Towards Materialized View Selection for Distributed Databases

Leonardo Weiss F. Chaves[1], Erik Buchmann[2], Fabian Hueske[1], Klemens Böhm[2]
[1] SAP Research CEC Karlsruhe, Germany
{leonardo.weiss.f.chaves|fabian.hueske}@sap.com
[2] Universität Karlsruhe (TH), Germany
{buchmann|boehm}@ipd.uni-karlsruhe.de

ABSTRACT

Materialized views (MV) can significantly improve the query performance of relational databases. In this paper, we consider MVs to optimize complex scenarios where many heterogeneous nodes with different resource constraints (e.g., CPU, IO and network bandwidth) query and update numerous tables on different nodes. Such problems are typical for large enterprises, e.g., global retailers storing thousands of relations on hundreds of nodes at different subsidiaries.

Choosing which views to materialize in a distributed, complex scenario is NP-hard. Furthermore, the solution space is huge, and the large number of input factors results in non-monotonic cost models. This prohibits the straightforward use of brute-force algorithms, greedy approaches or proposals from organic computing. For the same reason, all solutions for choosing MVs we are aware of do not consider either distributed settings or update costs.

In this paper we describe an algorithmic framework which restricts the sets of considered MVs so that a genetic algorithm can be applied. In order to let the genetic algorithm converge quickly, we generate initial populations based on knowledge on database tuning, and devise a selection function which restricts the solution space by taking the similarity of MV configurations into account. We evaluate our approach both with artificial settings and a real-world RFID scenario from retail. For a small setting consisting of 24 tables distributed over 9 nodes, an exhaustive search needs 10 hours processing time. Our approach derives a comparable set of MVs within 30 seconds. Our approach scales well: Within 15 minutes it chooses a set of MVs for a real-world scenario consisting of 1,000 relations, 400 hosts, and a workload of 3,000 queries and updates.

1. INTRODUCTION

Materialized views (MV) are a well-known technique to reduce the response time of complex queries in database management systems (DBMS). MVs can improve the query performance by avoiding re-computation of expensive query
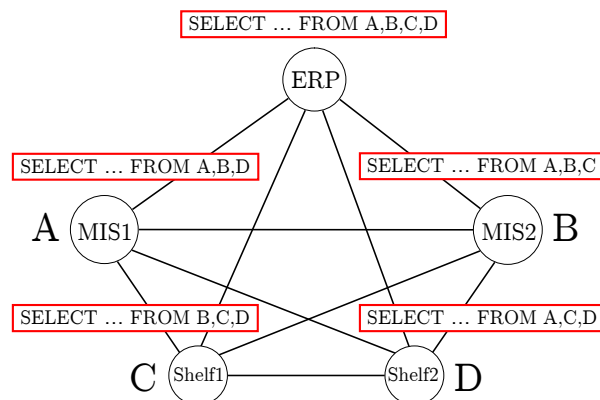

Figure 1: Simple example scenario

operations. In a distributed DBMS, MVs can materialize query results near to the query issuer and reduce network transmissions. On the other hand, MVs have to be recomputed when the underlying relations are updated, and various resource constraints have to be considered. Thus, it is far from trivial to find the optimal set of MVs in complex, distributed scenarios.

Example 1: Think of a large retailer that wants to adopt RFID technology. Figure 1 illustrates a simple scenario consisting of one central Enterprise Resource Planning System (ERP), two Merchandise Information Systems (MIS) storing the relations A, B and two smart shelves managing the tables C, D. ERP, MIS and the smart shelves store different data. A typical query in this scenario joins data from different nodes, e.g., "Join the product code of blue pants from ERP with the inventory data of all MIS in shops within a radius of 20km, and count the tuples where the attribute *sold* is *false*". Many aspects have to be considered when optimizing such queries. If the tables are updated very frequently, no MV should be materialized. In the case of high-speed links, the central ERP node should materialize the tables from all MIS. Otherwise, the MIS could materialize the product codes from ERP, etc.

Currently, there is no approach to compute *which view* has to be materialized *on what machine* in a *decentralized DBMS* consisting of many *heterogeneous nodes* with different constraints on CPU, IO and network bandwidth, where each node issues many *different queries and updates* at *different rates*. We call this the *distributed view selection problem*.

Solving this problem for complex, distributed scenarios is challenging, for multiple reasons. First, the distributed view selection problem is known to be NP-hard [10]. Second, distributed scenarios consisting of nodes with many different resource constraints result in non-monotonic cost models. Thus, greedy algorithms which propose MVs based on optimal solutions for subproblems cannot be applied because the benefit of materializing a certain view might depend on all other MVs materialized. Third, the number of possible views to materialize grows exponentially with the number of computer nodes and queries, and with the numbers of columns, join predicates, grouping clauses and tables referenced in each query. Due to the huge solution space, brute-force strategies, e.g., backtracking, as well as biology-inspired solutions like ant colony optimization or genetic algorithms cannot be applied directly. Finally, a practical solution for the distributed view selection problem must fulfill two non-functional requirements:

1. **Robust results for inaccurate costs:** Database optimization is based on cost estimations, which might deviate from the real cost to some extent. However, it is important to obtain "good" MVs even if the underlying cost model is simplified or inaccurate.

2. **Flexible cost models:** Business IT scenarios feature different optimization goals, e.g., guaranteed quality-of-service agreements or staged pricing models, and some machines might be tailored for specific queries. Thus, a view selection algorithm must allow to exchange the model without depending on restrictions like linearity or monotonicity.

Approaches for estimating a good set of MVs according to a given query mix have been implemented in centralized DBMS for years [1, 2, 6, 7, 21, 24, 26, 28, 29]. However, these approaches depend on monotonic cost models, while considering updates and distributed data results in non-monotonic models [4, 11]. All existing approaches to derive MVs in distributed scenarios we are aware of make prohibitive restrictions in order to obtain monotonic cost models or reduce the complexity, e.g., no update costs [4, 10, 12, 25] or only one querying node [10, 11, 12, 13, 16].

In this paper, we propose an algorithmic framework which reduces the solution space of the view selection problem for complex scenarios so that the problem becomes solvable by a genetic algorithm. Therefore, we have to (1) represent the solution space as a bit matrix that is small enough to be fed into a genetic algorithm, and (2) propose mechanisms that let the genetic algorithm converge quickly. We start by choosing a set of "costly" tables. As this is a common basis for view selection approaches, we use a standard algorithm [2] to derive this set from our workload definition and cost model. In a second step, we identify MV candidates. Unlike other approaches which build numerous query execution plan alternatives to extract MV candidates, our approach is based on a syntactic analysis: We decompile the workload into aggregation operations, joins and predicates, and we recombine these operations to construct promising MV candidates. In the final step, we use a genetic algorithm to propose MVs and the nodes to materialize them on. Therefore, we let the genetic algorithm start with initial populations containing individuals that either minimize query execution costs or update costs. Finally, we devise

a selection function which restricts the solution space by taking the similarity of MV configurations into account. Although genetic algorithms do not guarantee to find optimal solutions, they are well suited for this kind of problem as they can avoid getting stuck in local optima.

We evaluate our approach both with artificial settings and a large real-world RFID scenario we have devised together with a large retailer. As our experiments show, an exhaustive search over the complete solution space of a small scenario with 9 hosts and 24 tables requires 10 hours to compute the optimal set of MVs. For the same scenario, our approach provides within 30 seconds a solution that is only 0.01% worse. Our experiments confirm that our approach scales very well. Within 15 minutes, it proposes a good set of MVs for very large scenarios consisting of 1,000 relations distributed over 400 hosts, which together process a complex workload of 3,000 queries and updates.

Paper outline: In the next section we discuss the related work. In Section 3 we describe the RFID scenario that has motivated our work, and outline how MVs can optimize queries on RFID data. In Section 4 we present our approach. We evaluate our approach in Section 5, and we close with a conclusion.

2. RELATED WORK

We divide related work into (1) scenarios with one computer node that holds all tables, and views are materialized on this computer node (non-distributed MVs), (2) scenarios with tables distributed across several computer nodes, and views are materialized on one computer node (semi-distributed MVs), and (3) scenarios where tables and MVs can reside on any computer node in a network (distributed MVs). The third scenario corresponds to our problem.

(1) Non-distributed MVs: Choosing MVs in centralized scenarios is a well researched problem. In such scenarios, storage is considered to be the limiting factor.

[2] describes a heuristic for automated selection of materialized views and indexes in centralized relational databases without considering updates. The approach introduces a system model and an algorithm for choosing a set of "costly" tables, which we have adapted as a starting point for our approach. Extensions of [2] include horizontal partitioning [1] and the materialization of frequently accessed rows only [28]. All of these approaches iterate over subsets of MV candidates to construct the final set of MVs from optimal solutions for sub-problems. This is only possible with monotonic cost models. Otherwise the benefit of materializing a certain view depends on the complete set of MVs proposed.

A number of approaches consider updates when proposing MVs. [24] describes an algorithm to derive MVs from an existing query access plan. However, as the algorithm might get stuck in local optima, the savings potential of this algorithm is limited. Similar algorithms face comparable problems, e.g., [3] only considers a "representative" query subset of the workload. A more recent approach is [29], which introduces an algorithm that uses the query optimizer to suggest and evaluate multiple candidate MVs and indexes.

Many approaches have been proposed for data warehouses, which are tailored for specific update strategies, storage requirements or multidimensional data sets. [6] describes how to automatically distribute storage space among MVs and indexes. [21] proposes a heuristic to choose which aggregate views to precompute for multidimensional datasets. [7] de-

scribes how MVs can be incrementally updated using log data. In [19], a method for maintenance of MVs is described which exploits common subexpressions of different MVs. [27] proposes an algorithm to lazily maintain materialized views. This decreases update costs and allows the aggregation of updates. Algorithms to choose which MV should be used for each query are presented in [7, 20].

[26] uses genetic algorithms to choose MVs from a genome that represents a complete query access plan consisting of multiple data warehouse queries. However, [26] cannot be adapted to a distributed scenario: Here, the query access plan depends on the nodes where the MVs are materialized on, and the solution space is too large to be directly evaluated by a genetic algorithm.

(2) Semi-distributed MVs: Many approaches consider choosing which views to materialize in semi-distributed scenarios. [12] presents a model for analyzing materialized and virtual views for distributed data where views are maintained on one mediator. The model calculates query response time, view freshness and system load though parameters like query frequency, query complexity, update frequency and network delay. [10] proposes a heuristic for choosing which views to materialize in a data warehouse. The heuristic is based on view graphs. The authors show their problem is NP-hard, and their solution yields at least 63% of the optimal benefit. In [11], this solution is expanded to choose MVs under the constraint of view maintenance costs in contrast to the usual storage constraint. This seems to be the first paper to present an algorithm for this type of constraint. The problem of choosing which views to materialize has also been researched in federated databases: [16] analyzes which views to materialize in front-end databases, and [13] analyzes which views to materialize in back-end databases. A self-tuning algorithm for data placement in distributed databases is proposed by [15]. A method for incrementally maintaining MVs in semi-distributed scenarios is shown in [22].

(3) Distributed MVs: There is little work on choosing which views to materialize in a distributed scenario. [4] seems to be the first paper to address the problem of materializing views in distributed scenarios. The concepts are explained for data warehouses, but they should apply for RDBMS. The algorithm is based on aggregation lattices. However, [4] does not consider load or network transmission, which is very important in distributed scenarios. Furthermore, [4] explains that the approach presented cannot handle updates, because this would result in a non-monotonic cost model. [25] uses a greedy algorithm based on view derivation cubes to choose where to materialize views in a distributed data warehouse. A linear cost model is used and update costs are not considered. The paper emphasizes that there is still no effective solution for distributed scenarios.

3. APPLICATION SCENARIO

In this section, we introduce a generic RFID scenario for retail [23], which we have devised together with a large retailer. The scenario demonstrates that complex enterprise scenarios cannot be handled by existing approaches (cf. Section 2). We will refer to this scenario as a running example, and we will use it to evaluate our approach. Three aspects make it challenging to solve the view selection problem for such scenarios: the scale of the infrastructure, the heterogeneity of the hardware, and the complexity of the workload.
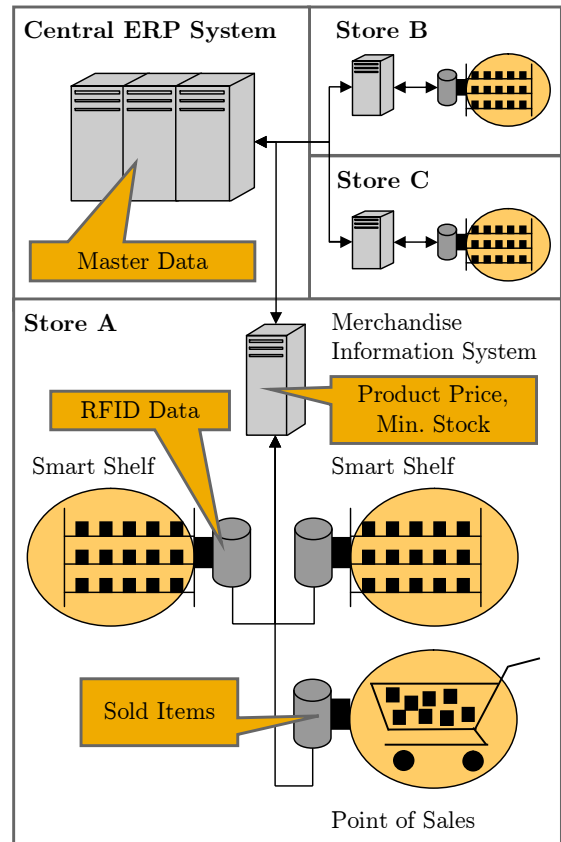

Figure 2: An IT hierarchy from a retail scenario

Infrastructure. Figure 2 sketches a typical IT infrastructure of a large retailer. One Enterprise Resource Planning System (ERP) supports business tasks for all stores, e.g., logistics, data warehousing or purchase planning. Assume the retailer has 100 stores. Each store has its own Manufacturing Information System (MIS). The MIS holds information about product orders, product sales and product data. Each store has one point of sales (POS) and two RFID readers. The POS hosts the product prices and manages all cash registers in the store. The two RFID readers hold the data from all smart shelves in the front store and back store. Any node in this setting – even a RFID reader – is capable of materializing a view.

Hardware. Since the IT infrastructure of an enterprise grows over time, all nodes have different resource constraints. Machines like ERP systems or RFID readers are tailored for particular tasks. The network bandwidth varies at a large scale. The MIS of large stores are connected with the ERP over high-speed connections, while small stores use dial-up telephone lines. The nodes within a store use Ethernet.

Workload. The workload consists of a large number of complex queries and updates that are executed with different frequencies and from different nodes. Due to different products sold, bonus offers or peak sales, the workload is unique for each node. The product information and summarized sales data managed by the ERP system are updated infrequently. The POS and RFID readers in a store have to cope with approx. 25,000 updates per day. The MIS in each store performs approximately 9,000 updates per day.

Symbols of Scenario	
$f(q), f(u)$	The frequency of a query or update
N	The set of computer nodes
Q	The set of queries
U	The set of update statements

Symbols of Algorithms	
C	The set of all MV candidates
$cost_{query}(x, c_1, n_1,$ $c_2, n_2, \cdots, c_i, n_i)$	The costs of a query or update statement x when the view c_w is materialized on node n_w
$cost_{table}(x, t)$	The costs of table t in query or update statement x
Q_{ts}	The set of queries that contains the table subset ts
s_{pop}	The population size
TS	The final set containing table subsets of the table selection algorithm
θ	Threshold for a table subset to be relevant
θ_{expTab}	Threshold for expensive tables
θ_{sim}	Threshold for similarity of populations
U_{ts}	The set of update statements that contains the table subset ts

Table 1: Symbols used

4. MATERIALIZED VIEW SELECTION

In this section we describe our approach to solve the view selection problem for complex, distributed scenarios. We consider a distributed database system consisting of a set of nodes N, a set of queries Q, a set of update statements U and their respective frequencies f (cf. Table 1). Our solution space is huge: Let m be the number of materializeable views. Then there are $2^{m*|N|}$ possible solutions of which view to materialize on which node. m can be very large. For each query $q \in Q$, MVs can be generated for all combinations of columns and predicates in q, i.e., for each query there are at least $m' = (2^{|columns|} - 1) * (2^{|predicates|} - 1)$ possible MVs. Joins and grouping clauses further increase the size of m.

Our approach uses background knowledge on database optimization to exclude inferior MVs from the solution space, so that the view selection problem becomes solvable by a genetic algorithm [5]. Therefore, we reduce the columns and predicates that need to be evaluated by considering the similarity of queries, and by focusing on "expensive" tables. Approaches for the view selection problem consist of steps that select relevant tables, identify promising MV candidates and enumerate over all candidates in order to obtain a "good" set of MVs (cf. [2]). Our approach follows this structure, but proposes novel solutions for the last two steps.

1. **Table Selection**: This step analyzes the workload and chooses table subsets which have a significant impact on the workload costs. Therefore, we adapt the metrics of a table selection algorithm [2] for centralized DBMS so that it considers distributed scenarios.

2. **Candidate Generation**: Based on the table subsets, this step generates MV candidates. Our approach is based on a syntactical analysis of the workload. We decompile the workload into aggregation operations,

joins and predicates, we recombine base tables, operations and predicates to create MV candidates, and choose the most promising ones using our cost model.

3. **MV Selection**: In this step, we use a genetic algorithm to choose (1) the views from the candidate set that are actually materialized and (2) the nodes they are materialized on. Therefore, we devise an initial population, a fitness function and a selection function, and we develop a heuristic for the placement of MVs to reduce the solution space evaluated.

Cost Model. Our approach requires cost estimates from a cost model. As explained in Section 1, we must not depend on specific cost models or restrictions like monotonicity or linearity. Thus, we build on any cost model that provides two generic measures $cost_{query}()$ and $cost_{table}()$.

$cost_{query}(x, c_1, n_1, c_2, n_2, \cdots, c_i, n_i)$ estimates the costs of a query or update statement x when i views are present, while view c_w is materialized on node n_w. One view can also be materialized on multiple nodes. The views c and the nodes n are optional. If they are not given, this function estimates the cost without considering nodes and/or views.

$cost_{table}(x, t)$ estimates the costs of accessing a single table t in a query or update statement x, i.e., costs of table-scans, selection and sending data.

The cost model we have used for our evaluation will be described in Subsection 5.2.

4.1 Step 1: Table Selection

We start by choosing tables with a high impact on the runtime of the query mix, e.g., tables that have a high cardinality, which are queried very often, or located on low performance computer nodes. At this step, the solution space includes $2^x - 1$ possible combinations of relevant tables, where x is the number of different tables referenced by all queries in the query mix. The naive approach would be iterating through every possible subset of tables in order to filter the relevant ones. But this approach is not feasible due to the large number of table combinations possible. Instead, we adapt the metrics of a common table selection algorithm [2] to consider distributed scenarios.

The algorithm works as follows: It iterates from $i = 1, 2, \cdots$ to the number of different tables specified in the query mix. For each iteration it generates table subsets S_i consisting of i tables, and it computes a coarse estimation for the cost savings if this subset would be materialized. If the coarse cost estimation of such a table subset is smaller than a threshold θ, the subset is discarded. Finally, the algorithm uses a detailed cost metric to remove insignificant subsets from the final result set. An evaluation of this algorithm and of the choice of parameters can be found in [2].

The algorithm requires a cost threshold θ and two cost metrics: $TS–Weight(ts)$ calculates the detailed costs of all queries in the query mix that use a given table subset ts, and $TS–Cost(ts)$ estimates approximate costs only, with $TS–Cost(ts) \geq TS–Weight(ts)$. We derive $TS–Cost(ts)$ and $TS–Weight(ts)$ from our generic cost measures $cost_{query}()$ and $cost_{table}()$ as follows:

$$TS-Cost(ts) = \sum_{q \in Q_{ts}} cost_{query}(q) * f(q) \qquad (1)$$

Q_{ts} is the set of queries that reference the queries in ts. $cost_{query}(q)$ returns the costs of a query q without using MVs, and $f(q)$ is the frequency of q in the workload.

[2] defines $TS{-}Weight$ as the sum of query costs weighted by the cardinalities of the tables. However, this definition is not suitable for distributed scenarios where the cardinality is not the only influencing factor. We base our calculation of $TS{-}Weight$ on $cost_{query}()$ to capture the performance of the nodes hosting the tables:

$$TS{-}Weight(ts) =$$
$$\sum_{q \in Q_{ts}} \left(cost_{query}(q) * f(q) * \frac{\sum_{t \in ts}(cost_{table}(q,t))}{\sum_{t \in T_q}(cost_{table}(q,t))} \right) \quad (2)$$

Q_{ts} is the set of queries that refer to the tables ts, T_q is the set of tables needed to answer query q, and $f(q)$ is the frequency of a query q in the workload. $cost_{table}(q,t)$ returns the costs of a table t in q, i.e., the costs of table-scans, selection and sending data, which depend on the resources of the node storing t. This definition captures the cardinality of a table, its query frequency and the performance of its allocation node. Note that as the cardinality of ts increases, the values of $TS{-}Cost(ts)$ and $TS{-}Weight(ts)$ will decrease since there will be fewer queries that use all tables in ts.

> *Example 2:* To illustrate how the table selection algorithm works, we apply it to our simple scenario (Fig. 1). The algorithm starts with table subsets containing one table, i.e., $S_1 = \{\{A\}, \{B\}, \{C\}, \{D\}\}$, and it iterates over subsets with two, three and four tables. Assume the threshold θ is the cost of a table subset used by at least three queries. Thus, the result is $TS = \{\{A\}, \{B\}, \{C\}, \{D\}, \{A, B\}, \{A, C\}, \{A, D\}, \{B, C\}, \{B, D\}, \{C, D\}\}$.

4.2 Step 2: Candidate Generation

We now devise promising MV candidates from the queries and the table subsets $ts \in TS$ identified in the previous step. In the following, we refer to the tables in ts as *base tables*. Existing approaches, e.g., [29], obtain MV candidates by generating numerous alternatives of query execution plans, which are filtered for "expensive" subgraphs. The plans can be derived from multi-query optimization [19, 29] or by merging multiple query plans [10, 11]. However, these methods require optimizer calls that are expensive for complex scenarios. Furthermore, as the number of processing alternatives grows exponentially with the number of computer nodes, query operators and tables involved, such approaches produce huge candidate sets for complex settings.

We follow a different approach, that considers each table subset $ts \in TS$ in isolation and decomposes the queries into selection and join predicates, grouping clauses and aggregation operations. We generate MV candidates in two phases: First, we create two generic MV base candidates. One base candidate materializes the join of all tables in ts and supports the intersection of all predicates of all queries Q_{ts} that refer to the tables in ts. The other base candidate supports grouping and aggregation functions in addition. Second, we generate a number of refined MV candidates from the generic candidates. The intuition is as follows: Computing the intersection of all predicates over the join of all tables results in one costly MV with a high cardinality. On the other hand, dividing the base candidate into multiple specialized MVs that support only a few tables and predicates might

reduce the update cost of the MVs. In order to obtain distinct candidates at different levels of specialization, we use an approach based on a binary tree to continuously split the predicates of the base candidate. Thus, the root of the tree is the base candidate that supports all queries in Q_{ts}, and the leaves are the most specific MV candidates which support only the predicates and tables of a single query.

Generation of Generic Candidates

We distinguish between (1) *general MV candidates* and (2) *aggregation MV candidates*. General MV candidates can be used by any query in Q_{ts}, while aggregation MV candidates only support queries with the same aggregate functions and the same selection predicates on the aggregated columns. For each table subset, we generate one general base candidate and if applicable one aggregation MV candidate. The candidates consist of the following components:

1. The **general base candidate**:

   - Base tables: The join of all tables in ts.
   - Predicates: Selection and join predicates that are common for all $q \in Q_{ts}$ and refer to a table in ts. If queries have a predicate on the same column but select different value ranges, the MV predicate selects the union of these values ranges.
   - Projections: Every column in ts that is part of the query result or needed to compute a predicate, join, grouping or aggregation in a query in Q_{ts}.

2. The **aggregation candidate** includes the general base candidate together with the following components:

   - Grouping clauses: Every column in ts that is grouped by a query $q \in Q_{ts}$ or required for the computation of other query predicates.
   - Aggregation functions: Every aggregation function that is part of any query $q \in Q_{ts}$ and refers to a table in ts.

Example 3 shows an intuition for the candidate generation.

Refinement of MV Candidates

Instead of materializing one MV with a general selection predicate or aggregation function, it might be more efficient to materialize multiple specialized MVs with highly selective predicates or aggregation functions. Such MVs have a smaller cardinality, thus they can be queried and updated at smaller costs. For example, consider the base candidate $c_{ts,Q}$ from Example 3, which supports the queries q_1 to q_5. A MV candidate that supports only q_2, q_4, q_5 has a smaller cardinality, because its predicates are more selective: $\sigma_{A.a1 \, = \, B.b1 \wedge A.a2 \, > \, 100}$. In the case the MV supports q_5 only, its predicates would be $\sigma_{A.a1 \, = \, B.b1 \wedge A.a2 \, > \, 100 \wedge B.b2 \, = \, 10}$.

In this phase, we generate specific MV candidates, which the genetic algorithm will evaluate together with the generic candidates. A straightforward approach would generate the permutation of all predicate combinations. However, the number of combinations grows exponentially with the number of predicates. Our approach exploits the similarity of queries. Because all queries in Q_{ts} use the same base tables in ts, their selection predicates must refer to the same columns. Thus, it is possible to identify clusters of queries

Example 3: Consider the Queries $Q = \{q_1, q_2, q_3, q_4, q_5\}$ referring to the tables A, B and C:

q_1 : "SELECT A.a1, A.a2, B.b2, C.c2
    FROM A, B, C,
    WHERE A.a1 = B.b1 AND A.a1 = C.c1"

q_2 : "SELECT A.a1, A.a2, B.b2
    FROM A, B
    WHERE A.a1 = B.b1 AND A.a2 > 100"

q_3 : "SELECT A.a1, A.a2, A.a3, B.b2
    FROM A, B
    WHERE A.a1 = B.b1"

q_4 : "SELECT A.a1, SUM(B.b3)
    FROM A, B
    WHERE A.a1 = B.b1 AND A.a2 > 120
    GROUP BY A.a1"

q_5 : "SELECT A.a1, A.a2, B.b2
    FROM A, B
    WHERE A.a1 = B.b1 AND A.a2 > 100 AND
        B.b2 = 10"

One table subset $ts \subseteq TS$ which we obtained in the last step from all tables used in Q is $ts = \{A, B\}$. Thus, the general base candidate $(c_{ts,Q})$ and the aggregation candidate $(c'_{ts,Q})$ for this subset and the queries in Q are:

$c_{ts,Q}$: "SELECT A.a1, A.a2, A.a3, B.b2, B.b3
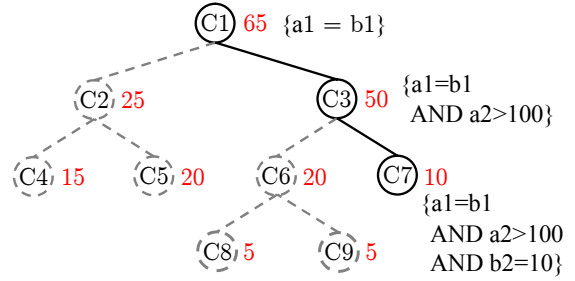   FROM A, B
    WHERE A.a1 = B.b1"

$c'_{ts,Q}$: "SELECT A.a1, SUM(B.b3)
    FROM A, B
    WHERE A.a1 = B.b1
    GROUP BY A.a1"

Example 4: Assume a candidate tree whose base candidate supports the queries $q1$ to $q5$ from Example 3:

Candidate $C1$ represents $c_{ts,Q}$ in Example 3. The example shows how query sets are split according to its similarity. The first split separates the queries with selection predicates $(q2, q4, q5)$ from those without $(q1, q3)$. The red labels in the tree outline the cost reduction of the nodes. Observe node $C2$, which provides a cost reduction of 25. Its children $C4, C5$ cover the same range of predicates, but reduce the cost by $15 + 20$. Thus, our algorithm removes $C2$ and keeps $C4, C5$. The final set of candidates is $C3$, $C4$ and $C5$.

which take advantage of one highly specific MV. For example, q_2, q_4, q_5 are similar, because they benefit from a single MV that materializes Table A with the predicate $\sigma_{A.a2 > 100}$ and the projection $\pi_{A.a1, A.a2}$. We define the similarity of queries as the cardinality of the intersection of the query results. Other metrics for similarity might also be used.

We use a binary tree (cf. Example 4) to organize and structure all predicates in Q_{ts} according to their similarity and degree of specialization. The root contains the MV base candidate with predicates that include the domain of values for all queries, and the leaves contain MV candidates with specific predicates for single queries. Thus, we only consider $(2 * |Q_{ts}| - 1)$ cases. We use the tree to choose MV candidates that minimize costs of query execution. However, we must not neglect MVs that make updating other MVs less expensive, e.g., one MV might materialize intermediate results which are useful when updating many other MVs. We create such MV support candidates in a subsequent step.
(1) Raise the tree. In order to build the binary tree, we first assign the root with the base candidate and all queries in Q_{ts}. For each pair of children, we divide the set of queries in the parent so that each child is assigned with queries having similar predicates. In particular, for every candidate

in the tree the set of supported queries is split into two subsets. Both subsets are used to build new child candidates, which are appended to the tree. We split a set of supported queries Q by sorting all queries $q \in Q$ into two buckets based on their similarity, i.e., according to the cardinality of the intersection of the query results. We stop when all leaves of the tree are candidates that support only one query. For a base candidate that supports $|Q_{ts}|$ queries, $(2 * |Q_{ts}| - 1)$ candidates are generated.
(2) Remove inferior candidates. After building the tree, we remove inferior candidates. Therefore, we calculate the net reduction in costs r_{net} for each candidate c, i.e., the difference between the minimal update costs $cost_{upd}$ and the maximal cost reduction r_{max}:

$$r_{net}(c) = r_{max}(c) - cost_{upd}(c) \tag{3}$$

$$r_{max}(c) = \tag{4}$$
$$\sum_{q \in Q_{ts}} f(q) * \max_{n \in N} \left(cost_{query}(q) - cost_{query}(q, c, n) \right)$$

$$cost_{upd}(c) = \tag{5}$$
$$\sum_{u \in U_{ts}} f(u) * \min_{n \in N} \left(cost_{query}(u, c, n) - cost_{query}(u) \right)$$

To compute the maximal cost reduction of c, we sum up the maximal cost reduction for every query q in Q_{ts}. Therefore, we assume c is materialized on every node, and compute the cost of executing q (Equation 4). The minimal update costs of c are computed likewise (Equation 5), while U_{ts} is the set of update statements that alter tables in ts. We use the minimal update costs, since the computation of updates is much more expensive than replicating tables. Thus, our approach updates a view on the node with the smallest update costs, and replicates the view to other nodes.

We now traverse the tree in reverse level-order and calculate the net reduction in costs for each candidate in the tree. If the reduction of a candidate is larger than the sum

of reductions of its children, the children are removed from the tree. Otherwise, the children are adopted by the parent of the candidate. The tree loses the binary property in this step. We stop if the tree has only one level.

(3) Add support candidates. When the trees of all base candidates have been built and cut, we refine the result set by adding candidates that reduce the update costs of the MV candidates proposed. The candidates generated so far consider the costs of query execution only. But there can also be MVs that reduce update costs of other MVs. For example, a MV could compute a complex intermediate result that is needed when updating a number of other MVs. We call these additional candidates *support candidates*. For every candidate with two or more base tables, we identify *expensive* base tables which have a big influence on the update costs of the view. A table t is considered to be expensive for a MV candidate c, if Equation 6 holds:

$$\sum_{u_t \in U_t} cost_{query}(u_t, c) > \left(\sum_{u \in U} cost_{query}(u, c) \right) * \theta_{expTab} \quad (6)$$

$cost_{query}(u, c)$ computes the update costs of the MV c for the update statement u, U is the set of update statements that alter any base table of c, U_t is the set of update statements that alter t and θ_{expTab} is a threshold. Having identified the expensive base tables of c, we compute the support candidate's set of base tables TS_{sup} by subtracting the expensive tables from the set of c's base tables. The support candidate c_{sup} is build from TS_{sup} and the set of queries supported by c. We check if c_{sup} supports c by comparing the update costs for c with and without c_{sup} being materialized. If c's update costs are reduced, we add c_{sup} to the result set.

The Candidate Generation Algorithm

Now we describe how the concepts introduced in this section, i.e., generic candidates and their refinement, are combined into one algorithm, cf. Algorithm 1. The algorithm generates a set of MV candidates for a given set of queries Q, a set of updates U, and a set of computer nodes N. For each table subset ts the set Q_{ts} of queries and the set U_{ts} of update statements that contain ts are obtained (Lines 5, 6). We build the generic MV candidates from these sets (Line 7). For each MV candidate we build a tree and generate specialized MV candidates (Lines 10, 11), and add the remaining candidates to the result set of candidates in Line 12. After that, we improve the result set further by adding candidates that reduce the update costs of already selected candidates (Line 14). When all candidates have been checked for support candidates, the algorithm returns the result set (Line 15) and terminates.

4.3 Step 3: MV Selection

This step evaluates the base MV candidates together with the specialized MV candidates obtained from the last steps. In particular, we find out (1) which MVs should be realized, and (2) what are the optimal nodes to materialize them on. Because even after applying our heuristics the solution space is still too large to enumerate every possible MV configuration, we adapt a genetic algorithm to solve the view selection problem.

Genetic algorithms [5] belong to the class of probabilistic optimization algorithms. The algorithm organizes possible solutions (*individuals*) in *populations*. New individuals are derived by random *mutations* and *crossings* of individuals. The next *generation* is composed from individuals that are *selected* based on their quality (*fitness*). In order to solve the view selection problem by a genetic algorithm, we code all the possible configurations (the individuals) as binary allocation matrices that specify if a certain MV candidate (column) is materialized on a certain node (row). We refine the genetic algorithm for the view selection problem by (1) generating initial populations based on knowledge on database tuning, and (2) devising a selection function that is specific to our problem. Both extensions help the genetic algorithm to converge faster than the standard approach [9].

1: **input** set of table subsets TS, set of queries Q and a set
    of updates U
2: $C_{base} = \{\}$
3: $C_{res} = \{\}$ // initialize result set
4: **for all** $(ts \in TS)$ **do** // Generate base candidates
5:    $Q_{ts} = selectReferencingQueries(ts)$
6:    $U_{ts} = selectReferencingUpdates(ts)$
7:    $C_{base} = C_{base} \cup generateBaseCandidates(ts, Q_{ts}, U_{ts})$
8: **end for**
9: **for all** $(c_{base} \in C_{base})$ **do** // For each base candidate
10:   $buildTree(c_{base})$ // build tree for base candidate
     // remove candidate w/ inferior quality from tree
11:   $cutTree(c_{base})$
     // add remaining candidates to result set
12:   $addTree(c_{base}, C_{res})$
13: **end for**
14: $addSupportingCandidates(C_{res})$
15: **output** C_{res}

Algorithm 1: Candidate Generation Algorithm

---

(1) Initial population

Although the previous steps reduced the solution space for the view selection problem by selecting promising MV candidates only, the number of alternatives to be evaluated still grows exponentially with the number of computer nodes where the MVs can be materialized. Standard implementations of genetic algorithms start with a randomly selected initial population. However, if we can devise an initial population that is close to the optimal solution, the genetic algorithm will converge after a few iterations instead of browsing a large number of inferior solutions. Therefore, we apply the following heuristic: *A candidate c is preferably materialized on a node which either hosts one or more of c's base tables (minimizes update costs) or is the issuer of a query supported by c (minimizes execution costs).* Note that this heuristic does not inhibit the materialization of views on nodes than neither host nor query a table; such nodes can still be chosen for materialization through mutations. Following this heuristic we define for each candidate c a set of 'promising' materialization nodes N_c:

$$N_c = \{n \mid n \in N, (\exists t, t \in ts_c, n \in N_t) \vee \\ (\exists q, q \in Q_c, n = origin(q))\} \quad (7)$$

ts_c are the base tables of c, N_t is the set of computer nodes hosting the table t, Q_c is the set of queries supported by c and $origin(q)$ gives the origin node of a query q.

(2) Selection function

When composing a new population, a genetic algorithm selects the 'fittest' individuals from the previous population and introduces some mutation and crossover of the genome. In the case of the view selection problem, this means selecting MV sets according to their cost reductions. However, selecting individuals only because of their fitness might lead to populations consisting of individuals with similar genetic information, i.e., the genetic algorithm evaluates the fitness of similar individuals in parallel. To reduce the number of iterations and diversify the population, our selection function takes the similarity of individuals into account. We define the similarity $s(i_1, i_2)$ between two individuals i_1, i_2 as the sum of identical bits in their allocation matrices:

$$s(i_1, i_2) = \left(\sum_{i=1}^{|N|} \sum_{j=1}^{|C|} x_{i,j} \right) / (|N| * |C|) \qquad (8)$$

$$x_{i,j} = \begin{cases} 0 & \text{, if } A_1[i][j] \neq A_2[i][j] \\ 1 & \text{, if } A_1[i][j] = A_2[i][j] \end{cases} \qquad (9)$$

A_1, A_2 are the allocation matrices of i_1 and i_2 respectively. The average similarity of an individual i to all individuals in a population P is calculated as follows:

$$avgSimilarity(i, P) = \left(\sum_{i' \in P} s(i, i') \right) / |P| \qquad (10)$$

1: **input** population P, population size s_{pop},
   similarity threshold θ_{sim}
2: $P_{sort} = sortPopulationByFitness(P)$
   // sort individuals by fitness in descending order
3: $P_{next} = \{\}$ // initialize the next population
4: **for all** (individual i in P_{sort}) **do**
5:    **if** $(avgSimilarity(i, P_{next}) < \theta_{sim})$ **then**
6:       $P_{next} = P_{next} \cup i$
         // add individual i to P_{next}
7:    **end if**
8:    **if** $(|P_{next}| = s_{pop})$ **then**
9:       **break** // Exit loop
10:   **end if**
11: **end for**
12: **output** P_{next}

Algorithm 2: Selection function

Our selection function $select(P, s_{pop}, \theta_{sim})$ is described in Algorithm 2. Its input parameters are the current population P, which is the combined set of former selected individuals and those derived from mutation and crossing, the maximal population size s_{pop} and a similarity threshold θ_{sim}. First, we sort P according to the fitness of the individuals in descending order (Line 2). We iterate over the sorted set starting with the fittest individual (Line 4) and check for each individual if its average similarity to the next generation is less than the threshold θ_{sim} (Line 5, Equation 10). An individual that passes the check is added to the next generation (Line 6). We stop adding individuals to the next population, if either the size of the next generation reaches the desired population size (Line 8) or all individuals have been processed. Finally, the next generation is returned (Line 12).

The right choice of θ_{sim} involves a tradeoff between runtime and the quality of the result. A small threshold will result in small populations (P_{next}) that can be evaluated quickly. On the other hand, a large threshold results in a more comprehensive set of alternatives to be evaluated, and might produce better results. In our experiments we achieved a good tradeoff with a setting of θ_{sim} that required at least $(0.2 * |C|)$ cells of the allocation matrices to differ (i.e., $x_{i,j} = 0$ in Equation 9).

The MV Selection Algorithm

Algorithm 3 shows how we have integrated our heuristics into a genetic approach [5]. First, the algorithm generates an initial population (Line 2). $generateInitialPopulation()$ randomly generates an initial population. Then the algorithm steps into an evolution loop and computes descendant populations until the termination condition is fulfilled (Line 3). We define the termination condition as $(genCount \geq genThreshold)$, where $genThreshold$ is a threshold for the number of consecutive iterations without significant cost reduction and $genCount$ counts the iterations without cost reduction. A descendant population is generated by mutation (Line 6), crossing (Line 11) and selection (Line 16) of its ancestor population's individuals. For the next iteration of the evolution loop, the descendant population becomes the ancestor population (Line 17). Finally, $getBestIndividual(P_{suc})$ returns the fittest individual of the population.

1: **input** mutation probability p_m, crossover probability p_c
2: $P_{anc} = generateInitialPopulation()$
   // generate initial population
3: **while** (Termination Condition is not fulfilled) **do**
4:    $P_{new} = \{\}$
5:    **while** $(i < |P_{anc}|)$ **do**
      // do for every individual of P_{anc}
6:       **if** $random() < p_m$ **then**
         // Mutate i-th individual and add mutant to P_{anc}
7:          $P_{new} = P_{new} \cup mutate(P_{anc}, i)$
8:       **end if**
9:       **if** $(random() < p_c)$ **then**
         // Cross i-th and random individual and
         // add offspring to P_{anc}
10:      $j = (random() * |P_{anc}|)$
11:      $P_{new} = P_{new} \cup crossover(P_{anc}, i, j)$
12:     **end if**
13:     $i = i + 1$
14:   **end while**
15:   $computeFitnesses(P_{anc})$
     // select individuals for next generation
16:   $P_{suc} = select((P_{anc} \cup P_{new}), s_{pop}, \theta_{sim})$
17:   $P_{anc} = P_{suc}$
18: **end while**
19: **output** $getBestIndividual(P_{suc})$

Algorithm 3: Genetic Algorithm

---

5. EVALUATION

In this section we evaluate our approach by simulating different scenarios consisting of a query mix, a workload definition and a distributed setup of machines with different resource constraints. Genetic algorithms do not guarantee optimal solutions. However, we show that our approach de-

vises 'good' MVs that are intuitively reasonable for database administrators. We compare our approach with a brute-force variant that finds the optimal set of MVs by browsing through the whole solution space, and we measure how our approach copes with misestimated cost parameters. We demonstrate that the results of our approach are stable, i.e., there is little deviation in the results of multiple test runs. Finally, we show that our approach copes with huge distributed scenarios consisting of up to 400 nodes, 1,000 tables and about 3,000 queries and update statements.

Note that the right choice of the parameters for genetic algorithms is not the focus of this paper. We obtained our results with a population size of 64 individuals, a crossing probability of 25%, a mutation probability of 15%, and termination if none of the last 4 iterations had a cost reduction of at least 0.1%. An in-depth analysis of parameters of genetic algorithms can be found in [9].

5.1 Experimental Setup

For our experiments we programmed a Java prototype of the algorithms described in Section 4. We implemented a simulated distributed database environment including computing nodes, database tables and a cost-based query optimizer. The cost model for the optimizer will be described in the next section.

The optimizer generates query execution plans for our simulated database environment, i.e., it decides which MV are used in which queries. Further, it calculates the execution costs of queries and updates needed for our algorithms. The implementation of the optimizer is based on well-known algorithms [8, 18] which we adapted to consider MVs and distributed databases. The optimizer follows the dynamic programming approach and computes an execution plan from a restricted set of operators. In particular, it supports a subset of SPJG without nested subqueries. To obtain a distributed execution plan the optimizer computes a join order and allocates operators on computer nodes of the distributed database. In presence of materialized views, the optimizer checks for every intermediate result if it can be computed from a MV. The optimizer estimates the size of intermediate results based on the principles of independence and equal distribution. It computes the costs for update statements that update base tables for an eager incremental maintenance strategy.

We use a simple and a real-world test scenario:

Simple Scenario. We use a simple scenario as described in Figure 1 to show how our approach works. It contains four tables and five computer nodes with the same constraints for CPU, IO and network bandwidth. All tables have the same structure, cardinalities and update frequencies.

Real-world Scenario. This scenario is based on the retail scenario described in Section 3. The setting consists of one ERP system, and varies between 20 and 100 stores. For each store, 4 nodes, 10 tables, 15 queries and 14 update statements are added. In contrast to the simple scenario, the workload is different for each node. The scenario and the test data come from a large retailer; except for the RFID table which we estimated from an ongoing field trial.

5.2 Cost Model

For our evaluation, we use a cost model similar to [18], which we have extended for distributed settings. For each relational operator we consider its most simple implementa-
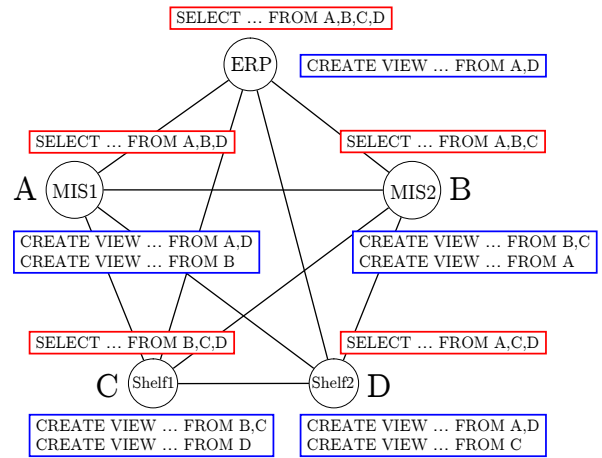

Figure 3: Example scenario

tion, e.g., sequential table scans and nested-loop joins. Our cost model assumes consistent data, i.e., if a view is stalled, it has to be updated before querying. We compute the cost of a query as the sum of all normalized individual costs incurred by the required relational operations. Therefore, we consider the computing and IO costs of a computer node (CPU, IO), and the network transmission cost $NET_{N1,N2}$ between two computer nodes $N1, N2$. Furthermore, we use catalog information: $iCard$ is the cardinality of a table. iW is the summed width of one or more input tuples, i.e., the number of bytes fetched to process one input row. $compW$ is the summed width of the columns checked by predicates. $aggW$ and $groupW$ are the summed widths of the rows being aggregated or grouped. We use these data in our cost model as follows:

SELECT Fetching a table, incl. projection and selection.
$$cost_{sel} = (IO * iW * iCard) + (CPU * compW * iCard)$$
JOIN Nested loop join, evaluation of join predicates.
$$cost_{join} = (CPU * compW * iCard_1 * iCard_2)$$
GROUP and AGGREGATE Sorting the input and sequential comparison or aggregation.
$$cost_{agg} = (CPU * groupW * iCard * log_2(iCard)) + (CPU * iCard * (groupW + aggW))$$
DATA SHIPPING Costs for transferring data.
$$cost_{net} = (NET_{N1,N2} * iW * iCard)$$

Note that our approach is not limited to this kind of simple cost models. As we have pointed out in Section 4, more complex models [14, 17, 18] can be used as well, e.g., in order to consider resource contention.

5.3 Experiment 1: Providing an Intuition

In order to show that our approach proposes 'good' MVs that are intuitively reasonable, we conducted a number of experiments with the simple scenario. Figure 3 shows the MVs which are devised by our approach according to the given workload and query mix. The edges symbolize the network connections, and the vertices the computer nodes. Since all nodes have the same resource constraints, and all tables have the same update ratio and cardinality, we expected each node to compute its query only from local tables and MVs to save transport costs. The figure shows that our expectation holds: the MVs proposed for this setting are spread over all nodes. Our approach suggests creating views
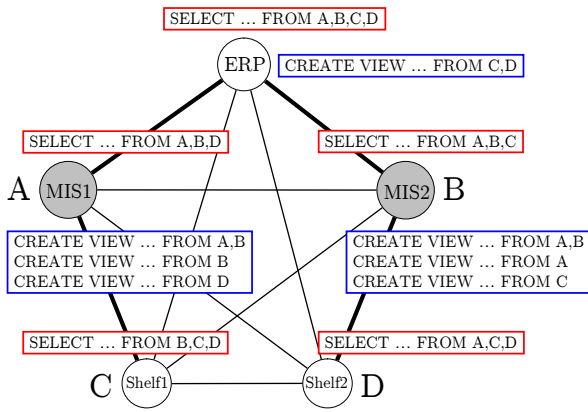
Figure 4: MVs after changes in computer nodes

Figure 5: Materialization after workload changes

Figure 6: Variation in relative cost reduction

over one and two tables on the nodes MIS1, MIS2, Shelf1 and Shelf2, and a view over two relations on ERP. This setup enables MIS1, MIS2, Shelf1 and Shelf2 to compute their queries without any direct network access. Note that the VIEW on A,D is only computed once and then replicated to other nodes. Because of the symmetry of the scenario and because the genetic algorithm is indeterministic, there are other solutions with equal quality, e.g., materializing B,C instead of A,D on node ERP.

Now we decrease the CPU and IO costs of the MIS nodes and their network costs to the ERP and Shelf nodes by 50%. If our approach works well, more views should be materialized on these nodes, as they can do faster computations and network transmissions now. Figure 4 shows that our approach works as expected: Tables previously materialized on Shelf1 and Shelf2 are now materialized on MIS1 and MIS2. The thick lines in the figure show the communication costs, e.g., MIS1 and MIS2 have a fast connection to the ERP, but the connection from MIS1 to MIS2 is slow.

Finally, we switch back to the original costs, but change the workload: We increase the number of tables queried by Shelf1, Shelf2 from three to four, and we decrease the number of tables queried by ERP from four to two. The nodes MIS1, MIS2 now query three external tables, instead of two external and one internal table. We expect that the result favors local computation of queries as all nodes have the same performance parameters. The modification of the workload should cause a shift of views in a way that more views are materialized on the shelves. The results can be seen in Figure 5. In comparison to the first experiment, the new workload results in MVs that let every node process its queries from local data. Again, because of the symmetry of the scenario there are multiple equal solutions, e.g., materializing C on Shelf2 instead of materializing D on Shelf1.

5.4 Experiment 2: Result Variation

Since the genetic algorithm introduces randomness into the process of selecting MVs, the set of MVs proposed varies from one test run to another. However, we assume that many sets of MVs provide comparably good results. In order to verify this assumption, we ran 10 tests with our real-world scenario consisting of 20 stores (80 nodes), and we recorded the relative cost reduction of each generation. We define the *relative cost reduction r* as the difference of the costs of the original scenario (c_o) and the costs when the
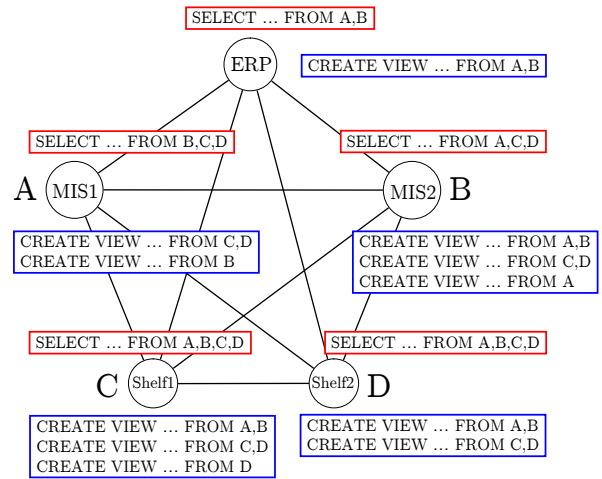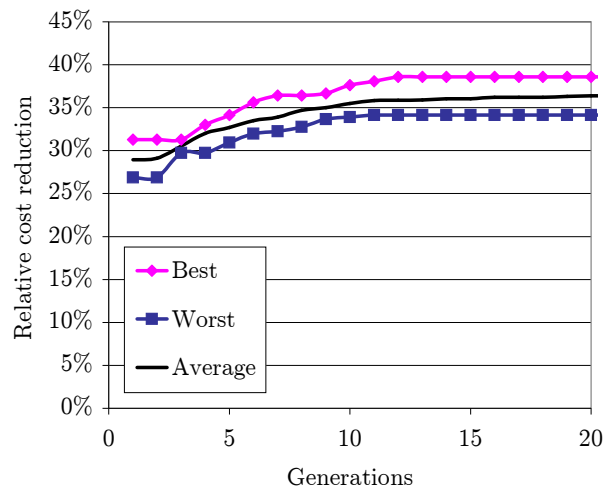
MVs are applied (c_m), normalized by the costs without MVs: $r = (c_o - c_m)/c_o$.

Figure 6 shows the results of this series of experiments. Each dot on the curves represents one generation of our genetic algorithm. For each generation the best, worst and average relative cost reduction of 10 test runs is shown. For this experiment we relaxed the termination condition so each test would run for many more generations than needed. We only show the results for the first 20 generations of the genetic algorithm, since further generations did not introduce significant improvements. The runtime of each test was less than 4 minutes. The experiments confirm that our heuristics are effective: the genetic algorithm converges in surprisingly few generations. Furthermore, the worst and best result do not deviate more than 2% from the average value during optimization time. The experiment also shows that the results are stable. Even one run produces a result that saves a high share of the processing costs.

5.5 Experiment 3: Cost Savings and Runtime

We now explore how our approach works in a complex setting. We run experiments with the real-world scenario, and
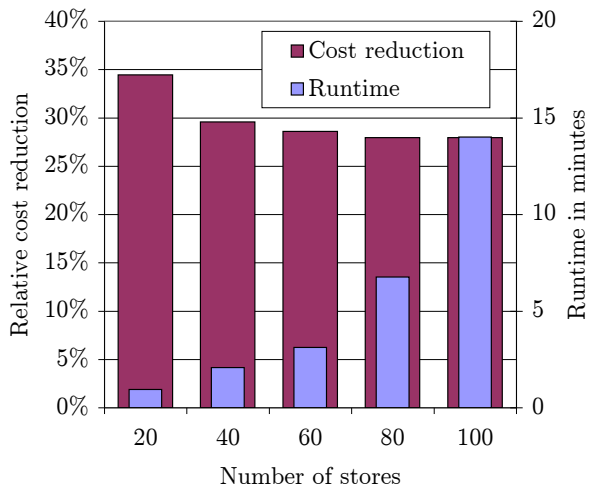
Figure 7: Runtime and cost reduction in dependence of the scenario size

Figure 8: Influence of the cost model on our approach

we measure both the runtime and the relative cost reduction of the MVs proposed. In order to investigate the influence of the complexity on our approach, we varied the number of stores from 20 to 100. We executed 10 test runs for each scenario size and computed the average of the results. We expect the size of the scenario will have little influence on the quality of the MVs proposed, and the runtime will stay low even for huge scenarios.

The results are shown in Figure 7. All tests show a relative cost reduction of at least 28%, which is slightly decreasing with an increasing complexity of the scenario. Even though runtime significantly increases with the size of the scenario, it is still low. The scenario with 100 stores which has more than 400 computer nodes and more than 1,000 tables has a runtime of around 14 minutes. As expected, the experiment confirms that the quality of the results is not significantly influenced by an increasing scenario complexity. Furthermore, we have shown that our approach is applicable to large-scale distributed environments.

5.6 Experiment 4: Optimality

Genetic algorithms do not guarantee to find the global optimum, because of their probabilistic behavior. However, we can show that our approach provides 'good' results.

In this experiment we compare the set of MVs proposed by our approach with the optimal set of MVs. Therefore, we implemented a straightforward brute-force algorithm that finds the optimal solution by browsing trough all sets of MVs and MV placements possible. Since this is very time-consuming, we had to reduce our real-world scenario to the ERP node and two stores, i.e., to 9 nodes and 24 tables.

After running both our approach and the brute-force variant, we compare the quality of the MVs proposed. In the scenario described, our approach proposes MVs that are near the optimum. In particular, the relative cost reduction of the set of MVs proposed by our genetic algorithm is less than 0.01% worse than the relative cost reduction of the optimal set of MVs. However, when comparing the runtimes of the algorithms, our genetic approach required 30 seconds while browsing through the whole solution space of the NP-hard problem took 10 hours processing time.
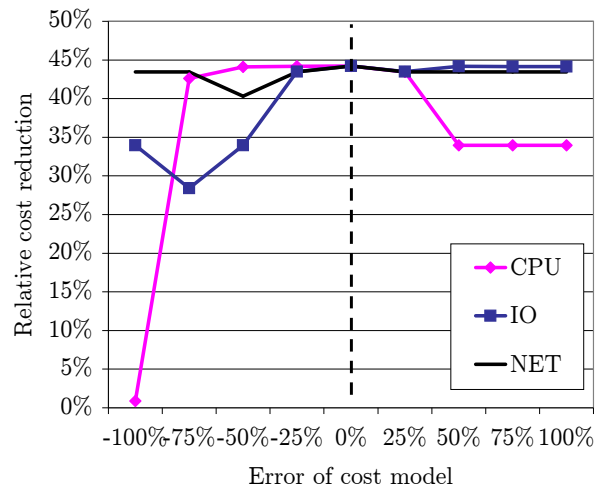
5.7 Experiment 5: Robustness of the Result

Obviously, the effect of the MVs proposed for the query runtime will degrade with misestimated cost parameters. For example, if the resources available at a certain node are overestimated, our approach might materialize many MVs at a slow machine. A practical solution for the distributed view selection problem must devise good sets of MVs even with (slightly) misestimated cost parameters.

With this series of experiments we want to explore the impact of such inaccurate cost parameters on the cost reductions that result from the proposed set of MVs. Again, we use the real-world scenario in this experiment. We define one cost model of the scenario to be 100% accurate, i.e., we assume it correctly models the actual database costs. First we run our approach using this model. Then, we vary the costs of the CPU, IO and NET parameters independent from each other and run our approach again.

The results are summarized in Figure 8. The x-axis graphs the relative increase and decrease of the parameter value, and the y-axis graphs the relative cost reduction. If CPU costs are underestimated more than 75% or overestimated more than 50%, results of our approach will be worse. The modeling of the IO costs influences the results if costs are underestimated more than 50%. Misestimating the network costs NET has surprisingly little impact. This can be explained when looking at the heuristic for generating initial populations. Since this heuristic places MVs on nodes that query or update them frequently, the genetic algorithm starts with a population that minimizes network transfers even if they are not assigned with any costs. Observe that even in the presence of strongly deviating cost parameters, our experiments never proposes MVs that produce higher costs than they save. Thus, our approach is robust towards misestimated cost parameters.

6. CONCLUSION

Materialized views (MV) promise to improve the performance of queries in centralized and distributed database scenarios. However, considering distributed settings where numerous nodes with different resource constraints issue complex queries and updates at different rates result in a huge

solution space and non-monotonic, NP-hard optimization problems. Therefore, greedy algorithms or brute-force approaches cannot be applied directly, and all existing solutions we are aware of make prohibitive assumptions, e.g., no update costs or only one querying node.

In this paper we have introduced a practical approach that reduces the solution space in order to solve the view selection problem with a genetic algorithm. Therefore, we have restricted the set of queries and tables to be considered for materialization, and we have identified clusters of queries with similar predicates that can be supported with one specialized MV. In order to let the genetic algorithm converge quickly, we have generated initial sets of MV that are close to an optimal solution, and we have developed a selection function based on the similarity of queries.

The evaluation confirms that our practical approach is applicable for real world problems. In particular, we have shown for artificial settings and a large RFID scenario from retail that our approach scales well and proposes almost optimal sets of MVs. For example, our approach generates within 30 seconds a set of MVs comparable to the optimal solution, which we obtained after 10 hours processing time with an exhaustive search over the complete solution space.

7. REFERENCES

[1] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, and M. Syamala. Database Tuning Advisor for Microsoft SQL Server 2005. In *Proceedings of VLDB'04*, 2004.

[2] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In *Proceedings of VLDB'00*, 2000.

[3] E. Baralis, S. Paraboschi, and E. Teniente. Materialized views selection in a multidimensional database. In *Proceedings of VLDB'97*, 1997.

[4] A. Bauer and W. Lehner. On solving the view selection problem in distributed data warehouse architectures. In *Proceedings of SSDBM'2003*, 2003.

[5] D. Beasley, D. Bull, and R. Martin. An overview of genetic algorithms: Part 1, fundamentals. *University Computing*, 15(2):58–69, 1993.

[6] L. Bellatreche, K. Karlapalem, and M. Schneider. On efficient storage space distribution among materialized views and indices in data warehousing environments. In *Proceedings of CIKM'00*, 2000.

[7] R. G. Bello, K. Dias, A. Downing, J. James J. Feenan, J. L. Finnerty, W. D. Norcott, H. Sun, A. Witkowski, and M. Ziauddin. Materialized Views in Oracle. In *Proceedings of VLDB'98*, 1998.

[8] S. Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of PODS'98*, 1998.

[9] A. E. Eiben, R. Hinterding, and Z. Michalewicz. Parameter control in evolutionary algorithms. *IEEE Trans. on Evolutionary Computation*, 1999.

[10] H. Gupta. Selection of views to materialize in a data warehouse. In *Proceedings of ICDT'97*, 1997.

[11] H. Gupta and I. S. Mumick. Selection of views to materialize under a maintenance cost constraint. *Lecture Notes in Computer Science*, 1999.

[12] R. Hull and G. Zhou. Towards the study of performance trade-offs between materialized and

[13] H. Jiang, D. Gao, and W.-S. Li. Exploiting correlation and parallelism of materialized-view recommendation for distributed data warehouses. *Proceedings of ICDE'07*, 2007.

[14] D. Kossmann and K. Stocker. Iterative dynamic programming: a new class of query optimization algorithms. *ACM Trans. on Database Systems*, 2000.

[15] M. L. Lee, M. Kitsuregawa, B. C. Ooi, K.-L. Tan, and A. Mondal. Towards self-tuning data placement in parallel database systems. *Proceedings of SIGMOD'00*, 2000.

[16] W.-S. Li, D. C. Zilio, V. S. Batra, C. Zuzarte, and I. Narang. Load balancing and data placement for multi-tiered database systems. *Data Knowl. Eng.*, 62(3):523–546, 2007.

[17] T. Liu. Cost-based query optimization in a heterogeneous distributed semi-structured environment. In *The VLDB PhD workshop*, Vienna, Austria, 2007.

[18] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for local queries. In *Proceedings of SIGMOD'86*, 1986.

[19] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. *Proceedings of SIGMOD'01*, 2001.

[20] R. Pottinger and A. Y. Levy. A scalable algorithm for answering queries using views. In *Proceedings of VLDB'00*, 2000.

[21] A. Shukla, P. Deshpande, and J. F. Naughton. Materialized view selection for multidimensional datasets. In *Proceedings of VLDB'98*, 1998.

[22] H. Wang, M. Orlowska, and W. Liang. Efficient refreshment of materialized views with multiple sources. In *Proceedings of CIKM'99*, 1999.

[23] L. Weiss Ferreira Chaves, E. Buchmann, and K. Böhm. Tagmark: Reliable estimations of RFID tags for business processes. *Proceedings of KDD'08*, 2008.

[24] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in data warehousing environment. In *Proceedings of VLDB'97*, 1997.

[25] W. Ye, N. Gu, G. Yang, and Z. Liu. Extended derivation cube based view materialization selection in distributed data warehouse. In *Proceedings of WAIM'05*, 2005.

[26] C. Zhang and J. Yang. Genetic algorithm for materialized view selection in data warehouse environments. In *Proceedings of DaWaK'99*, 1999.

[27] J. Zhou, P.-A. Larson, and H. G. Elmongui. Lazy maintenance of materialized views. In *Proceedings of VLDB'07*, 2007.

[28] J. Zhou, P.-A. Larson, J. Goldstein, and L. Ding. Dynamic materialized views. *Proceedings of ICDE'07*, 2007.

[29] D. C. Zilio, C. Zuzarte, G. M. Lohman, H. Pirahesh, J. Gryz, E. Alton, D. Liang, and G. Valentin. Recommending Materialized Views and Indexes with IBM DB2 Design Advisor. In *Proceedings of ICAC'04*, 2004.