A maintenance centric approach to the view selection problem

Ray Hylock a, *, Faiz Currim b

a Interdisciplinary Graduate Program in Informatics, University of Iowa, 200 Hawkins Drive, 281 MRF, Iowa City, IA 52242, United States
b Department of Management Information Systems, University of Arizona, Tucson, AZ 85721, United States

A R T I C L E   I N F O

Article history:
Received 17 October 2012
Accepted 5 March 2013
Available online 20 March 2013

Keywords:
Data warehouse
Optimization
Heuristics
VSP

A B S T R A C T

The View Selection Problem is an optimization problem designed to enhance query performance through the pre-computation and storage of select views given resource constraints. Assuring the materialized views can be updated within a reasonable time frame has become a chief concern for recent models. However, these methods are crafted simply to fit a solution within a feasible range and not to minimize the resource intensive maintenance process. In this paper, we submit two novel advances in terms of model formulation and solution generation to reduce maintenance costs. Our proposed model, the Minimum-Maintenance View Selection Problem, combines previous techniques to minimize and constrain update costs. Furthermore, we define a series of maintenance time reducing principles in solution generation embodied in a constructor heuristic. The model and constructor heuristic are evaluated using an existing clinical data warehouse and state-of-the-art heuristics. Our analysis shows our model produces the lowest-cost solution relative to extant models. Also, they indicate algorithms seeded with our constructor heuristic to be superior solutions to all other methods tested.

© 2013 Elsevier Ltd. All rights reserved.

1. Introduction

In business and clinical organizations, information is stored in multiple, independent, and heterogeneous data sources. Business analysts pose complex queries to these information sources to identify, e.g., trends; typically, this is done through decision support systems (DSSs) and online-analytical processing (OLAP) queries. Likewise, healthcare researchers and clinicians use clinical decision support systems (CDSSs) to mine healthcare data, leading to improvements in the cost, efficiency, and quality of patient care. A data warehouse is essentially a data repository that collects and stores data from the aforementioned information sources; functioning even when the original sources are unavailable. The goal is to create a single, integrated system by extracting, transforming, and loading (ETL) relevant information into a consistent view [1–6]. Briefly, the relational structure of a data warehouse consists of a central table, called the fact table (denoted as $F$), to which, in most cases, all other tables, dimensions ($D$), connect—known as a star-schema design [4]. The fact table stores tuples in the form of dimension keys and values for aggregation. The dimensions function essentially as dictionaries, although more complex dimensional designs exist (see Kimball and Ross [7], Sen and Sinha [8], Gardner [4], Inmon [5] for more details). Since data warehouses are primarily intended for decision support, they are optimized for retrieval instead of routine transaction processing. More formally, a data warehouse can be characterized as a subject-oriented, integrated, nonvolatile, time-variant collection of data used to support decision making. Each characteristic is as follows [4,8,9]:

- **Subject-oriented:** The data warehouse is organized around a key subject known as the fact table (e.g., customer sales or patient encounters).
- **Integrated:** All data in the warehouse conforms to a single canonical standard. For example, if one source...
stores male and female as “male” and “female” and another as “m” and “f”, one representation is chosen and the other(s) converted. This ensures data integrity and improves query performance.

- Nonvolatile: The data is generally loaded and refreshed in a single process and is not editable by end users.
- Time-variant: Data is stored as a time series so one can identify, e.g., trends.

Fig. 1 illustrates a typical data warehouse architecture. Breaking it down, the monitors observe the information sources and notify the integrator when a change occurs. The integrator, during allotted times, amalgamates data from the information sources into the data warehouse. At the apex are the users who interface directly with the data warehouse instead of the individual information sources [7,10–12].

Decision support queries are generally quite complex and time consuming to perform. A commonly used technique to reduce the overall query response time is to precompute and store, in the form of materialized views, the most frequently used and “beneficial” queries in a data warehouse. Materialized views can significantly improve the speed of the analytical process; from hours or even days to seconds or minutes. Therefore, one of the most important issues in data warehousing is the selection of views to materialize known as the View Selection Problem (VSP) [1,2,13,14,15].

In this paper, we will discuss the expansion of two VSP components with a view maintenance bias. Although attempts have been made to address the minimization and constraining of update times, none have sought to minimize these costs while confining their expansion. As the cost of storage devices decrease, time not space should be the bounding principle. However, simply fitting a problem within a specified update window does little to impact the maintenance process. Therefore, we propose a new VSP model which minimizes and bounds update times and a constructor heuristic with the express intent of driving down update costs with little, if any, effect on query performance. Our model, the Minimum-Maintenance View Selection Problem (MMVSP), combines the Space-Constrained VSP [13,15] and the Maintenance-Cost VSP [16] to both constrain and minimize update costs. The constructor heuristic uses patterns from high quality solutions identified from current and previous research to fabricate “good” initial solutions used as seeds by other algorithms. We evaluate MMVSP by comparing it with the two previously mentioned models and the constructor heuristic against state-of-the-art VSP heuristics using an existing clinical data warehouse [17]. These algorithms include random search, first and best improvement, simulated annealing, genetic algorithms, and genetic local search—each discussed in detail in Section 4.

The rest of the paper is organized as follows. Section 2 provides a brief summary of previous work in VSP modeling, graphical representations of candidate views, and heuristics. In Section 3, a more thorough background of the VSP is given along with our proposed extension. Section 4 describes the algorithms used. Section 5 covers the experimental setup while Section 6 presents our findings. Finally, we wrap up with our concluding remarks and plans for future research in Section 7.

2. Previous work

2.1. Mathematical models

There are three main VSP formulation categories presented in literature, each with two sub-formulations: (1) unbounded (UVSP) [13,18–30], (2) space-constrained (SCVSP) [10,13,15, 19,22,31–39], and (3) update-time-constrained (MCVSP) [10,16,34,40–45]. The description of each will be discussed in this section with a more formal mathematical definition presented in Section 3.

The first unbounded version minimizes only the query cost (Model (1)) [13,19,22]. The second adds the update cost to the objective function (Model (2)) [13,18,20,21,23–30]. Neither is constrained by space or time. These formulations are the most frequently used of the three since 2005 in part because of the ease of implementation. However, for even moderately sized data sets UVSP is not feasible as space, time, or both impacts the solution space.

SCVSP is designed to minimize query response time plus the update cost (or portion of) subject to a space constraint (Model (4)) [10,15,19,22,32–36,38,39]. The most used formulation since the conception of the VSP, this has the benefit of minimizing, as a piece of the objective function, the update cost which is generally measured in terms of an order of magnitude (or more) greater than the query cost. However, the update cost is unbounded and thus may not be realizable. A subset of the literature removes the update cost entirely (Model (3)) [13,31,37]. Thus, update time may vary wildly with only slight improvements to the objective function and, as with the previous, may still yield unrealistic maintenance times.

The initial definition of the MCVSP minimized the query response time with a constraint on update time (Model (5)) [10,16,40–42,45]. By having the update time bound, it assures that we can refresh the views in a specified time frame. Gupta and Mumick [16] contend that the SCVSP is a special case MCVSP where update costs are static and thus they omit the space constraint.
However, query and update times may not be related so some have explicitly stated the space constraint (Model (6)) [34,44,43,35,37,39,46–48]. Proposed by Harinarayan et al. [13], the cube lattice is a directed acyclic graph (DAG) $L$ whose nodes represent views $\mathcal{V}$ (or queries) with edges $\mathcal{E}$ corresponding to partial orderings between the views. That is, if there is a path from view $u$ to a view $v$, then queries on $v$ can also be answered or updated by $u$ (i.e., $v \leq u$). This relationship can be seen from top-down with larger, more generalized views (e.g., $u$) higher up in the lattice and smaller, more specific views (e.g., $v$) further down until we reach the base relations at the bottom. The cube lattice can contain any set of views that one wishes to consider for materialization. However, since the number of possible views is exponential in terms of dimensions, the cube lattice is generally formed by queries found in the query log. The benefit of this approach is the utilization of materialized views to directly answer non-materialized and update materialized views. However, except for building up from the set of base relations, the cube lattice does not support views lower in the lattice constructing those higher up. Moreover, the root node(s) in the lattice, must be materialized, according to Harinarayan et al. [13], as it cannot be constructed otherwise (they preclude direct construction from dimension and fact tables) and is the foundation for all update costs. Since 2005, the most implemented graphical representation for the VSP is the Multi-View Processing Plan (MVPP) [20,21,23,24,26–30]. As defined by Yang et al. [26], the MVPP is a DAG representing a query processing strategy. The root (source) nodes are the queries themselves, the leaf (sink) nodes are the base relations, and all other intermediate nodes are selection/projection/join views that contribute to the construction of a given query in a bottom-up manner. Each query has its own processing plan. When all plans are generated, they are merged into a single MVPP. All views in the resulting MVPP are candidates for materialization. These candidates can be used to either answer or update a query or set of queries. The drawback to MVPP is it does not allow for partial ordering. That is, an ancestor view cannot be used to answer or update a descendant query.

2.2. Graphical representations

One of the earliest generated VSP representations is the Data Cube Lattice (simply cube lattice) [13,18,25,35,37,39,46–48]. Proposed by Harinarayan et al. [13], the cube lattice is a directed acyclic graph (DAG) $L$ whose nodes represent views $\mathcal{V}$ (or queries) with edges $\mathcal{E}$ corresponding to partial orderings between the views. That is, if there is a path from view $u$ to a view $v$, then queries on $v$ can also be answered or updated by $u$ (i.e., $v \leq u$). This relationship can be seen from top-down with larger, more generalized views (e.g., $u$) higher up in the lattice and smaller, more specific views (e.g., $v$) further down until we reach the base relations at the bottom. The cube lattice can contain any set of views that one wishes to consider for materialization. However, since the number of possible views is exponential in terms of dimensions, the cube lattice is generally formed by queries found in the query log. The benefit of this approach is the utilization of materialized views to directly answer non-materialized and update materialized views. However, except for building up from the set of base relations, the cube lattice does not support views lower in the lattice constructing those higher up. Moreover, the root node(s) in the lattice, must be materialized, according to Harinarayan et al. [13], as it cannot be constructed otherwise (they preclude direct construction from dimension and fact tables) and is the foundation for all update costs. Since 2005, the most implemented graphical representation for the VSP is the Multi-View Processing Plan (MVPP) [20,21,23,24,26–30]. As defined by Yang et al. [26], the MVPP is a DAG representing a query processing strategy. The root (source) nodes are the queries themselves, the leaf (sink) nodes are the base relations, and all other intermediate nodes are selection/projection/join views that contribute to the construction of a given query in a bottom-up manner. Each query has its own processing plan. When all plans are generated, they are merged into a single MVPP. All views in the resulting MVPP are candidates for materialization. These candidates can be used to either answer or update a query or set of queries. The drawback to MVPP is it does not allow for partial ordering. That is, an ancestor view cannot be used to answer or update a descendant query.

Similar to the MVPP is the implementation of Roussopoulos’ Logical Access Path Schema [49] in the form of AND, OR, and AND–OR view graphs [10,15,16,33,34,50]. These graphs specify the views necessary to build a query from the base relations up. As with MVPP, the benefit of this approach is the assembling of materialized views to directly answer non-materialized and update materialized ancestor views with the added benefit of not needing to materialize, by default, the root cuboid. The drawback is it does not allow for partial ordering. An OR view graph is a special case view graph where AND arcs (multiple, intermediate views required to update or answer a view) do not exist. That is, there are multiple, independent views which can be used to construct a view. The difference between MVPP, OR, and AND–OR view graphs is the OR notation which allows a single query to be answered and updated from multiple paths (AND view graphs are essentially MVPPs).

2.3. Heuristics

A number of heuristics have been presented in literature to solve the VSP. As shown by Karlo and Mihail [46], unless $P=NP$, the VSP is essentially inapproximable for general partial orders. Hence, research in this area primarily focuses on special cases and experimental analysis of heuristics (additional complexity results can be found in Abiteboul and Duschka [51]). In their seminal paper on the View Selection Problem, Harinarayan et al. [13] formalized the cube lattice and proposed two greedy heuristics: Greedy and Benefit Per Unit Space (BPUS). These algorithms work as follows. Greedy iterates over every view in the remaining candidate set (which is initialized to every view minus the root), selecting for materialization the view with the greatest benefit. The benefit of a view $v$ is simply defined as the current objective function value given a set of materialized views $\mathcal{M}$ minus the objective function value including $v$ in $\mathcal{M}$. The process continues until $k$ views are selected. BPUS operates almost identically to Greedy except instead of selecting $k$ view, there is an allotted space $S$. As with Greedy, each candidate view is added to the solution and the new objective function value is subtracted from its previous, but in addition, that value is then divided by the size (in terms of tuples) of the added view; thus the benefit per unit space. This is done for each view and the one with the greatest benefit is selected and added to the materialized set. The process is continued until the allotted space is full. As every view in the remaining candidate set is visited per iteration, Greedy and BPUS do not scale well. They prove both Greedy and BPUS are at least $(1-1/e)\approx 63\%$ of the optimal solution. Uchiyama et al. [25] constructed a greedy algorithm based on BPUS. Their heuristic, the Progressive View Materialization Algorithm (PVMA), computes, instead of the benefit per unit space, the benefit per blocking factor [1,52] of a view. Thus, PVMA has the effect of minimizing the cost of physical data access. Their algorithm builds a solution monotonically until no view can be added to increase the overall profit. They show when the number of dimensions is greater than 3, PVMA continually out performs BPUS.

---

1. Let view $a$ be updated or answered from view $b$ (i.e., $a \leq b$). If $a$ were allowed to aid in the construction of $b$, then a cycle in the graph would occur.

2. Let view $c$ require views $a$ and $b$ for construction. If view $d$ requires view $c$ and $e$ for construction, then $c$ is a partial order of $d$ (i.e., $c \leq e$). However, if $c$ is allowed to be answered or updated by $d$ then a cycle in the graph would occur.
Gupta and Mumick [10,16] first proposed the use of update time as the constraining factor instead of space in their optimization function. Prior models grew monotonically as views were added, making greedy algorithms easier to construct. Update costs, however, are non-monotonic and thus existing algorithms can produce arbitrarily bad solutions. To combat this new class of problem, they proposed two new algorithms: the Inverted-Tree Greedy and an A*Heuristic. The Inverted-Tree Greedy algorithm is specifically designed to function on OR view graphs as any subset of connected nodes are trees (with inverted edges) by definition. They propose the benefit per unit of effective maintenance-cost of adding a set of views $T$ to $M$ which is the change in objective function value divided by the update cost with $T$. The algorithm iterates over every $T$, compares the benefit of adding $T$ to the current best subset, storing the larger of the two. This process repeats until the update time constraint is saturated. For OR view graphs, they prove that their algorithm is at least 63% of the optimal solution. However, there are an exponential number of subtrees, so this algorithm does not scale well. The A* Heuristic searches an AND–OR view graph for an optimal solution where each node represents a candidate solution [53]. The algorithm produces an optimal solution, however it is extremely time consuming as its run-time is exponential. They compared these two heuristics only to one another, making a general statement as to their overall performance impossible.

Several papers have been conceived in which Genetic Algorithms (GA—see Section 4.6 for more details) have been used to solve the VSP. Zhang and Yang [30] compared GA to a simple greedy algorithm (which it outperformed) to show GA can be used effectively on the VSP. Similarly, Horng et al. [33] use a GA variant called Genetic Local Search (GLS). Their GLS builds an initial population through an undisclosed constructor heuristic and performs a local search on each subsequent generation. No comparison with any other algorithm is made. The simple goal of the paper, as with Zhang and Yang [30], was to prove the viability of genetic algorithms on the VSP. Lawrence [35] takes a different approach to genetic algorithms by way of using Multi-Objective Genetic Algorithms (MOGAs). The objective function is broken into two components (query and update costs) that are solved independently. The advantage of a multi-objective implementation over a single, linearly combined one is the ability to perform pairwise comparisons of solutions in terms of query and update cost independent of one another. A solution is said to dominate another if both the query and update costs are more favorable. They compare their MOGA to BPUS and conclude MOGAs are competitive with the greedy algorithm and thus deserve further study.

Kalnis et al. [34] experimented with four randomized algorithms: Iterative Improvement (II Section 4.3), Simulated Annealing (SA Section 4.4), Random Search (RS Section 4.2), and Two-Phase Optimization (2PO combines the faster converging II with the higher quality solution finding SA). The goal was not to compare the algorithms so much as show the robustness of randomized search algorithms within the VSP. They did, however, perform a run-time analysis experiment between Greedy, SA, and 2PO with Greedy being the fastest by a substantial margin. Outside of the case where the allowable update time was 100% of the total required for all views, 2PO was superior. The rest of their experiments were direct comparisons between the four proposed algorithms. These results show that SA and 2PO are generally better.

Derakhshan et al. [20] experimented with SA on the data warehouse schema provided by Yang et al. [27]. A standard GA and “heuristic” (for which they provided no details) were compared to their SA with SA winning handily. Derakhshan et al. [21] later parallelized this implementation (PSA). Comparing it to the single-machine SA and “heuristic”, PSA continually outperformed the competition as long as the number of threads was greater than 1 (as 1 is SA).

3. The view selection problem and data cube lattice

The VSP is an NP-Complete problem dealing with the selection of views to materialize in order to efficiently answer queries posed to a data warehouse [10,13,15,16,18,31,37,39,46,54,55]. We conceptualize our VSP instance, and therefore all proceeding definitions, following the cube lattice description discussed in Section 2.2 with $L$ representing the cube lattice, $V$ the views (vertices), and $E$ the partial orderings (edges); our reasoning resolves to simple $JOIN$ versus $PROJECTION$ operations. To construct an ancestor view, a $JOIN$ must be performed using the fact table and all relevant views/base relations. This is an expensive process (see Section 3.4 on join cost estimation). The further down one goes from the ancestral view in question, the more expensive the joins become until base relation construction is reached. On the other hand, the assembly of a descendant view (i.e., top-down), requires a $PROJECTION$ operation which runs in time proportional to the size of the relation from which we are extracting said view. Given a view $v \in V$, the following relationships can be defined:

$$parent(v) = \{u|v \subseteq \Omega; \forall x, v = x \wedge x \subseteq \Omega\}$$

$$siblings(v) = \{u|\text{Level}(u) = \text{Level}(v) \wedge parent(v) = parent(u)\}$$

$$children(v) = \{u|u \subseteq \Omega; \exists x, u \subseteq x \subseteq \Omega\}$$

$$ancestors(v) = \{u|v \subseteq \Omega\}$$

$$descendants(v) = \{u|u \subseteq \Omega\}$$

Fig. 2 is an example cube lattice from a clinical data warehouse using 4 of its 15 dimensions: Interventions (I), Outcomes (O), Diseases (D), and Patients (P) [17]. There are $2^P$ or 16 possible views to materialize [3,4,11,56]. Fig. 3 shows the effect hierarchies have on the lattice. A hierarchy is a definition of functional dependencies in a table, delineating a path of granularity. For example, dates can contain a hierarchy with the following four levels: day $\rightarrow$ month $\rightarrow$ quarter $\rightarrow$ year (a week can span months, quarters, and years so it is a separate hierarchy). As shown in Fig. 3, there exists a dimension, Outcomes, with a hierarchy containing two levels: Label $\rightarrow$ Class (the middle levels in the Nursing Outcomes Classification hierarchy [57]). Since Label and Class are from the same dimension, they cannot co-define a view. This also allows for directed edges to connect siblings and thus, the leading
3.1. View selection problem formal definition

The view selection problem, can be formally defined as follows. Given a relational data warehouse schema $R$, containing a set of dimensions $D = (d_1, d_2, \ldots, d_k)$, maximum storage space $S$, maximum update cost $U$, and a workload of queries $Q = (q_1, q_2, \ldots, q_m)$, select a set of views $M \subseteq V$ to materialize, whose combined size is at most $S$ (Models (3), (4), (6) and (7)) and, for Models (5)–(7), whose combined update cost is at most $U$. Each view $v \in V$ has an associated query frequency $f_v$, an update frequency $g_v$, and a reading cost $r_v$ (the number of tuples in $v$), and a cost for building from the base tables $b_v$ (derivation described in Section 3.4). For the presented experiments, the resulting sizes of each query is known beforehand, but if unknown (e.g., too expensive to calculate), view-size estimation can be used [13,58–60]. The cost of answering a query $q$ corresponding to a view $v \in V$ is

$$q(v, M) = \begin{cases} r_v & \text{if } v \in M \\ \min(r_v, b_v) & \text{if } v \notin M \end{cases}$$

We implement the linear cost model provided by Harinarayan et al. [13] so, if $v \in M$, the query cost is the number of tuples in $v$ ($r_v$). If $v \notin M$, then the query cost is the minimum between the smallest materialized ancestor ($r_v$ if $v \in M$) or the cost of building from the base tables ($b_v$).

The cost of updating a materialized view $v \in M$ is equal to the number of changes in the view(s) from which $v$ is updated. This, according to Lawrence [35], Yousri et al. [28], and Kalnis et al. [34], can be expressed as a fixed percentage $p$ of the view(s). Therefore, the cost of updating $v \in M$ is

$$u(v, M) = \begin{cases} p(\min(r_v, b_v)) & \text{if } v \in M \\ 0 & \text{if } v \notin M \end{cases}$$

If $v \notin M$, then the update cost is the same as the query cost if $v \notin M$ times the percentage $p$. If $v \in M$, then there is no update cost.

3.2. View selection problem models

Following the definitions in Section 3.1, the mathematical formulations for the VSP categories presented in Section 2.1 can now be described. Note, $q(v, M)$ and $u(v, M)$ are constructed for a top-down cube lattice approach (per Section 3) and must be reformulated for MVPPIs and View Graphs if so desired.

Models (1) and (2) represent the UVSP without and with update time minimization respectively. As neither have constraints, the resulting solution may or may not be realizable in terms of space, update time, or both. Therefore, we do not consider this model further.

Minimize $\sum_{v \in V} f_v q(v, M)$

Subject to $\sum_{v \in M} r_v \leq S$

(3)

Minimize $\sum_{v \in V} f_v q(v, M) + \sum_{v \in M} g_v u(v, M)$

Subject to $\sum_{v \in M} r_v \leq S$

(4)

Unlike the previous, the MCVSP variations are based on the number of constraints. The initial, proposed model only supports an update time bound (Model (5)) while subsequent implementations add a space constraint (Model (6)).

Minimize $\sum_{v \in V} f_v q(v, M)$

Subject to $\sum_{v \in M} g_v u(v, M) \leq U$

(5)

Minimize $\sum_{v \in V} f_v q(v, M)$

Subject to $\sum_{v \in M} r_v \leq S$

$\sum_{v \in M} g_v u(v, M) \leq U$

(6)

Definition 1 (Monotonic Function). A function $f$ is said to satisfy the monotonicity property if it is entirely non-
increasing or non-decreasing (i.e., its first derivative does not change signs).

The size of the materialized set $S(\mathcal{M}) = \sum_{v \in \mathcal{M}} r_v$ is monotonic as for each view $v$ added to the set of materialized views $\mathcal{M}$, $S(\mathcal{M} \cup \{v\}) \geq S(\mathcal{M})$ as $r_v \geq 0$.

**Definition 2 (Non-Monotonic Function).** A function $f$ is said to satisfy the non-monotonicity property if it is both increasing and decreasing (i.e., its first derivative does change signs).

The update cost of the materialized set $U(\mathcal{M}) = \sum_{v \in \mathcal{M}} g_v u(v, \mathcal{M})$ is non-monotonic as for each view $v$ added to the set of materialized views $\mathcal{M}$, $U(\mathcal{M} \cup \{v\})$ could be greater than, less than, or equal to $U(\mathcal{M})$. That is, if $v = x_v$ ($v$ is a materialized ancestor of $x$) for some view $x \in \mathcal{M}$, then $U(\mathcal{M} \cup \{v\})$ may be greater than, less than, or equal to $U(\mathcal{M})$ depending upon the relationship between $u(v, \mathcal{M} \cup \{v\}) + u(x, \mathcal{M} \cup \{v\})$ and $u(x, \mathcal{M})$.

### 3.3. Minimum-maintenance view selection problem

While Model (4) (SCVSP with update cost minimization) seeks to minimize both query and update costs, it does not constrain maintenance time and, therefore, may present unrealizable solutions. Model (6) (MCVSP with the space constraint) was crafted in direct response to this issue. However, as data and therefore view sizes increase, too does the amount of time and resources required to update the materialized set. Therefore, the potential exists to combine these two methods to capitalize on their advantages.

The proposed MMVSP model minimizes the query cost plus a portion $w$ of the update cost subject to space and update time constraints; effectively combining Models (4) and (6). It is trivial to see that SCVSP is a special case of the MMVSP where $U = \infty$.

Minimize $\sum_{v \in \mathcal{V}} f_v q(v, \mathcal{M}) + w \sum_{v \in \mathcal{M}} g_v u(v, \mathcal{M})$

Subject to $\sum_{v \in \mathcal{M}} r_v \leq S$

$\sum_{v \in \mathcal{M}} g_v u(v, \mathcal{M}) \leq U$  \hspace{1cm} (7)

### 3.4. Estimating the cost of a join in tuples

Calculating the cost of a join is typically done in terms of the number of block accesses [1,2,31,61,62]. In this paper, however, the basis for comparison is in tuples; therefore, changes to any chosen model must be made.

We employ two methods for determining the cost of a **JOIN** with the minimum taken: (1) a variant of the **sort-merge join** cost model [1] and (2) a **hash-join** estimation function [1,62]. Sort-merge join (SMJ) requires both relations to be physically stored in sorted order based on the keys being used in the join operation. We assume each $x \in D$ is bulk loaded in order of their primary key attribute; a reasonable assumption in data warehousing. However, $F$ (or any other relation in a join consisting of a foreign key) cannot be stored in sorted order on every foreign key so there is an additional cost incurred for sorting. The original cost model is as follows. The cost of a sort-merge join equals $k_f + k_s + (js\|R \times S|/kfr)$ where $R$ and $S$ are the relations being joined, $k$ is the number of blocks used by the given table, the join selectivity $f = |R \cap S|/|R \times S|$, and $kfr = |block size in bytes/average number of bytes per tuple|$ is the blocking factor. Essentially, it estimates the cost of reading $R$ and $S$, matching their tuples, and writing the results (estimated by $js\|R \times S|/kfr$) back to the database for additional processing. In order to model the sorting requirements, $k_f$ (the unsorted table) is exchanged with $2k_f(1 + \log_{572}(ks))$ where $M$ is the number of available memory buffers [1]. Thus, the augmented cost is $2k_f(1 + \log_{572}(ks)) + k_s + (js\|R \times S|/kfr)$. This is easily converted to tuples as follows. Set $k_f = |R|$, $k_s = |S|$, and $kfr = 1$. Therefore, the resulting equation is $2|R(1 + \log_{572}(|R|)) + |S|(\log_{572}|R \times S|) = 2|R| + (1 + \log_{572}|R|)|S| + (\log_{572}|R|S|)$.

In order to compute $b^m_{p1}$, the fact table $F$ is joined with the first relation $x_1 \in X$: $E[F \bowtie x_1] = 2r_x(1 + \log_{572}(1 + r_x + r_{x_1})) + r_{x_1} + r_{F \bowtie x_1}$. To estimate the cost of $(F \bowtie x_1) \bowtie x_2$, $b^m_{p2} = E[F \bowtie x_1] + 2r_{F \bowtie x_1}. (1 + \log_{572}(1 + r_{F \bowtie x_1}) + r_{x_2} + r_{F \bowtie x_1 \bowtie x_2})$. This process is repeated for all $x_i \in X$ (see Eq. (8) for a general formula). Note, if updates are known, then a fraction $p$ of each relation (as with updates in Section 3.1) can be taken.

\[ b^m_{pi} = \sum_{i=2}^{k_f} E[F \bowtie x_i] + \sum_{i=1}^{k_f} 2r_{F \bowtie x_1}(1 + \log_{572}(1 + r_{F \bowtie x_1}))r_{x_i} + r_{F \bowtie x_1 \bowtie x_i} \]

where

$$\mathcal{J} = \begin{cases} \emptyset & \text{if } i = 1 \\ x_1 \cdots x_{i-1} & \text{otherwise} \end{cases}$$

and $X' = \mathcal{M} \cup \mathcal{B}$ is sorted in ascending order on $r_i$ while still maintaining JOIN feasibility.

**Example 1.** Compute $b^m_{SD}$. For $F =$ Patients (P) and Diseases (D). Let $|F| = 1,378,572$, $|P| = 654,578$, $|D| = 1,012,461$, $|F \bowtie P| = 1,378,572$, $|F \bowtie P \bowtie D| = 1,744,923$, and $M = 3$. Then, $E[F \bowtie P] = 2(1,378,572(1 + \log_{572}(1,378,572))) + 654,578 + 1,378,572 = 61,021,538$, $b^m_{SD} = 61,021,538 + 2(1,378,572(1 + \log_{572}(1,378,572))) + 1,012,461 + 1,744,923 = 122,767,310$ tuple reads.

Hash-joins (HJ) are the typical method of choice for query optimizers when dealing with larger relations and equi-joins. Basically, it takes the smallest $3$ of the two relations (either a base relation or result set—called the inner relation), creates a hash table based on the join attribute(s), and probes it for each tuple in the larger (outer) table by hashing its respective join attribute and performing a direct comparison. This, in the worst case, can be estimated using recursive partitioning incurring a cost of $2(k_f + k_s)(|log_{572}(ks)| - 1) + k_f + k_s + (js\|R \times S|/kfr)$. For $2(k_f + k_s)(|log_{572}(|S|)| - 1) + k_f + k_s$ is the number of block transfers and $(js\|R \times S|/kfr)$ is the cost of writing the results back to the disk [162]. Following the same

---

3 The relation with the fewest tuples is chosen as it has a greater probability of fitting into memory, thus greatly enhancing performance.
substitutions presented above yields $2(|R| + |S|)|\log_{M-1}(|S| - 1)| + |R| + |S| + (|R \bowtie S|)$ where $S$ is the minimum relation; Eq. (9) represents its general form

$$b_v^H = \sum_{i=1}^{K} 2\left(r_{f SM} + r_{x_i}(|\log_{M-1}(v_i) - 1|) + r_{s SM} + r_{x_i} + r_{f SM} / |\text{s}|ight),$$

where $v = \min(r_{f SM}, r_{x_i})$.

**Example 2.** Compute $b_v^H$ for $v = PD$ adhering to Example 1. $E[|RF \bowtie P|] = 2(1,378,572 + 654,578)|\log_{2}(654,578 - 1)| + 1,378,572 + 654,578 + 1,378,572 = 80,671,422$. Therefore, $b_v^H = 80,671,422 + 2(1,378,572 + 1,012,467)$

![Equation](image)

**4. Algorithms**

Comparison heuristics were drawn from the VSP literature and range from purely random to evolutionary approaches with hill-climbing. These algorithms were chosen based on either their speed and/or solution quality. The employed methods are: Random Search (RS) [34], First and Best Improvement (FI and BI respectively—iterative improvement algorithms) [23,24,34], Simulated Annealing (SA) [23,24,34,41], Genetic Algorithms (GA) [29,30,32,33,36,38,41,43–45], and Genetic Local Search [32,33]; each explained in their respective section. GA and GLS were both seeded with the results from RS, BI, and our constructor (C) heuristic. Additionally GLS performs either a single or double removal prior to replacement (GLS1 and GLS2 respectively). For the smaller datasets, direct comparisons were performed with an optimal algorithm while larger ones necessitated heuristic-to-heuristic evaluations.

**4.1. Global methods**

There are two main global methods used by the presented algorithms; SetCosts and ComputeOF. The SetCosts method determines the query and update costs based on the $q(v, M)$ and $u(v, M)$ definitions in Section 3.1. This algorithm runs in $O(|\mathcal{E}|)$ for each view as it amounts to a breadth-first search with directed edges. The ComputeOF method calculates the objective function value based on the model used; it runs in $O(|\mathcal{V}|)$ as it need only visit each node in the lattice once.

**4.2. Random search**

Random Search (RS) simply selects a random number (line 3) of random views (lines 4–8), trying to “stumble” across a good solution. If this configuration is feasible (line 10), the solution is returned, if not, the process repeats itself until a feasible solution is found (line 11). The benefit of this algorithm is speed as it takes little time to find a single, feasible solution. However, since it is simply picking a random solution from an extremely large solution space, the quality is generally quite poor [63]. Algorithm 1 depicts the implemented random search algorithm, having an execution time of $O(|\mathcal{V}|^2)$.

**Algorithm 1. Random Search ($\mathcal{L}$).**

**input:** cube lattice $\mathcal{L}$

**output:** $O_{F}$

1. $O_{F} \leftarrow 0$ // objective function value
2. $s \leftarrow \emptyset$ // solution
3. $r$ = random number between 1 and $|\mathcal{V}|$
4. for $i = 1$ to $r$ do
5. $k$ = random number between 1 and $|\mathcal{V}|$
6. $s = s \cup \{ k \}$
7. $V = \mathcal{V} \setminus \{ k \}$
8. end for
9. SetCosts($M, p$)
10. if valid then $O_{F} \leftarrow \text{ComputeOF}((M))$
11. else RandomSearch($\mathcal{L}$)
12. end if

**4.3. Iterative improvement**

A simple hill-climbing algorithm, Iterative Improvement (II) seeks only to improve its current condition; downhill moves only. Beginning with a random state, it searches for a nearby solution (neighbor) with a better objective function value and moves there. This comparison can also be done within a group (neighborhood) where the best solution in the neighborhood is selected not simply the first one located. After some termination criteria is met, the best value amongst all points is returned [64–68]. For this paper, a neighborhood is defined as follows. Let $\Phi$ be the solution space. For each solution $i \in \Phi$, a set $N(i) \subseteq \Phi$ is defined as a neighborhood and $j \in N(i)$ is defined as a neighbor of $i$ where $i \neq j$. Both FI and BI receive three parameters: the number of neighborhoods $N$, the number of neighbors in each neighborhood $n$, and the amount of variation about the size of the initial solution $\sigma$. For these experiments $j \in N(i)$ where $\max(0, |i| - \sigma) \leq |j| \leq \min(|\mathcal{V}|, |i| + \sigma)$; that is, the size of the neighbors in $N(i)$ are within $\sigma$ of the size of the initial solution $i$, subject to a bound of $[0, |\mathcal{V}|]$. For example, if $|i| = 15$ and $\sigma = 2$, then the size of any $j$ can be between 13 and 17 (assuming $|\mathcal{V}| \geq 17$). This way, the neighborhood is more diverse in hope of avoiding regions with few, if any, feasible solutions.

The improvement algorithm, as described by Algorithm 2, implements both first and best improvement. For first improvement, it randomly generates a feasible solution $i$ (for each of the $N$ neighborhoods), then it randomly generates up to $n - 1$ feasible solutions $N(i)$ until one better than $i$ in terms of objective function value is found; then it continues on to the next neighborhood. The overall best solution found is then returned. There are two termination criteria, one for the neighborhoods and one for the

---

1. It might require a few iterations to produce a feasible solution, but the presented algorithm is more efficient than adding a view and setting the lattice which yields a run-time of $O(|\mathcal{V}|^2)$. Algorithm 1 is more efficient.
neighbors, consisting of \( k \) times \( N \) and \( n \) respectively, to make ensure the algorithm terminates within a reasonable amount of time. Line 9 is intended to somewhat restrict the size of a neighborhood to between 2 and \( |V|−2 \) to allow for a reasonable number of neighbors to search. For example, for \(|V| = 30\), a neighborhood of size 0 or 30 yields only 1 solution (i.e., \( V(0) = V(30) = 1 \)) and a neighborhood of size 1 or 29 has 30 solutions; whereas \( 2 ≤ N ≤ 28 \) has a minimum of 435 combinations. If smaller combinations are allowed, then we may not find any feasible solutions and/or restrict our neighborhood searches to too few in order to realize any benefit from this search mechanism.

For best improvement all \( n−1 \) feasible solutions \( j \) are consulted before moving on to the next neighborhood. The method \texttt{CalculateImp} (line 23) is responsible for distinguishing between FI and BI and validating solutions. This algorithm, independent of first or best improvement,\(^5\) runs in \( O(k^2N|V|/|E|) \).

**Algorithm 2.** Improvement(\( L, σ, N, n, \text{type}, k \)).

**Input:** the cube lattice \( L \), variation \( σ \), neighborhoods \( N \), neighbors \( n \), type FI or BI, and termination criteria \( k \).

**Output:** \( O_{F spoiled} \) and \( M_{best} \).

1. \( O_{F}=\emptyset \) // best objective function value
2. \( M_{best}=() \) // best solution
3. \( S_{set}=\emptyset \) // the set of solutions already tried
4. \( N_{term}=0 \) // termination iterator for \( N \)
5. for \( i = 1, N \) do // termination iterator for each neighborhood
   6. if \( N_{term} > kN \) exit end if
   7. \( n_{term}=0 \) // termination iterator for \( n \)
   8. \( found=false \) // FI found
   9. \( r_{beg}=\text{random number between } 2 \text{ and } |V|−2 \)
   10. if \( r_{min}==\omega \text{ and } r_{beg}−σ < 0 \) then \( r_{new}=\text{end if} \)
   11. if \( r_{max}==\omega \text{ and } r_{beg}+σ > |V| \) then \( r_{max}=|V| \) end if
   12. for \( j = 1, n \) do // for each neighbor
      13. if \( N_{term} > kN \) exit end if
      14. \( s=\emptyset \) // solution
      15. \( V_{temp}=V_{j} \) // copy of vertices
      16. \( r=\text{random number between } r_{max} \text{ and } r_{min} \)
      17. for \( m = 1, r \) do
         18. \( l=\text{random number between } 1 \text{ and } |V_{term}| \)
         19. \( s=\text{add } V_{l} \text{ to } V_{temp} \)
         20. \( V_{temp}=\text{add } V_{l} \text{ to } V_{temp} \)
      21. end for
      22. if \( S_{set} ≠ \emptyset \) then
         23. if \( \text{CalculateImp}(s, \text{type}) \) valid then \( j=j−1 \) end if // add one back to \( j \)
      24. else
         25. \( j=j−1 \) // add one back to \( j \)
      26. end if
      27. \( n_{term}=n_{term}+1 \)
      28. end for
      29. \( N_{term}=N_{term}+1 \)
   10. end for
   11. \( r_{new}=\text{end if} \)

4.4. Simulated Annealing

*Simulated Annealing* (SA) is similar to FI only allowing *uphill* moves with decreasing probability (\( V \)). SA will transition between two states if either the neighboring node produces a higher quality solution (i.e., a downhill move) or probabilistically based on a *cooling schedule* (i.e., an uphill move). The cooling schedule consists of an initial value (called the *temperature*) which progressively decreases with each iteration (or set of iterations), thus allowing fewer and fewer uphill moves in the latter stages of execution. The probability of selecting any move at time \( t \) is \( \min(1,\exp(-(OF(j)−OF(best))/T_{j}t)) > R[0,1] \) where best is the current solution, \( j \) is the current neighbor under consideration, \( T_{j} \) is the temperature at time \( t \), and \( R[0,1] \) is a uniform random number. This addresses an issue found with II of getting trapped in local minima early, preventing a more thorough search of the solution space [69]. The algorithm is virtually identical to Algorithm 2, with the following modification. First, based on FI principles, if \( i \) is less than \( j \), then the algorithm transitions between them, else probabilistically (added to \texttt{CalculateImp} on line 23). Second, the temperature \( T \) is decremented after \( n \) feasible iterations \( T:=V−\text{addted between lines 28 and 29} \). As the algorithms for SA and FI are otherwise identical, we do not present one for SA. Subsequently, the run-time for SA matches that of FI at \( O(k^2N|V|/|E|) \).

4.5. Constructor

In this section, we present our constructor heuristic. The aim of any constructor is to produce a seed set, for algorithms such as GA, that are initially better than random. That is, when a random solution generator is used, the only information required is the number of feasible solutions to generate. Whether or not a particular solution makes sense or will aid in post processing is not of concern. Thus, the onus of crafting suitable solutions falls on the consuming heuristic. With a well devised constructor heuristic, however, this burden is diminished and the algorithm is free to spend its allotted computation time refining its results.

In order to define these governing set of rules, we draw from our knowledge and experience with cube lattices and prior VSP research to propose the following three construction principles.

**Principle 1.** Views with the greatest number of downward edges (out degree) in the upper tiers of the lattice should be selected for materialization. This will have the effect of decreasing the query response time (as larger views are expensive to generate in the higher tiers) and building a foundation for a smaller update cost (implicit enforcement of the update time constraint).

**Principle 2.** Only descendant views of those selected under Principle 1 should be candidates for materialization. The logic behind this has to do with the effect adding a view \( \nu \) to the materialized set \( M \) has on both the query and update times, along with how savings propagate downward. When a view higher in the lattice is set to be materialized, all descendant views without an intermediate materialized ancestor, benefit in terms of either a decrease in query response time, if that view is not materialized, or update cost, if materialized. It is this very set of descendant nodes which should be considered for further materialization in order to continue to drive the
objective function value down through materialization compounding.

**Definition 3 (Materialization Compounding).** The process by which an objective function value $OF(M)$ capitalizes on materializing views whose ancestors are themselves in the materialized set $M$. That is, if $v \in M$, then all of its descendant views, $V_2^v \not\subseteq M$, query costs most likely decrease (if $q(i,M) = r_v \forall i \in V_2^v$). Compounding occurs when $v$ is then selected to be materialized as $q(i,M) = r_v \leq r_v^*$ and (the compounding) $u(i,M) = r_v$ which is probably less than $b_v$. Terms such as “most likely” and “probably” are used because these assumptions do not hold explicitly as there may be a case when some $q(i,M) \neq r_v$, where $i \not\in M$, and some $u(i,M) \neq r_v$, where $i \in M$.

**Principle 3.** From the candidate set ensuing from Principles 2, only views that do not strictly increase the objective function will be considered further as defined by Eq. (10). Consider $v$ iff:

$$f_j(q(v,M) - r_v) \geq \omega_{pg}(u(v, M \cup \{v\}))$$

Equation (10)

Essentially, the discounted query response time, due to materialization of $v$, must be at least as large as the amplified update cost. That is, the savings in tuples by $v$ should not be considered a candidate for materialization.

**Lemma 1.** Given a VSP model $\Psi$, a view $w \in M$, and a view $v \notin M$ where $v_1 = u$ and $r_1 = r$, such that $2b_{M \cup \{v\}} \leq \sum_{u \in V} x_u \not\subseteq \Pi$, $v$ should not be considered a candidate for materialization.

**Proof.** Let $V_2^v$ be the set of views that currently answered from ($\forall v \subseteq V_2$) or updated by ($\forall v \subseteq V_2$) view $u$, $OF_u$ is the objective function value, and $\omega_{kg}$ is the update cost. Assume parameters $g_i, w, p > 0 \forall i \in V$. If view $v$ is added to set $M$, the cost associated with $V_2^v = V_2$ since $\sum_{i \in V_2^v} q(i,M) = r_v \sum_{i \in V_2} q_i = r_v \sum_{i \in V_2} f_i$ and $\sum_{i \in V_2} u(i,M) = p_{r_v} \sum_{i \in V_2} g_i = p_{r_v} \sum_{i \in V_2} g_i$. However, view $v$ must be updated from view $u$ which increases the total update cost by $\sum_{i \in V_2} g_i$. Therefore, $\omega_{kg} = \omega_{kg} + p_{r_v}$ for $\Psi = (5)$ and (7). Furthermore, the objective function value increases by $\omega_{kg}$. Therefore, $OF_u = OF_u + \omega_{kg}$. For $\Psi = (4)$ and (7), the query cost does not change since $f_j(q(v,M) - r_v) = f_j r_v$ for $v \notin M$ and $f_j r_v$ for $v \in M$ (recall $r_v = r$, therefore $f_j r_v = f_j r_v$). Thus view $v$ should not be considered a candidate for materialization.

Now to the algorithm. In order to fulfill Principle 1, the uppermost layer(s) to be used must be defined. The decision to use the highest two tiers (line 8) is based on the number and size of the views, which can be set to suit the problem. Next, the algorithm populates a portion $\kappa$ of the overall allotted space $S$ with views having the largest degree, with a selection probability of $1 - \epsilon$—this ensures identical solutions are not generated at each iteration while constructing the final population $\rho$ (a max of $kp$ cycles—line 4). Then, the selected set is sorted by number of tuples in ascending order (line 15); this assures the smallest materialized ancestor will be used in the proceeding step. Now, a candidate list of views is built from the $\beta$-descendants of the views in the partial solution per Principles 2 and 3 (lines 16–21). This will have the effect of implicit enforcement of the update cost constraint by utilizing materialization compounding. After generating the candidate list $\zeta$, it is sorted by level then benefit descending (line 22). With probability $1 - \epsilon$, views from $\zeta$ are appended to the solution until no view can be added without violating the space constraint. Since only the space constraint has been explicitly enforced, the final solution must be checked against the update constraint and if not in violation, is added to the population (via CalculateConst on line 37). The algorithm iterates until the population is built or the termination criteria $k$ is reached. The overall run-time is $O(kp|V|/|\zeta|)$.

**Algorithm 3. Constructor($\zeta$, $j$, $k$, $s$, $\kappa$, $\epsilon$).

input: $\zeta$, the cube lattice $\zeta$, the population size $\rho$, termination criteria $k$, the max space $s$, the percentage $\kappa$, and the selection percentage $\epsilon$.

output: $\mathcal{J}$.

1. $j = \emptyset$ // resulting population
2. $\rho_{term} = 0$ // termination iterator for $\rho$
3. for $p = 1, \ldots, \rho$ do // build up the population
4. if $\rho_{term} > k \rho$ exit end if
5. $S(M, j, \zeta, \rho_{term} = 0) // size constraint, iterator, counter
6. $M, \zeta, m = \emptyset$ // solution, list of candidate view, results from Eq. (10)
7. sort $\zeta$ from top to bottom by $\zeta$ and by degree descending // Principle 1
8. for $i = 1, \ldots, |V|$ in levels $V \cdot \text{getTopLevel}()$ and $V: \text{getTopLevel}() - 1$ do
9. $m$ = random number between 0 and 1
10. if $m > \epsilon \alpha s(M) + r_v \leq s$ then
11. $M = M \cup \{v\}$
12. $S(M) = S(M) + r_v$
13. end if
14. end for
15. sort $M$ by row ascending // assures the smallest materialized ancestor is used
16. for $j = 1, \ldots, |M|$ do // will only look at descendant views (line 19), Principle 2
17. if $j = \beta$ Principle 3 holds then
18. $e = \sum_{M \in \mathcal{J}} \sum_{v \in M} |A_{v,M}| // add descendants of those in $M$ not already in $M$
19. $m = m + e$
20. end if
21. end for
22. sort $\zeta$ and $m$ by $\zeta$ descending then $m$ descending
23. continue = true
24. while $S(M) < s \alpha$ continue do
25. continue = false
26. while $f < |\zeta|$ do
27. $m = \text{random number between 0 and 1}$
28. if $m > \epsilon \alpha s(M) = f_r \leq s$ then
29. continue = true
30. end if
31. end while
32. end while
33. end while
34. end for
35. end for
36. end if
37. end for
38. end for
39. end if
40. end if
41. end if
30. $\mathcal{M} = \mathcal{M} \cup \{p'\}$ // add view to solution
31. $\mathcal{S}(\mathcal{M}) = \mathcal{S}(\mathcal{M}) + r_{\mathcal{M}}$ // add size to total
32. $\ell = \ell + 1$ // remove view
33. else $f_{\text{inf}} + 1$ end if // increment counter
34. end while
35. end while
36. if $\mathcal{M} \notin \mathcal{J}$ then
37. if CalculateConst($\mathcal{M}$) is valid then $\mathcal{J} = \mathcal{J} \cup \mathcal{M}$ else $p = p - 1$
38. end if
39. else $p = p - 1$ end if
40. $\rho_{\text{size}} = \rho_{\text{size}} + 1$
41. end for

4.6. Genetic algorithms

Genetic algorithms (GAs), a specific form of evolutionary algorithms (EAs) commonly used to solve large, complex problems such as the VSP, were designed by Holland [70] and imitate the process of evolution to solve optimization problems. As this algorithm simulates biological processes, the terminology follows suit. GAs begin with an initial population $\rho$ of solutions that are typically randomly generated (known as a generation). Each solution is called a chromosome. The actual chromosome structure is called a genotype which consists of genes, the individual components of the solution, which have possible values called alleles. The meaning of each solution is known as its phenotype. Simply put, consider a solution to be a binary string of 5 elements. The chromosome is a binary string, the genotype is the binary string of length 5, there are 5 genes with two possible allele values (0 or 1), and the phenotype could then be the binary number's base 10 value or, in our case, the set of views to materialize [70–73].

Offspring (future generations) are generated by combining two parents in $\rho$. To create children, a crossover point is selected. All genes from the first parent up to and including the crossover point and all genes from the second parent from the gene after the crossover point to the end are combined to make the first child; the second is simply the remaining genes. For example, let parent $P_1 = 10010$ and $P_2 = 00111$ with a crossover point of 60% (i.e., $0.6 \times 5 = 3$), then offspring $O_1 = 10011$ and $O_2 = 00110$. While building each offspring, genetic mutation is applied which introduces diversity into a population that might otherwise be or become too homogeneous. This is done probabilistically by flipping a bit with a likelihood of, e.g., 0.1% [70–73].

Algorithm 4 depicts this crossover and mutation portion of our GA. Method CalculateCross (line 10) takes each offspring and, if it is original, tests it for feasibility and, if it is, retains the solution. This method runs in $O(|\mathcal{V}| |\mathcal{E}|)$ as for each element in the candidate solution, the SetCosts function is invoked. Overall, the algorithm requires $O(|\mathcal{P}|^2 |\mathcal{V}| |\mathcal{E}|)$ time. The resulting generation of offspring can be merged with the parents or taken alone as the new population for subsequent generations. Either way, a population strategy to either diversify or intensify is employed. Intensification takes an elitist approach in which only the best solutions compose the resulting generations whereas diversification incorporates a step size $\lambda$ in the selection of the population; meaning every $\lambda$th solution from the merged set is selected.

For this paper, three population creation methods were used: (1) randomly generated (resulting in a running time of $O(k \rho |\mathcal{P}| |\mathcal{V}| |\mathcal{E}| - k \rho$ maximum random solution generation attempts), (2) the results from a best improvement search (resulting in a running time of $O(k^2 N |\mathcal{V}| |\mathcal{E}|)$, and (3) seeded from our constructor (having an execution time of $O(k \rho |\mathcal{P}| |\mathcal{V}| |\mathcal{E}|)$ as with random). The genotype is a binary string consisting of $|\mathcal{V}|$ elements with possible allele values of 1 (selected for materialization) or 0 (not selected).

**Algorithm 4. Crossover($\mathcal{L}$, $\mathcal{J}$, $\mathcal{X}$, $\eta$).**

**input:** the cube lattice $\mathcal{L}$, initial population $\mathcal{J}$, crossover $\mathcal{X}$, and mutation factor $\eta$ ($\eta = \text{flip bit with probability } \eta$)

**output:** $\mathcal{J}$'s solution $s$ and $OF$

1. $OF = 0$ // objective function value
2. $S_{\text{found}} = \emptyset$ // the set of solutions already tried
3. for $i = 1, \ldots, |\mathcal{J}|$
4. for $j = 1, \ldots, |\mathcal{J}|$
5. $p_1 = J^i$ // parent 1
6. $p_2 = J^j$ // parent 2
7. $O_1 = [\neg \chi_1(p_1), \ldots, \neg \chi_{\ell_j}(p_1)] \cup [\chi_1(p_2^{\ell_j} + 1), \ldots, \chi_{\ell_j}(p_2^{\ell_j})]$
8. $O_2 = [\neg \chi_1(p_2), \ldots, \neg \chi_{\ell_j}(p_2)] \cup [\chi_1(p_1^{\ell_j} + 1), \ldots, \chi_{\ell_j}(p_1^{\ell_j})]$
9. if $O_0 \notin S_{\text{found}}$ and $k = 1.2$ then
10. if CalculateCross($O_0$) is valid then $S_{\text{found}} = S_{\text{found}} \cup O_0$ end if
11. end if
12. end for
13. end for
14. sort $\mathcal{J} = \mathcal{J} \cup S_{\text{found}}$ from smallest $OF$ to largest

4.7. Genetic local search

Genetic local search combines the advantages of population-based search and local optimization [33,74]. In this paper, we implement three population generation approaches: random, best improvement, and via our constructor. Local search iteratively searches its neighborhood, moving from one solution to another if it is an improvement.

In order to implement any local search method, one must first define the relationship between solutions and how to move from one state to another. We employ a state change similar to Kalnis et al. [34]. A 1-removal is equivalent to move 2 under a space constraint where one view is removed from $\mathcal{M}$ and the remaining space is systematically filled with selected candidate views. To be considered for candidacy, a view must be a relative of a view in $\mathcal{M}$ (line 13). This has the effect of implicitly enforcing the update constraint. We extend this to the removal of two views as well. Algorithm 5 describes the 2-removal local search portion of GLS. Essentially, each materialized (line 8) pair $(j,k)$ (defined on lines 6 and 7 respectively) is removed from $\mathcal{M}$ (lines 8–9) and the candidate set constructed (lines 11–16). The candidate set retains frequency counts for inter-level sorting (line

---

7 It is possible to have multiple crossover points. In this case, each child receives every other segment from each parent; an extension of the single crossover conjunction.

8 The step size is not a factor in this algorithm as the results from the search provide, in general, more than $\rho$ solutions.
Algorithm 5. LocalSearch2(ℒ, ℋ).

input: the cube lattice ℒ and the initial population ℋ
output: ℋ_{new}
1. ℋ_{new} = ∅ \hspace{1cm} // resulting population
2. for i = 1,...,|ℒ| do // for each solution
3. \((\tau_i, \phi_i)\) \hspace{1cm} // temporary copy
4. \(\mathbf{c}_i = \mathbf{c}\) \hspace{1cm} // list of candidate views, list of counts for each view in \(\mathbf{c}\)
5. \(q = 0\) \hspace{1cm} // space available
6. for j = 1,...,|ℒ| do // for each view in the solution
7. for k = j+1,...,|ℒ| do // for every view further down the line
8. if \(\{\mathbf{r}\} = \text{materialized} \land \{\mathbf{r}\} = \text{materialized}\) then
9. \(\{\mathbf{r}\}, \{\mathbf{r}\} = \text{not materialized}\)
10. \(q = S - S(\tau_i^f) + r_{\tau_i^f} + r_{\tau_i^b} \hspace{1cm} // \text{total-solution} + \text{removed views}\)
11. for c = 1,...,|ℒ| do // for every view
12. if \(\{\mathbf{r}\} = \text{materialized}\) \hspace{1cm} if materialized, build the candidate list
13. \(c = Q \cup d_{\tau_i^f} \cup d_{\tau_i^b} \cap r_{\tau_i^f} \cap r_{\tau_i^b} \hspace{1cm} // \text{add new anc's and desc's}\)
14. \(q = q + |d_{\tau_i^f}| + |d_{\tau_i^b}| - |r_{\tau_i^f}| \cap |r_{\tau_i^b}| \hspace{1cm} // \text{increment counts}\)
15. end if
16. end for
17. sort \(c\) by \(\mathbf{c}\) from top to bottom then by \(\phi\)
18. for t = 1,...,|\(\mathbf{c}\)| do // for each view in the candidate list
19. if \(q - r_{\tau_i^f} \geq 0\) then // if the view fits, add it
20. \(\{\mathbf{r}\} = \text{materialized}\)
21. \(q = q - r_{\tau_i^f}\)
22. end if
23. if \(q = 0\) then break end if
24. end for
25. CalculateLS(\(\mathbf{c}, \phi\))
26. end if
27. end for
28. end for
29. end for

4.8. Algorithmic progression

The algorithms in the previous sections build upon themselves. Random Search, Improvement, and Constructor are placed into a genetic algorithm and therefore should produce better solutions. Likewise, Random, Improvement, and Constructor GAs are used as seeds for GLS; so the same solution progression follows. Fig. 4 illustrates this process.

5. Experimental setup

As pointed out in Section 3.2, the SCVSP is a special case MMVSP where the update constraint is never violated. Thus, for these experiments, we simply compare MMVSP to MCVSP under the guise that if the update constraint is not violated, then the optimal solution obtained by MMVSP will be identical to SCVSP (and therefore the comparisons with MCVSP). If not, then any solution acquired by SCVSP will not be feasible. All results shown are averaged across 10 iterations unless otherwise specified.

The computer used for experimentation was an Intel Core 2 Duo T7500 CPU @ 2.20 GHz machine with 3 GB of DDR2 RAM running Windows XP Professional SP3. The program was written in Java using JDK/JRE 1.6.

5.1. Dataset

We use the bridge group formulation and multidimensional design from Hylock et al. [17] based on the dataset from Lu et al. [75]. It is a real-world nursing data set used for analyzing outcomes consisting of 15 relations, 6 hierarchies, and covers basic information such as demographics, encounters, diseases, nursing diagnoses, interventions, and outcomes. The tables range in size from 2.5 to 4.1 million tuples with an average of 187 thousand. Three query sets (20, 25, and 30; 25 and 30; 30 being the maximum we could optimally process in a reasonable amount of time—a fashion similar to Chaves et al. [76]) were manually generated ensuring no query could be merged with another. Then, values for \(f_r\) and \(g_r\) were randomly assigned based on uniform and Gaussian distributions resulting in 6 test datasets. To evaluate scaling, additional experiments were performed using larger value for \(\nu\) ((250, 500, 750, 1000, and 1250), with heuristic-to-heuristic comparisons.

5.2. Parameter settings

The following parameters were thoroughly tested for the improvement algorithms. The settings were as follows: the number of neighborhoods \(N = (1, 10, 25, 50, 75, 100)\), the number of neighbors \(n = (25, 50, 75, 100)\), and the variation about the initial solution \(\sigma = (0.2, 5)\). All combinations were tested 10 times and an average for each was recorded. The results show the following parameter settings to be the best in terms of objective function value and computational time: \(N = n = 50\) and \(\sigma = 5\). Also, the
termination criteria $k$ was set to 10 and can be changed to suit the problem; we found this to be more than adequate.

For Simulated Annealing, we began with a temperature $T$ of 90 and cooling schedule ($c$) of 0.7 per Derakhshan et al. [20,21]. Experiments were performed for $T = (50, 70, 90, 110)$, a cooling of $(0.3, 0.5, 0.7, 0.9)$, external ($N$) iterations of $(25, 50, 75, 100)$, and internal ($n$—the number of evaluations before $T$ is decremented) iterations of $(25, 50, 75, 100)$. The best parameter settings were found to be $T=90, c=0.7$ (verifying Derakhshan et al. [20,21]). $N = n = 50$ (as with II).

Genetic algorithms requires the tuning of four parameters. The first is the population size $\rho$. According to Burke and Kendall [71], a population size of 50 is a good starting point. The second is the crossover rate $\chi$. Burke and Kendall [71] suggest a value of 0.6 (i.e., $o_1$ is the first 60% of $p_1$ and the last 40% of $p_2$). The third is the mutation factor $\eta$ for which a starting value of 0.05 is recommended [71]. Finally, the fourth is the population strategy of intensification or diversification. The following parameter settings were tested: $\rho = (50,100), \eta = (0.005,0.05,0.1), \chi = (0.25,0.6,0.75)$, and population strategy $=$ (intense, diverse) $-$ for diverse, we use every other $j$th $= 5$th (unless the results size is less than 5 times the size of the population in which we use the largest possible divisor).

Each test was performed 10 times and the results show the following parameter values to be the best in terms of objective function value and computational time: $\rho = 50, \eta = 0.05, \chi = 0.25$, and strategy $=$ intense.

For the constructor heuristic, $\kappa$ was tested at $(0.75, 0.8, 0.85, 0.9, 0.95, 0.99)$ with 0.95 preferred and $\epsilon$ at $(0.01, 0.05, 0.1, 0.15, 0.2)$ favoring 0.1. For the individual tests, $\rho$ was set to 50 based on $N$ and $n$ values from II and SA and $\rho$ from GA.

Finally, the update cost percentage $p$ was set to 10% following Lawrence [35], Yousri et al. [28], and Kalnis et al. [34] and $w$ to 1 to minimize update time fully.

### 6. Computational results

#### 6.1. Benefits of materialization

Briefly, we would like to preface this section with the average query response times for each size $|V|$ without any materialized views. For $|V| = 20$, $|V| = 25$, and $|V| = 30$, the query response times are 4.46, 3.88, and 3.42 s respectively. Therefore, any materialization strategy must outperform these base values.

#### 6.2. Algorithmic average execution time

Table 1 presents the cumulative execution times in milliseconds per algorithm segmented by distribution and dataset size (MMVSP and MCVSP combined). By far, the fastest executing family of algorithms are the Constructor-based approaches—with C finishing 1st in uniform 20, 2nd everywhere else, CGA and CGLS 1 third and fourth respectively, and CGLS 2 6th @30, 7th @25, and 5th @20; only random-based heuristics co-mingle. It is interesting to note that Random GA and Random GLS 1 & 2 require more time than all except Constructor GLS 2. This appears to be counter-intuitive since generating $\rho$ random solutions should take the same if not less time than the constructor. One potential explanation may revolve around the solution space and the number of feasible solutions given the constraints. That is, with tight constraints, it may be difficult to randomly generate valid seeds. Random Search simply produces random solutions of random size. If the solution space provides more infeasible than feasible solutions, then it follows that random search will find more infeasible solutions than feasible. Furthermore, since the constructor heuristic follows a specified set of rules that are designed to lead to not only higher quality, but feasible solutions (through explicit and implicit enforcement of constraints), the constructor should find $\rho$ feasible solutions with few discards. Thus, given these conditions, the constructor heuristic should execute faster than or at worst proportional to any form of random search (even though the run-time indicates a $k_p$ time increase).

#### 6.3. Algorithmic averages by model

In the proceeding section, the heuristics are compared via VSP formulation. As shown in Tables 2 and 3, minimizing the update cost ($U$) appears to have a distinct advantage. For example, CGLS 2 for MMVSP has a higher

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Gaussian</th>
<th>Uniform</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>30</td>
<td>25</td>
</tr>
<tr>
<td>Random search</td>
<td>6.84</td>
<td>6.08</td>
</tr>
<tr>
<td>First improvement</td>
<td>1292.02</td>
<td>962.88</td>
</tr>
<tr>
<td>Best improvement</td>
<td>1423.86</td>
<td>962.65</td>
</tr>
<tr>
<td>Simulated annealing</td>
<td>1807.30</td>
<td>1399.44</td>
</tr>
<tr>
<td>Constructor</td>
<td>24.20</td>
<td>17.3</td>
</tr>
<tr>
<td>Random GA</td>
<td>187.35</td>
<td>151.87</td>
</tr>
<tr>
<td>Improvement GA</td>
<td>890.49</td>
<td>699.01</td>
</tr>
<tr>
<td>Constructor GA</td>
<td>79.82</td>
<td>43.53</td>
</tr>
<tr>
<td>Random GLS 1</td>
<td>225.42</td>
<td>171.09</td>
</tr>
<tr>
<td>Random GLS 2</td>
<td>428.86</td>
<td>293.60</td>
</tr>
<tr>
<td>Improvement GLS 1</td>
<td>951.75</td>
<td>728.79</td>
</tr>
<tr>
<td>Improvement GLS 2</td>
<td>1370.05</td>
<td>966.32</td>
</tr>
<tr>
<td>Constructor GLS 1</td>
<td>90.76</td>
<td>48.14</td>
</tr>
<tr>
<td>Constructor GLS 2</td>
<td>200.54</td>
<td>171.91</td>
</tr>
</tbody>
</table>
percentage of optimal solutions found as well as being closer to the optimal objective function value (O). Furthermore, the averages for all models using CGS 1 appear to be similar, however, MCVSP fails to find as many optimal solutions. Finally, it is worth noting that most update percentage values for the uniform distribution (with the exception of RS, SA, and C) are greater than the optimal solution. That is, the solutions found take more time to update than the optimal one; this is not seen anywhere in the results for MMVSP. This, we believe, is an artifact of $w=1$ as MMVSP will favor heavily decreasing update over query costs. Thus, this data provides further evidence supporting the use of MMVSP over MCVSP.

### 6.4. Algorithmic averages by distribution

Table 4 shows the cumulative results for all models by algorithm and distribution. The number of optimal solutions found by CGS 1 & 2 are significantly greater than all other implemented techniques. Additionally, those values are nearest the optimal objective function value and, when evaluating each component of the objective function separately (i.e., Q and U), they sacrifice the least in terms of query response times while making a slight to significant improvement in update cost; varying by distribution. This, in conjunction with the preceding sections, provide strong evidentiary support to our proposed constructor.
principles, as supplied in Section 4.5, and its use as a seed generation tool for genetic local search.

6.5. Extensions using large lattices

The previous experiments were performed on smaller datasets to allow for direct comparison with known optimal values. They do not, however, lend themselves to understanding scalability. Therefore, we present the following five assessments using the MMVSP formulation, synthetically generated cube lattices for $|V| = (250, 500, 750, 1000, 1250)$, and CGLS 1, IGLS 1, and RGLS 1 heuristics. Each experiment was performed 10 times for $|V| = 250–750$ and 3 times for $|V| = 1000$ and 1250 with the averages recorded in Tables 5–9. Our reasoning behind the selection of GLS 1 as well as Improvement- and Random-based seeding is as follows. First, although GLS 2 produces the highest quality solutions, it comes with a significant growth in processing time which cannot be justified by its savings; thus our choice of GLS 1. Furthermore, the second best GLS 1 heuristic is the Improvement (i.e., best improvement) seeding method, which our constructor must outperform in terms of solution quality. Finally, for speed comparison, Random GLS 1 was chosen as its running time is comparable, albeit slower, than our constructor. It is, however, much faster than IGLS 1, giving us a solid baseline for time. As for the generation of $f_v$ and $g_v$ frequencies, we chose a uniform distribution as the constructor. It is, however, much faster than IGLS 1, giving us a solid baseline for time. As for the generation of $f_v$ and $g_v$ frequencies, we chose a uniform distribution as the constructor. It is, however, much faster than IGLS 1, giving us a solid baseline for time. As for the generation of $f_v$ and $g_v$ frequencies, we chose a uniform distribution as the constructor. It is, however, much faster than IGLS 1, giving us a solid baseline for time.

Tables 5–9 present our computational findings. In every instance, CGLS 1 formulates the highest quality solution. Between CGLS 1 and IGLS 1, a slight to moderate improvement in query cost and overall objective function value exists, but a significant enhancement is shown in update costs. This is directly attributable to the principles laid out in Section 4.5, which implicitly enforce the update time constraint and thus force this value down. As expected, CGLS 1 handily outperforms RGLS 1 in these terms.

Concerning execution time, for $|V| = (250, 500)$, very little separation is noticed with CGLS 1 actually being the slowest in both cases. However, these discrepancies are minute due to their scale. Separation can be observed starting with $|V| = 750$ which continues to increase quite rapidly with each lattice iteration. At $|V| = 750$, CGLS 1 improves over RGLS 1 by 27.68% and IGLS 1 by 48.07%. An interesting event occurs at $|V| = 1000$ with CGLS 1 taking roughly the same amount of time as it did at $|V| = 750$; while both RGLS 1 and IGLS 1 increased. We believe this to be an anomaly pertaining to the generation of the cube lattices and their similar solutions. That is, the objective function values are proportional to one another while all others are not. Likewise, the solution sizes (which have been omitted from this paper to conserve space) between $|V| = 750$ and 1000 are virtually identical for CGLS 1 while...
quite different for the other tested methods. Finally, at \(|V| = 1250\), there is a staggering gap of nearly 21 h (83.97%) between CGLS 1 and IGLS 1. Likewise, over RGLS 1, CGLS 1 improves in execution time by almost 7 h or 63.95%.

It might not appear feasible for a seed set to have such a profound effect on execution time, however it is entirely consistent for the following reasons. First, as constraints tighten, the ratio of feasible-to-infeasible solutions decreases. Currently, GLS forces RS to return its full complement of seeds whereas C and II retain their \(k_P\) and \(k^2/nN\) bound on attempts respectively; II, however, is asked to create 50 times as many be default. Therefore, it may take RS a substantial amount of time to produce \(\rho = 50\) feasible solutions and II may approach or exhaust all 250,000 of its allotted iterations.\(^9\) C, on the other hand, is designed to not only produce high-quality solutions, but, more importantly, feasible ones. Second, II and RS randomly select a solution length whereas C simply fills as much room as possible. Therefore, the removal-replacement phase in GLS may require several times as many iterations for II and RS as C has fewer replacement candidates due to its increased length.\(^10\) Taken in combinations, these two explanations further substantiate the results.

The preceding experiments have shown genetic local search seeded with candidates from our constructor heuristic, to maintain high quality solutions while efficiently scaling to meet real-world cube lattice sizes where others cannot.

7. Conclusions and future work

A data warehouse is designed to integrate multiple, heterogeneous information sources to facilitate decision support and data analysis. One major component to such a design is the selection of views to materialize to decrease query response time given resource constraints. In this paper, we presented a new model for solving the VSP called the Minimum-Maintenance View Selection Problem and formulated a constructor heuristic to build an initial population for seeding other algorithms with the direct intent of reducing view maintenance costs.

As predicted, MMVSP has been demonstrated to decrease the update cost at little expense to query performance when compared to MCVSP; leaving more time for other tasks. Likewise, since SCVSP is consumed by MMVSP, MMVSP produces solutions no worse than SCVSP and, when the update constraint is a determining factor, it ensures feasible solutions where SCVSP cannot. Hence, our claim that MMVSP does not suffer from the short-comings of its predecessors has been substantiated. Moreover, we have verified the power of a constructor heuristic designed specifically for the VSP against well known and accepted heuristics. The constructor alone has been shown to take time near a purely random approach, but routinely provides superior solutions even when compared to RGA and RGLS 1 & 2. A genetic local search seeded with the results of this constructor heuristic produces solutions of higher quality than all others examined. Furthermore, it has been shown to effectively scale to more realistic sizes while existing methods do not. Finally, the prevalence of SCVSP and MCycle models may be seen as a barrier to the adoption of MMVSP, but switching directly to MMVSP requires \(O(1)\) space and time.

In the future, we plan to expand the MMVSP to various multi-system environments. Previous work on the distributed VSP is limited and cannot account for a federated environment.

References