as the count and complexity of queries in the workload, cardinalities of fact tables, storage requirement, etc. Another direction our future work may take is the periodic monitoring of workload queries to identify static and dynamic filter predicates. The filter predicates that are static (i.e., their constant values do not change over time) can be included in the candidate materialized view definitions thereby making rewrites more efficient. We also plan to incorporate in the GGR algorithm the expected materialized view maintenance cost, which can be predicted by the neural-net-based machine learning algorithm.

9. ACKNOWLEDGEMENT

We wish to thank Murali Thiyagarajan, Mohamed Ziauddin, Srinivasan Ramakrishnan, and Peter Damron, the members of the auto-MV team, for their help and support in the implementation.

10. REFERENCES

[1] Agarwal, S., Chaudhuri, S., and Narasayya, V., *Automated Selection of Materialized Views and Indexes for SQL Databases*, Proc. of the 26th Int. Conf. on VLDB, Cairo, Egypt, 2000.

[2] Agarwal, S., Chaudhuri, S., Kollar, L., Marathe, A.P., Narasayya, V., and Symala, M., *Database Tuning Advisor for Microsoft SQL Server 2005*, Proc. of the 30th VLDB Conf., Toronto, Canada, 2004.

[3] Ahmed, R., Sen, R., Poess, M., and Chakkapen, S. Of Snowstorms and Bushy Trees. PVLDB, 7(13):1452-1461, 2014.

[4] Ahmed, R., Lee, A., Witkowski, A., Das, D., Su, H., and Cruanes, T., *Cost-Based Query Transformation in Oracle*, Proc. of the 32nd VLDB Conf., Seoul, S. Korea, 2006.

[5] Annette, J.D. and Barnett, A.G., *An Introduction to Generalized Linear Models*, Fourth Edition, 2018.

[6] Arfati, F. and Chirkova, R., *Selecting and Using Views to Compute Aggregate Queries*, Journal of Computer and System Sciences, vol. 77, no. 6, 2011.

[7] Arlot, S. and Celisse, A., *A survey of Cross-Validation Procedures for Model Selection*, Statistics Surveys. vol. 4, p. 40–79, 2010.

[8] Bello, R., Dias, K., Downing, A., Feenan, J., Finnerty, J., Norcott, W., Sun, H., Witkowski, A., and Ziauddin, M., *Materialized Views in Oracle*, Proc. of the 24th Int. Conf. on VLDB, New York, U.S.A., 1998.

[9] Charikar, M., Chaudhuri, S., Motwani, R., and Narasayya, V., *Towards Estimation Error guarantees for Distinct Values*, Proc. of the 19th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, May 2000.

[10] Chaudhuri, S. and Narasayya, V., Auto-Admin: *'What-If' Index Analysis Utility*, Proc. of ACM SIGMOD, 1998.

[11] Chaudhuri, S., Datar, M., and Narasayya, V., *Index Selection for Databases: A Hardness Study and a Principled Heuristic Solution*, IEEE Trans. Knowl. and Data Engg. 16(11), 2004.

[12] Chaudhuri S. and Narasayya, V., *Self-Tuning Database Systems: A Decade of Progress*, Proc. of the 33rd VLDB Conf., Vienna Austria, 2007.

[13] Chaudhuri S. and Narasayya, V., *An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server*, Proc. of 23rd VLDB Conf., Athens, Greece, 1997.

[14] Chirkova, R. and Yang, J., *Materialized Views*, Foundation and Trends in Databases, vol. 4, no. 4, p. 295-405, 2011.

[15] Dageville, B., Das, D., Dias, K., Yagoub, K., Zait, M., and Ziauddin, M., *Automatic SQL Tuning in Oracle 10g*, Proc. of the 30th VLDB Conf., Toronto, Canada, 2004.

[16] Das, S., Grbic, M., Ilic, I., Jovandic, I., Jovanovic, A., Narasayya, V., Radulovic, M., Stikic, M., Xu, G., Chaudhuri, S., *Automatically Indexing Millions of Databases in Microsoft Azure SQL Database*, ACM SIGMOD, Amsterdam, Netherlands, 2019.

[17] Goldstein, J. and Larson, P.A., *Optimizing Queries Using Materialized Views: A Practical Scalable Solution*, ACM SIGMOD, Santa Barbara, U.S.A, 2001.

[18] Gupta, H., and Mumick, I.S., *Selection of Views to Materialize Under Maintenance Cost Constraint*, Intl. Conf. on Database Theory, Jerusalem, Israel, 1999.

[19] Jindal, A., Karanasos, K., Rao, S., and Patel, H. Selecting Subexpressions to Materialize at Datacenter Scale. PVLDB, 11(7):800-812, 2018.

[20] Kathuria, T. and Sudarshan, S., *Efficient and Provable Multi-Query Optimization*, PODS, Chicago, U.S.A., 2017.

[21] Kubat, M., *An Introduction to Machine Learning*, Springer, 2015.

[22] Lehner, W., Cochrane, B., Pirahesh, H., and Zaharioudakis, M., *Applying Mass Query Optimization to Speed Up Automatic Summary Table Refresh*, Intl. Conf. on Data Engineering., 2001.

[23] Leis, V., Gubichev, A., Mirchev, A., Boncz, P., Kemper, A., and Neumann, T., *How Good are Query Optimizers, Really?*, PVLDB 9(3), November 2015.

[24] Lohman, G., *Is Query Optimizer a 'Solved' Problem?*, http://wp.sigmod.org/?p=1075, 2015.

[25] Roy, P., Seshadri, S., Sudarshan, S., and Bhobe, S., *Efficient and Extensible Algorithms for Multi Query Optimization*, ACM SIGMOD, 2000.

[26] Shapiro, G.P., *The Optimal Selection of Secondary Indices is NP-Complete*, SIGMOD Record 13(2), 1983.

[27] Silva, Y. N., and Larson, P-A., Zhou, J., *Exploiting Common Subexpression for Cloud Processing*, ICDE, 2012.

[28] Talebi, Z. A., Chirkova, R., Fathi, Y., and, Stallman, M., *Exact and Inexact Methods of Selecting Views and Indexes for Performance Improvement*, EDBT, Nantes, France, 2008.

[29] Zhou, J., Larson, P-A., Freytag, J-C. and Lehner, W. *Efficient Exploitation of Similar Subexpressions for Query Processing*, ACM SIGMOD, Beijing, China, 2007.

[30] Zilio, D. C., Rao, J., Lightstone, S., Lohman, G., Storm, A., Garcia-Arellano, C., and Fadden, S., *DB2 Design Advisor: Integrated Automatic Physical Database Design*, Proc. of 30th VLDB Conf., Toronto, Canada, 2004.

[31] Zilio, D. C., Rao, J., Lightstone, S., Ma, W., Lohman, G., Cochrane, R., Pirahesh, H., Colby, L.S., Gryz, J., Alton, E., Liang, D., and Valentin, G., *Recommending Materialized Views and Indexes with IBM DB2 Design Advisor*, Proc. of Intl. Conf. on Autonomic Computing, 2004.