An Automatic Physical Design Tool for Clustered Column-Stores

Alexander Rasin
DePaul University, 243 S. Wabash Av.
Chicago, IL 60604
arasin@cdm.depaul.edu

Stan Zdonik
Brown University, Box 1910
Providence, RI 02912
sbz@cs.brown.edu

ABSTRACT

Good database design is typically a very difficult and costly process. As database systems get more complex and as the amount of data under management grows, the stakes increase accordingly. Past research produced a number of design tools capable of automatically selecting secondary indexes and materialized views for a known workload. However, a significant bulk of research on automated database design has been done in the context of row-store DBMSes. While this work has produced effective design tools, new specialized database architectures demand a rethinking of automated design algorithms.

In this paper, we present results for an automatic design tool that is aimed at column-oriented DBMSes on OLAP workloads. In particular, we have chosen a commercial column store DBMS that supports data sorting. In this setting, the key problem is selecting proper sort orders and compression schemes for the columns as well as appropriate pre-join views. This paper describes our automatic design algorithms as well as the results of some experiments using it on realistic data sets.

Categories and Subject Descriptors

H.2 [**Database Management**]: Physical Design, Systems—Relational DBs

General Terms

Design, Algorithms, Performance

1. INTRODUCTION

As the cost of hardware falls, application demands grow. After all, if it is feasible to buy a 100 machine cluster, then that level of horsepower can potentially enable much more demanding data processing capabilities. Complex hardware coupled with new and complex application demands has dramatically increased the cost of administration. In order to offset this cost, there has been recent interest in no-knobs systems that can, in effect, administer themselves.

There has also been a recent trend toward specialized database systems that are architected to perform really well for specific workloads. For example, column-store architectures are particularly well-suited to OLAP workloads. Each new architecture brings with it a set of physical design choices that are different from those of previous systems. It is our belief that we must therefore revisit the topic of automated physical design to address these new choices and thus exploit fully their benefits.

Physical database design is well-known to be a difficult problem [14, 9]. Many competing demands must be balanced, and there is a very large space of potential designs. The goal is to create a system that can produce all or part of a design automatically, or at least with a minimum amount of intervention from the DBA. We aim to produce designs that are competitive with what a human might produce since, in practice, we know that designs produced by such a tool are often used as a starting point.

Others have approached this problem by building tools that perform automatic secondary index selection [6, 7] or automatic data partitioning [22]. All of this work has been done in the context of traditional row-stores. In this paper, we describe the problem of automatic physical design in the context of a column-store that supports clustering (i.e., sorting). We call such systems Clustered Column Stores (CCS). The most prominent CCS examples are C-Store [24], its commercial realization, Vertica [4, 20] and the work done in [17]. We will use a commercial CCS referred to as C-Store throughout this paper. In Section 2 we discuss additional architectural features of C-Store such as compression and insert-handling that must be considered as well.

Column-stores can perform dramatically better than row-stores in data warehousing environments where the workload is dominated by read-only queries. A typical data warehouse query reads a small fraction of the columns of each table. Thus, a column-store performs far less I/O since it will only read the columns needed by a query. Physical design for our column-store can be challenging since it must determine column sort orders, individual column compression schemes, and materialized join views.

In this work, for simplicity we assume that queries are star (or snowflake) compliant - or that it is possible to split the user queries to make them star-compliant. That is to say, joins must be based on a PK-FK relation with a single underlying fact table (see Sections 2.1 and 5.2 for more details). If a query contains a join that does not comply with

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EDBT/ICDT '13 March 18 - 22 2013, Genoa, Italy
Copyright 2013 ACM 978-1-4503-1597-5/13/03 ...$15.00.

this condition, it must be split into pieces that do (e.g., [25]). To our knowledge, such adjustments are easily performed in most OLAP applications.

Our algorithm builds design candidates consisting of materialized views and corresponding clustering indexes. We will discuss why this makes sense in this setting. Our algorithm does not support materialized aggregate views. Aggregate views make sense when the query workload is very predictable; however, in the face of ad hoc queries, they may not provide enough advantage to outweigh their maintenance cost.

From a very high-level, all automatic design algorithms do the same thing. They generate alternative designs, apply a cost model to each one, and pick the design with the lowest cost. Of course, the space of all possible designs is very large making an exhaustive search impractical. Thus, all such design tools use heuristics to prune the space of designs that are considered. We will show that good designs in our column-store are different from good row store designs. Using row store heuristics to prune the search space is likely to miss the best column-store choices.

We will discuss at length the features of a C-Store physical design that make it different from a conventional row-store design. We will also show that some of the key C-Store considerations can also have a positive effect on a row-store design although they are less dramatic in the row-store setting. Note that here and throughout the rest of the paper "database design" always refers to the physical database design.

Contributions:

- We present and evaluate an accurate, correlation-aware cost model for a clustered column-store DBMS with compression.
- We present an algorithm for single-query MV candidate design that is sensitive to the query disk access pattern.
- We present an improved approach for merging indexes – an algorithm that designs new MV candidates based on workload query groups.
- We present a framework for incorporating inserts into a clustered column-store that employs insert buffering. We also use an LP solver to provide a solution that accounts for insert-heavy workload.
- We present a comprehensive design algorithm for a clustered column-store DBMS.

The paper proceeds as follows. We first describe relevant features of the C-Store architecture. We then give a precise description of the problem. Next, we describe our algorithm in detail and the rationale for its design. We present an experimental analysis of its performance, and close with a comparison with previous work on automatic design tools and provide some ideas for future work.

2. THE CLUSTERED COLUMN-STORE

In this section we give an overview of the features of C-Store that must be considered in formulating a physical design.

2.1 Sorted Materialized Views

C-Store is a column-store in that it stores each column of a relation separately with its own sort order. If column A is sorted by itself we write (A | A) and say that this represents column A sorted by column A. Of course, a column can be sorted by some other column as in (A | B) which represents column A sorted by column B. We can then group all columns with the same sort order as in (A B C | A) which is the three columns A, B, and C, all sorted by A. We call such a column grouping a sorted Materialized View (MV). Note that an MV still preserves the separate storage of each column. An MV can also be sorted on multiple columns as in (A B C | A B). It must be emphasized that an MV with these three columns is not the same as a three-column relation in a row store. Here, each column is in a separate, autonomous file.

The columns in an MV need not all come from the same logical relation. An MV that has columns from multiple base relations is a materialized JOIN view. With star schemas, we typically create one or more materialized JOIN view with the Fact table. Given the 1:n foreign-key relationships, the number of rows in these MVs is the same as that of the Fact table. In a non-Star Schema, queries might use more than one Fact table. In order to accommodate such queries, we break them up into multiple queries - one per fact table (as is done in [19]).

Every column in an MV is stored in the same order and thus C-Store does not require explicit RowIDs or join indexes to match fields belonging to the same row in an MV. The offset position of each field value in the column serves as an implicit RowID. In order to reconstruct the complete i^{th} row in an MV we need to retrieve values at the ith position in every column in that MV.

2.2 Column Compression

Every sorted column is stored in a sequence of large disk pages (64K) each containing a consecutive range of column values (based on the sort order). In order to access a value, C-Store uses a sparse index that locates the right page. We need such an index for every column in an MV.

Each C-Store disk page has some compression technique applied to it. The default is Lempel-Ziv-Oberhumer (LZO) [27], but for a column whose domain has a small number of values compared to the cardinality of the relation, we can do much better. For example, for the MV P = (A B C | A), if A is few-valued, since it is also self-sorted we can use Run-Length Encoding (RLE) to compress it.

RLE represents a sorted column as a set of (value, run-length) pairs. Thus, a column with a domain of cardinality n will require n pairs in its RLE representation. If a column contains order 1000 values, then a column with hundreds of millions of rows will compress into a single page. Even if a few-valued column is not first in the sort order, RLE will still compress columns into a small number of blocks as long as the column cardinality is low. For example in an MV (A B C | A B) where cardinality of B is 50 and A is 1000, RLE compression would reduce B to 50K RLE pairs which would take approximately 7 (64K) disk pages, regardless of how many rows the MV contains.

Several additional standard compression techniques can be applied to the data pages when it is deemed to be more effective than LZO. For unsorted low cardinality columns, dictionary compression can be applied to every page. Delta compression can be used for a many-valued, sorted column.

The space that is saved through compression can be used to store columns redundantly but in different sort orders.

Thus, it might make sense to store (A B C | A) and (A C | C) in the same physical design. Each of these MVs would serve a different class of query.

MVs can have a composite sort order, which loosely corresponds to a composite clustered index in a row store. The sort order also determines the compression scheme that we are able to apply to a column. Because of compression and because sort columns contain the actual data, we can use the sorted columns more effectively than a composite B-tree index. A predicate on any subset of the sort order columns will benefit as opposed to restrictions to a prefix of the sort order as in composite B-tree indices. This will be illustrated in the discussion of our cost model (Section 4).

2.3 Managing Inserts in CCS

C-Store is designed for data warehouse applications in which the workload is read-mostly. Thus, storing many redundant sort orders is good in general to accelerate the largest number of query types. The potential downside of MV proliferation is that inserts could put a heavier burden on the system.

Performing inserts in-place is not practical in a column-store architecture. Consider a 17 column fact table from SSB Benchmark [21]). A single insert into this table requires updating 17 non-contiguous pages. Moreover, if the column-store supports page-based compression then each of these 17 pages will need to be de-compressed, updated and re-compressed before being written back to disk. Appending new rows is more practical, but is not possible in a column-store that supports data sorting. Recent work in [17] describes one possible approach to handling inserts in a CCS.

C-Store addresses this issue by batching inserts in main memory unsorted, and then (eventually) sorting and writing large groups of tuples to disk as a single operation. This amortizes the overhead of the sort and the I/O.

We call the sorted columns that are stored on disk the Read-Only-Store (ROS), and the memory-resident MVs are called the Write-Only-Store (WOS). Each MV in the ROS has an analogue in the WOS. Inserts are transactionally appended to the WOS MVs. When a main memory limit is exceeded, some MVs in the WOS are sorted and written to disk.

The sorted WOS tuples are first written as a separate structure called a mini-ROS in a step called a move-out. A background process will merge the mini-ROS's in a step called a merge-out. Similar buffering approaches have been used in other systems [3] although, in these cases, data is always merged into a single structure (thus it is equivalent to having at most one mini-ROS at all times). C-Store's buffering approach amortizes the cost of performing a full data merge, which can be extremely expensive. However, it causes MVs to be stored in multiple fragments: each query needs to process all fragments of the MV it uses. A background process is responsible for gradually merging such MV fragments.

Although C-Store approach is different from the one presented in [17] the idea of batching inserts in memory remains the same. A discussion of batch merging policies is beyond the scope of this paper. Here, we only consider the immediate penalty associated with the insert cost. In Section 4, we explain how we estimate the insert cost penalty.

2.4 Covering Indexes in Row-Stores

The existing research on automatic design largely focuses on the design of secondary indexes. It is also well known that secondary indexes are not effective in the data warehouse setting. Therefore, a common practice is to build *covering* indexes, which are secondary indexes that include a superset of the columns that a query needs.

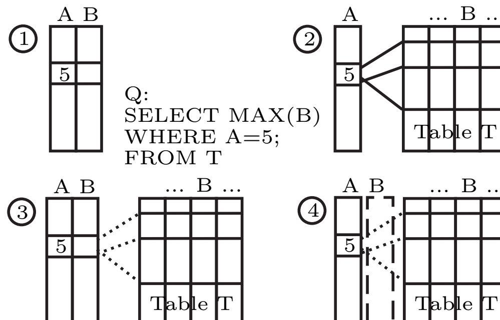

Figure 1: Secondary Indexes - 4 options

Consider query Q in Figure 1 that accesses column B and has a predicate on column A. The best structure is index type 1 which is the MV (A,B) clustered on A. The corresponding secondary index would be index type 2, but the cost to follow the index pointers into table T is likely to degenerate into a full table scan. Index type 3 represents a covering index. The advantage of the covering index is that query Q does not need to follow the index pointers (which are still there) as it did with index type 2. In other words, a query can be answered by using the index alone. Note that this structure is very similar to the MV in index type 1, since the secondary index is sorted (i.e., clustered) on its own key.

Maintaining a long composite key is expensive in a row store. Therefore, several commercial database systems support a special variation of the secondary index (e.g., *included columns* in MS SQL Server [1]) as shown in index type 4. The data from column B is physically stored with the index over attribute A, but is not included in the index key. Type 4 is an explicit implementation of type 3 and can therefore be considered to be a covering index as well as types 3 and 4 and equivalent to type 1. In a data warehouse setting, indexes of type 1, type 3 and type 4 can be effectively used as materialized views. Thus, there is little difference between the design of our sorted MVs and the design of secondary indexes of type 3 and type 4. We will later show that our techniques are applicable for designing MVs (in particular covering indexes) in row-stores.

3. THE PROBLEM

Briefly, the problem can be stated as follows. The following inputs are given.

1. A relational schema S.
2. A set of n training queries with associated weights: TS = (Q_i, W_i) for $0 \leq i \leq n$ with each Q_i over S. The weights can be interpreted as an indication of the relative frequency or importance of each query.
3. A disk space budget B (in MBs).
4. An expected level of insert batching, I_b (rows/batch).

5. An expected number of batches per workload, I_c.
6. Initial estimates of dataset sizes, column domain cardinalities, and column selectivities (or a sample dataset from which these statistics can be gathered).

The problem is to produce a set of projections P_d such that

$$\sum_{i \leq n}(W_i \times Cost(Q_i)) + (I_c \times Cost(I_b)) \quad (1)$$

is minimized. The first part of equation 1 is the read workload cost and the rest is the cost of the inserts. We discuss estimating the cost of a query and an insert batch in Section 4. The design must fit into the disk space budget B. Furthermore, any legal query Q over schema S that is not in the training set can still be processed. There are no guarantees about the performance of a query that is outside the training set; however, we would like a design to be resilient in the sense that there exist large classes of ad hoc queries that will still perform well.

The training set of queries TS represents an expected workload over some period of time. One proposed way to acquire such a set is to log all queries over several days. Increasing a weight gives us the ability to treat some queries as more important than others (e.g., a query from the boss).

4. COST MODEL

Automatic design relies on the ability to generate a number of feasible *design candidates*. An accurate cost model is necessary to evaluate the relative merits of these design candidates. In addition, the same principles that underlie the cost model also serve as a foundation for the heuristic algorithms used in MV candidate design (Section 5.3).

Consider an example MV (A B C D E F | A B C) shown in Figure 2. Column A splits the MV rows into 2 *buckets*. Column B splits each of the 2 buckets into 3 buckets, for a total of 6. And, finally, column C brings the total bucket count to 12.

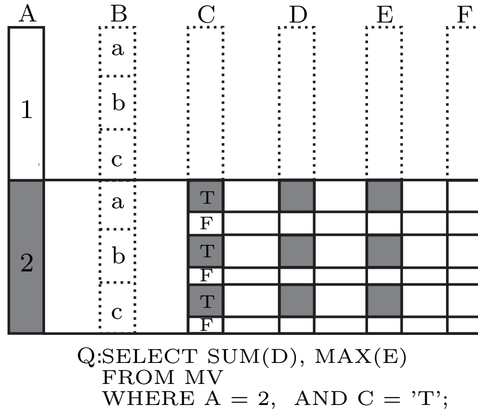

Figure 2: Cost Model Example - MV Access

Observe that the product of the cardinalities of the sort order columns determines the number of buckets (12) – one bucket for each unique combination of values in columns A, B and C. Of course, if correlations exist between these columns, the combined cardinality, and thus the number of buckets could be smaller. The number of buckets is therefore determined by the effective cardinality of the composite sort order.

Query predicates determine which of these 12 buckets need to be accessed (3 in this case). Query Q in Figure 2 selects the 2nd half of column A which corresponds to the final 6 buckets. The predicate on column C selects three of these 6 buckets. We refer to the pattern of buckets to read as a *read bitmap*. The read bitmap determines the cost to access all subsequent columns such as columns D and E in Figure 2.

Once the bitmap pattern is determined, the cost to read the subsequent columns accessed by query Q is a function of the physical column size on disk. Thus, in our example, if column D takes 4 pages on disk to store 12 buckets, and these buckets are the same size, each page would contain 3 buckets. Q would then have to read 2 of these disk pages (or 50%). However, the same column D with size of 40 pages with the same assumptions would require reading about 10 pages (or 25%) with 2 seeks to relocate between buckets. The same column can take on a different size depending on a particular compression techinque applied to that column (we discuss choosing column encoding in Section 5.3.1). In general, the cost model computes the cost of the sequence of reads and seeks as applied to each accessed column. Note that it is important to consider the seek time since in a column store the number of seeks to read the relevant part of each column is multiplied by the number of columns accessed by the query. This can make seeks a dominant part of the overall cost. We use physical disk parameters (e.g., seek time, I/O time) to combine read cost and seek cost into a single estimated time.

RB	Read Bitmap.
P_T	The size of column T (pages, post-compression).
$Buckets$	Cardinality of the composite key (see Figure 2).
B	Bucket size in pages ($\frac{P_T}{Buckets}$).
C_{seek}	The cost to do a disk seek.
C_{read}	The cost to read a single disk page.
$RUNS_a$	The set of all bit sequence lengths in a bitmap. (e.g. $RUNS_1$([1,1,0,0,0,1,1,1,1,0,0,1]) = (2,4,1)
$DTables$	The set of dimension tables joined by an insert.

Table 1: Cost Model Variables

Table 1 lists the variables that are considered in determining the cost to access a column T as expressed in equation 2. For every consecutive run of ones we estimate the amount of data that the query will scan. For every sequence of zeros in a bitmap, we take the lesser between the cost to seek over or to read the unneeded data. Note that the seek cost is roughly an order of magnitude higher than the cost to read a single page. For example, a read bitmap that accesses one in every 10 pages will have a similar cost to a column scan.

$$ColumnCost(T) = \sum_{x \in RUNS_1(RB)} \lceil x * B \rceil * C_{read} + \sum_{y \in RUNS_0(RB)} min(\lceil y * B \rceil * C_{read}, C_{seek}) \quad (2)$$

The cost of a query Q is then

$$Cost(Q) = \sum_{T \in Columns(Q)} ColumnCost(T) \quad (3)$$

Recall that inserts are handled by placing new tuples with the appropriate pre-join in the WOS. This requires accessing each dimension table to perform the join. Thus, the cost

of a single insert batch is equivalent to the cost of reading all relevant dimension tables. *DTables* is the union of all dimension tables used by the physical design MVs (without pre-joins that set is empty) and equation 4 summarizes the cost of each insert batch (more on this in Section 5.5).

$$Cost(I_b) = \sum_{DT \in DTables} (\sum_{T \in Columns(DT)} ColumnCost(T)) \quad (4)$$

5. THE DESIGN ALGORITHM

5.1 Overview

At some very high level all database design algorithms, including ours, follow the same basic steps. As these algorithms progress, they generate a growing pool of design candidates. A final design will be chosen from this pool. The steps are characterized as follows:

1. For each training query, heuristically generate one or more physical candidate structures (e.g., indexes, MVs) that enhance that query's performance.
2. Generate *shared* candidate structures based on groups of similar queries that can serve multiple queries simultaneously. This step is typically to save space or to reduce maintenance cost
3. Pick a set of candidate structures that minimize total query runtime and do not violate the user (space) budget constraints. These candidates are added to the growing pool of candidates.
4. Repeat steps #2 and #3, until we do not see significant improvement from one iteration to the next.

A number of simple greedy techniques can produce a reasonably good result for step #3, particularly for limited space budgets because smaller budgets tend to result in designs with fewer candidates. The *Greedy(m,k)* technique, described in [11], introduces a greedy search over a candidate set produced by a limited (by parameter m) exhaustive enumeration. Our algorithm uses an ILP solution (Section 5.4) that will, for most problems, produce the optimal solution with respect to the candidate set in a reasonable period of time. Notice therefore, that the choice of good candidates is critical which is why we emphasize candidate generation in our approach.

Algorithm 1 Overall Design Algorithm

```
1: ΔRuntime ← X% {Stopping condition}
2: queryGroups ← {(Q_1), (Q_2)...(Q_n)}
3: MVPool ← {}
4: (Design, Runtime) ← ({BaseTables},{QueryRuntime})
5: repeat
6:    newQGSet ← groupQueries(queryGroups, maxJoins)
7:    for QGroup_i in newQGSet do
8:       MVPool ← MVPool ∪ buildCandidates(QGroup_i)
9:    end for
10:   queryGroups ← queryGroups ∪ newQGSet
11:   oldRuntime ← Runtime
12:   (Design, Runtime) ← BuildDesign(MVPool)
13: until (oldRuntime-Runtime)< ΔRuntime
14: return Design
```

In algorithm 1, on each iteration, we group the training queries such that each group represents queries that are similar in some respect. We anticipate that such groups will have similar needs. Therefore, we use these query groups to produce MV candidates.

Note that quality of designs produced at each iteration is monotonic since we only add candidates. The iteration stops when the overall training set runtime improvement over the previous iteration is less than some threshold. There are no hard rules for how to choose a good value of this threshold, particularly because it is affected by how good the previous pass was, but we found by trial and error that anywhere between 1% and 3% will normally not miss good designs and will complete in a reasonable amount of time. For the remainder of this paper, we set it to 2%.

The algorithm performs the following steps: first, it generates a set of query groups to serve as the basis for MV candidate generation (line 6). The details and the justification of query grouping are covered in Section 5.2; we can use the *maxJoins* parameter to control the number of candidates that are generated (more details in Section 5.2). This number is approximated based on the total number of candidates that the CPLEX solver can handle.

Once the new set of query groups has been generated, we proceed to build MV candidates. Our approach to MV candidate generation is discussed in Section 5.3. The new MV candidates are appended to the overall candidate pool (line 8). We then record the new query groups for the following iteration (line 10), cache the last known design (produced prior to the addition of the new candidates) and create the new optimal design (line 12) using the ILP solver. The details of applying the ILP solver are explained in Section 5.4.

5.2 Query Grouping

In this section, we discuss query grouping as the basis for MV candidate design. We defer the discussion of how to generate MV candidates given a query group until the next section. A query group is a set of similar queries (in the training set) that can all benefit from a single shared MV.

Previous work [8] has created designs by first generating candidates for a single query and then merging these candidates to reduce the total number of MV's in the design. In contrast, we group queries that are similar and design MVs for each group. In Section 5.3.2 we will explain why our approach is better than the approach sketched above for our setting. Several parts of our query grouping method have been adopted from the work in [19].

Intuitively, since we are looking for queries that will be well served by a shared MV, finding queries with similar predicates is important. In order to detect query similarity, we represent each query with a *selectivity vector*. The selectivity vector has one dimension (slot) for each of the M attributes in the schema. If query Q uses N distinct attributes (ignoring the join keys), it will be represented by an M-dimension vector with N non-default entries (default = 1), one for each attribute's selectivity (between 0.0 and 1.0). All other dimensions will be set to one, although for brevity, we omit these default values in the examples. Using the SSB [21] Query 2.1:

```
SELECT SUM(lo_revenue), d_year, p_brand1
FROM lineorder, dwdate, part, supplier
WHERE ... PK-FK joins ...
AND p_category = 'MFGR#12'
AND s_region = 'AMERICA'
GROUP BY d_year, p_brand1
ORDER BY d_year, p_brand1;
```

the selectivity vector might look as follows:

$$QV_{2.1} = (p_category : 0.025, s_region : 0.2, d_year : 1.0, p_brand1 : 1.0, lo_revenue : 1.0)$$

If attribute A is correlated with attribute B, then the selectivity of attribute B is dependent on the selectivity of attribute A. To address this, we apply *selectivity propagation* using the correlation data. The process of collecting correlation data and selectivity propagation is described in more detail in [19]. Briefly, for a uniform distribution, the selectivity of an attribute B that is correlated with attribute A is the $Selectivity(A) \times \frac{Card(AB)}{Card(B)}$. The significance of data correlations for MV design will be explained in detail in Section 5.3. The following selectivity adjustments can be made to $QV_{2.1}$ if there are two functional dependencies as shown.

$$QV_{2.1} = (p_category : 0.025, \rightarrow \mathbf{p_mfgr : 0.2}, s_region : 0.2, \rightarrow \mathbf{s_nation : 0.2}, d_year : 1.0, p_brand1 : 1.0, lo_revenue)$$

A partially pre-joined MV might be a better design candidate compared to the full pre-join either because it takes less space on disk or because it has a lower maintenance cost. Thus, we extend the framework adopted from [19] to support partially pre-joined MVs. Intuitively, if query Q1 accesses dimension tables D1, D2, and D3, and Q2 accesses dimension table D1, D4, and D5, then these queries are unlikely to be similar since they have only one dimension in common. However, if we are only considering a partial pre-join of the Fact table with D1, then in that context Q1 and Q2 would likely be similar. To capture this similarity we introduce a projection operator that lets us focus on the attributes relevant to a specific pre-join.

Here a projection of the selectivity vector eliminates attributes that are not available in the pre-join. It does so by setting all other attributes in the vector to 1. To use the same example query, a possible vector for Q2.1 is:

$$QV_{2.1}[lineorder, dwdate] = (d_year : 1.0, lo_revenue : 1.0)$$

In this case we want to capture a similarity that is relative to the join between *lineorder* and *dwdate*.

We now use these vectors to form query groups. For each considered pre-join, we use hierarchical clustering [15] to build a merge tree thereby generating new query groups, one for each node in the tree. When possible we consider the exhaustive set of pre-joins; the allowed maximum is provided as an input in algorithm 1.

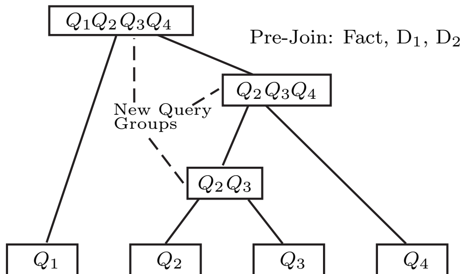

Figure 3: Query Merging - A Single Merge Pass

We begin with a known set of query groups (initially, singleton groups) and perform pair-wise merging based on a distance function that we will describe shortly. A single *merge pass* (Figure 3) produces a fully merged tree. Algorithm 1 will perform a new merge pass with all groups formed so far until it reaches the stopping condition. Figure 3 shows a hypothetical 4-query set that is being merged conditioned on a particular pre-join. If we were to perform a second merge pass, the new groups would be added to the initial merge set (i.e., the leaves). At every merge step the closest pair of query groups (i.e., most similar according to our distance function) is merged. In order to produce interesting new candidates, we avoid merging overlapping query groups. The distance between query groups is based on standard Cartesian distance normalized using the expected size of the MV candidate for each query group. It is impossible to compute the exact candidate size since the candidates are not yet generated. However, we use the LZO-compression estimate for all attributes in order to approximate the size. For example, *d_year* compresses to approximately 1.00 byte per row, while *lo_revenue* compresses to 3.74 bytes per row. The distance function is therefore:

$$D(V_1, V_2) = Cartesian(V_1, V_2) \times \frac{LZO(V_1 \cup V_2)}{LZO(V_1) + LZO(V_2)}$$

The absolute value of this distance does not have a concrete meaning; however, we expect that queries that are similar will have lower distances than queries that are not. The idea of merging similar structures has already been established by [8]. Even though we merge queries instead of merging MVs (as is done in [8]), there are some conceptual similarities. In the case of [8], the basic similarity measure is the overall query penalty (i.e., performance deterioration) resulting from MV merge. This similarity measure is normalized by the estimated size penalty resulting from that merge. In our algorithm we choose to merge queries instead of MVs, since important merging information may be lost when a user query is "transformed" into a dedicated MV (Experiment 2 in Section 6 demonstrates the dangers of merging MVs directly). In the next section, we will discuss a number of reasons why that is the case and detail our approach to generating MVs for a query grouping.

5.3 Generating Candidate MVs

In this section we present our approach to generating MV candidates that are tailored to a specific group of queries. First, we explain how single-query (i.e., *dedicated*) MVs are generated and then we extend our approach to the generation of multi-query MVs. Row-store literature typically approaches these two tasks in a fundamentally different manner: dedicated MVs are generated with heuristic query analysis and shared MVs are generated by merging the already-generated MVs [7]. Before we move on to explain multi-query MV design, we present the shortcomings of the MV merging strategy in the context of C-Store.

5.3.1 Single-Query Candidate MVs

The problem of picking tables to pre-join is not very different in C-Store as compared to a typical row-store. Even though query execution in C-Store differs significantly, the strategy for minimizing the cost of a join remains the same and consists of *a)* minimizing the I/O cost for reading the tables and MVs that are being joined and *b)* minimizing the size of the intermediate join results. In our work we try to consider all possible pre-joins limited by the threshold that was discussed in the previous section. Thus, in the following discussion we will concentrate on the problem of choosing the MV clustering key (i.e., sort order) for a given pre-join.

In order to avoid the well-known problem of secondary indexes for data warehouses, the typical use of B-Trees in this context is to create a covering index with a composite clustering key. Composite B-Trees support arbitrary predicate evaluation over the first attribute in the index. Multi-attribute access demands (1) that a prefix of the key must be used, (2) all but the last of the queried attributes in the prefix have an equality predicate. Thus, the clustering key for a dedicated MV for a particular query, should contain all of the equality predicates from that query followed by the single lowest-selectivity range predicate. This indexing strategy minimizes the amount of data that the query accesses thereby proving the single best answer to designing a dedicated composite index in a row-store.

The same problem is far more complex in C-Store. The expected quality of an index is hard to anticipate both because the index attributes are RLE-compressed and because index and target columns are accessed individually. Each of these factors affects index performance in its own way. RLE compression means that most of the RLE-encoded attributes are cheap to read (few I/O's) and process. Column-store architecture allows reading just the columns that are needed - which makes index access more flexible than in a row-store. However, each column read requires a seek to a different file. We will explain the impact of both of these factors as we explain the algorithm pseudo-code.

Algorithm 2 MV Design Algorithm (single query)

1: Parameters: Query, PreJoin
2: ModQuery ← Query [PreJoin]
3: Prefixes ← buildPrefixes(ModQuery)
4: **for** Prefix in Prefixes **do**
5: cost = 0
6: **for** Attr_i in ModQuery.Attrs **do**
7: cost = cost + evalPrefix(Attr_i, Prefix)
8: **end for**
9: bestPrefix ← record lowest cost prefix
10: **end for**
11: Suffix ← {}
12: RankedAttrs ← rank (ModQuery.Attrs−bestPrefix.Attrs) (rank the non-prefix attributes using Equation 5)
13: **while** Cardinality(Prefix + Suffix) < $\frac{MVRows}{3}$ **do**
14: Suffix.add(RankedAttrs.best())
15: **end while**
16: newMV ← ModQuery.Attrs sorted on (Prefix+Suffix)
17: **for** Attr_i in (ModQuery.Attrs − (Prefix + Suffix)) **do**
18: newMV.AssignEncoding(Attr_i)
19: **end for**
20: return newMV

Algorithm 2 describes the design process for a single MV with an already specified pre-join. For purposes of designing a sort order, we will ignore the attributes not covered by the specified pre-join (line 2) and introduce the necessary foreign keys. Next we generate all possible prefixes (line 3), where a prefix is the leading part of the sort order that determines the read bitmap as described in Section 4. We add columns to the composite sort order until the resulting bucket size is equal to one disk page (since that is the I/O unit). As we generate prefixes, we apply a number of filtering heuristics (e.g., equality predicate will always result in better performance compared to a range predicate). The space of all possible prefixes is limited, because the number of pages per column is relatively small due to compression. For example, a column with 10M integer values can easily fit into about 100 disk pages (without resorting to RLE) and prefix generated can stop when the number of buckets exceeds 100. This means that prefixes will in general be fairly shallow. In lines 4-10 we evaluate the set of prefixes and select the one with the lowest cost. The *evalPrefix* function in line 7 is a cost model call that estimates the expected read bitmap cost for each column accessed by the query.

Once the best prefix is chosen, we extend that prefix by adding more columns (building a *Suffix* or the *rest* of the sort order) to further improve column compression through RLE. We do not have the space to present the evidence, but the C-Store update framework (Section 2.3) nearly eliminates the penalty associated with long composite indexes that we see in row-stores. Therefore, we extend the sort order to the maximum length for which RLE still provides any compression benefits (line 13): RLE compression is effective as long as we are encoding runs of at least 3 identical values. We use the following equation

$$AttrBenefit = \frac{(1 - Selectivity_{predicate})}{Log(Cardinality_{attribute})} \quad (5)$$

to rank the remaining query attributes in line 12 of algorithm 2. The intuition here is as follows - the numerator of the formula minimizes the selectivity value as is important in a row-store. The denominator accounts for the fact that the maximum length of the sort order is constrained by its composite cardinality (condition in line 13). Therefore, attributes with lower cardinality are preferable since more of them can be added to the sort order. This is a heuristic approximation which has shown to do reasonably well in our experiments.

Finally, we need to assign individual column encoding schemes for the remaining (i.e., non-RLE) columns (lines 17-19). We choose a custom compression scheme (e.g., dictionary or delta compression) based on the data type and the cardinality of the column. The estimated size of each compression scheme is compared against the default LZO compression. Recall that compression in C-Store is page-based, therefore we estimate dictionary compression using per-page cardinality rather than general attribute cardinality. The per-page cardinality may decrease due to correlation with the sort order - for example if the MV is sorted on *state* then each page in the *city* column is going to have fewer unique values (e.g., the first 2% of the cities are going to be from AL only and so on).

Our algorithm has been tailored to C-Store, but our techniques can be applied in row-stores, since row-store DBMSes have some similar features. For example, Oracle's *skip-scan* mechanism is a way to query an index with more than one range predicate. This is done by replacing the leading range predicate by a number of equality queries (e.g. using "(A=5 AND B>10) OR (A=6 AND B>10)" instead of "5≤A≤6 AND B>10"). See Figure 5 in Section 6 for an example. A more general approach to efficiently apply multiple predicates is through using bitmap indexes for the columns in the sort order. Such a bitmap index would also be very amenable to compression because matching values are collocated.

5.3.2 Merging Materialized Views

Row-store designers typically merge MVs in order to limit the total number of MVs in the design. We discuss some of the possible pitfalls of merging MVs in C-Store. An MV is a combination of a pre-join and a sort order. Merging a

pre-join is not difficult. The problem of merging indexes has been covered in [13] and the accepted solution in a row-store is to concatenate the two parent index keys eliminating any redundant attributes. Row-stores cannot access individual columns of the index, interleaving of the index attributes has a very deleterious effect on all of the queries that were helped by the original indexes. In contrast, since column-stores can access columns individually, interleaving of index attributes is often a better choice than simple concatenation. Evaluation of all feasible attribute interleavings is expensive – therefore we choose to build a new MV for each group of queries (Section 5.3.3) that we encounter. For us, query grouping substitutes for index merging.

In some cases, even a row-store may benefit from attribute reordering during index merging. For example, a common second attribute that appears in both parent indexes could be beneficial as the first attribute in the merged child index. Thus, our approach of reconstructing the index based on the queries may be useful in a row-store.

5.3.3 Multi-Query Candidate MVs

Next, we present our approach to generating MVs for a multi-query group. We extend Algorithm 2 presented in Section 5.3.1. Most of the algorithm can be used without any modifications – we only need to make two adjustments.First, we build a prefix for multiple queries. We do this by computing the average expected cost of each prefix over all queries in the group. The candidate prefixes are chosen in the same way as before.

The second necessary modification concerns the extension of the sort order. In addition to averaging the attribute ranking over all queries in the group, we also modify Equation 5 in the following way:

$$AttrBen_{Q_i} = Read(Prefix, Q_i) \times \frac{(1 - Selectivity_{pred})}{Log(Cardinality_{attr})} \quad (6)$$

This formula is augmented with an additional term that accounts for the amount of data read by each query. Intuitively, a predicate with selectivity of 0.10 will eliminate 90% of column rows. However, if Q_i only reads half of the column based on the prefix, then the same predicate will only filter out 45% of column rows since it is being applied to half of the column.

5.4 Picking the Best Candidates

Using an ILP solver to find the optimal solution has been proposed in [23] and [19]. The goal is always the same - pick a subset of MV design candidates from the candidate pool, subject to a budget. The summary of the variables that we use in the problem formulation are listed in Table 2.

m	An MV_m from the candidate pool M.
q	A query from query set Q. $q = 1, 2, .., \|Q\|$.
B	Space budget.
s_m	Size of MV_m.
$t_{q,m}$	Estimated runtime of query q using MV m.
$p_{q,r}$	r-th fastest MV for query q. $(r_1 \leq r_2 \Leftrightarrow t_{q,p_{q,r_1}} \leq t_{q,p_{q,r_2}})$.
$x_{q,m}$	Whether query q is penalized for not having MV m. $0 \leq x_{q,m} \leq 1$
y_m	Whether MV_m is chosen.

Table 2: ILP Formulation Variables

The ILP Objective function is:

$$\sum_q \left(t_{q,p_{q,1}} + \sum_{r=2\ldots|M|} x_{q,p_{q,r}} (t_{q,p_{q,r}} - t_{q,p_{q,r-1}}) \right) \quad (7)$$

and it is subject to the following constraints:

$$1 - \sum_{k=1}^{r-1} y_{p_{q,k}} \leq x_{q,p_{q,r}} \leq 1 \quad (8)$$

$$y_m \in \{0, 1\} \quad (9) \qquad \sum_m s_m y_m \leq B \quad (10)$$

The objective function is designed to minimize the total runtime of the query workload. Intuitively, it sums up the penalties incurred by each query depending on which MVs were chosen. When every query gets its dedicated MV, the sum of penalties is 0. If an inferior MV is used due to budget constraints, then each $(t_{q,p_{q,r}} - t_{q,p_{q,r-1}})$ element represents the penalty incurred as a result of choosing $(r)^{th}$ best MV instead of $(r\text{-}1)^{st}$. Equation 8 constraints $x_{q,m}$ based on which MVs were chosen (y_m). Equation 9 ensures that y_m is boolean, since MV_m can is either chosen or not. Finally equation 10 limits the total size of the chosen MVs to the budget B.

We solve the stated ILP problem using a commercial LP solver (ILOG CPlex [2]). The values of y_m returned identify the ideal subset of MVs that will constitute the output design.

5.5 Handling Inserts

Most data warehouse workloads contain inserts. The maintenance costs increase dramatically as the design size increases. In fact, the space budget sometimes serves as a metaphor for expected maintenance overhead. Note, as stated earlier, we assume that inserts arrive in batches of known size and frequency.

In row-stores, the insert cost is directly determined by the number and size of indexes in the design. Indexes are expensive to maintain because they keep data sorted and each new insert is likely to touch a new disk page. C-Store amortizes the maintenance cost both by buffering inserts in memory (similar to what InnoDB [3] does) and by writing these batches to disk. The amortized insert cost cannot be estimated based on the size or number of MVs in the design because MVs are not immediately updated.

We adapt our ILP solution to account for the insert costs by partitioning the MV candidates based on which pre-joins they require. We then run the ILP solver on each partition to select the best design for that partition. For every design, the widest (union of used dimension tables) required pre-join will determine the immediate overhead incurred with every inserted batch of rows. Intuitively, a design built from the candidate partition that avoids pre-joins with a large dimension table will incur lower insert overheads.

Inserts also incur move-out and merge-out penalties in the long run. However, these costs are amortized and thus negligible when compared to the immediate cost of the inserts. For example, assuming a 500MB WOS buffer, most SSB designs would be able to buffer over 1M rows before the first move-out has to be performed. Since a move-out occurs infrequently and does not take significat time (e.g., a few seconds every hour), we can ignore this in our cost formula.

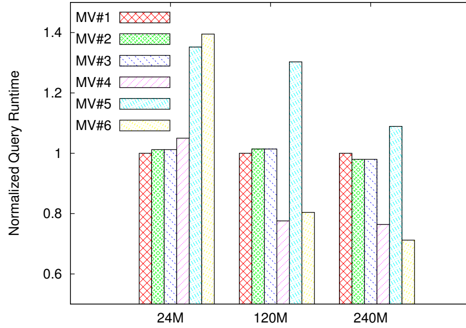

Figure 4: Different Sort Orders in C-Store

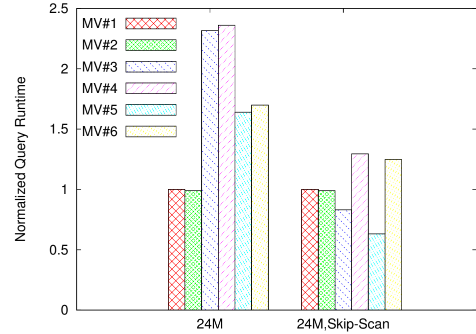

Figure 5: Covering Indexes and with and without skip-scan in DBMS-X

6. EXPERIMENTS

We ran our experiments using the C-Store on machines running Fedora Core 6 with AMD64/3000+ processor, 2G of RAM and a 320G 7200RPM SATA II disk. We flushed the cache between runs by using the Linux /proc/sys/vm/-drop_caches mechanism. In order to get stable results we averaged multiple (at least 3) repeated runs after removing outlier results. We use the SSB [21] benchmark, which is a variation of the TPCH benchmark. Unless otherwise specified, we use a Scale 4 sized dataset.

In some of our experiments we use a popular commercial row-store DBMS to demonstrate the architectural differences between column and row-stores. We will refer to this row-store as DBMS-X. Both the DBMS-X and C-Store results are normalized with respect to their respective baseline performance, to avoid comparing row-store and column-store performance directly. This baseline is computed differently for each experiment (as specified in the text).

Experiment 1: The Effect of Sort Orders

In this experiment, we plot the runtimes of the following query using the dataset described above.

SELECT SUM(revenue)
FROM lineorder, dwdate
WHERE ... joins ... **AND** DayNumInYear > 240
AND Quantity > 24 **AND** Discount > 2;

The cardinalities of DayNumInYear, Quantity, and Discount are 365, 50 and 11 respectively, while the selectivities are 0.3, 0.5 and 0.7, respectively. The goal here is to demonstrate that unlike in a row store, the choice of the sort order in C-Store will depend on the size of the MV columns on disk. We don't always want to use selectivity as a guide when choosing the sort order because the read bitmap is often more important in determining how many I/O's are required. As observed earlier, the mapping of the read bitmap to physical pages changes as the physical size of the column changes.

In this experiment, we generate 6 pre-joined MVs representing all possible sort orders and run the same query against each MV while varying the scale (# rows) of the dataset. The sort orders are:

SO#1: DayNumInYear, Discount, Quantity
SO#2: DayNumInYear, Quantity, Discount
SO#3: Discount, DayNumInYear, Quantity
SO#4: Discount, Quantity, DayNumInYear
SO#5: Quantity, DayNumInYear, Discount
SO#6: Quantity, Discount, DayNumInYear

All runtimes in Figure 4 are normalized with respect to MV #1 that uses the sort order based on selectivity (SO#1). Note that at 24M rows, DayNumInYear is actually a good leading choice (SOs #1 and #2 are doing well). However, as the number of rows increases, other sort orders begin to dominate. At 120M rows, MV #3 and MV #5 are winning by 20%, and by the time we reach 240M rows MV #6 is the best choice, while MV #1 and MV #2 are close to the worst choices. The difference may not seem dramatic, but keep in mind that in this case every column except for the revenue column is RLE compressed. If the query had predicates over more columns, some of these columns might not appear in the RLE portion of the sort order making the performance gap larger.

The reason for this variation in performance with dataset scale is fragmentation. Intuitively, when the row count is small, each accessed bucket is relatively small (compared to the page size), so the number of pages read is commensurate with the selectivity of the leading column in the sort order. However, as the target column (i.e., revenue) becomes larger, the buckets are large enough that the I/O cost of the query will depend on the particular RLE prefix. Buckets will often contain a high proportion of irrelevant data. In that case, leading with a lower cardinality attribute is often a good idea even if that attribute has a high selectivity.

Figure 5 shows how using skip-scan in a row-store also benefits from our MV design technique. The set of bars on the left show how DBMS-X performs using the same sort orders as in Figure 4, while the bars on the right show what happens with these sort orders when we simulate skip-scan described in Section 5.3.1 (note: DBMS-X does not support this directly). The better performance on the right is due to the fact that the skip-scan can take advantage of the sort orders that we design. Thus, a row-store designer could benefit from our MV design technique as well, although not as much as in our column-store.

Experiment 2: Merging Sort Orders

Consider the two queries Q_1 and Q_2 with two predicates each as shown:

Q_1:
SELECT SUM(revenue)
FROM ... **WHERE**
quantity **BETWEEN** 25 **AND** 30 **AND**
nation **BETWEEN** 'Germany' **AND** 'Japan';

Q_2:
SELECT SUM(revenue)
FROM ... **WHERE**
orderdate **BETWEEN** 94-05-01 **AND** 94-06-25
city **BETWEEN** 'France 0' **and** 'Japan 0';

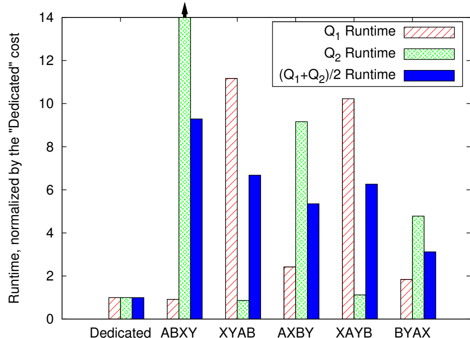

Figure 6: Merging sort orders in C-Store

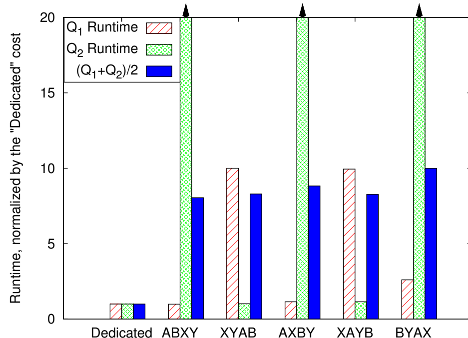

Figure 7: Merging indexes in DBMS-X

We use labels A=quantity, B=nation, X=orderdate, Y=city to save space in the figure. Assume that the best sort orders are as given (AB for Q_1 and XY for Q_2), and their performance is labeled as "dedicated" in the figure. Now consider the problem of merging these two sort orders. Recall that in a column-store, it is easy to use a subset of the columns in a sort order, so interleaving the two parent sort orders is a viable option. Furthermore, we will show that the presence of functional dependencies can have a significant effect on the result.

We have produced indexes with different permutations of the sort order columns from the example above. Figure 6 shows the real system runtimes in C-Store normalized by the runtime of the dedicated indexes (i.e. we measure the slowdown resulting from the merge). Note that the first two merges (ABXY, XYAB) correspond to what a designer would choose in a row-store. The results are as expected. One of the queries (the one with the predicates that lead in the merge) exhibits no slowdown and the second query slows down by an order of magnitude. The two following indexes are simple interleavings (AXBY, XAYB) and the average performance of both is better than either of the concatenated indexes. When indexes are interleaved, both queries slow down, but each is able to benefit from the combined index to some degree. Finally, we come to the most surprising result of all. Notice that the average performance of the index BYAX is much better than all other indexes, despite the fact that the leading column of the merge is the 2nd column of the first dedicated index followed by the 2nd column of the second. This can be explained by a functional dependency that exists between B and Y (city→nation). This reduces the read bitmap fragmentation (as explained in Section 4) by reducing the number of buckets induced by the sort order prefix and lets Q_2 benefit from the 2nd column of the sort order (city) as if it were the leading column in the index. Or to put it another way, because of the functional dependency between city and nation, a predicate on the city column carries an implied predicate on nation.

To confirm that the state-of-the-art wisdom about index merging in a row store is accurate, we re-run the same combinations in DBMS-X. Figure 7 shows the corresponding results. Interleaving indexes is not beneficial in a row-store. Although in this case interleavings AXBY and XAYB slow down the "winning" query only by a little (approximately 15%) it still does not benefit the "loser" query in any way. The final interleaving (BYAX) is even worse.

Experiment 3: A Human DBA Evaluation
Next we compare our design for the SSB Benchmark to one produced by an experienced C-Store DBA. This particular DBA produced the physical design by partitioning the queries into a different number of groups and producing one fully pre-joined MV per group (i.e., 1 MV, 3MVs and 4MVs). Figure 8 shows the design curve produced by our design tool and the corresponding cost model estimate curve.

The designs compare pretty well as they were done by an experienced DBA. However, the DBA did not consider designs for budgets of less than 275MB. Note that the reason one of the DBA designs stands out as a seemingly "bad" design point (at 520MB), is, once again, compression. The "bad" design point is a single materialized view (i.e. single query group) and compresses relatively poorly. It is difficult for the DBA to anticipate the final design size. We also include a curve that shows our cost model's predicted runtime, and it is shown to be extremely accurate.

Experiment 4: The Row-Store Strategy
In this experiment, we evaluate a popular MV design technique for row-stores in the context of our CCS. We build the sort order based solely on predicate selectivity instead of using our algorithm. Note that in the absence of special DBMS functionality (such as skip-scan), this is a best choice for a row-store (see Experiment 1).

As can be seen in Figure 9, row-style MVs design does not produce good candidates in our setting. As we had argued in Section 4 and demonstrated in Experiment 1, considering the actual bitmap access pattern (which depends both on the particular sort order prefix and the target size) is important for designing a good sort order. Note that the designs do not diverge immediately and are closer at smaller budgets. That is because smaller MVs have fewer columns, thus there are fewer opportunities to make a mistake ordering the columns by selectivity.

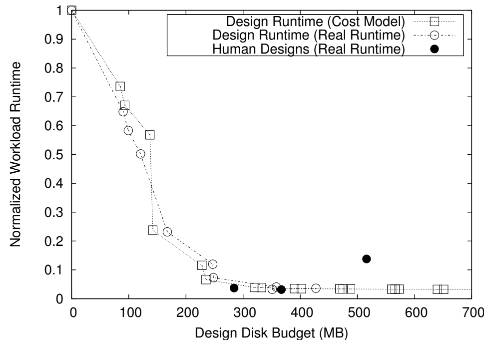

Figure 8: SSB Benchmark Runtimes

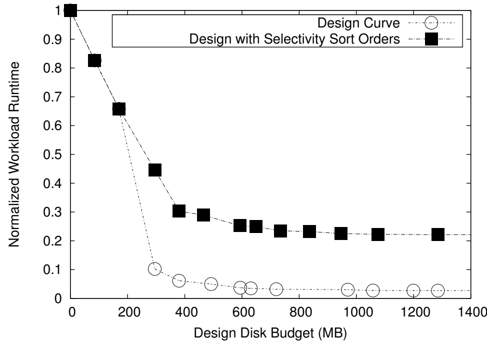

Figure 9: Sort Order Design vs Selectivity-based Ordering

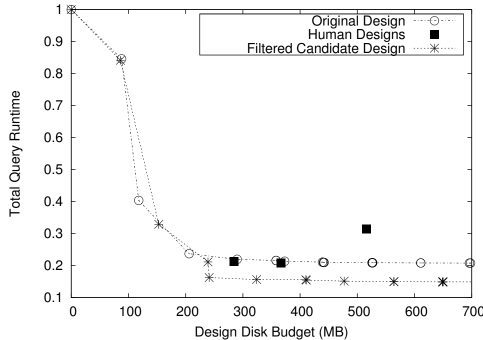

Figure 10: SSB Workload with High Insert Rate

Experiment 5: Insert Heavy Workloads
The design curve in Figure 10 shows the effect of high insert rate on the workload runtime. We assume a workload that, in addition to the read original queries, includes 12 batches of 10K row inserts. With a sufficiently high insert rate, the insert component of the workload cost becomes non-negligible (see Section 3). The first two curves are the designs from Figure 8 that now include the insert penalty caused by the pre-join penalty associated with every inserted batch. The 3rd curve in Figure 10 corresponds to the design built using a subset of MVs using the MV candidate partitioning technique as described in Section 5.5. Although such designs may be inferior when there are no inserts, avoiding MV candidates that incorporate expensive pre-joins becomes an advantage in the presence of high insert rates.

7. RELATED WORK

There has been a lot of research on automatic database design over the last decade. In fact, most commercial products have a tool or wizard that embodies some of that work.

The bulk of the previous work has been done in the context of row-stores. As observed earlier, the bulk of the problem lies in generating good design candidates. There are some fundamental differences in the way we explore the design space when we consider the problem in the context of a column-store, and we have discussed these differences throughout this paper.

Building a set of MVs in a column store is similar to picking materialized views and indexes in a row store. Selecting a pre-join and a column set for a projection corresponds to picking a materialized view, and generating sort orders loosely corresponds to composite index selection. However, in our work, we are essentially picking primary indices while in previous work, attention has mostly been focused on secondary index selection. As we explain in Section 2.4, the design of primary and covering secondary indexes is similar.

AutoAdmin [12] represents pioneering work in automatic design. Their basic approach begins by generating the best possible single-column indices. They then widen the space by considering two-column indices that contain an already generated one-column index. While in theory this process could consider deeper keys, very wide composite secondary indices do not work well in a row-store. In contrast, we indirectly start with ideal MVs for each query as the base clustering round, however, we never limit our sort orders to a small number of columns (quite the contrary), and we do not require that longer sort orders contain "good" prefix orders.

AutoAdmin also does a post pass to merge MVs that were created in the first phase. They do this as a way to account for constrained space. For us, adding columns to the sort order can actually make MVs smaller because of the effect of RLE compression. Thus, we cannot separate the two activities. It is tempting to say that view merging is like our query clustering, but query clustering is done in advance of sort order selection, and it is done using different metrics.

Workload compression has been studied in [26, 6, 10] and allows clustering queries based on common features. It merges queries that are similar to reduce the size of a large workload. For example, a simple scheme would merge queries that are identical except for constants. The AutoAdmin [10] algorithm uses pre-processing to eliminate columns that should have little impact. This is reminiscent of our query clustering.

The work in [19] has used an approach similar to ours to produce a design for a row-store. However, in the context of a row store, the problem of designing dedicated MVs is a trivial one. Kimura, et al. use a similar query grouping mechanism, but rely on sort order merging to produce the shared MVs. In our work, we also consider partial pre-joins in order to improve our MV candidate pool and to handle insert workloads (the latter does not require a special treatment in a row store).

The DB2 Design Advisor [26] identifies dependencies between different features (such as Materialized Query Table or Index) and searches the space accordingly, ensuring that dependent features are searched in tandem. Virtual features are enabled and plans are generated given the real and the virtual features. If the plan makes use of one of the virtual features, that feature is suggested as a possible addition to the physical design. Different features are each given a different fraction of the space. The DB2 Design Advisor also relies on query compression to reduce the need to re-compute estimated query cost with every step.

The work in [18] presented a mechanism for dynamically reorganizing data in the column store context. The column data is partially sorted based on what is accessed by query workload. That approach has the advantage of continuously adapting to workloads, while the standard design requires an expensive transition period to implement. However, it also prevents the DBMS from taking advantage of the compression opportunities.

MV selection has been studied at length [14, 16]. This work tends to explore MV's with pre-computed aggregates. It is the savings in aggregate computation that the MV is

designed for. For us, MV's are not lossy which is to say that our MV's do not reduce rows, and can, therefore, be used for general queries. In [5, 16] a simple linear cost model is used. This cost model estimates the number of rows a query would have to process. Instead, we use an I/O based cost model that estimates the number of disk blocks accessed and seeks performed.

AutoPart [22] is concerned with choosing a good vertical partitioning of large database tables to speed up queries. In a column store the vertical partitioning is not an issue since each column is partitioned separately. An MV looks a little like a vertical partition, but it is important to remember that even though we write a list of columns together to describe an MV, each column remains as a separate file. The AutoPart algorithm also uses categorical attributes to generate a horizontal partitioning. It then combines the horizontal and vertical partitions to form composite fragments that can be thought of as rectangular chunks of the table.

Recent work in [20] describes the architectural decisions made by Vertica, including the differences with the original C-Store paper [24]. Vertica includes a physical design tool, but description of the internals of that tool was outside the paper's scope. When selecting the best encoding for non-RLE columns, we rely on computing empirical compression estimates, similar to the approach adopted in [20].

8. CONCLUSION AND FUTURE WORK

We have presented the design of a tool for automatically producing physical database designs for a CCS. Our algorithm selects effective materialized views with compound sort orders. We have also showed the effect of compression and the use of functional dependencies on these designs. We demonstrated that even though our algorithms were inspired by CCS's, some of the techniques can be applied to row-store designers as well. We also presented an experimental analysis of our algorithms and observations.

We are currently considering a number of extensions. Although beyond the scope of this paper, we have done some preliminary work for distributed database design. C-Store relies on MV replication to achieve fault tolerance. The safe but naive approach is to replicate the design as necessary. However, the replicated MVs can have different sort orders since the hash-partition key is typically orthogonal to the sort order. Therefore, the idea of query grouping can also be used to generate multiple sort orders.

We are also interested in building a tool that produces incremental designs. The idea here is to produce a new design that is "close" to the existing design. Since a warehouse can contain hundreds of terabytes of data, it is infeasible to load a new design every time query workload changes. Instead, it would be better to create a new design that is easier to achieve from the old one, and that can deliver, say, 80% of the benefit of the best one.

9. REFERENCES

[1] Create indexes with included columns. http://msdn.microsoft.com/en-us/library/ms190806.aspx.

[2] Ibm ilog cplex optimizer. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/.

[3] InnoDB Engine. http://www.innodb.com/.

[4] Vertica. http://www.vertica.com/.

[5] D. J. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD Conference*, pages 671–682, 2006.

[6] S. Agrawal, S. Chaudhuri, L. Kollár, A. P. Marathe, V. R. Narasayya, and M. Syamala. Database tuning advisor for microsoft sql server 2005. In *VLDB*, pages 1110–1121, 2004.

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

[8] N. Bruno and S. Chaudhuri. Automatic physical database tuning: A relaxation-based approach. In *SIGMOD Conference*, pages 227–238, 2005.

[9] S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Record*, 26(1):65–74, 1997.

[10] S. Chaudhuri, A. K. Gupta, and V. R. Narasayya. Compressing sql workloads. In *SIGMOD Conference*, pages 488–499, 2002.

[11] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for microsoft sql server. In *VLDB*, pages 146–155, 1997.

[12] S. Chaudhuri and V. R. Narasayya. Autoadmin 'what-if' index analysis utility. In *SIGMOD Conference*, pages 367–378, 1998.

[13] S. Chaudhuri and V. R. Narasayya. Index merging. In *ICDE*, pages 296–303, 1999.

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

[15] J. Han and M. Kamber. *Data Mining: Concepts and Techinques*. Morgan Kaufmann Publishers, 2nd edition edition, 2006.

[16] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *SIGMOD Conference*, pages 205–216, 1996.

[17] S. Héman, M. Zukowski, N. J. Nes, L. Sidirourgos, and P. A. Boncz. Positional update handling in column stores. In *SIGMOD Conference*, pages 543–554, 2010.

[18] S. Idreos, M. Kersten, and S. Manegold. Database cracking. 2007.

[19] H. Kimura, G. Huo, A. Rasin, S. Madden, and S. B. Zdonik. CORADD: Correlation Aware Database Designer for Materialized Views and Indexes. In *Proceedings of the 36th International Conference on Very Large Data Bases*. VLDB Endowment, September 2010.

[20] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandier, L. Doshi, and C. Bear. The Vertica Analytic Database: C-Store 7 Years Later. *CoRR*, abs/1208.4173, 2012.

[21] P. O'Neil, E. O'Neil, and X.Chen. The Star Schema Benchmark (SSB). http://www.cs.umb.edu/~poneil/StarSchemaB.PDF.

[22] S. Papadomanolakis and A. Ailamaki. Autopart: Automating schema design for large scientific databases using data partitioning. In *SSDBM*, pages 383–392. IEEE Computer Society, 2004.

[23] S. Papadomanolakis and A. Ailamaki. An integer linear programming approach to database design. In *ICDE Workshops*, pages 442–449. IEEE Computer Society, 2007.

[24] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A Column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[25] C. Yang, C. Yen, C. Tan, and S. R. Madden. Osprey: Implementing MapReduce-style fault tolerance in a shared-nothing distributed database. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 657–668. IEEE, 2010.

[26] D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. J. Storm, C. Garcia-Arellano, and S. Fadden. DB2 design Advisor: Integrated Automatic Physical Database Design. In *VLDB*, pages 1087–1097, 2004.

[27] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.