Abstract of "Design Tool for a Clustered Column-Store Database" by Alexander Rasin, Ph.D., Brown University, May 2011.

The goal of an automated database designer is to produce auxiliary structures that speed up user queries within the constraints of the user-specified resource budget (typically disk space). Most existing research on automating physical database design has been performed in the context of commercial row-store databases such as Microsoft SQL Server or IBM DB2. In fact, every commercial database offers some sort of a tool that can provide design recommendations for the consideration of the database administrator. An automated tool is necessary not only because a database administrator is not always available but because the complexity of the design problem is constantly increasing: new auxiliary structures and query processing methods continue to be introduced, and more users and queries need to be serviced.

This dissertation extensively investigates the problem of automating physical database design in the context of a column-store that supports clustered indexing. In the experiments presented here, we primarily used Vertica, a commercial column-store database that is based on the C-Store research project jointly developed at Brandeis, Brown and MIT. Although on the surface it seems like only the underlying storage system has changed, while the problem of designing the physical structures remains essentially the same, we found several fundamental differences that make physical design in a clustered column-store database a unique problem. Many of the basic axioms that are used in a row-store design are invalid in a column-store design (and vice versa). In this dissertation, we demonstrate the construction of an effective design tool and an analytic cost model for use in a column-store. We show that certain techniques from machine learning such as clustering can reduce and simplify this design problem. To our knowledge there has been little previous work on the problem of physical design in the context of column-stores and none in the context of column-stores such as C-Store or Vertica.

Design Tool for a Clustered Column-Store Database

by

Alexander Rasin

Sc. M., Brown University, 2003

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island

May 2011

This dissertation by Alexander Rasin is accepted in its present form by the Department of Computer Science as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date ______________ ______________________________
Stan Zdonik, Director

Recommended to the Graduate Council

Date ______________ ______________________________
Uğur Çetintemel, Reader

Date ______________ ______________________________
Tom Doeppner, Reader

Approved by the Graduate Council

Date ______________ ______________________________
Peter M. Weber
Dean of the Graduate School

Vita

Alexander Rasin was born on August 20, 1978 in Dnepropetrovsk, Ukraine. He had received his B.Sc. degree in Computer Science and Mathematics from Brandeis University, MA in 2001. He then joined the doctoral program in the Computer Science Department at Brown University in Providence, Rhode Island and received his Sc.M. degree in Computer Science in May, 2003.

Acknowledgements

First, I would like to thank my advisor, Stan Zdonik. He has been a very patient mentor for many years – and taught me everything I needed to know about research. Stan taught me that in databases one never has to settle for anything because databases come in contact with all research areas. He had taught me how to write papers and how to channel the inner "grumpy reviewer." I would not be where I am now if it were not for Stan.

I am very grateful to Uğur Çetintemel for his regular advice and the perpetual willingness to answer any question. Uğur provided impartial feedback on my work and regularly reminded to use my time at Brown wisely. I would also like to thank Tom Doeppner who was always ready to meet me when I asked, and who was extremely patient listening to my (hand-waving) presentations.

I am grateful to Sam Madden for serving as an inspiration on how to quickly and efficiently revise a paper and for helping me find a job. I would also like to thank Hideaki Kimura. I was able to draw on many useful ideas from our collaborative projects, and even assimilate some of his code for my experiments.

There are literally hundreds of people who have made my graduation possible. I would like to thank all of my colleagues from the Aurora/Borealis research group, as well as the C-Store research group. I would also like to thank the technical and administrative staff at Brown that kept my computers and my paperwork in order for many years. The experiments in this dissertation were made possible by the assistance from many friendly engineers at Vertica. I am also grateful to all of my friends, roommates and officemates that had made it possible for me to survive Providence. Finally, I would like to thank my family whose continuous support (sometimes expressed through letting me work undisturbed) made this dissertation possible.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction to Databases

A database is a collection of data organized for efficient user access. The data usually belong to a particular category, such as historical data (e.g., data from a stock market ticker or store transactions history) or spatial data (e.g., the astronomical survey data described in [SDS]). Almost every organization in the world maintains a certain amount of data and thereby needs a database to manage it. As the amount of managed data grows, database performance becomes a progressively greater concern. In this work, we study the problem of optimizing large databases (there is no precise definition for *large*, but small databases usually run fast enough without additional tuning). Such databases, which typically house historical are referred to as *data warehouses* [DBB07].

The contents of a database are managed by a *database management system* (DBMS). The DBMS is responsible for storing the data, providing access to the data, keeping track of modifications to the data and recovery in case of failure. There are a large number of widely used DBMSs ranging from open source [pos, Mon, CSt] to commercial [sqlb, DB2, Syb, Ver]. Companies typically employ *database administrators* (DBAs), who are responsible for fine-tuning the performance of the DBMS. Most DBMSs come with wizard tools (designed for users or DBAs) that attempt to simplify and automate the process of configuring the database. Chapter 3 examines the research, published in the context of the most widely used DBMSs, that has addressed the problem of tuning a DBMS.

This chapter introduces and explains the basic concepts and terminology necessary to understand the context in which DBAs and automated design tools carry out database tuning work. We begin by describing the concept of *relational databases* and the difference between a *column-store* database and *row-store* database. We then cover the query language used in databases and finally formulate the problem of database design.

1.1.1 Relational Databases

The *relational data model* has been the dominant approach used in database design and management over the past few decades. In this model, data is grouped based on real-world connections. For example, a database entry about a certain customer's transaction forms a *tuple* containing a transaction ID, purchase date, purchase time and purchase amount (it might also contain a number of other pieces of information such as customer ID). Each entry in the transaction tuple forms an *attribute* of the tuple, and the collection of these tuples forms a *relation*. The manipulations performed on these tuples and relations can be described and analyzed using *relational algebra* [SKS02]. We will consider this notion further when we discuss the *query optimizer* component of a DBMS in Section 2.2. In a relational database, attributes are referred to as *columns*, tuples correspond to *rows* and relations are *tables* (see Figure 1.1). Database tables are manipulated using *Structured Query Language* (SQL, see Section 1.1.4) and table definitions are a part of what is collectively known as the *database schema* (Section 1.2.1).

1.1.2 Row Stores

The most intuitive (and thus the most frequently used) way to manage relational data is by storing the table rows sequentially on a hard disk. Each table is stored in a separate file (Figure 1.2), and when the DBMS needs to retrieve data for a user, it reads the relevant file from the hard drive. Because users frequently access only a subset of the table, DBMSs

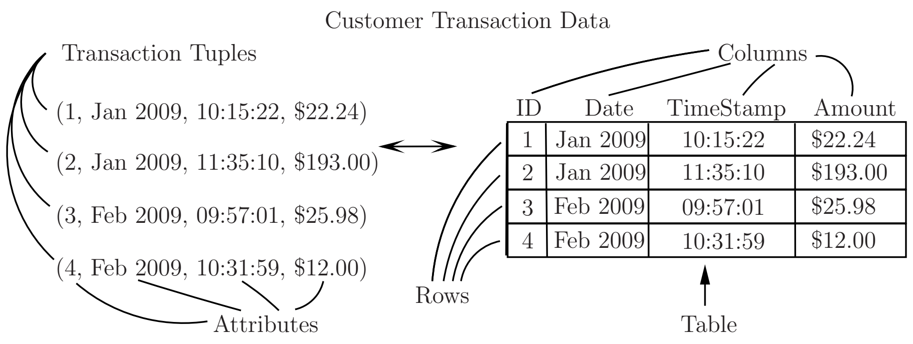


Figure 1.1: **Relational Database Model** – Tuples versus tables

employ additional auxiliary structures such as *indexes* and *materialized views* to avoid reading the entire file from disk (referred to as a *full table scan*) every time the user needs to access certain rows. The collection of these structures is known as the *physical design* (see Section 1.2.2).

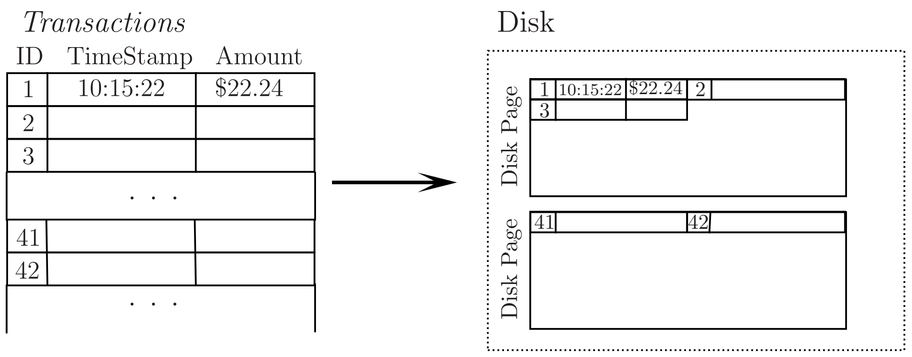


Figure 1.2: **Row-store Tables** – Row storage on disk

Owing to the nature of hard disk drives, DBMS data is stored and accessed in terms of disk *pages*. The size of the page varies [SKS02], but the smallest page in modern systems is 4 KB [sqlb]. Therefore, each page typically contains at least a few dozen rows. For instance, the row sizes in the SSB [POX] and TPC-H [TPC] benchmark tables vary between 40 and 100 bytes, which corresponds to between 100 and 40 rows per 4 KB disk page. Data is accessed using the disk page unit for several reasons. First, the average expected time to perform the *disk seek* (i.e., the act of physically moving the disk head to read the magnetic platter in a particular place) is approximately 9 ms [HDD], which is much higher than the cost of reading 4 KB of data. Thus, reading several disk pages can be worth the extra cost if it helps avoid a disk seek. Second, even when the disk read only needs a few bytes of data, the hard disk controller will read more data and discard the rest. Finally, for the purposes of *caching data* (i.e., keeping data in RAM to avoid going to disk), managing the data in pages is preferable because it requires less meta-data to keep track of the RAM cache [SKS02].

Each DBMS has a *storage layer* component that is responsible for tracking the data and fetching pages from disk. Some DBMSs such as MySQL [MyS] even allow the dynamic substitution of storage layers, as discussed further in the next chapter.

1.1.3 Column Stores

The focus of this dissertation is the problem of tuning a column-store DBMS. Column-store DBMSs provide a similar interface to that of a row-store DBMS by utilizing SQL, as discussed in the next section. Under the hood, however, a columnar DBMS is different, thus requiring a different approach to the problem of database design. Instead of storing the data row after row on a page, column stores keep the values from each table column in a separate file (Figure 1.3). This approach comes with a number of important trade-offs when compared with a row-store DBMS.

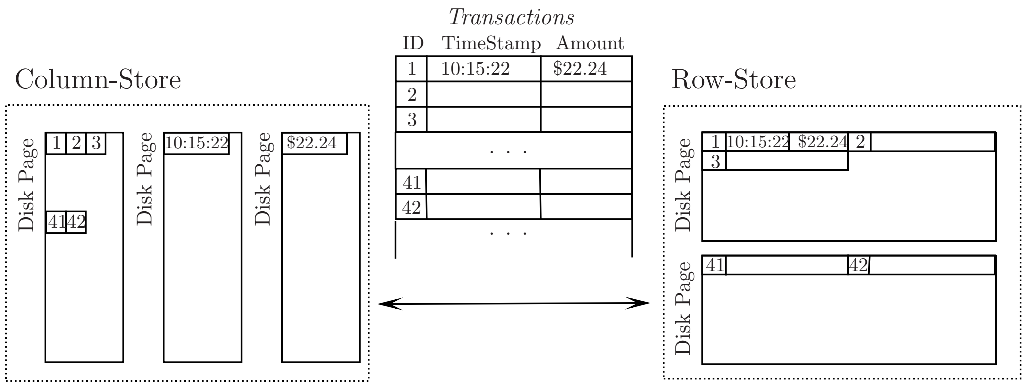


Figure 1.3: **Column-store vs. Row-store** – Differences in storing the same table data

First, let's discuss the advantages enjoyed by column stores. The major benefit is that the columns of the table are stored in individual files; therefore, when the user needs some of the table columns, the DBMS can access the columns of interest, ignoring the rest. When table data is stored row by row, accessing only some of the columns is impossible (again, because we read in page units that contain multiple rows as shown in Figure 1.3). Second, storing data of the same type in the same file makes it more amenable to compression (this notion is further discussed in Section 2.6.1, Section 5.1.1 and in [AMF06]) and late materialization (see also Section 2.6.2 and [AMF06]) during query execution. The implications for query execution in columnar DBMSs are further discussed in Section 5.1.

Column stores do come with some drawbacks, however. Additional work – unnecessary in a row-store DBMS – is required to connect data from different columns and form the corresponding row. For example, if a query needs to fetch certain transaction dates and amounts, the corresponding values need to be extracted from different files. This is a type of *join* problem and join operations can become expensive (see [AMF06], [SKS02]). Another issue arises when the database needs updating. Because new data is always inserted in terms of rows, the updates need to modify a different file for every column in order to insert or modify a complete row. By contrast, in a row-store, the whole row is likely to be on the same disk page. We will further examine the problems of updating a column-store DBMS in Section 2.6.3.

1.1.4 Using SQL

A user can operate on data and on auxiliary design structures in a DBMS using SQL [BC74]. Although we will not attempt to provide a full definition of SQL (which can be found in any number of textbooks, such as [SKS02]), we will recapitulate the necessary basics of the language. Throughout this dissertation, we use several example queries written in SQL. The structure of the *read* or *SELECT* SQL query has the following form:

SELECT [columns, **column** aggregations]
FROM Table1, Table2, ...
WHERE [predicates]
GROUP BY [columns];

The *SELECT* clause contains the set of columns in which the user is interested. This can be either simply the contents of the column (e.g., *SELECT Customers.Name* when looking for customer names) or one of the predefined *aggregate* functions over the contents of the same column (e.g. *SELECT SUM(Revenue)* to compute a total of the revenue). We will frequently refer to the column set in the *SELECT* clause as the *target column set*, because the user ultimately requests these columns.

The *FROM* clause of the query contains the set of tables that the query will use to provide an answer. The details of how the tables are joined to answer the query are discussed further in Section 1.2.1.

The *WHERE* clause contains query *predicates*, which restrict the return values to a subset of data rows that the user is looking for (e.g., *Year* = 1998 or *Discount BETWEEN* 5 *AND* 10). The most important property here is the *selectivity* of the predicate. The selectivity (a value ranging between zero and one) is the normalized fraction of the rows in the table that pass the predicate filter. Thus, if the data contains 10 years' worth of data, the predicate *Year* = 1998 might have a selectivity of 0.10, meaning that 10% of the data rows contain information about the year 1998.

The *GROUP BY* clause contains the *aggregation* columns, which work with the *aggregate* functions in the *SELECT* clause. For example, a data analyst who needs to request the total store revenue for each state would issue a the following query:

SELECT state, **SUM**(revenue)
FROM Stores
WHERE ...
GROUP BY State;

Importantly, SQL is *case insensitive*: for example State, state, STATE and even StAtE are considered identical in SQL. In addition to the *read* queries, SQL also supports *write* queries in order to insert a row into (or delete one from) the table. The syntax of inserts is not important here because, as we will later explain, the inserts in our problem setting use a bulk loading tool to amortize costs. Thus, the cost of a single *INSERT* query can be considered constant regardless of the particular values involved. Vertica's approach is described in detail in Section 2.6.3, and the maintenance issues are covered in Section 5.3.

1.2 Database Design

The design process consists of two parts: the logical design and the physical design. Although the work in this dissertation concerns the latter, we will define both for clarity.

1.2.1 Logical Design and Schema

The *logical design* is the process of organizing the data into a relational model, or a *schema*. The schema used in most of our experiments is shown in Figure 7.1. The design of the schema includes assigning data column types and declaring any interrelations that exist among the columns. The process of organizing the data into tables and connecting them with proper join keys is known as *normalization* [Cod71].

Intuitively, the idea of normalization is to recognize and eliminate unnecessary data duplication. For example, if our database contains customer purchases, we do not need to store all customer information (name, address, etc.) with every transaction because first-time customers can become repeat customers. Instead, we assign a unique ID to every customer, storing his or her data in a separate (smaller) *customers* table. Then, we store the unique customer ID with every transaction and look up detailed customer information (e.g., name, address) only when necessary.

Consider the example schema in Figure 1.4 that contains a *transactions* table and a *customers* table. The identifying number for each customer transaction is *Transactions.TID* (often simply referred to as *TID* if the name is unique). This number uniquely identifies each entry in the *transactions* table. Any attribute that can uniquely identify each row is known as a *candidate key*. The schema typically declares one of the candidate keys as the *primary key* (PK). In our example, the PK of the *customers* table is *CustID*. A PK can be used to look up and identify any row, and the DBMS monitors the data to make sure the PK is never duplicated.

To confirm, the normalization process has divided the transaction data into two tables to avoid duplication. Thus, the *customers* table contains data specific to the customer (which will remain unchanged every time the same customer buys a new item). Customer address is one example of an entry that does not need to be recored with every purchase. To analyze the data, queries might need to connect the customer and transaction data (i.e., join the *customers* and *transactions* tables). The *Transaction.CID* attribute is used to determine which customer bought a particular item. For instance, the first two purchases in the transactions table were made by customer #1. The attribute that is used to join a table is called a *foreign key* (FK), because it is an attribute in one table (*transactions*) that refers to a PK in another table (*customers*). The PK–FK key connection allows us to join tables by matching the corresponding rows.

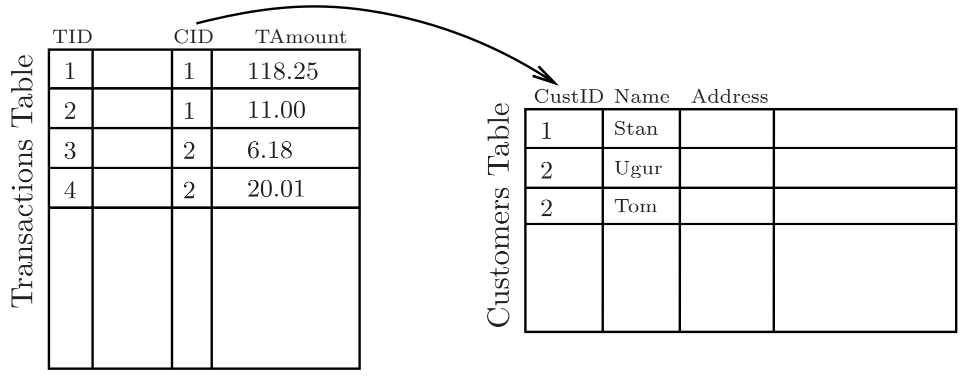


Figure 1.4: **A Simple Schema** – An example schema with two tables

1.2.2 Physical Design

Once the logical design has been produced, we can load our data into the tables and begin executing queries. The *physical design* is the task of rearranging and duplicating the data (i.e., partially denormalizing the schema) to speed up user queries. The contents of the tables can be sorted in order to collocate the data and speed up query access. Additionally, databases support the creation of auxiliary physical structures (different auxiliary structures are described in Chapter 2) that can be added to the preloaded data tables.

1.2.3 Problem Statement and Contributions

The goal of the physical design tool is to produce a set of auxiliary database structures (a design) to minimize the runtimes of the queries that a database user wishes to run (referred to as a *training query set*) without exceeding the amount of resources allotted by the user. Our design algorithm will recommend a set of physical structures that minimizes the runtime of the training query set without exceeding the user-defined disk budget (specified in MBs). The following list contains the expected user inputs:

- *Database Schema*: The database schema includes the list of tables, data types, PK-FK relations, etc. (Section 1.2.1).
- *Data Sample*: The data sample could be a set of statistical information or raw data that our design tool will sample. Our tool samples the raw data to extract the basic statistical information about correlations in the data (Section 2.1.2).
- *Current Physical Design*: The current physical design is the list of the auxiliary structures that already exist in the database. This is primarily relevant to the problem of generating an *incremental design* (see Chapter 8). In the absence of such information, we assume the database to be loaded with the *default design* that contains data tables sorted by their respective PKs (this is the typical default configuration of a data warehouse, because it allows any query to be answered).
- *Training Query Set*: This is the set of *SELECT* user queries. These queries can have weights attached to them in order to reflect the relative frequency and importance of each query (we aim to minimize the total runtime of the entire query set).
- *Insert Query Set* and *Batching Rate*: *INSERT* queries are also part of the training query set. In Section 5.3, we will explain why knowing the batching rate (i.e. how many rows are inserted in a single operation) is important in this setting. The insert rate should be specified for each table individually.

- *Query Cycle*: This is how often queries arrive. In particular, if our query workload has a high insert rate, we need to know how often we can expect the workload to be repeated (e.g., the specified *SELECT* query workload executes once every 10 minutes). Without knowing this *SELECT* frequency, we cannot compare its cost with that of the *INSERT* queries.

- *K-Safety Level*: The K-safety level is the required level of fault tolerance, which is defined as the number of machines that can fail without compromising data integrity (see Section 2.6.4).

The goal of the design tool is then to produce the design with the lowest total runtime (for *SELECT* and *INSERT* queries combined) without exceeding the user-specified disk budget.

Chapter 2

Background

This chapter outlines the basic information necessary to understand the nature of database design. We discuss the requirements for data statistics, introduce the basic notion of query execution and describe the number of different auxiliary structures available in DBMSs including an overview of Vertica.

2.1 Data Statistics

As it will become abundantly clear throughout this dissertation, having the right information about the contents of a database is crucial for both query execution and design. All databases collect and use basic data statistics, while many collect more special purpose statistics (the specifics vary among DBMSs). This statistical information is used both for query processing and for database design (see Section 5.4.2). Here, we explain the basic and the more advanced statistics that we rely on in our design tool and the SQL query generator.

2.1.1 Histograms

The basic structure used to describe column data is a *histogram* [SKS02]. A histogram stores the value frequency count, providing an aggregate overview of data distribution within the column. For example, Figure 2.1 contains two different types of histograms – *equi-width* and *equi-height* histograms – for the *lo_discount* column. Even though a table in a data warehouse may contain millions of rows, the *lo_discount* column contains 10 unique values in our benchmark scenario; and this particular data skew was invented for the sake of example. The difference between the two histogram types is in the way the value ranges are mapped. With the equi-width histogram, we divide the x-axis domain into a number of equal *buckets* or ranges (hence, equi-width). Thus, 10 unique *lo_discount* values could be divided into five buckets with a width of two as shown in Figure 2.1. This approach is simple to apply (all one needs to know is the min and max values of the column), but it is an inefficient way to represent data with skewness. For instance, consider a degenerative case where all the values are located in the range of [0,2]. The equi-height histogram method takes the complementary approach, arranging the buckets so that value frequencies are equal – this lets us capture the information about data skewness. Intuitively, we take a more detailed view of the ranges that have a higher value frequency. Thus, Figure 2.1 shows that the value range of [0,4] contains three buckets, while the value range of [4, 10] only contains two buckets that reflect data skewness.

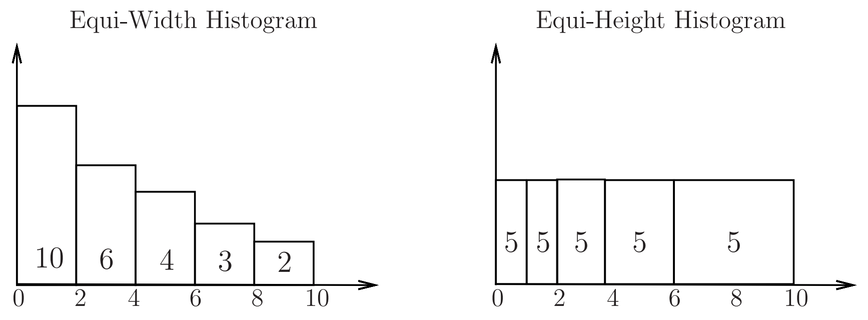


Figure 2.1: **Histogram Example** – Equi-width and Equi-height Histogram Example for *lo_discount*

A *multi-dimension histogram* is a histogram that records the frequency of the existence of attribute value pairs (or triplets, etc.). This is one way to capture correlation in data (correlations are discussed in the following Section 2.1.2) between data attributes. However, storing and maintaining multi-dimensional histograms is expensive [MS03, PR03]; thus, we rely on a more economical approach by keeping single-column histograms and some additional simple information that describes the degree of the inter-column correlation.

2.1.2 Tracking Correlations in Data

There is a large body of work demonstrating that awareness of correlations among data columns can greatly improve query execution time and cost model accuracy in a DBMS [BH03, GSZZ01, GGZ, KHR+09, KHR+10]. Although most of the work on this idea has been limited to research prototypes, some limited forms of correlation-aware functionality have been integrated into mature commercial databases [SQLa, Oraa]. It is also worth noting that multi-dimensional histograms, which are supported by most DBMSs, are also a limited (and expensive) way of tracking correlations between columns.

Why are correlations so important? The cost of query execution depends on the amount of data that the query accesses and the layout of this data on the hard disk. The amount of intermediate data that has to be kept in RAM (and joined as necessary) as the query is executed also affects query runtimes. Moreover, in a clustered column store, the value distribution affects the compression rate of each column.

When the value of one attribute (A) always uniquely determines the value in another attribute (B), this is known as a *functional dependency* [SKS02] and is denoted as $A \rightarrow B$. Functional dependencies are important for the normalization process in logical design (see Section 1.2.1). For example, the PK in a table (defined as a unique row identifier) determines the value of every attribute in the table. Consider the example in Section 1.2.1. The transaction ID (TID) determines the customer ID (CID) as well as the transaction amount. Thus, $TID \rightarrow CID$ and $TID \rightarrow TAmount$. In practice, data columns correlation ranges between no correlation and a functional dependency (i.e., a "perfect" correlation). The degree of correlation between two attributes A and B was defined in [IMH+04] as s *correlation strength* or a *soft functional dependency*:

$$SoftFD(A, B) = \frac{Cardinality(|AB|)}{Cardinality(|B|)} \tag{2.1}$$

A higher value for SoftFD means greater column correlation – with one being the highest possible value. be achieved. Every PK has a correlation strength of 1 (which is enforced by the DBMS). However, a correlation strength of one between two non-key attributes, does not prove the existence of a functional dependency (new data might later violate the functional dependency). However, the performance benefit of correlation degrades gracefully; thus, query performance and compression will be similar between a correlation strength of 1 and 0.999. For instance, consider the correlation that exists between the city and state in the customer address. The city name is often sufficient identify the customer state; otherwise, we can usually narrow down the list of possible states. Sorting the table by state name will partition the cities so that only cities from each state are collocated in the same area on the disk. Thus, even in the absence of a correlation strength of one, most of the correlation-related benefits can be realized. Note that correlation strength is not symmetric; thus, assuming that $Cardinality(A) \neq Cardinality(B)$:

$$SoftFD(B,A) = \frac{Cardinality(B)}{Cardinality(BA)} = \frac{Cardinality(B)}{Cardinality(AB)} \neq SoftFD(A,B) \tag{2.2}$$

Finally, we also note that there exists a special category of correlations that can be described as an algebraic relation For instance, a correlation such that the value of B is always within a difference of 10 from the current value of A (i.e., $\forall A, B \in (A, A+10)$). The idea was first presented in [BH]. It is also possible to capture the same information by bucketing values (as suggested in [KHR+09] and briefly described in Section 3.2.2) and storing the correlation information between *value ranges* instead of individual values.

The work in [Gib], [CCMN00] and [KHR+09] presents efficient approaches for detecting correlation in data. This topic is further discussed in Chapter 3.

2.2 Query Execution Process

Without getting embroiled in the unique architectural differences between row stores and column stores, this section describes the basics of the query execution process. Consider the following SQL query (similar to one of the SSB queries, but with a simplified target list with only one column), Q$_{ex1}$:

```
SELECT SUM(lo_extendedprice)
FROM lineorder, dwdate
WHERE lo_orderdate = d_datekey
AND d_year = 1993
AND lo_discount BETWEEN 1 AND 3
AND lo_quantity < 25;
```

The query selects the total price, subject to some predicates. In this case, query Q$_{ex1}$ needs data from two tables; thus, it must join the *lineorder* (fact) and the *dwdate* tables. The first entry in the *WHERE* clause (*lo_orderdate = d_datekey*) tells us which columns are going to serve as the join key. *d_datekey* is the primary key (PK) in the *dwdate* table and *lo_orderdate* is the corresponding foreign key (FK) key in the *lineorder* table. The rest of the *WHERE* clause entries are predicates limiting the set of rows from which *lo_extendedprice* values are to be summed.

Once the query has been received by the DBMS, the *query optimizer* component of the DBMS generates a *query plan* based on the available physical design. Figure 2.2 shows a possible query plan. It is then the job of the *query executor* to process this plan from the bottom up.

A query plan can become very complex, particularly when multiple auxiliary structures play a role in its execution. For a given query plan chosen by the query optimizer (we will discuss cost models, including our own, in Section 5.4), the cost of the query will primarily depends on two components: the cost to read the data from the relevant tables or other structures and the cost to process the intermediate results, such as applying predicates or executing joins. The I/O cost typically determines the bulk of the query cost.

The query optimizer has the task of selecting the fastest query plan, including which auxiliary structures to use. For example, a secondary index over *lo_discount* (secondary indexes are described in Section 2.5.1) could provide the list of rows matching the corresponding *lo_discount* predicate, thereby speeding up the process of filtering the *lineorder*

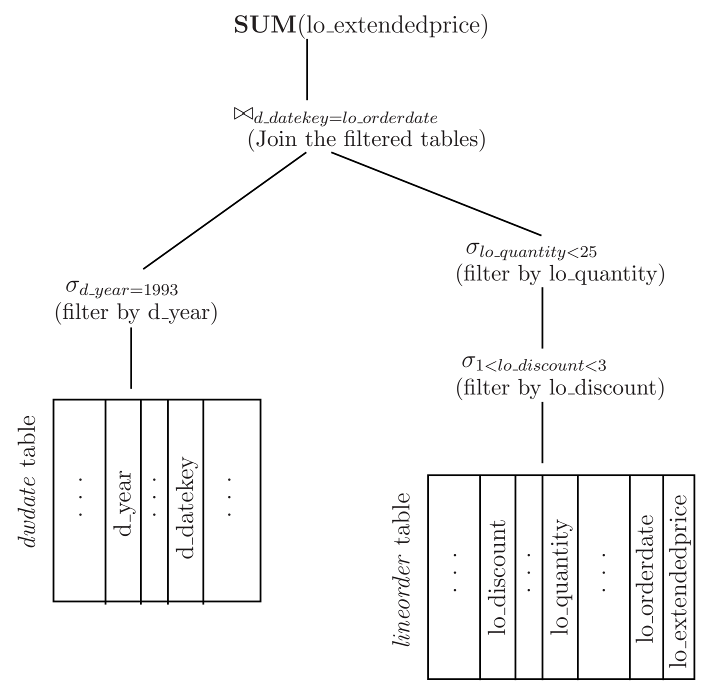


Figure 2.2: **A Query Plan** – An architecture agnostic query plan

rows by Q_{ex1}. Alternatively, if we had an additional copy of the *lineorder* table sorted on *lo_discount* (i.e., a primary index – see Section 2.5.3), that would speed up the filtering of the *lineorder* table further, since rows matching the predicate are then collocated on the disk. Once the data is read from the table(s), it is kept in the memory while the query is being processed. For example, after filtering the *lineorder* table by the *lo_discount* predicate, but before applying the *lo_quantity* predicate, we retain the *lineorder* rows that matched the predicate in memory. In fact, the chosen order of predicate application depends on the expected size of the intermediate results that need to be stored. In the case of the query Q_{ex1}, the *lo_discount* predicate filter is chosen first, since it has a lower selectivity (0.27) compared to the selectivity of the *lo_quantity* predicate (0.50). Note that the selectivities depend on the particular dataset – here, they are estimated using the default SSB [POX] dataset. Of course, the example Figure 2.2 is an over-generalized example. In practice, we can expect these these two predicate filters will be applied to the *lineorder* table simultaneously, while in a column-store intermediate materialization will be more complex.

2.3 Row-Store vs. Column-Store Architectural Differences

One might be tempted to think that differences between row-store and column-store DBMS can be reduced to the different storage layer (i.e., the component that stores data on disk), particularly because it is possible for a database to have multiple storage layer implementations. In [AMH08], Abadi and colleagues argued that this is not the case. They explored a number of ways to simulate column-oriented storage in a commercial row-store database and compared the resulting performance to C-Store [SAB$^+$05]. There are a number of ways to simulate column-store storage: for example, by creating a vertically partitioned schema or by creating single-column indexes for every column (that approach is reminiscent of [KM05, IKM07a]), which is discussed in 3.1). However, the work in [AMH08] conclusively proves that query execution in a row-store database cannot efficiently use such vertically partitioned storage. Many of the difficulties stem from the row-store's inability to operate on compressed data (forcing the DBMS to decompress or *materialize* results early) and the overhead of joining the values corresponding to the same row. In Section 4.1, we discuss the important factors that differentiate row stores and column stores in detail.

2.4 Query Manipulation

It is important to understand the similarity that exists between queries in the training query set. We discuss this in detail in Chapter 6. For example, a significant part of the design process involves generating *shared* physical structures that serve multiple queries at once (see Section 6.3). Intuitively, in order to find *similar* queries, the designer needs to look for queries that access the same columns and share similar predicates. We measure the similarity between queries by representing queries with feature vectors (see Sections 2.4.1 and 6.2.1). We also compare our approach to the *Jaccard coefficient*-based measure described in Section 6.5.2.

We rely on similar intuition in designing a query generator to supplement training query workloads and further test the database design tool. Although the SSB [POX] benchmark includes a query workload, it proved insufficient to thoroughly test different physical design.s Having spent significant time working with SSB [POX], we have observed several shortcomings inherent to that workload (we explain the issues in more detail in Chapter 7). In some of our experiments, We do augment and alter queries from the SSB workload; however, a more general purpose query generator provides us with the flexibility to generate large query sets. First, we describe the concept of the *selectivity vector* and then the query generator itself.

2.4.1 Selectivity Vector

Throughout this work, we have to operate with queries, both for query generation (as in Section 2.4.2) and for query grouping (Chapter 4 and Chapter 7); We represent SQL queries using a *selectivity vector*. A selectivity vector is the collection of query attributes that the query accesses and the respective selectivity of each attribute. Recall that the selectivity of a predicate is defined as a value between zero (i.e., filter out all values) and one (i.e., keep all values). Thus, each query can be represented as an N-dimensional vector. Consider, for example, the query shown in the beginning of Section 2.2. The corresponding selectivity vector would contain the following:

$(d\_year : 0.14, lo\_quantity : 0.47, lo\_discount : 0.27, lo\_extendedprice : 1.0)$

We demonstrate the implications of data correlation that were discussed in Section 2.1.2. The query vector listed above contains the explicit predicates used in the example query. However, the presence of data correlation results in additional, *implicit* (i.e., derived) predicates. The correlation strength ratio (described in Section 2.1.2 and defined in [IMH+04]) can be used to compute an implied upper bound on column selectivity. For example, a predicate that selects the *d_yearmonth* value of 1994-01-01 implies that only the rows with 1994 in *d_year* will be selected. The process of selectivity propagation has been described in [KHR+10], but we will briefly recapitulate it here.

The idea is that the presence of a correlation between two attributes can let us compute an (average expected) upper bound on the selectivity of a correlated column. Suppose that there exists a soft functional dependency of a certain strength between attributes A and B ($SoftFD(A, B) = c_{a,b}$) and that there is a predicate on column A such that $Selectivity(A) = s_a$. In that case, the expected average bound on selectivity of attribute B is computed as follows:

$$Selectivity(B) = \frac{Selectivity(A)}{SoftFD(A, B)} = \frac{s_a}{c_{a,b}} \tag{2.3}$$

We discuss this idea further and present some examples in Section 6.2.1.

2.4.2 Query Generator

In order to satisfy the need for multiple query workloads with reasonably simple, easy-to-describe properties, we developed a standalone SQL query generator. The query generator project was originally submitted as a Master's project [Hus] at Brown University, and we briefly describe it here for completeness. The basic intuition behind the query generator is that data warehouse queries have some inherent natural grouping. In fact, the workload in [POX] is arranged in four groups (called *flights*). Similar queries are likely to be issued by the user for two reasons:

- The same query is re-issued with different predicates (i.e., store's revenue over year 1992 or store's revenue over year 2000). The Microsoft SQL Server, for example, supports *parameterized queries* [SKS02] that are essentially SQL queries with place-holders instead of proper constants. This mechanism allows the query optimizer to reuse (i.e., cache) the query plan.

- A similar query can be re-issued by the user to look at the data from a different point of view. Thus, if one query has computed the average store revenue for each state (in which stores are located), the user might then want to issue a more detailed query that collects average revenues by city instead.

Of course, any pair of queries in the workload may share any number of predicates or columns, and hence, one could expect some overlap in any query workload. For the purposes of the design, it clearly matters *how much* overlap there exists between the training queries. For example, a workload consisting of two queries that do not access any of the same data columns will have very different design opportunities compared to a workload consisting of two queries that access the same columns. This idea of query overlap is the driving force behind the query generator. We are able to specify (with a random distribution) which columns the query accesses and the selectivity of the predicates that are applied.

The query generator accepts as an input a set of *query structures*; a query structure is similar to the parameterized query concept, letting us specify randomized placeholders instead of constants; we can randomize predicated columns as well as predicate selectivity. We can even specify the probably distribution for the predicate selectivity, as desired. For example, an example input such as this:

(*d_year* : Uniform(0.2, 0.5), *lo_extendedprice* : 1.0)

would generate a requested number of random queries with a randomly chosen predicate on *d_year* that would have a selectivity drawn uniformly at random from the range between 0.2 and 0.5; the query will also simply read lo_extendedprice. It is also possible to specify aggregate functions and other random distribution types.

2.5 Auxiliary Design Structures

This section discusses the different structures that are used in DBMSs to speed up user queries. Some implementation details vary among different DBMS systems, but the basic idea of an optional pre-join combined with data reorganization remains the same.

2.5.1 Secondary Indexes

An index is a mapping between the values that the query is looking up (such as a predicate on *lo_discount*) and the table rows in which these values are located. Therefore, going back to the example query Q_{ex1} from Section 2.2, a secondary index on *lo_discount* would contain the mapping between its values and the row IDs of the *lineorder* table. The query will then look up the values that it needs (1, 2, and 3 in this case) and find the rows that match the query predicate. The indexed attribute does not have to be unique, however it does typically have to have an entry for the indexed row (i.e., secondary indexes are *dense*). For example, even though there are 11 unique values in the indexed column *lo_discount* (based on the SSB benchmark data generator), the secondary index has to contain every key-value pair – and there are millions of value entries (24 million rows for a scale-4 data set). We will use the notation: $\mathrm{I}_{lo\_discount}$ to describe the secondary index over a particular attribute. Keep in mind that the secondary index is always associated with a table (e.g., the *lineorder* table in this case) and cannot stand on its own. The most common indexing structure that is used in such cases is a B-Tree [Com79], used due to its ability to amortize the costs of read and write to a logarithmic time by continuously rebalancing the index structure.

A secondary index can be created over multiple columns by indexing a concatenation of multiple column values. Such indexes are referred to as *composite* secondary indexes. $\mathrm{I}_{lo\_discount,d\_year}$ would denote an index mapping between value pairs in *lo_discount* and *d_year* and the indexed table. These two columns have 11 and 7 unique values respectively and are uncorrelated; thus, there are 77 indexed keys in this case (i.e., each possible pair). Note that although $\mathrm{I}_{d\_year,lo\_discount}$ also indexes 77 unique values and can be used in a similar manner, the structures are slightly different (e.g., the former contains a key "0 1997," while the latter contains a key "1997 0").

As we will demonstrate at the end of this section, simple secondary indexes are not effective in the OLAP setting. Secondary indexes have the following limitation: once the query looks up the values requested by the predicate using the index (which could be very efficient if an index is built well), the result is the set of rows. Using the row pointers, the query has to follow the row IDs to read each matching row in the indexed table. These target rows are likely to be scattered throughout the indexed table, and thus, the query reads a great deal of extraneous data (as the minimum I/O read unit is a disk page) and performing many disk seeks. In the worst case scenario, the query must read a page per requested row. Consider an example table T = (PK, A, B, C) and a query Q_{abc}:

SELECT SUM(C)
FROM T
WHERE A = 5 **AND** B < 3;

Clearly, if the table has no useful indexes (by default, we cluster and index tables on their primary key), then we scan the entire table for the needed rows. Next, consider adding a composite secondary index on columns A and B ($I_{A,B}$) to improve query performance; now, instead of scanning the table, query Q_{abc} can access $I_{A,B}$ first and determine the exact rows matching its predicates. Both of these scenarios are shown in Figure 2.3. Knowing the locations of the requested rows can be helpful. However, the cost of seek to every row location and reading at least a page per row may be very high and such queries often degenerate into a full table scan. Queries in the OLAP environment access a relatively large number of rows; thus, secondary indexes are rarely useful in this setting.

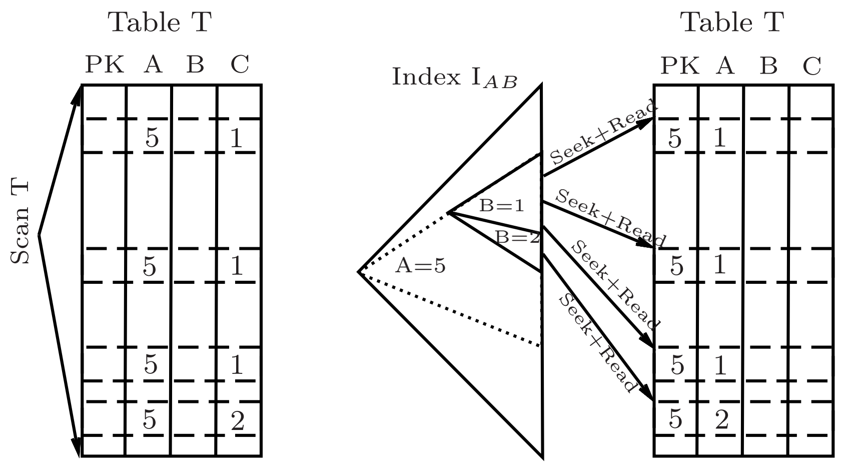


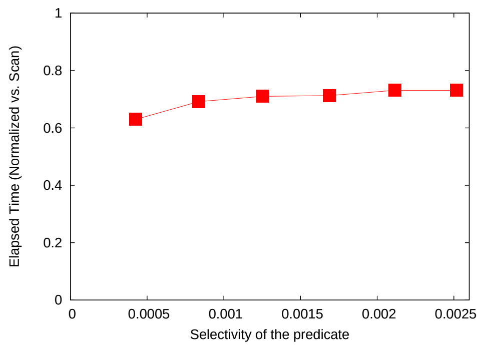
Figure 2.3: **Indexed and non-Indexed Table Access in a DBMS** – Example of Secondary Index Use

To evaluate the feasibility of secondary indexes in OLAP, we consider an experiment using DBMS-X and Vertica; we simulate a secondary index in Vertica because Vertica does not explicitly support secondary indexes. We use an SSB schema and the following query Q:

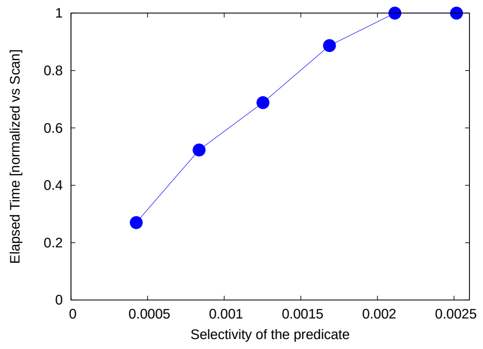
SELECT MAX(lo_revenue)
FROM lineorder

WHERE lo_orderdate **BETWEEN** 19920101 **AND** [**date**];

This query selects the maximum revenue collected within a certain range of days. We vary the [date] value of the order date range between one day (i.e., substitute 19920101) and six days (i.e., 19920106). We build a secondary index on *lo_orderdate* and measure runtimes of the query for ranges of different numbers of days. Recall that in the absence of any auxiliary structures, query cost matches the scan of the whole *lineorder* table. Therefore, we normalize the runtimes by the scan cost in order to observe the improvement derived from using a secondary index. Figure 2.4 shows the resulting runtimes; we use selectivity of the query predicate for the x-axis. As we see in Figure 2.4(b), the secondary index ceases being beneficial above a selectivity threshold of 0.002, while at most other selectivities the benefit of the secondary index is rather limited. This is precisely the effect shown in Figure 2.3. Of course, for queries that require very few rows (i.e., a selectivity of 0.0001 or less), the secondary index can be a useful structure. However, such queries are rare in an OLAP environment. Although there are no secondary indexes in Vertica, we will nevertheless simulate the behavior of a secondary index to show that secondary indexes are even less useful in a column-store than they would be in a row-store. The full table scan remains unchanged; to simulate a secondary index, we use a sort order where *lo_orderdate* is the third column. The *lo_orderdate* column is still compressed – thus, the cost to read it and apply the predicate is negligible, but target data rows become declustered, once *lo_orderdate* is not the first column in the sort order. The result shown in Figure 2.4(a) demonstrates that even at a selectivity of 0.0005, the benefit of applying the date predicate saves less than half of the runtime compared to the full table scan. Performance issues caused by accessing scattered data (which forces the query engine to read unnecessary data and perform extra disk seeks) are significantly worse in a column-store. By contrast, the secondary index in Vertica Figure 2.4(a) is still winning at 0.0025, unlike in DBMS-X. This is an artefact of our secondary index simulation – *lo_orderdate* is still part of the primary index, so some benefit is still available to the query. A proper secondary index implementation in Vertica may perform even worse at that selectivity.



(a) Simulated secondary index in Vertica



(b) Secondary index in DBMS-X

Figure 2.4: **The Benefit of the Secondary Index** – The Performances of the Secondary Indexes Normalized by the Full Table Scan Cost

2.5.2 Covering Indexes

These problems with secondary indexes, described in the preceding section, are normally resolved by using a variation of secondary indexes: *covering secondary indexes*. Since following pointers into the indexed relation is inefficient, we avoid that step by including every column the query needs (both the predicates and the target columns) in the index key itself. Returning again to our earlier Q_{abc} example, we replace the index over the two predicated columns I_{AB} by a full covering index I_{ABC}. That solution is available in any DBMS. DBMS-X used in our row-store experiments supports a special version of the covering secondary indexes, which allows us to physically include additional data columns sorted according to the indexed key, but without including these columns in the key itself. For example, following on from the example query Q_{abc}, in DBMS-X we can create an index $\mathrm{I}_{AB}[C]$ (instead of I_{ABC}) in which the values of column C are sorted on *(A,B)* and then stored with the index. This is a more efficient approach than covering indexes, since I_{AB} may still be accessed as before (preserving its original number of indexed keys), while I_{ABC} is a larger index by due to having more unique keys. Figure 2.5 shows the intuition, using the previous example. Importantly, column C is not simply copied to index I_{AB}, but is also sorted as the indexed key. We observe that such structure closely resembles the regular materialized view (MV) *(A,B,C)* that is clustered on A, B (see Section 2.5.4 for materialized view description).

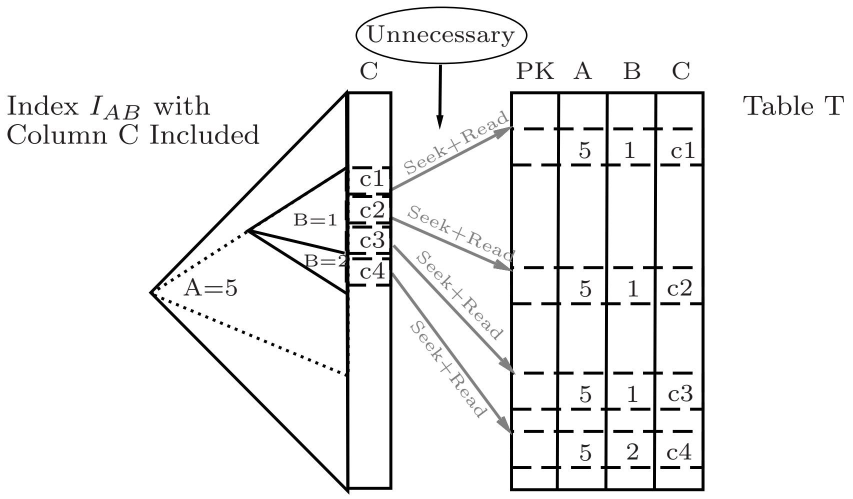


Figure 2.5: **Table Access with a Regular Secondary Index vs. Included Columns** – Example of how Table Access Changes while Using a Secondary Index with Included Columns

One notable difference between a secondary index with included columns and an MV is that the former can still be used as a regular secondary index (albeit by paying the corresponding penalty for storing the pointers into the indexed table). We will present results from DBMS-X using materialized views or secondary indexes as appropriate. Another important (DBMS-X-specific) difference is that in DBMS-X the clustering key of the materialized view has to be unique, unlike a secondary index. Finally, we note that although secondary indexes can be built without included columns, it is unusual for the design tool that is provided with DBMS-X to build a secondary index that does not utilize the additional included columns. In a DBMS where such a feature is not implemented, the design tool would have to resort to simple covering indexes.

2.5.3 Primary Indexes

As explained in the previous section, the problem with secondary indexes is the extraneous reads and seeks required for when looking up the scattered values that the query needs. A much better alternative is to create an indexing structure that also sorts the data (collocating data with the same keys, thereby minimizing the number of seeks and extraneous reads). Such a structure is called a *primary index* (or a *clustering index*) and it is supported by most DBMSs. Thus, going back to a primarily index example query Q_{ex1} in the previous section, a primary index $PI_{lo\_discount}$ over the *lineorder* table would still contain the mappings between the values of *lo_discount* and the row locations, but the *lineorder* table itself would be sorted (i.e., clustered). As a result, all rows with the same value of *lo_discount* would be collocated, minimizing the extraneous reads and seeks required when processing the query.

The primary index can be composed of multiple columns, just like the secondary index. In fact, as stated previously, a secondary index can be made primary by either sorting the indexed data or adding included columns as was described in the previous section. A significant part of designing physical auxiliary structures involves designing a primary index. We typically refer to this primary index as the *clustering index* or *sort order*, because this index determines how the data contents of the table are clustered (or sorted). The term clustered is particularly appropriate when the entire sort order is not unique (i.e., it does not correspond to a precise deterministic ordering of rows). For example, if we chose to sort the *dwdate* table on (*d_year*), this would only mandate that the *dwdate* table should be kept in a yearly order. However, unless we created a longer composite sort order, the months within each year would not be sorted in any particular manner.

2.5.4 Materialized Views

As in most other DBMSs, the common unit of physical design in Vertica is a *Materialized View* (MV). An MV is a pre-computed (hence, materialized) SELECT query stored in the DBMS. Adding MVs to a design requires disk space and incurs maintenance overhead due to the need to keep MVs current as the database contents change. The database contents will change through a sequence of *INSERT*, *DELETE*, or *UPDATE* queries that come in addition to the user *SELECT* queries that we are trying to optimize. MVs serve to speed up user queries by pre-computing some of the query's work and by organizing the data for faster access. As we will later discuss in detail, the physical design is a combination of MVs that fits into some resource allotment. In fact, one of the main purposes of the query optimizer is to correctly incorporate the available MVs into the query execution plan to speed up user query as much as possible.

The language and feature set supported by MVs in any particular DBMS is almost always a subset of the query language supported by the underlying DBMS. Below is a list of important feature categories that MV creation might support. Depending on the system architecture, some or all these might be available in the DBMS:

Projection: The MV can contain a subset of the columns from the source data tables.

Sorting: The MV data rows can be sorted (clustered) on a key composed of one or several columns in that MV.

Pre-join: The MV can contain a pre-join of several source data tables. Projection and sorting can be applied to the table pre-join.

Pre-filter: The MV can contain a subset of the rows from the source table. For example, an MV might contain data rows for the year 2009 only.

Pre-aggregation: The MV can contain a pre-computation of an aggregate function over the original data. For example, we might create an MV with the average revenue generated by each store.

Note that the first two features are *lossless* – in the sense that they contain all the rows from the source tables but with a different organization. Or, in other words, the original table data can be recovered from the MV. The projection feature is supported in almost every DBMS. In contrast, sorting is not supported in every DBMS; however, the two representative column-store and row-store DBMSs that we use in our experiments support MV sorting. Although most row-store DBMSs support sorting, this feature is rarely available in column-stores.

The pre-join may or may not be lossless: if there exists a PK–FK relation between all pre-joined tables, then the relationship between the rows in these tables is 1:N, and the MV has as many rows as the largest table in the pre-join. The table on the “1” side of the 1:N join relationship, which is also the larger table in the pre-join is referred to as the *anchor* table (*lineorder* table in Figure 7.1). Lossy pre-joins (as well as pre-joins that contain more rows than the largest base table) are not supported in Vertica.

The logical schema in which all tables are connected with PK–FK relations but only a single table participates exclusively in the “1” side of the 1:N PK–FK relationships, is called a star-schema (one level of joins) schema or a snowflake-schema (multiple join levels) schema. The large anchor table in the center of the schema is referred to as the *fact* table. The tables connected to the fact table via the FKs are called the *dimension* tables. Clearly, a star-schema is a restricted case of the snowflake-schema (see Figure 2.6 for an example). The most important property of the snowflake schema is that any pre-join defined in that schema is lossless, since there is only one fact table connected by 1:N key relations to the rest of the tables. Therefore, any possible MV in such a schema will be lossless.

Note that in many parts of this dissertation, we often assume a snowflake-schema setting, because it allows some simplifying assumptions. However, to our knowledge, most data warehouses have relatively simple schemas, and thus, this is not a serious limitation. Moreover, if the schema is more complex, it is possible to “break it up” into multiple stars or snowflakes to compensate. The work in [KHR+10, YYTM10], for example, discusses ways to rewrite the queries in order to simplify. As should become clear throughout our discussion of the algorithms, we only need to make this rewriting change to the *queries*, since any pre-joins are done for their benefit. Consequently, if all of the user queries are star-compliant while the schema is not, we need not do any rewrites. For instance, a schema consisting of two independent star-schemas (i.e., two fact tables with their own dimensions that are not connected; sometimes referred to as a *dumbbell* schema), does not require any rewrites – all queries are trivially star-schema compliant.

Finally, pre-filtering and pre-aggregation operations always result in lossy MVs. This

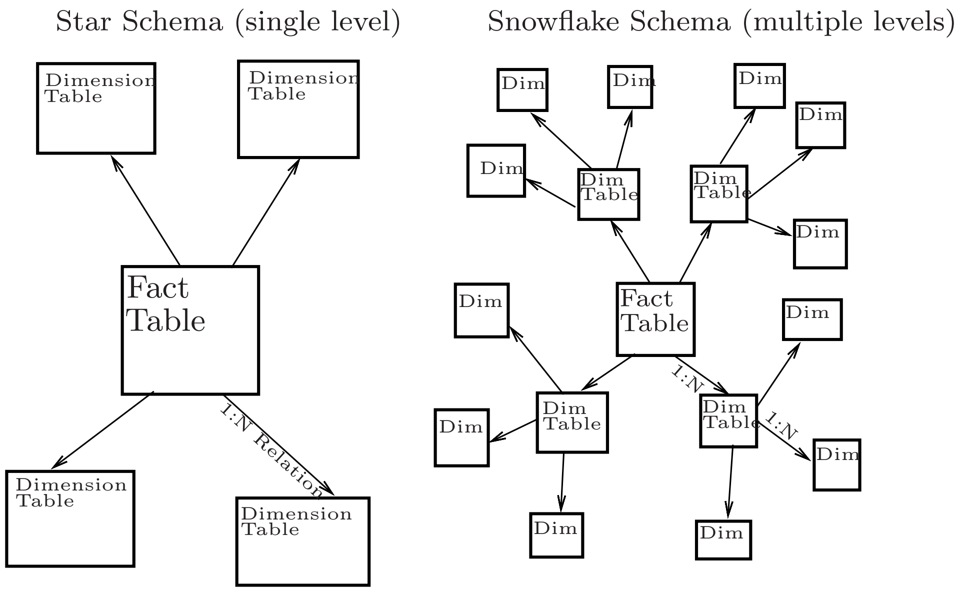

Star Schema (single level)
Snowflake Schema (multiple levels)
Dimension Table
Fact Table
1:N Relation
Dim
Dim Table
1:N

Figure 2.6: **Data Warehouse Schemas** – Examples of star-schema and snowflake-schema

means that the original table data can no longer be recovered from that MV and that the set of queries that can utilize a lossy MV is more limited. For example, if the MV only contains the data for year 2009, then any query that needs even one row from year 2008 cannot use it. Lossy MVs also complicate the job of the optimizer, since matching predicate overlap is a difficult task (i.e., asserting that query Q does not need any data from year 2008 is hard). Vertica does not support lossy MVs; thus, we are not going to spend much time discussing them here.

The implementation and supported features of an MV will vary significantly from DBMS to DBMS, both in row-stores and in column stores. In this work, we use a very popular commercial database (DBMS-X) as a representative of the row-store DBMSs. A significant amount of research using the same DBMS has already been published in the area of automatic database design, allowing us to gain some insight into state-of-the-art approaches to the design problem. Although no single DBMS can represent all row stores perfectly, since every DBMS has a number of unique features that are not available elsewhere, we believe that DBMS-X is a good choice for our purposes. For a column store DBMS, we use

Vertica: as we demonstrate in this dissertation, Vertica has some very interesting features that make database design more challenging. Note that column stores are a relatively young branch of the DBMS world, but we believe that Vertica has made some good choices in set of the features that it supports, and hope that more other column stores follow Vertica's lead. In particular, its support of explicitly sorted MVs is currently rare in column-store DBMSs. We explain the relevant features unique to each of the systems that we use in our experiments.

2.6 Vertica

Finally, this section describes the basics of Vertica [Ver] – the column-store database that we use throughout most of our experiments. As a column-store that supports clustered indexes (i.e., primary indexes), Vertica has the set of features that we need to demonstrate the advantages and proper design techniques for a clustered column store DBMS.

2.6.1 Compression

Compression is an integral part of Vertica, which supports a number of different encoding methods. The relevant encoding methods are described next – and we will explain why it is particularly effective in a column store such as Vertica. Encoding the data provides significant savings in space and thus speeds up queries because a significant part of the query cost is that of reading data from the disk (I/O). The experimental evaluation and explanation of how compression rates are estimated is covered in Section 5.1.1. What follows here describes only the basics of the available compression methods in Vertica.

LZO

Lempel-Ziv (LZO) [ZL77a] is the default compression method used for all disk pages in Vertica that are not using a custom encoding method. This is a relatively heavyweight method, in that it is comparably expensive to encode and decode and that it accessing any value on a page requires decompressing the entire page. However, it can provide a reasonable amount of space savings, since in a column store, the data in each file tends to be fairly homogeneous.

When the data is sorted or when we have additional information about data distribution in the column, custom compression often achieves better compression rates.

RLE

Perhaps the most effective compression mechanism available in Vertica is *Run Length Encoding* (RLE). RLE provides the highest compression rate for sorted data in a column store. It is also unique to a particular kind of DBMS, in that neither a row-store nor a column-store without support for sorting can effectively employ this technique. The data is encoded by recording any run of repeating values (*value run*) as a pair: ([repeated value], [number of repetitions]). Therefore, (1, 1, 1, 3, 3, 3, 3, 3, 2, 2, 2, 2) would be stored as ([1,3], [3,5], [2,4]). This encoding method is very effective, as it can encode any number of identical values using a single pair of values.

However, the data still has to be sorted, and this storage method does not support random access: in the example above, most queries that need to access any of the 2s would have to first decode the initial run of 1s and 3s. The reason for this is the relative run-length size. By reading the compressed data above, we can determine that there are four sequential 2s in the data. However, this information is relative to the preceding RLE-ed runs. In this case, the run of 2s begins at the 9^{th} row, because there are eight RLE-ed values in front of it (three 1s and five 3s).

Dictionary Encoding

Dictionary encoding is a way to represent the values in a column with a small set of unique values. The idea is to record all the unique values present in the column and assign a small code to represent every unique value. Then we can store the codes instead of the actual values on the disk. This saves disk space, assuming that the codes are smaller than are the encoded values.

Vertica builds a dictionary by recording every unique value and assigning a code to each value based on how many unique values were found. For example, a column containing the values (10, 10, 12, 12, 12, 14, 15, 15) might contain the mappings (10→0, 12→1, 14→2, 15→3). Once the dictionary is built, the data would be encoded as (0, 0, 1, 1, 1, 2, 3), where each encoded entry needs 2 bits. In Section 5.1.4, we discuss the factors considered in estimating the compression rate.

Delta Encoding

Next, we describe *delta encoding*. Vertica supports two different types of delta encoding: Delta Value (*DeltaV*) and Delta Common (*DeltaC*). Although the two mechanisms are similar, *DeltaC* works best when the values are sorted and fall into a narrow range domain, while *DeltaV* does not require that the values are sorted.

To use DeltaV encoding, we choose an anchor value (for each disk page) and encode each value as a difference (a delta) between that value and the anchor (i.e., $value - value_{anchor}$). Thus, our example dataset (10, 10, 12, 12, 12, 14, 15, 15) from the previous section would encode to (Anchor: 10; 0, 0, 2, 2, 2, 4, 5, 5). This encoding requires $log_2(value_{max} - value_{min})$ bits per value, which comes to 3 bits. DeltaV encoding is effective when all the values on a page fall within a narrow range – note that data does need to be sorted for that to work. Instead of choosing the anchor, DeltaC uses a delta between each adjacent pair of values (i.e., for each value, the preceding value serves as its anchor). Our example dataset would, therefore, encode to (10, 0, 2, 0, 0, 2, 1, 0). Each value would need $log_2(delta_{max})$, or 2 bits in our example. DeltaC encoding outperforms DeltaV encoding when the values are at least partially sorted within the page. Section 5.1.5 discusses the delta encoding compression rates further.

2.6.2 Query Execution in Vertica

Query processing in Vertica is similar to that in any DBMS. The query optimizer generates a query plan, and the data is then accordingly accessed and processed by the query engine. Naturally, there are a number of details involved in result materialization and accessing the data.

The most significant difference that we want to discuss in detail is Vertica's capacity to operate directly on compressed data. The compression that is employed most frequently (both in row- and column-stores) is page-based, such as the LZO compression described earlier. In order to process the compressed page, the query engine has to first de-compress its contents. However, the custom encoding methods described above (RLE, dictionary, etc.) can be processed directly. For example, if the table contains columns *(A, B)*, we do not need to de-compress the contents of column *A*, and column *A* is encoded using RLE (of course, it has to be sorted in order for RLE to compress it efficiently), in order to process it. Thus, for a query such as:

SELECT SUM(A)
FROM ...
WHERE A=10;

instead of materializing the contents of column *A*, which compactly represents millions of data rows, we can process the RLE contents of the column directly. Suppose our table contains 10 million rows. In a row-store, we would have to read all the rows from the table, retaining the rows where A is equal to 10. The query executor then would have to sum the values in *A*. In a column store that does not support table clustering, we would have to do the same thing but without the need to read column *B*. Suppose that column *A* only has three values (10, 20 and 30). If the column was sorted and RLE compressed in Vertica, it would contain the following three hypothetical entries: (10, 2.5 million), (20, 3.5 million), (30, 4 million). In order to process this RLE column, the query engine can apply the predicate A=10 without decompressing the data. Then, still without having to decompress the values, we can compute the sum of all values of *A* and get 25 million.

Similarly, Vertica can also process a dictionary-encoded column without decompressing it. Consider the same example query and the same column A. When encoded with dictionary encoding, the same column would look as follows. For example, consider a dictionary with (10→0, 20→1, 30→2) and a page containing a sequence of dictionary entries (0, 1 or 2): 0, 2, 1, 0, 2, 1, 2, 1, 2, 0. Since the predicate is looking for values of 10, it would simply sum the occurrences of the code zero without explicitly decoding the entries on the page. Note that if the data in A were sorted as shown in the previous RLE example, the compression and data access would be even more efficient, since most pages would contain identical values (i.e., 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 or 1, 1, 1, 1, 1, 1, 1, 1, 1, 1).

2.6.3 Updates in Vertica

As discussed earlier, under our assumptions, updates in data warehouses are dominated by *INSERT* queries (rather than by *UPDATE* or *DELETE* queries). Thus, we are going to discuss *INSERT* queries in more detail in this work. Functionally speaking, an *UPDATE* query performs both a delete of the old row as well as an insert of a new row, so at the end of this section, we describe the delete mechanism for completeness.

Vertica employs an insert-buffering mechanism that is different from what is done in any row-store database. As we will demonstrate later, this approach allows Vertica to sustain a much higher insert rate compared to a typical row-store data clustering support. Note that buffering has to be used, because Vertica belongs to the category of column stores that supports data clustering – therefore, it does not have the option of appending the data in arrival order (unlike [pos] or [Mon]). In Vertica each new row is inserted into the Write Only Store (WOS) memory buffer (see Figure 2.7). Of course, this extra level of buffering incurs costs of its own – the rows of the table that are stored in the memory have to be accessed when a query uses a table, and the memory buffer needs to be cleared when full. When the WOS memory buffer fills up, instead of merging the data into the main part of the table – still too expensive to rewrite the table every time the WOS buffer fills up – Vertica makes a new mini-fragment of the table on disk. The on-disk table structure is referred to as a

Read Only Store (ROS) in contrast to the memory buffer WOS. The smaller structures, or mini-ROSs, are sorted and encoded in the same way as are the original ROS of the MV. Figure 2.7 summarizes the process.

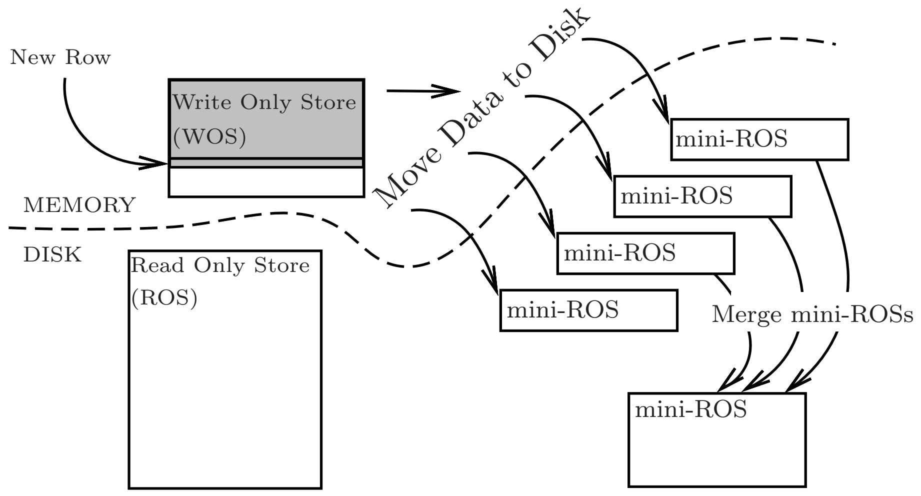


Figure 2.7: **Inserts in Vertica** – Vertica's Insert-buffering Approach

This mini-ROS buffering mechanism has some interesting implications on the cost of inserts in Vertica. The immediate cost of the insert is reduced, and the additional cost of sorting, encoding, and writing newly inserted data to disk is delayed (and thus amortized), as it only happens when the WOS buffer fills up. In addition to that, we must also account for the cost of the maintenance required to merge mini-ROSs in the background, as the table cannot keep becoming more and more fragmented indefinitely. The merging process is managed by the background maintenance process in Vertica; we will be presenting a discussion of possible merging policies in Chapter 6.

Note that the cost of the insert is doubly amortized by first buffering the inserts to memory and then delaying the merge with the "main" ROS by continuously merging mini-ROSs in the background. This amortization does not happen for free, however. In addition to the cost of merging the mini-ROSs, the fragmentation of the table imposes an overhead on SELECT query execution. Each piece of the table requires a separate query execution and merging of the query results; moreover, as described earlier in this section, Vertica suffers from relatively high minimal query cost, for, no matter how good the design is, a query has to perform at least one seek and one read per column of each mini-ROS accessed. Therefore, the fundamental difference is that while updates in a row-store have an observable linear cost that increases with the number of tables added into the design, in Vertica the update cost is split into the amortized maintenance cost (moving WOS contents to disk and merging mini-ROSs) and the fragmentation penalty incurred by the *SELECT* queries. The upshot of this is that inserts are cheaper in Vertica, but a penalty is imposed on *SELECT* queries. In contrast to that, a row-store DBMS impose higher penalties on inserts but avoids additional penalty on *SELECT* queries.

We finish the discussion on updates in Vertica with a brief description of *DELETE* queries. Just like inserts, performing deletes in-place is prohibitively expensive: deleting a single row would require accessing a page of each column touched by the deleted row. Moreover, since pages are always compressed (LZO or otherwise), modifying the page would require decompressing the page first. Thus, when the *DELETE* arrives, Vertica determines which row was deleted and stores the ID of that row in the WOS alongside the inserts in a *delete vector*. Similar to the *INSERT*s, each read query has to ignore all deleted rows during execution. The maintenance process also has to integrate all pending deletes when merging ROS fragments (i.e., it must not write the deleted row in the newly merged structure).

2.6.4 Data Recovery in Vertica

When using a database, one has to take precautions against data loss in case of a hardware failure. Databases solve this problem by employing a combination of logging, data replication, and snapshots [SKS02]. Here we will discuss machine-level failure (caused by things like power failures or operating system crashes). Such failure is the only case where the database design tool can offer some solutions and is therefore most interesting to us. Data loss can also be caused by other kinds of failure (such as hard disk corruption), but these issues cannot be solved by the physical design (hard drive corruption is typically solved by employing RAIDs [RAI]). Since we are talking about machine-level failure, data recovery can only be discussed presuming the presence of multiple machines.

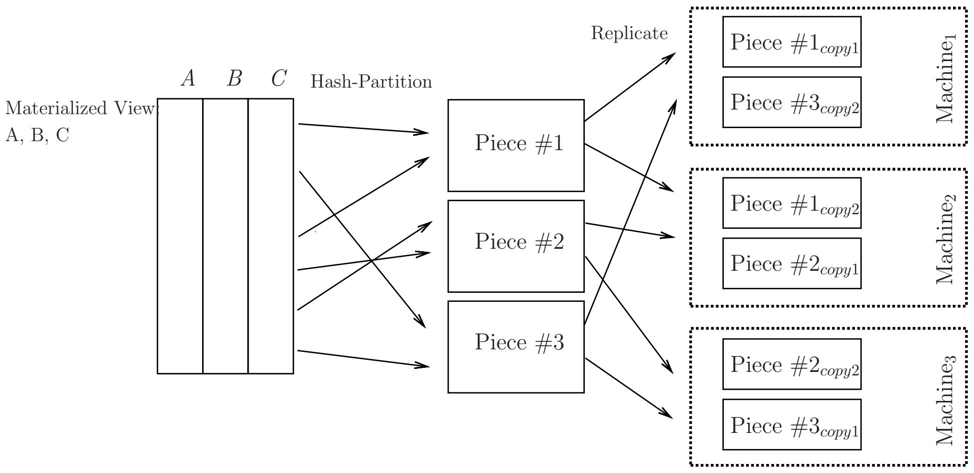


Figure 2.8: **Replication in Vertica** – 1-safe Design

In Vertica we rely on data replication as a precaution against loss by creating extra copies of table rows across different machines. Intuitively, we want to keep a spare copy of data to replace every row from the failed machine using the still-operational machines. The replication process can be extended to store multiple independent copies in order to tolerate multiple failures. In Vertica this is referred to as *K-safety*, where the value of K is the number of machines that can be lost without suffering irreparable data loss. The table is segmented into multiple pieces using a hashing function, and each of the table slices is replicated on at least $K + 1$ different machines. Note that table partitioning can be (and often is) independent from the sort order of the table. The replication approach for $K = 1$ (1-safe design) is shown in Figure 2.8. The default design that does not employ any replication is said to be 0-safe.

The obvious cost of employing data replication is the extra disk space spent on the redundant copies of the same data table. However, all of the costs are increased; thus, even if disk space is not a concern, the maintenance costs (which can become a bottleneck in a DBMS and will be discussed in Chapter 5) also increase by a factor of K. We will present a method for generating a more efficient K-safe design in Section 6.4.

Chapter 3

Related Work

Over the past decade, there has been a great deal of research on the topic of automatic database design [CS95, CD97, CN98a, CN98b, ACN00, ACK+04, VZZ+00, ZZL+04, ZRL+04a, ZRL+04b]. Most commercial products include a tool or wizard that implements some of the ideas developed by the research community. This chapter summarizes existing design tools and the closely related published research to clarify the context within which we developed our design tool.

The bulk of the previous research has been performed in the context of row stores; hence, a new tool must be created for a clustered column-store database such as C-Store or Vertica. From a very high-level point of view, all automatic design tools perform the same set of tasks. First, they enumerate a heuristically chosen set of structures (based on the query training set) and apply a cost model to evaluate the benefits of each structure. Then, the best subset of the candidate structures is chosen using a search algorithm, varying between using a simple, greedy algorithm and an an exhaustive or optimal solution when possible. However, the particulars of each of these generic steps have to be modified to account for the fundamental differences between DBMSs. Most notably, the mechanism used to search the candidate space is significantly different between a clustered column-store and a row-store database. In this chapter, we introduce the existing research tools and compare them to the design tool presented in this dissertation.

3.1 Design Tools in a Column-store Database

The amount of research published on automatic column-store database design is relatively limited. Furthermore, a significant proportion of the research has been performed in the context of MonetDB [Mon], which faces different challenges to those of a clustered column-store database such as Vertica. For example, although MonetDB is also a column store, it does not support data clustering [BZN05]. In order to keep the data sorted, Vertica has to sort every auxiliary design structure according to the sort key and maintain that sort order in the presence of updates. MonetDB uses *data segmentation* (or *partial sorting*) to avoid incurring such maintenance overheads. That is, rather than being fully sorted as in Vertica, the data in MonetDB is divided into different segments or partitions, with meta-data describing the particular range of values placed in each segment. This mechanism is similar to the row-store design approach presented in [PA04] and described later in more detail in this chapter. The idea behind generating a good physical design in the MonetDB setting (known as *database cracking* [KM05, IKM07a]) is to continuously re-segment the data within columns as the queries access the contents of the database. There are two advantages to this approach. First, the database dynamically adapts to the query workload as queries arrive. Therefore, if the current query workload were to change significantly, we could expect the DBMS to (eventually) repartition itself accordingly. Second, we gain an advantage by piggybacking the dynamic design, namely using the work that has already been performed by the user queries (query execution is likely to do more work than is necessary, but this work cannot be avoided). For example, suppose the user query accesses column A by reading all values that are less than 10. Consequently, an intermediate result containing all A values less than 10 will be created during query execution. We might then take this opportunity to partition the contents of column A into two segments: values less than 10 and values greater than 10, accordingly. After we record the necessary meta-data to reflect this separation, the subsequent query that needs values less than 10 can reuse the results that have already been produced. Moreover, any query that needs, say, values greater than 15 will only read the segment containing values greater than 10 and can further partition that segment into two pieces: $(10, 15)$ and $(15, \infty)$.

Later work in this context has investigated efficiently supporting *INSERT* and other update queries in *cracked* database [IKM07b]. Similar to Vertica, the aim of this approach is to delay inserts and deletes in order to amortize the average cost. The individual segments are not sorted in any particular way (the DBMS is only aware that a particular segment contains values in the range $(10, 15)$). Therefore, we append all updates to the end of the segment. Every time the segment changes (i.e., when it is split or merged), or periodically, the contents can be compacted by applying the deletes and rebuilding the segment.

The advantage of database cracking is primarily in its low startup time; the user expects to have to have immediate access to the operational default design and, as the queries arrive, the database redesigns itself accordingly. However, this approach still assumes that the current workload is representative of future queries (otherwise, the database will adapt poorly). This reorganization also depends on the order in which queries arrive (i.e., if different queries arrive on Monday compared with those that arrive on Tuesday, then the database will continuously rearrange itself). Thus, it is reasonable to assume that a representative query workload will be provided in advance, possibly by logging user queries over a few days. We will also demonstrate throughout this dissertation that sorting data (rather than partitioning it) can lead to a very high compression ratio in a column store database.

A recent study by [HZN$^+$10] has investigated ways to support data clustering in the MonetDB setting. By recognizing that data clustering can provide significant performance benefits, this study presented a mechanism that delays updates in memory while keeping track of the position where the change belongs. As mentioned earlier, the most intuitive way to model an *UPDATE* query is through a composition of *DELETE* and *INSERT*.

A special indexing structure (essentially a B-Tree with some additional meta-data and access methods) is used to keep track of inserted and deleted values. Moreover, similar to Vertica, each query has to confer with the delayed set of updates to be processed correctly. However, by contrast, this approach does not support moving out such memory-indexing structures to the hard disk. Instead, different positional index trees can be merged with each other or with the main table.

3.2 The Basics of the Physical Database Design

At a 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 from which a final design is chosen. The steps are characterized as follows:

1. For each training query, heuristically generate one or more physical candidate structures, materialized views (MVs), indexes that enhance the performance of that query.

2. Generate *shared* candidate structures based on groups of similar queries that can serve multiple queries simultaneously. This step is typically performed to save space or to reduce maintenance costs.

3. Select the set of candidate structures that minimizes total query runtime but remains within the user's (space) budget constraints. These candidates are then added to the growing pool of candidates.

4. Repeat steps #2 and #3 until there is no significant improvement from one iteration to the next.

As highlighted above, the majority of research work involves determining the specifics details for each of the above four steps. In the remainder of this section, we will discuss how these steps are approached in a row-store setting.

3.2.1 Microsoft AutoAdmin

Microsoft AutoAdmin [Aut] is an umbrella project that spans various database design projects over the past decade. Most of this research has been experimentally evaluated in the context of the Microsoft SQL Server DBMS [sqlb], and much of it has been incorporated into the commercial release of that same DBMS.

Early pioneering work quickly established a row-store solution for finding a good set of secondary indexes (Section 2.5.1) to speed up user queries. Although only one of the table indexes can be promoted to a primary index (since the table can only be sorted in one way; see also Section 2.5.3), this problem has not been investigated in detail. The basic approach proposed in [CN98a] starts by generating the best possible single-column indexes. The authors then widen the space by considering two-column indexes that contain a previously generated one-column index. That is, the leading column must have already be selected as a single-column index. While in theory this process could consider deeper keys, because of some of the earlier observations, very wide composite secondary indexes that are very wide do not work well in a row store. By contrast, our design tool starts with ideal MVs for each query as the base clustering round; however, we avoid limiting 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. We consider most reasonable prefix permutations.

This work on discovering indexes was followed by a publication on choosing MVs (see Section 2.5.4) with indexes in [ACN00], which had recognized that benefits from indexes alone (most of which are secondary indexes) are limited. Similarly, we also aim to select MVs and indexes (i.e., sort orders) in this research study.

AutoAdmin [ABCN06] also makes a post pass to merge views that were created in the first phase and "view merging" to account for the constrained space. In our setting, adding columns to the sort order may actually reduce the size of the entire MV because of the effects of compression. Thus, we cannot separate the two activities. It is tempting to say that view merging is similar to query clustering; however, query clustering is performed before sort orders are selected, and is based on different metrics.

The problem of workload compression has been studied in [CGN02]. This study clustered queries based on common features and merged similar queries to reduce the size of a large workload. For example, the most trivial query compression scheme merged queries that were identical except for constants. This is reminiscent of our query-grouping approach, although we use this method to drive our candidate generation phase rather than to reduce the initial set of MV candidates, as was done in the context of the row-store design.

AutoAdmin also uses a two-phase process, generating MVs and indexes independently, each within its own budget. In addition to the obvious issue of determining how to divide the existing budget between two different (yet interacting) sets of physical structures, pre-joins and sort orders must be considered together to avoid recomputing the cost of the same sort order multiple times; thus, we cannot take the same approach in our present study. In contrast, much of the cost model computation can be reused, because columns are independently accessed in a column store. The additional pre-joins or extra columns do not change the clustering associated with the same sort order.

3.2.2 CORADD: CORrelation Aware Database Designer

The *CORADD* designer uses a a similar approach to the work presented here. The CORADD is implemented in the context of a major commercial row store rather than in a column store. In generating MV candidates, the *CORADD* relies on a set of special *correlated map* (CM) index structures that are introduced in [KHR+09], which are discussed in more detail in Section 3.2.2. The *CORADD* reserves a fixed amount of space for each design candidate to have a set of CM indexes that are built using the algorithms described in [KHR+09]. This works, because CM indexes are usually very small.

Introduction to CMs

Similar to the secondary index described earlier in Section 2.5.1, a CM is a mapping between the values in the indexed column and the rows that contain those values, except that CMs support creating mappings between *value ranges* instead of between single values stored in a secondary index. Thus, the mapping in a CM index contains an entry for a value range such as (01-01-1992, 01-31-1992); of course, the range could also contain a single value. The corresponding row pointer would also contain a set of ranges – in this example, all row ranges that contain any day in January 1992. Normally, this approach would offer little benefit over a regular secondary index because January days could be anywhere. Thus, the prerequisite to effectively using CMs is to sort the data in the table on a correlated attribute (i.e., to include a correlated column somewhere in the composite sort order). Once the data is at least somewhat sorted by a correlated attribute, the CM index becomes highly compressed. For example, if the indexed table were sorted on the *year* column, the row range for (01-01-1992, 01-31-1992) could contain a single range corresponding to the year 1992. Alternatively, if the indexed table were sorted on the *year_month* combination, then the index would contain the range of rows corresponding to 01-1992. Note that in the former case, the CM index permits *false positives* to achieve better compression, since it will only restrict us to the right year and that will include all months of the year 1992. In practice, that is not a problem, since the query is still responsible for eliminating the rows it does not need: CMs only provide a hint by narrowing the range of rows that need to be read.

CM indexes effectively permit a row store to enjoy some of the benefits that a column store such as Vertica supports naturally. They can create *sparse* secondary indexes (i.e., indexes that do not contain a mapping for every row in the table), thereby avoiding most of the problems caused by typical secondary indexes. The CM index is compressed by storing ranges of values instead of individual values (i.e., CM is *sparse* rather than *dense* and thus stores fewer entries by an order of magnitude); thus, CMs are much smaller on disk and (through the special mechanism of delayed updates described in [KHR+09]) can be much cheaper to update. Using CMs can also allow us to use a subset of the composite index (i.e., access the range of rows matching a predicate over the second or third column of the composite index, instead of a prefix only). Vertica retains the advantage of compressing the data itself and although CMs are well compressed, they contain a copy of the indexed data and the original data (which is stored in a row-store MV) is not compressed.

Moreover, until CMs are built into row-store DBMSs, they require a custom front-end implementation using *query rewrite* [KHR+09, YYTM10]. Every *SELECT* and *INSERT* query has to be rewritten. Read queries are changed by adding the CM-supplied hints, and write queries are used to update the structure of CM. In the design tool presented here, we do not rely on query rewrite. However, we believe that Vertica could greatly benefit from employing index structures similar to CMs.

MV Design in the CORADD

The process of MV generation in the CORADD bears some similarity to that presented in this dissertation (the CORADD also uses selectivity vectors to produce query groupings, although in a different manner). However, we use iterative hierarchical merging to form query groups and take an entirely different approach to generating MV candidates for the resulting query group. In particular, our approach to generating MV candidates is specifically designed for a clustered column store, such as Vertica and is a significant part of our contribution (see Chapter 6). Moreover, we also generate partially pre-joined candidates because one of the goals of our design tool is to improve insert-heavy workloads. The work in the CORADD may benefit from considering partially pre-joined MVs (Chapter 6 explains this assumption), but an even bigger motivation for considering partial pre-joins in this work is to generate designs that can tolerate high insert rates. The latter reason is not applicable in row store, because of their different update framework.

3.2.3 DB2 Design Advisor

The work in [VZZ+00] presented an approach to generating a set of feasible index recommendations using the query optimizer. Relying on the query optimizer (i.e., the component that selects the auxiliary structures that queries use when they are executed) to provide a good set of indexes is similar to what is done in [ACN00]. [VZZ+00] provides more details regarding design of the initial per-query indexes. The objective is to permute query columns based on their types (predicate type, aggregation, etc.) and generate candidate indexes. These candidate indexes are then tentatively (i.e., through simulated entries in the database catalogs instead of through explicit materialization) added into the design, and the best possible query plan is constructed for each query, marking the hypothetical indexes that were used as candidates for consideration. Once a set of indexes for each query has been chosen, the problem is formulated as a knapsack problem (one for chosen indexes, zero for ignored indexes) with a custom solution implemented to shuffle the selected indexes until no better answer can be found or until the user-specified time limit expires. This approach is an early precursor of the linear programming solution used previously in [PA07], [KHR+10] and in this dissertation.

Later work in the context of the DB2 Design Adviser [ZZL+04, ZRL+04a] identified dependencies between different features (such as materialized query tables or indexes). It also recommended searching the space accordingly to ensure that dependent features are searched in tandem, as in the context of the query optimizer. Virtual features are enabled, and plans are generated by considering both the real and virtual features. If the plan uses of one of the virtual features, that feature is suggested as a possible addition to the physical design. Different features are given a different proportions of the available space. The DB2 Design Advisor also relies on query compression and uses the top-K most expensive queries to compensate for the need to re-estimate query costs at every step. With no compression, 80–100 query designs require over two hours.

IBM was the first to consider using the automatic horizontal partitioning of tables across multiple nodes in a parallel, shared-nothing database system. In the current version of our design tool, we solve this problem in the following manner: we choose a high cardinality (i.e., many valued) attribute with significantly more values than the number of available nodes and range-partition all MVs across nodes using that key. We intend to investigate this problem for our setting, particularly in the presence of highly skewed data, in much more detail in future work.

3.2.4 AutoPart

The *AutoPart* project [PA04] employs both horizontal and vertical partitioning to partition large database tables in order to speed up queries. Horizontal partitioning is based on popular columns (i.e., those frequently accessed by training queries) and offers a solution similar to both sorting data (as employed by most row stores and Vertica) and partitioning data (used by [KM05] with MonetDB). The vertical partitioning of the data tables is also based on the column ranges accessed by the training queries. The goal of vertical partitioning is to offset the penalties inherent to data access in row stores. Of course, like any other column-store database, Vertica vertically partitions every column in the table.

Horizontal and vertical partitioning divide the table up into rectangular chunks of different proportions, and the job of the optimizer is thus to determine which chunks need to be read and joined to answer the query. The authors of [PA04] observed that a typical row-store design approach ends up creating *covering indexes* (i.e., indexes that cover all the query's columns, either through the index itself or through the additional use of the included columns described in Section 2.5.2). Such designs typically do well in execution of *SELECT* queries, but replicating a number of columns in several indexes significantly increases the insert overheads. Thus, when using multi-terabyte data warehouses, it might be beneficial to limit the amount of data replication instead of relying on data partitioning. The AutoPart project replicates a limited number of columns, focusing on particularly popular attributes. Vertica uses the two level buffering approach described in Section 2.6.3 to address the very same problem of sustaining a high insert rate. As a result, we have to deal with different performance bottlenecks in Vertica.

Chapter 4

Materialized View Candidates

4.1 Column-Store vs. Row-Store Databases

We start this chapter by briefly summarizing the fundamental architectural differences that come into play when adapting design tool algorithms designed for a row-oriented DBMS setting to a column-oriented DBMS setting. The design problems for column-store and row-store DBMSs can be distinguished on the basis of four prominent factors, which are summarized in the following sections. Following these summaries, we discuss the considerations involved in choosing a suitable materialized view (MV) for a query in a column-store DBMS.

4.1.1 Columnar Access

The most apparent difference between column-store and row-store DBMSs is that column-store DBMSs store (and thereby access) data columns as individual files. Typically, this is an advantage because a query is able to read only the proportion of the data necessary for its task. In fact, row-store DBMSs often achieve improved performance by vertically partitioning the table (i.e., creating an additional MV that contains a subset of the table) This advantage is inherent in the design of column-store DBMSs – every table is already partitioned into individual column files.

However, storing table columns individually also presents some inherent drawbacks. A query that needs to access multiple columns encounters two problems. First, the query executor must process each column individually, keeping track of the necessary meta-data (e.g., the rows that have already been chosen by the predicates) as it processes the query. Second, the values in each individual column need to be matched to form the data rows that the query returns to a user. Such value matching is a form of a join operation (even if the query accesses a single table and thereby contains no explicit table joins), and join operations are notoriously expensive. In Vertica, the problem of matching rows is simplified by keeping most table data in the immutable Read Only Store (ROS) structures (see Section 2.6.3), where row values can always be matched by their relative positions within the data file. A similar solution has been proposed in [HZN+10], which has some implications on *INSERT* and *DELETE* operations, as discussed below.

An alternative approach, employed in [Mon], is to explicitly keep track of column positions. This allows reorganizing each column individually, which may entail less update work but makes column join operations more expensive.

4.1.2 Compression

The second significant difference arises from the ubiquitous presence of compression. Using compression in a database is not a new concept per se, because the bottleneck in query performance in a DBMS has always been determined by the amount of accessed data on the hard disk (i.e., I/O cost). However, as pointed out earlier in Section 2.6.1, a clustered column store has a higher propensity for compression because each file necessarily contains homogeneous data. In Vertica, compression can often reduce the size of the MV columns by one order of magnitude or more. In addition, estimating the size of the MV candidate becomes a more challenging issue. According to the literature, estimating the size of an MV in a row-store, although not necessarily trivial, is relatively simple to perform. If we consider the work in [KHR+10, KHR+09], experimental cost model (further information on cost models is in Section 5.4) evaluation compares the real runtime to runtime estimates.

It is therefore implied that the x-axis (i.e., the space budget) is already accurate.

The second important issue, which might be unique to a clustered column store, is handling data decompression. The conventional approach is to read and decode every disk page as the query is processed. However, in a column store, the query processing engine has a viable alternative because it can operate directly on the compressed data (see Section 2.6.2). Keeping the intermediate results of the query compressed in RAM is almost as important as the I/O savings achieved by column compression. Although I/O costs tend to dominate the cost of a query, processing millions of compressed rows in a column store can result in a bottleneck as columns are processed one by one. For example, in a scale-4 SSB [POX] dataset containing 24 million rows, Run Length Encoding (RLE) compression may reduce the size of the column to less than one disk page, but materializing such a column and applying a simple predicate would require 24 MB of memory and 24 million operations.

Here, it is worth noting that it is theoretically possible for queries to operate directly on the compressed data in a row-store DBMS. However, some of the encoding methods that we use are unavailable in a row-store DBMS (RLE only works on data that is both sorted and homogeneous). Other encoding methods such as dictionary compression are available in row-store DBMSs. However, each disk page in a row-store would have to employ several different encoding methods (one for every column), making it much more difficult to directly process compressed data. Moreover, as explained in Section 4.1.1, in a column store columns are processed sequentially; and thereby each column is a potential bottleneck.

4.1.3 Disk Access Patterns

The next important distinction between column-store and row-store DBMSs that affects database design is the importance of the disk read pattern. The I/O cost of accessing DBMS data is essentially a combination of seeks and reads to obtain the requested ranges of rows. This statement is true of both row-store and column-store DBMSs. However, estimating the exact disk read pattern in a clustered column store is far more challenging for numerous reasons.

First, because each column is read from a separate file any read pattern needed by the query (as it is induced by the filtering predicates) is applied to every column file individually. The cost of every extra seek or extra page read (e.g., because of an outlier row) is effectively multiplied by the number of columns that the query accesses. As our experiments will demonstrate, the cost of extraneous reads and seeks often comprises the bulk of the query execution cost. Note that the order in which columns are processed is also significant because the read pattern is continuously updated as the columns are processed and as query predicates are applied.

Second, the combination of large disk pages and compression often causes the DBMS query engine to read and decompress more extraneous data in a column-store than in a row-store database. Although the page size in a DBMS is subject to configuration, the aggressive use of compression in a clustered column store such as Vertica requires larger disk pages. For instance, Vertica implementation chooses to store a mapping dictionary on each page (we discuss the reasons for this choice in Section 5.1.4) but keeping the page small would negate the benefits derived dictionary encoding (see Section 2.6.1), since the dictionary would consume a large proportion of the page.

Third, comparing the performances of a row-store and a column-store DBMS that uses identical disk page sizes, reveals that the the column-store DBMS still has the propensity to read more extraneous data. In a systems with a typical 4 KB page and assuming approximately 100 bytes per row, a row-store DBMS would keep about 40 values on each page, whereas a column-store would contain approximately 500 values on each page (assuming 8 bytes per value). Therefore, we can conclude that, even with identical page sizes, a single outlier row would cause the query to read 39 unneeded values in a row-store and 499 unnecessary values in a column store. In practice, even setting aside RLE compression (which can achieve extremely high compression rates), a 64 KB page in Vertica can fit anywhere between 10K to 100K values. Therefore, monitoring and controlling which pages will be read by the query is considerably more important in a clustered column-store DBMS.

4.1.4 Insert Mechanisms

The implementation of a particular update mechanism depends on the DBMS in question (the Vertica update mechanism was detailed in Section 2.6.3). However, a certain type of in-memory buffering mechanism is always employed by a column store that supports data clustering [SAB$^+$05, Ver, HZN$^+$10, Gre]. Such an update buffering mechanism is necessitated by the inherent column-store differences listed earlier. Every update needs to modify a value in every column file of every relation (MVs and tables) that it touches. Moreover, owing to compression, after locating the right page with the modified value, the update will have to both read and decompress that page; and, before the page is written back to disk, it would need to be recompressed.

Thus, to sustain such a buffering mechanism, a clustered column-store must support a query execution engine that dynamically incorporates buffered updates into its query results. The design tool also has to be aware of how the buffering mechanism might affect the overall design. As we show through as experimental evaluation, ignoring this update architecture in the presence of a high update rate leads to inferior designs.

4.2 Materialized View Considerations

This section covers the set of decisions taken in order to generate an MV candidate. The algorithms for making these choices will be discussed in Section 6. As we explained in Section 2.5.4, an MV consists of two major components: an MV pre-join and an MV clustering key.

There are two steps to building an MV: the underlying pre-join and selecting the key on which the MV is clustered. Note that in this work, we do not consider *pre-aggregated* [HRU96] or *pre-filtered* MVs. A pre-aggregated MV is an MV that pre-computes an aggregate query, such as a list of average store revenues for every state. A pre-filtered MV is an MV that pre-computes a query with a filtering predicate, such as the query requesting the list of transactions made only in Rhode Island. We do not consider such MV candidates for three reasons:

- The pre-computation used in some of the aggregate types cannot be efficiently maintained. For example, MIN/MAX aggregates are extremely expensive (and thereby impractical) to maintain when the database is updated. Further, a pre-aggregated MV cannot be used in a join because each row in an aggregated MV represents information from several rows and thus cannot be joined with another table.

- Vertica does not support any *lossy* MVs, relying instead on compression to sustain a sufficient number of additional MVs. We reiterate that a lossy MV is an MV that cannot be used to recover the original data from the table or an MV that contains more rows than the largest of the pre-joined tables.

- We believe that the problem of selecting pre-computed MVs in a column-store is similar to the same problem in a row-store. The extensive literature on pre-aggregated MVs [AAD$^+$96, GSW98, CS96, ACK$^+$04, ACN00] remains relevant for column-store DBMS as long as the query workload benefits from the MVs and the underlying DBMS supports them. Similarly, when dealing with pre-filtered MVs, literature about row-store DBMSs is quite analogous to the column-store DBMS problem, since the ideal goal there remains building an MV that matches the query filters (i.e., pre-computing the query verbatim and storing it in a DBMS).

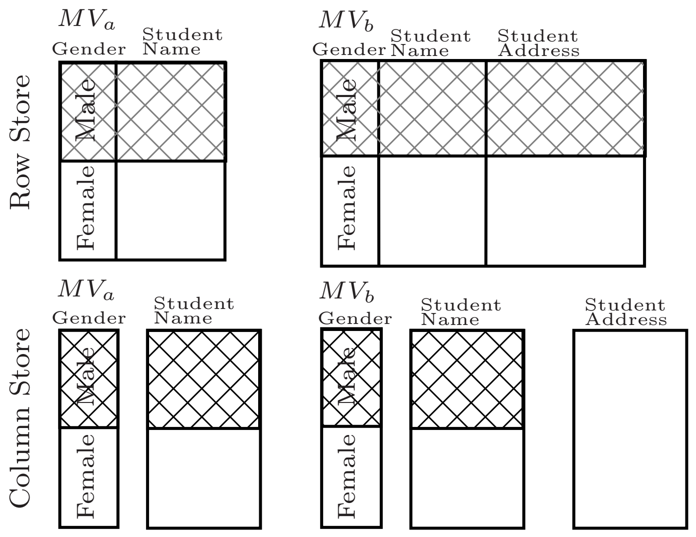
In the remainder of this section, we describe the differences in row-store and column-store DBMSs between choosing a pre-join and choosing the clustering key for an MV We then discuss the inherent cost of the updates incurred by MVs.

4.2.1 Pre-joins in MVs

We begin by revisiting the topic of column compression and how it creates a preference for a certain type of MVs. It should be remembered that one of the reasons a column-store DBMS can significantly outperform a row-store DBMS is by its ability to use individualized compression and storage methods for every column. Some row stores support compression (for instance, DBMS-X), but as discussed in Section 2.6.1 and as we explain in Section 5.1.1, the data in the column-store DBMS will be better compressed because of its homogeneity. MonetDB [Mon] and Greenplum [Gre] support individual column encoding similar to Vertica; however, without MV sorting (MonetDB) or with limited MV sorting (Greenplum), the compression ratio would be poorer than that in Vertica. As demonstrated later, a combination of MV sorting and individual column compression provides many opportunities to aggressively compress the contents of a database.

Selecting a set of tables for an MV to pre-join in a column store is similar to the same problem in a row store. In both cases, the tables (connected by a primary key–foreign key (PK–FK) relationship) can be selected for a pre-join to form an MV. In Section 4.2.5, we discuss the differences in the update penalties incurred by such pre-joined MVs. When considering different candidate MVs, the width of an MV (i.e., total sum of column sizes in bytes) is important. In Section 5.2 we explain the costs associated with having MVs of different widths and describe how this association is correlated with the MV on-disk size. Without going into details, the width of an MV directly affects the expected query runtime in a row store, whereas column-store queries are indifferent to this factor. This is because in a column-store DBMS, the query engine can access all columns individually and, thus, the underlying width of the MV does not affect the execution cost of the query. By contrast, in a row-store all accessed rows must be read in their entirety and, therefore, every new column added to an MV will necessarily slow all the queries using that MV.

In Figure 4.1 the two example MVs are clustered by gender, thereby allowing query Q to efficiently access only the data of the male students in the database. If we were to consider a wider MV$_b$ that also includes the students' address (for the benefit of another query), then in the row-store we immediately increase the cost of executing the original query Q, because it is now forced to read the addresses for all male students in addition to their names. In the column store, the existence of an additional column in the MV has no effect on the cost or execution plan of the original query Q because the columns are stored separately.



Q: **SELECT** (Student Name)
FROM Students
WHERE gender = 'Male';

Figure 4.1: **Varying MV Width** – A Detrimental Effect on the Query Runtime in a Row-store

As a result of this distinction between row-store and column-store DBMSs, there is a natural tendency for good row-store designs to contain narrower MVs. This tendency, in turn, has some implications on the space of the possible sort orders (clustering keys). The width of an MV is factor that determines how many feasible clustering keys exist. Technically, any combination of columns can be used as a clustering key. However, some clustering keys will be inferior. For example, a PK followed by any other column is inferior to a simple clustering on a PK, because sorting by a PK establishes a fixed order (a PK is unique by definition) and no further sorting is possible. The clustering key loses its usefulness before the composite cardinality of the key reaches the row count (sooner in a row store). The details of designing a sort order in Vertica as well as the corresponding limitations of a row store are discussed further in Section 6.1.2. Briefly, as Vertica is a column-store, it can take advantage of any part of the clustering key, whereas row store cannot. However, some commercial row stores have developed features that can provide some of the the benefits available to a column store. For example, CMs [KHR+09] (Correlation Maps – indexing structures that support approximate mapping of value ranges to value ranges) allow some random index access into an MV, while Oracle supports special purpose structures [Orab, Orac] that allow certain queries to achieve a performance that is similar to that of a column store. We believe that our design approach could be adopted to produce MVs for a row store that would benefit from indexing structures similar to [Orab] and [Orac].

4.2.2 Clustered Keys in MVs

The sort order for an MV is similar to the clustering key in an MV in a row store. The columns of the pre-joined MV are sorted and organized based on a subset of these columns that was chosen as a clustering key. As long as the clustering key is chosen well, the queries will be able to efficiently access only the necessary proportion of the MV data. The solution to selecting a “good” clustering key is more complex in Vertica compared to any row-store DBMS. For now, we only establish the underlying reasons for the difference and the increased complexity of the problem in Vertica; the discussion of how to build clustered keys and some further evaluation are given in Chapter 6. We provide a brief summary of the inherent differences in MV maintenance in a clustered column store; a more detailed discussion is presented in Chapter 5.

Each possible clustered key in a Vertica MV has an equivalent in a row store; thus, the space of available clustered key choices (i.e., the complexity of an exhaustive search) is theoretically the same. However, the number of “interesting” choices that must be considered in a column store is much larger than that in a row store for several reasons. First, the same argument made earlier about the effect of the width of an MV in a row store (see Figure 4.1) applies to the clustered key width. Adding more columns to the clustered key in a row store will cause the query performance to deteriorate and will increase the maintenance costs. For example, Figure 4.2, which shows the relative (normalized) performance of the clustering key in a row store compared with the equivalent sort order in a column store. We progressively add more columns to the clustering key, measuring the total key width in bytes on the x-axis. The query Q_l has a simple fixed predicate (*region*=’ASIA’) applied to the first column in the clustering key.

It should be noted that in Vertica, the performance of the sort order remains constant despite the changes in the length of the clustering key. The nature of the column store permits access to the relevant columns, independent of the rest of the columns. At the same time, the performance of the row store begins to deteriorate with the increased length of the index key. Although we are executing the same predicate, the underlying size of the index grows as we add more columns, and the same range of values becomes increasingly expensive to read from the disk. This further illustrates our assertion that in a row-store setting the length of the index is naturally limited.

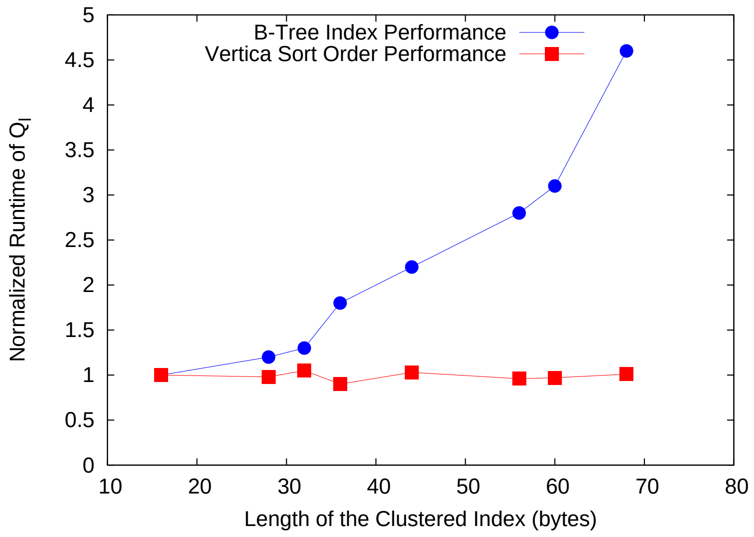


Figure 4.2: **Varying Clustering Key Length** – Normalized Runtimes of Q_l using a Clustered Index in Vertica and DBMS-X

In addition to the issue outlined above, row-store queries also have another problem: they can only use multiple index columns to apply a predicate if all but the last column have equality predicates (e.g., for the index over columns A, B and C, Q_i can only use the entire index if Q_i has equality predicates for both A and B). By contrast, because column stores can access every column of the index individually, each column in the sort order can serve as a bitmap index [bit] of itself (or, to be precise, we can easily compute a bitmap index of the column for any column in the sort order). Intuitively, we can read the individual column, apply the relevant predicate and compute the list of row positions that match the predicate. This implies that in a column store, most feasible permutations of the candidate columns for the sort order should be evaluated, whereas in a row store a significant number of feasible clustering keys can be eliminated from consideration. For example, consider a query with two predicates, A $= 5$ and B < 3; for this query any DBMS can use clustering keys "A, B" and "B." However, "B, A" is better than "B" in a clustered column store, and hence, must also be considered.

Finally, the selection of the clustered key in Vertica, unlike that in a row store, can have a significant effect on the resulting rate of column compression (potentially all columns in the MV, not just the columns in the clustered key; see Section 5.1.1) Thus, although compression can be used in a row-store and DBMS-X supports that functionality, the choice of the clustered key never affects the compression rate in a row store. In contrast, when designing a sort order in Vertica, the design tool has to be aware of the possible implications on compression.

4.2.3 The search space of MV candidates

Every column in a column-store MV is stored in a separate file and compressed using individual encoding. This enables greater flexibility and better compression rates, while simultaneously increasing the search space of feasible MV candidates. In theory, the number of feasible MV pre-joins and clustered keys in a column store is comparable to that in a row store, even though the number of "interesting" (i.e., worth considering) MVs is much larger in Vertica and the problem of selecting good MVs is far more challenging. However, individual column encodings provide an additional dimension that does not exist in row stores, thereby increasing the number of feasible MVs significantly. This is true despite our assumptions that sort order columns use RLE encoding and that not all encoding types can be applied to every column (for example, delta compression cannot be applied to a non-numeric column). We will illustrate our point through the following example. Consider a simple example query that uses the SSB schema [POX]. We enumerate all possible MV candidates below.

SELECT SUM(lo_revenue)
FROM lineorder, dwdate
WHERE lo_orderdate = d_datekey **AND**
d_year = 1997 **AND** lo_discount = 3;

The two pre-join choices are (*lo_discount*, *lo_revenue*, <u>*lo_orderdate*</u>) and (*lo_discount*, *lo_revenue*, *d_year*), where in the first case, we do not pre-join the tables and retain the *dwdate* FK (*lo_orderdate*, underlined) in the MV. For simplicity, we ignore subsets of the sort order (i.e., given the sort order "A, B" we will not consider "A"). For the first pre-join, the possible clustering keys are "*lo_discount*, *lo_revenue*," "*lo_orderdate*, *lo_revenue*," "*lo_discount*, *lo_orderdate*, *lo_revenue*" and "*lo_orderdate*, *lo_discount*, *lo_revenue*," and for the second pre-join, the possible clustering keys are "*lo_discount*, *lo_revenue*," "*d_year*, *lo_revenue*," "*lo_discount*, d_year, *lo_revenue*" and "*d_year*, *lo_discount*, *lo_revenue*."

The default behavior is to assign RLE to all columns in the sort order (see the discussion in Section 5.1.1). However, in each of the eight MVs above (two pre-joins each with four sort orders equals eight distinct MVs), the *lo_revenue* column can be encoded using two different delta encodings (see Section 2.6.1) and, possibly, using dictionary encoding. Similarly, when using the sort order with two columns (such as "*lo_discount*, *lo_revenue*"), the third column (*lo_orderdate* or *d_year*, respectively) can be compressed using dictionary compression or either of the two delta encoding techniques. Thus, we have demonstrated, that the availability of column encoding creates either four times the unique MV candidates (first case) or 12 times the unique MV candidates (second case). For example, the third sort order of the first pre-join, MV3 = (*lo_discount*, *lo_orderdate*, *lo_revenue* | *lo_discount*, *lo_orderdate*, *lo_revenue*) could have four possible encodings: (RLE, RLE, LZO), (RLE, RLE, DeltaC), (RLE, RLE, DeltaV) and (RLE, RLE, Dictionary). Here, because of individual column encodings, the total number of feasible candidates in Vertica is 64, instead of eight in a row store.

4.2.4 Demonstration of the differences between MVs

Having described the conceptual differences between the MVs in row-store and column-store DBMSs, we now demonstrate the performance implications using a simple example query. Note that we are not yet discussing the problem of choosing a good MV (see Chapter 6). Our only goal here is to demonstrate that what constitutes a “good” or “bad” MV differs significantly between a row store and a clustered column store. These differences go beyond the details of the implementation of the DBMS query execution engine but are fundamental to the different architectures. Consider a simple query Q_{ex3} using the SSB schema:

SELECT MAX(lo_revenue)
FROM lineorder, dwdate, supplier
WHERE dwdate.d_datekey = lineorder.lo_orderdate
AND supplier.s_suppkey = lineorder.lo_suppkey
AND s_region **BETWEEN** 'ASIA' **AND** 'EUROPE'
AND d_daynuminyear > 300
AND lo_ordertotalprice < 2000000;

There are three predicates in this query: *s_region* (selectivity = 0.401), *d_daynuminyear* (selectivity = 0.178), and *lo_ordertotalprice* (selectivity = 0.012). Based on this query, four pre-joins must be considered: neither of the dimension tables, one of the two dimension tables, and both of the dimension tables. To simplify, we only consider six different MVs:

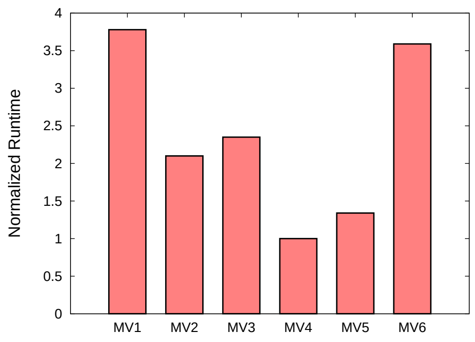
1. MV1: ($lo\_ordertotalprice$, $\underline{lo\_suppkey}$, $\underline{lo\_orderdate}$, $lo\_revenue$ | $lo\_ordertotalprice$)

2. MV2: ($d\_daynuminyear$, $lo\_ordertotalprice$, $\underline{lo\_suppkey}$, $lo\_revenue$ | $d\_daynuminyear$, $lo\_ordertotalprice$)

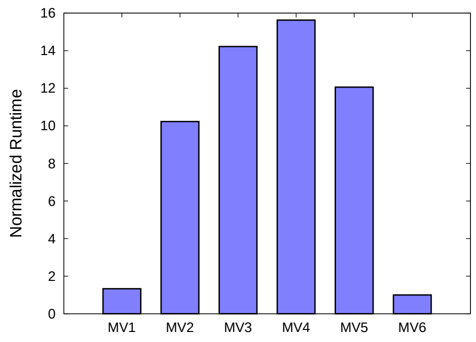
3. MV3: (s_region, $lo\_ordertotalprice$, $\underline{lo\_orderdate}$, $lo\_revenue$ | s_region, $lo\_ordertotalprice$)

4. MV4: (s_region, $d\_daynuminyear$, $lo\_ordertotalprice$, $lo\_revenue$ | s_region, $d\_daynuminyear$, $lo\_ordertotalprice$)

5. MV5: ($d\_daynuminyear$, s_region, $lo\_ordertotalprice$, $lo\_revenue$ | $d\_daynuminyear$, s_region, $lo\_ordertotalprice$)



(a) Normalized runtimes of Q_{ex3} in Vertica



(b) Normalized runtimes of Q_{ex3} in DBMS-X

Figure 4.3: Normalized Runtimes of Q_{ex3} in Different DBMSs

6. MV6: (*lo_ordertotalprice*, *d_daynuminyear*, *s_region*, *lo_revenue* | *lo_ordertotalprice*)

The FKs are underlined in every MV. The first MV requires us to perform a join with both dimension tables. MV2 and MV3 require query Q_{ex3} to perform a join with one of the dimension tables, as well as scanning the MV itself, whereas the last three MVs are fully pre-joined. This set of MVs can be easily translated into an equivalent set of MVs in DBMS-X. The only required modification involves appending the PK of the *lineorder* table to each clustered key (i.e., MV1 becomes (*lo_ordertotalprice*, <u>*lo_suppkey*</u>, <u>*lo_orderdate*</u>, *lo_revenue*, ***lo_orderkey***, ***lo_linenumber*** | *lo_ordertotalprice*, ***lo_orderkey***, ***lo_linenumber***), where ***lo_orderkey***, ***lo_linenumber*** is the PK of the *lineorder* table in the SSB schema. This modification is necessary because DBMS-X requires all clustering keys to be unique. This change does not affect our experiment because we normalize all the query runtimes.

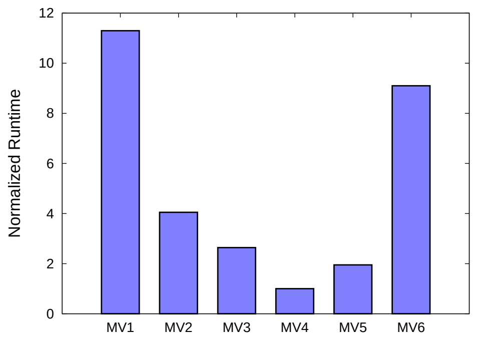
There is generally little insight gained in comparing the absolute runtimes of two radically different DBMSs. Our goal in this work is to establish the fundamental differences between the two systems, such as which MVs tend to be the "best" and which are the "worst." We avoid using any non-RLE encodings here because MVs with individual column encodings have no equivalent in a row store. Although DBMS-X does not support RLE encoding, we consider it an inherent part of the clustering key functionality in Vertica. In the following experiments in this section, we evaluate the benefit of data clustering in two different DBMSs.

Figure 4.3 shows the normalized runtimes of query Q_{ex3} in Vertica and DBMS-X using the six MVs defined earlier in this section. The runtimes are normalized using the "best" (i.e., the fastest) MV in each respective system; thus, we only compare the relative benefits of the MVs in each DBMS. Observe that the relative quality of the MV candidates between Vertica and DBMS-X is almost exactly the other extreme. For example, MV4, the fastest structure in Vertica, is the slowest in DBMS-X. Conversely, MV6, the fastest in DBMS-X, is a close second to the slowest structure in Vertica. This figure demonstrates that assuming the same behavior in a column store can result in the worst possible answer. The difference in the normalization scale is also significant (the performance difference between the fastest and the slowest runtimes is a factor of four in Vertica and a factor of 16 in DBMS-X) and will be discussed at the end of this section. Of course, these results do not imply that Vertica will always behave in direct contrast to that of a row store. In fact, a somewhat similar query Q_{ex4},

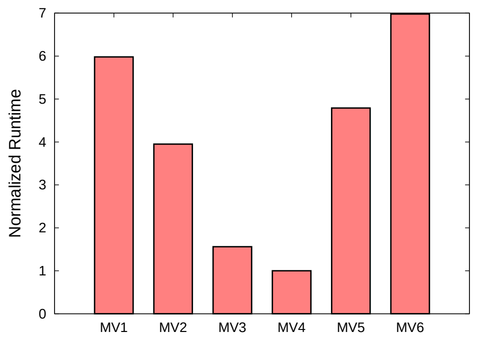
```
SELECT MAX(lo_revenue)
FROM lineorder, dwdate, supplier
WHERE dwdate.d_datekey = lineorder.lo_orderdate AND
supplier.s_suppkey = lineorder.lo_suppkey    AND
s_region  = 'EUROPE' AND d_daynuminyear > 300 AND
 lo_ordertotalprice  < 2000000;
```

will result in more conventional behavior. Figure 4.4 shows the runtimes of Q_{ex4} using the same set of MVs as used in Figure 4.3. Note that the relative behavior of the two systems is now almost identical; MV4 is the best in each system and MV1/MV6 are the worst choices in both systems. Thus, while the row-store behavior remains consistent and predictable (see Section 6.1.2), in Vertica we need to understand the exact query behavior to select the appropriate MV for a query. It should be noted that this example does not consider the additional complexity of individual column encoding, because we only use RLE in the example MVs.

It is tempting to assume that compression is the reason behind the disparity in the performances between Vertica and DBMS-X. Thus, we consider the disk sizes of our six



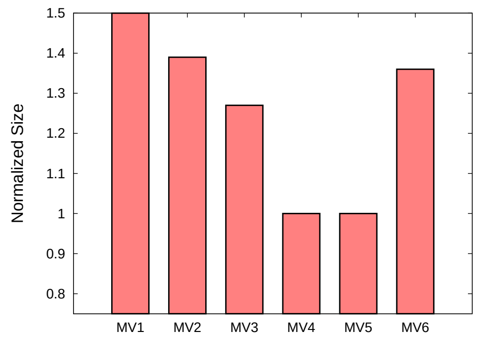
(a) Normalized runtimes of Q_{ex4} in Vertica



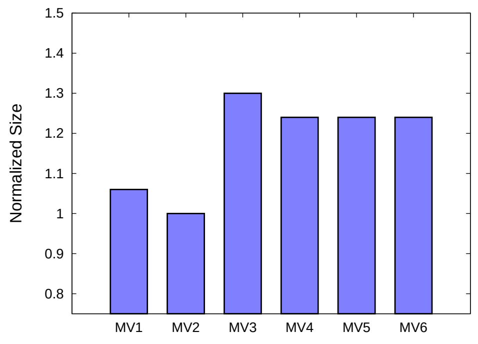
(b) Normalized runtimes of Q_{ex4} in DBMS-X

Figure 4.4: Normalized Runtimes of Q_{ex4} in Different DBMSs

example MVs to evaluate whether such an assumption is valid. As in the previous figures, we normalize the size of each MV by the smallest (MV2 for DBMS-X and MV4 for Vertica). Figure 4.5(a) and 4.5(b) compare the relative sizes of the MVs. Again, there is little reason to compare the storage requirements directly, so we compare the relative size differences instead. The most interesting observation here is that the two smallest MVs in DBMS-X (MV1 and MV2) are also the two largest MVs in Vertica. Even though we have not used compression in the row store, that is irrelevant; applying compression to the MVs in DBMS-X does not change the normalized sizes because the compression rate will be similar across all MVs (in fact, we have verified that the relative sizes remain the same after we apply DBMS-X compression to all example MVs). In a row store, the distribution (i.e., sorting) of the data does not affect the compression rate and, therefore, the clustering key of each MV has no impact on the resulting compression in DBMS-X. However, in a clustered column store such as Vertica, sorting of the MVs significantly affects the resulting compression rate because it determines the available encoding opportunity and data distribution within each column. As we will show in greater detail in Chapter 5, these two goals (the best sort order for a query and the best compression rate) can create a conflict when choosing a sort order for an MV. For now, we observe that the relative sizes of the MV candidates correlate with the relative runtimes of Q_{ex3} in Figure 4.3 but not with those of Q_{ex4} in 4.4. Therefore, we conclude that, at least in this example, the compression rate does not cause the observed



(a) The normalized on-disk size of MV candidates in Vertica



(b) The normalized on-disk size of MV candidates in DBMS-X

Figure 4.5: Normalized Candidate Sizes in Different DBMSs

difference in query runtimes. In Chapters 5 and 6, we elaborate on how the query runtime is affected by clustering and compression in a column-store.

Since we evaluate the relative difference between the performances of Vertica and DBMS-X, we finish this section by identifying another important distinction from Figure 4.3. As we have observed, the slowdown factor varies: the slowest Vertica MV candidate is approximately four times slower than the fastest, while the slowest MV in DBMS-X is approximately 16 times slower than the fastest. This can be explained by the inherent architectural differences in these two DBMSs. If we examine the absolute query runtimes instead of normalized runtimes, the fastest DBMS-X query runtime is much faster than is the fastest candidate in Vertica. The minimal achievable runtime in Vertica is bound by the costs of a single read and single seek per column accessed by the query. Although the amount of data processed is negligible, the cost of several disk seeks and reads can significantly increase the runtime of the query. Conversely, the slowest Vertica candidate is nevertheless faster than is the slowest DBMS-X candidate. This particular difference stems from compression: both RLE and LZO compression techniques reduce the size of the MV. As a result, a relatively poor MV can be processed and read faster in a column store than in a row store.

Table 4.1 shows the comparison between actual (wall-clock) minimum and maximum runtimes in each DBMS. The gap in behavior shown in the examples above may increase when considering multiple queries and wider MVs as we will demonstrate later in this

Candidate	Q_{ex3}	Q_{ex4}
Fastest Vertica MV	0.544	0.393
Fastest DBMS-X MV	0.249	0.299
Slowest Vertica MV	2.055	2.077
Slowest DBMS-X MV	3.899	2.818

Table 4.1: **Runtimes of Best/Worst MVs in each DBMS**

dissertation.

4.2.5 MVs and updates

In a conclusion to this chapter, we briefly discuss the cost differences between the maintenance of MVs in each environment. Note that although update queries refer to *INSERT*, *DELETE* or *UPDATE*, we typically focus on the implications of maintenance necessary for INSERT queries. In an OLAP environment (Online Analytical Processing deals with large amounts of slowly growing data), the number of *INSERT* queries tends to be significantly higher than that of other updates and hence it dominates the overall maintenance costs.

In a row store, the cost of an insert is relatively straightforward to calculate. Each affected MV must be updated with a new row; if any pre-joins are involved, the new row needs to be joined accordingly. Consequently, the new row needs to be inserted in the correct location according to the clustering key of the respective MV. Updates are performed in-place (i.e., the affected pages are modified directly on the disk). Although the modified pages are then stored in a memory cache instead of being written back to the disk immediately, the cost to read them is incurred immediately. Furthermore, any savings from the delayed page write are not insignificant because the likelihood of another insert hitting the same disk page is very low, given the nature of the stereotypical indexing structure, the B-Tree index [Com79]. Some row-store DBMSs reduce costs by buffering the inserts. For example, InnoDB [Inn] in MySQL [MyS] features a special insert buffer that will temporarily delay the inserts until they can be integrated into the MVs in bulk.

In Vertica, estimating the cost of the insert is much more complex. The necessary pre-joins occur when the newly inserted row arrives, just as in a row-store setting. However,

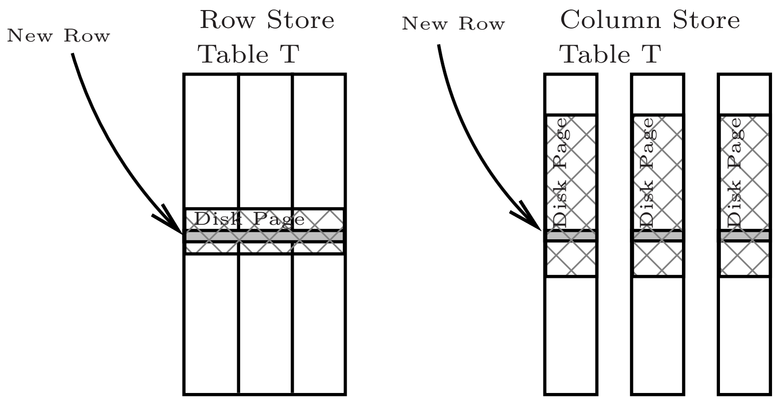


Figure 4.6: **Cost of Inserting one Row** – The Architectural Difference between Inserts in a Row-store and a Column-store Setting

the MV is not updated in-place because that would be prohibitively expensive (see Figure 4.6). In a row store, it is sufficient to find the correct page, read it, and make the necessary change (some additional pages are read while locating the target page in the index, but even if these pages are not already cached, the cost is constant and relatively small [Com79]). In a column-store, the same insert operation has to be simultaneously applied to every column in the affected MV. Moreover, the effective cost is even higher (it can be higher than thrice that for the three-column MV) because each disk page also needs to be decompressed and eventually recompressed before it can be written to disk.

The mini-ROS buffering mechanism (see Section 2.6.3) presents a number of interesting implications for the cost of inserts in Vertica. The immediate cost of the insert is reduced because only the necessary joins for each one need to be performed. The additional costs of sorting, encoding and writing newly inserted data to the disk as a mini-ROS is amortized because data is moved to the disk only when the WOS memory buffer is full. The amortization is replaced by an additional cost to merge mini-ROSs in the background because the MV cannot continue becoming more fragmented indefinitely. Finally, the existence of the mini-ROS fragments increases the cost of the *SELECT* queries. Although this was not the case in the experiments described in Section 4.2.4, it is yet another reason for the relatively (compared to DBMS-X) slow runtimes of the fastest MVs in Vertica. All of the associated trade-offs of the buffering mechanism will be discussed in detail in Chapter 5.

Chapter 5

Resource Allocation

In this chapter, we describe the breakdown of the costs incurred by adding a materialized view (MV) into the physical design. Databases have limited resources, and each MV, in addition to speeding up user queries, employs a share of those resources. For reference, building efficient MV candidates and devising a good physical design from a subset of those candidates within user-defined budget constraints will be discussed in Chapters 6 and 7. In this chapter, we focus on estimating the cost of the MV and explain some of the less obvious consequences of adding too many MVs into the physical design. We consider two distinct (although related) resources: disk space and memory (or RAM). We conclude this chapter by describing our analytic cost model.

Throughout this chapter we use four example queries and their individual *perfect* MVs (i.e., fully pre-joined MVs designed individually for a single query). The four example queries are shown in Figure 5.1 and their corresponding perfect MVs are listed below. We also consider an additional partially pre-joined fifth MV (termed MV_{1f}) that can be used by all four queries but that requires them to perform additional joins. Please note that MV_{1f} consists of all the columns from MV_1 as well as the foreign keys (which are underlined) required to perform a join with *dwdate* and *part* dimension tables. This allows all four queries to use MV_{1f}.

1. MV_1: *d_year*, *lo_extendedprice* | *d_year*

2. MV_2: *s_size*, *lo_quantity*, *lo_ordertotalprice* | *s_size*, *lo_quantity*

3. MV_3: *p_container*, *p_type*, *p_name*, *lo_revenue* | *p_container*, *p_type*, *p_name*

4. MV_4: *d_daynuminyear*, *lo_supplycost*, *lo_tax*, *lo_shipmode*, *lo_commitdate*, *lo_discount*, *lo_quantity*, *d_holidayfl*, *d_year* | *d_daynuminyear*

5. MV_{1f}: *d_year*, $\underline{\textit{lo\_orderdate}}$, $\underline{\textit{lo\_partkey}}$, *lo_extendedprice* | *d_year*, *lo_orderdate*

To confirm, the goal of our design tool is to produce a physical design that contains the best subset of the MV candidates and that fits within the user-specified constraints. For simplicity, we assume that the base tables (i.e., each table of the schema, such as *lineorder* in the SSB benchmark) are independent of the additional MVs in DBMS, namely that any valid SQL query can be answered despite the absence of additional MVs in the design (at a disk budget of zero). This assumption is trivially true in DBMS-X; the base tables are independent of the additional MVs in the design. The only possible optimization that can be applied to the base tables (which was used in [KHR+10] among others) is the addition of a clustering key, which resorts the table, potentially speeding up some queries. In Vertica, it is possible to further optimize the final design by merging one of the design MVs with the anchor base table (i.e., the fact table for that MV). However, unless otherwise noted, the base tables (in our work, a base table refers to an MV that contains all the columns from that table) will be kept separate. Furthermore, as is conventional in row-stores, the primary key of the table is, by default, used as the sort key for the MV.

First, we are going to discuss how to estimate the disk space requirement in the following section, and then, we move on to a discussion on budgeting RAM.

5.1 Disk Space

The most commonly used and easily understood budget constraint in database design is disk space. Each MV occupies a certain amount of disk space and the entire physical design has to fit into the overall user-specified disk budget. Because the concept is straightforward,

Q_1 :

```
SELECT SUM(lo_extendedprice)
FROM lineorder, dwdate
WHERE lo_orderdate = d_datekey
AND d_year = 1993
AND lo_quantity < 25;
```

Q_2 :

```
SELECT SUM(lo_ordertotalprice)
FROM lineorder, dwdate
WHERE lo_orderdate = d_datekey
AND d_dayofweek = 'Monday'
AND lo_discount between 1 and 3;
```

Q_3 :

```
SELECT SUM(lo_revenue)
FROM lineorder, supplier, part
WHERE lo_partkey = p_partkey
AND lo_suppkey = s_suppkey
AND s_nation = 'United Kingdom'
AND p_color = 'turquoise';
```

Q_4 :

```
SELECT SUM(lo_supplycost), SUM(lo_tax)
COUNT(lo_shipmode), MAX(lo_commitdate)
AVG(lo_discount), MAX(lo_quantity)
COUNT(d_holidayfl), MAX(d_year)
FROM lineorder, dwdate
WHERE lo_orderdate = d_datekey
AND d_daynuminyear between 1 and 7;
```

Figure 5.1: **Example Queries** – Four Example Queries (similar to SSB benchmark queries) Used in This Chapter

most of this discussion will focus on correctly estimating the on-disk size of an MV in Vertica. Computing the raw size of each column is simple (i.e., the size of the data attribute multiplied by the number of rows in the MV). However, the multitude of compression methods built into Vertica (see also Section 2.6.1) complicates the issue. In fact, it is worth noting that the default LZO [ZL77b] compression in Vertica cannot be disabled; LZO is applied in the absence of any other encoding method. Moreover, the Vertica query engine can directly operate on data encoded with most of the supported encoding methods (see Section 2.6). Otherwise, the cost incurred in the decompression and materialization of data would often negate the benefits of compression.

5.1.1 Estimating Compression

As explained in Section 2.6.1, Vertica supports various encoding methods, and it is important to correctly estimate the compression ratio of each encoding method to both make the correct cost estimates (see Section 5.4) and select the best encoding approach for each column when building MV candidates (Chapter 6).

In addition to the basic statistical information about the data, several other factors affect the resulting compression rate. First, the sort order of the MV can significantly affect the compression rate of any column in that MV (as touched on Section 4.2.4). This is true even for columns that do not belong to the sort order because of the nature of the data correlation (see Section 2.1.2). For example, suppose column A is correlated with column B. In the case that an MV (containing both A and B) is sorted on column A, the distribution of the values in column B will be affected by the sort order. Intuitively, when data is sorted on A, the values in B will be similarly collocated. For example, sorting USPS shipping data by "shipping date" will cause the "receipt date" to be similarly collocated, because most packages are received a few days after they are sent. In the following sections, we demonstrate how columnar compression works in Vertica and the effects of the correlation for each encoding.

5.1.2 Estimating LZO Compression

To estimate the compression ratio of LZO on column C, we use a random sample S of data from C and compute Size(LZO(S)). This estimate is accurate (to within 5%) as long as the MV sort key is not correlated with the column. More generally, the compression of sample S is suitable for estimating the compressed size of column C as long as C is sorted on a similarly correlated sort order (based on correlation strength as defined in Section 2.1.2). Employing a sort order with a different amount of correlation can introduce inaccuracy (see Figure 5.2). If we need to improve the accuracy, we can resort and LZO compress S. The implementation of our current cost model (which will be described in Section 5.4) accepts any potential errors because computing a custom LZO compression estimate is an expensive operation.

Let us consider how the compressed sizes of the two different columns from the SSB schema change as the columns are sorted by the three different sort orders. Figure 5.2 shows the sizes of the *lo_extendedprice* and *lo_shipmode* columns as the sort order changes between:

1. *lo_orderkey*, *lo_linenumber* (primary key)
2. *lo_revenue*
3. *lo_extendedprice*

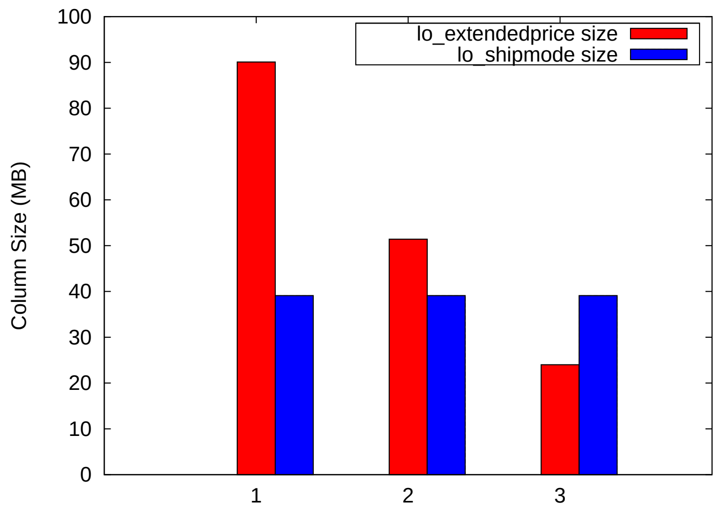


Figure 5.2: **LZO Column Compression** – The Effect of the Correlation on LZO Compression

None of the three sort orders above is correlated with the *lo_shipmode* column. Therefore, the compression rate of *lo_shipmode* remains the same in all three cases. By contrast, *lo_extendedprice* is (obviously) highly correlated to itself (#3), slightly correlated to *lo_revenue* (#2, correlation strength of 0.54) and not correlated to sort order #1. Accordingly, Figure 5.2 shows that the compression rate of *lo_extendedprice* varies in tandem. The sample-based estimate for sort order #1 is computed to be 90 MB, while resorting and recompressing the same sample column using sort order #2 produces an estimate of approximately 52 MB. Resorting the column on itself reduces the size to 24 MB. Finally, in the absence of additional information (i.e., if we are not willing to resort and recompress the sample), the size estimate from sort order #1 is used.

5.1.3 Estimating Run Length Encoding (RLE) Compression

Computing the size of the RLE-ed column is simple as long as we know the number of unique value sequences (or *value runs* [AMF06]) that are being RLE-ed. For the leading column in the MV sort order, the number of value runs is simply the cardinality of the column. However, for subsequent columns in the sort order, we need to know the composite cardinality of multiple columns in order to compute the number of values being RLE-ed. Thus, column C, which leads the sort order, has an RLE-ed cardinality of C (denoted by $|C|$). However, if the same column C were preceded by column A in the sort order, then the number of value runs in C would be the total number of unique values that can be composed from A and C (denoted by $|AC|$). The cardinality $|AC|$ cannot exceed the product of the individual cardinalities ($|A|*|C|$). However, in presence of a correlation, the composite cardinality of the two columns may be smaller. In fact, the correlation strength (see Section 2.1.2) is the measure of the ratio of $|AC|$ and $|A|*|C|$.

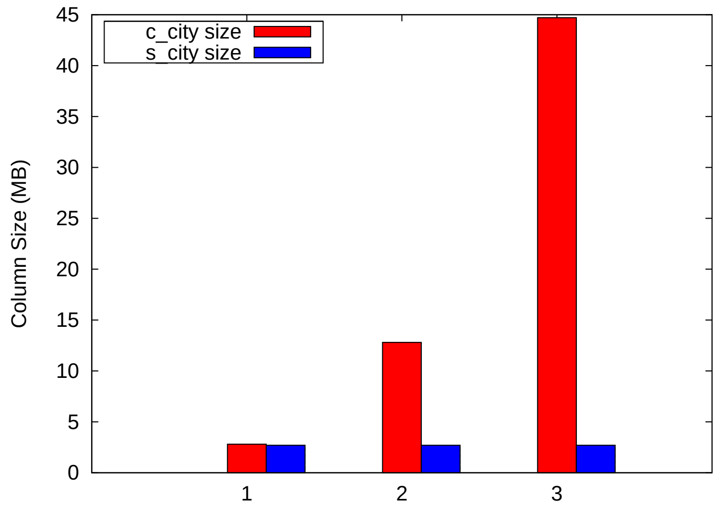


Figure 5.3: **RLE Compression** – The Effect of Correlation on RLE Compression

To illustrate the point through an experimental example, let us consider the RLE-compressed sizes of two columns: customer city (*c_city*) and supplier city (*s_city*). These two columns are chosen because they have similar cardinalities and the same data types (string) and thereby typically similar compression rates. We consider these column sizes when they are preceded by:

1. *lo_orderdate*, *lo_shippriority* (*lo_shippriority* has a cardinality of 1 and, therefore, does not affect RLE compression)
2. *lo_orderdate*, *s_region*

3. *lo_orderdate*, *s_nation*

Figure 5.3 shows the resulting compressions for each of the two city columns. The compression of *s_city* remains constant for all three cases owing to the correlation among *s_region*, *s_nation* and *s_city* that keeps the composite cardinality of the entire sort order low; thus, the compression rate of *s_city* does not deteriorate. By contrast, *c_city* does not enjoy the same benefit and its on-disk storage increases from 4 MB to 14 MB to 45 MB, as sort orders #2 and #3 increase the fragmentation (and subsequently the amount of value runs) in *c_city*.

5.1.4 Estimating Dictionary Encoding Compression

As described in Section 2.6.1, dictionary encoding is a way to represent the values in a column that have a low cardinality (i.e., they have few unique values). This technique can be successfully applied in a row store, although it is more effective in a column store because the column contains homogeneous values. Intuitively, the achieved compression rate depends on the size and number of encoded values, because these factors determine the size of the dictionary and the size of the code entries. For V_D distinct values in column D, $\lceil log_2(V_D) \rceil$ bits are required to represent a single encoded value. In addition, we also need to store the mapping dictionary itself, which requires approximately $(\lceil log_2(V_D) \rceil + Size(V)) * V_D$, namely the column value and its code stored for every entry.

Dictionary encoding in Vertica is page-based, meaning that each page contains all the information (such as the dictionary mapping itself) necessary to be decoded. This decision is a trade-off in that when the dictionary mapping itself is large, storing it on the page occupies a large proportion space, thereby reducing the effective compression rate. This issue is mitigated by using very large disk pages (i.e., 64 KB). Moreover, the advantage of using self-contained dictionary pages is that some value distributions can result in an improved compression rate. Intuitively, when the dictionary is limited to the disk page, its compression rate is based on the *page cardinality* rather than column cardinality (because values outside the current page do not exist for the purposes of the local dictionary). Similar

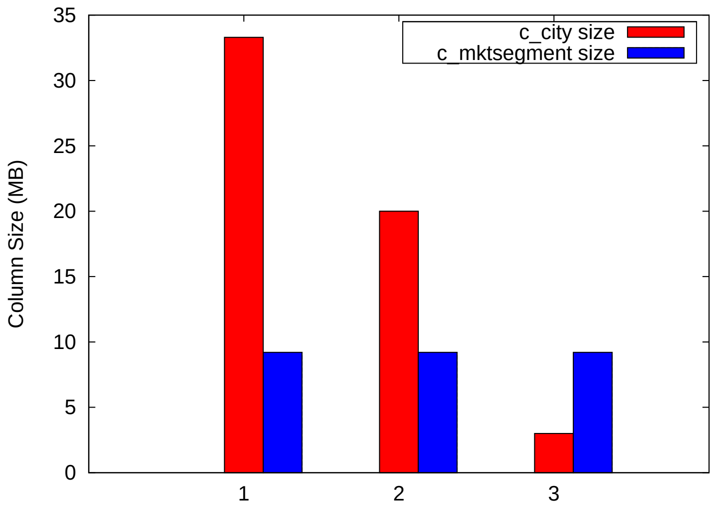


Figure 5.4: **Dictionary Column Compression** – The Effect of the Correlation on Dictionary Compression

to other encoding types, dictionary compression is greatly affected by either explicit or implicit sorting (i.e., sorting through correlation to the sort key).

Figure 5.4 shows the sizes of the *c_city* and *c_mktsegment* columns from the *customer* table sorted in different ways. Both columns are compressed using dictionary encoding and we use the following three sort orders:

1. *lo_orderkey*, *lo_linenumber* (the *lineorder* primary key)
2. *c_region*
3. *c_nation*

As in our previous examples, one of the columns (*c_mktsegment*) is not correlated with any of the sort orders and thereby dictionary compression reduces *c_mktsegment* to approximately 9 MB in all three cases. By contrast, the size of the *c_city* column changes from 34 MB to 20 MB to approximately 3 MB between the different sort orders. This change is caused by the sort orders #2 and #3, which reduce the effective per page cardinality of the *c_city* column. For example, *c_region* describes the geographic region of the customer (such as "ASIA" or "EUROPE"). The cardinality of the *c_region* column is 5 ($|c\_region|$=5), while that of the *c_city* column is 250 ($|c\_city|$=250). Once the data in *c_city* are partitioned by region, each region range has approximately 50 customer cities. Thus, we can estimate the compression rate as $\lceil Log_2(50) = 6 \rceil$ bit $= \frac{3}{4}$ bytes, and at 24 million rows we can estimate the column size as approximately 24 million * 0.75 bytes $= 18$ MB. If we incorporate the additional overheads spent on storing the dictionary on every page, we compute an estimate slightly less than 20 MB.

5.1.5 Estimating Delta Encoding Compression

Similar to the case of dictionary encoding, the delta encoding compression rate depends on the data distribution within the columns. As explained in Section 2.6.1, there are two types of delta compression mechanisms, both of which rely on encoding the differences between the values within a column. The number of bits required to store the Δ values determines the resulting compression rate (or, to be exact, $\lceil Log_2(\Delta) \rceil$). As a rule of thumb, if the data is sorted and the values on a page fall within a narrow range, then $DeltaV$ is a good option to select, whereas if the column data is not sorted but all the values within a page are in a narrow range, then $DeltaC$ is the best option.

As with other encodings, redistribution the data by sorting or indirectly sorting via the sort order correlation will affects the compression rates (it should be noted that the correlation is unlikely to produce highly sorted data, the kind preferable for $DeltaC$). However, a correlation between an encoded column and a sort order can cause the column to be partitioned so that each page has a limited range of values. One real-world example is the relationship between *shipdate* and *receivedate*. The correlation that connects these two columns is best described by an algebraic formula (as proposed in [IMH$^+$04]). For example, the correlation might be described as $receivedate \in [lo\_shipdate{+}3 : lo\_shipdate{+}7]$ (i.e., items are received within a time span of 3–7 days), or $receivedate = lo\_shipdate + 5 \pm 2$. As an alternative to the algebraic relations described in [IMH$^+$04], such correlations can also be obtained using through the bucketing method introduced in [KHR$^+$09], as described in Section 3.2.2.

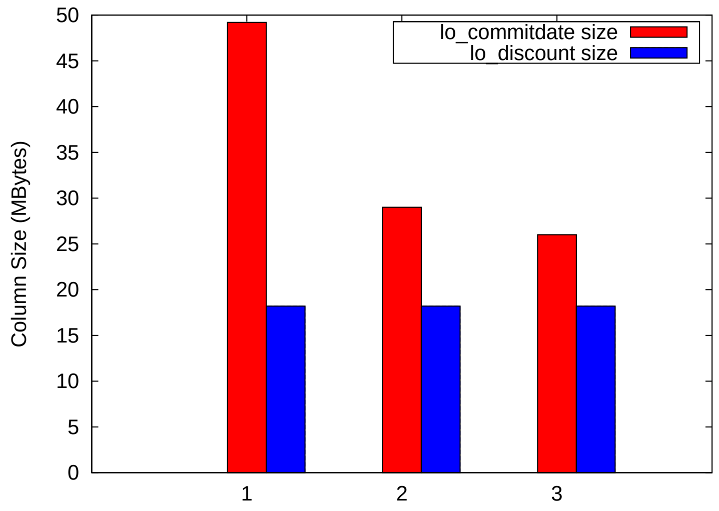


Figure 5.5: **Delta Column Compression** – The Effect of Correlation on DeltaV Compression

Figure 5.5 shows the sizes of the *lo_commitdate* and *lo_discount* columns from the *lineorder* table sorted in different ways. Both columns are encoded using the DeltaV compression, and we use the following three sort orders:

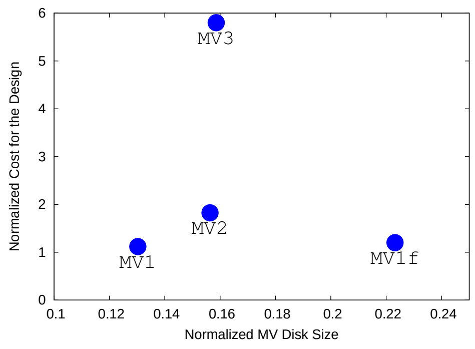
1. *lo_orderkey*, *lo_linenumber* (*lineorder*'s primary key)
2. *d_weeknuminyear*
3. *lo_orderdate*

In this example, the *lo_discount* column is not correlated with any of these three sort orders. Thus, its size remains constant for all sort orders. By contrast, *lo_commitdate* (receipt date) is correlated with sort orders #2 and #3. When sorting with *d_weeknuminyear* (#2), the compression is improved because many similar receipt dates fall within the same week (even though some receipt dates are from different years). Similarly, *lo_orderdate* is correlated with *lo_commitdate* because the item typically arrives within a few days of being ordered. Therefore, for each specific value of *lo_shipdate*, the *lo_commitdate* column will have a corresponding subset of all values from its value domain.

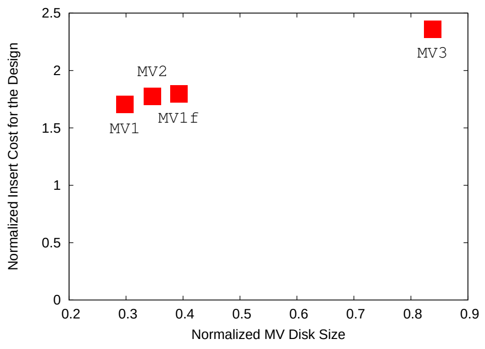
5.2 The Meaning of Disk Space

Why do we need to know the size of an MV? Typically, adding more MVs improves query the performance of a query (at least *SELECT* queries), but each MV takes up a certain amount more disk space. The original reasons for specifying space constraints were either the space availability of the user or the division of the available disk space between multiple databases by the system administrator. However, disks have become progressively cheaper in the past few years. Therefore, if database performance is important, additional hard drives can be purchased. Of course, this approach does not apply to multi-terabyte databases, but for many databases an additional disk configuration that is 10 to 50 times the size of the raw data can comfortably accommodate a highly optimized design. However, even if we had a hard drive of unlimited disk space capacity we would still not be able to deploy an arbitrary physical design. The deployed design needs to be regularly updated as the new data arrives at the DBMS and as each additional MV in the design potentially increases the maintenance cost. In an OLAP data warehouse setting, these updates are executed in batches to amortize the per row update cost (inserting a row batch allows the DBMS to share some of the common execution overheads). Nevertheless, the maintenance cost can still quickly spiral out of control in a large data warehouse [KHR+09]. Chapters 6 and 7 will discuss the budget concerns when generating and selecting candidate MVs for the final design. For now, we only cover the relationship between the size of the physical (disk) design and the associated maintenance cost.

As the following experiments demonstrate, in a row-store, the on-disk size of the physical design closely mirrors the maintenance costs. This relationship makes it easier to predict the expected maintenance costs. However, in a clustered column-store such as Vertica the relationship between the design size and the corresponding maintenance cost is far more complex. In general, it is still true that the maintenance of a larger design is likely to more expensive. For example, if we start with design D_A and add a new structure S_i to obtain D_{A+i}, the maintenance of the resulting design is going to be more expensive. However, other



(a) Vertica Cumulative Insert Cost



(b) DBMS-X Cumulative Insert Cost

Figure 5.6: **Insert Cost vs. Disk Size** – Normalized Performance of 10K Inserts in Vertica and DBMS-X

factors that are relatively insignificant in a row store play a significant role in determining the overall maintenance cost in a column-store. Indeed, it is easily possible to obtain a design D_A that is much larger than design D_B is but cheaper to maintain.

Using four of the five MVs defined at the beginning of this chapter (namely, MV_1, MV_2, MV_3, and MV_{1f}), we plot the maintenance (*INSERT* query) cost for a design containing a single MV. We adapt each of the four MVs to the row-store, as discussed in Section 4.2.3. Figure 5.6 shows the overall insert costs for each DBMS. Note that the insert cost does not include the delayed maintenance costs incurred by Vertica (see Sections 5.3.3 and 5.3.2), nor does it reflect the potential effect on *SELECT* queries (see Section 5.3.4). Figure 5.6(b) reflects the corresponding insert cost in DBMS-X (i.e., the cost of inserting a batch of rows including flushing everything to disk). All values are normalized by the insert cost of the *default design* (i.e., the design that contains only the base table MVs), computed in the same manner. Here, we use insert batches of 10K rows each; a discussion on varying the insert batch size will follow in Section 5.3.1. We reiterate that the maintenance resource (unlike the disk resource) is a function of both the design size and the insert rate. The MVs do not carry an inherent maintenance cost; in the absence of updates, the maintenance cost of any design is zero.

In a clustered column-store such as Vertica, the size of the on-disk design is one of the parameters for predicting the maintenance cost. As a result, when dealing with a high update rate, we have to correctly account for its penalty to the overall user query set runtime (an adjusted problem formulation is discussed in Chapter 7).

As shown in Figure 5.6(a), a relatively large (in terms of disk space) MV such as MV_{1f} may still have a low maintenance cost. In contrast to MV_{1f}, MV_3 requires five times the maintenance cost, even though it is 30% smaller. A more accurate approach (compared with using disk size) to estimating the update cost in Vertica is explained in detail throughout the rest of this chapter. The example in Figure 5.6(a) exemplifies how unreliable disk-based estimates can be. It shows that MV_2 and MV_3 are approximately the same size and yet their relative maintenance costs differ by a factor of three.

The corresponding designs in DBMS-X exhibit much more predictable behavior. As Figure 5.6(b) shows, the disk size of each design is perfectly proportional to its insert cost. MV_{1f} is neither the largest MV in DBMS-X and nor does it impose an unexpected update cost. Note that, particularly when using large designs, the cost of an insert in a row store might suffer additional penalties if the buffer that caches the disk pages that have been modified begins to overflow. However, we assume that as long as the insert rate remains fixed, adding or removing MVs from a design changes the maintenance cost proportionally to the sizes of the MVs added or removed. However, as we will demonstrate, this assumption does not hold in Vertica.

5.3 Maintenance Cost Breakdown

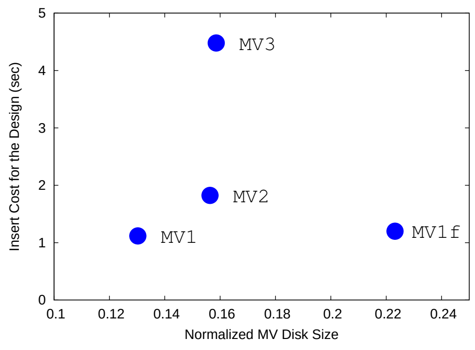
When evaluating a potential physical design, the automatic database design tool must consider the cost of the updates present in the query workload. Some database design tools accept an explicit set of SQL update queries as part of the query workload. However, to understand the overall update costs in Vertica, we need to list each individual item that affects the update cost. Some of these entries have an equivalent cost in a row-store setting. The overall update cost in a Vertica-like column-store DBMS is composed of the following four components:

- Cost of the insert: The immediate cost incurred when the insert is performed (as we will explain in Section 5.3.1, this is primarily the cost of the necessary joins employed by MVs).
- Cost of the move-out: The cost associated with dumping WOS buffers to disk when they are full. This cost is affected by the amount of available WOS memory, in-memory width of the MVs and the rate of insert (Section 5.3.2).
- Cost of merging: The cost imposed by the need to merge mini-ROSs formed by the move-out (see Section 5.3.3).
- Cost of the *SELECT*: The additional overhead incurred by *SELECT* queries when using a fragmented ROS (i.e., many mini-ROSs). Here, it is important highlight that this component does not directly contribute to the maintenance cost because it affects user queries rather than DBMS overheads (and the penalty does not apply if no one is using DBMS). However, this component is linked to a maintenance penalty because by delaying mini-ROS merges, we reduce the maintenance cost while penalizing *SELECT* queries (Section 5.3.4).

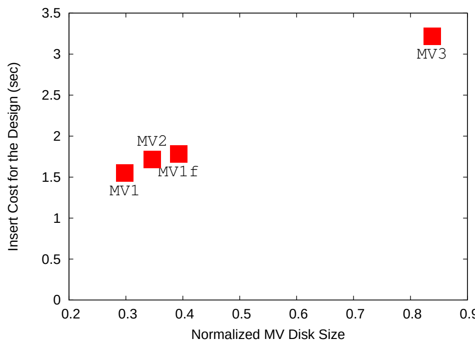
The first two bullets in this list have an approximate equivalent in a row-store; although their functionality is not identical, it can (arguably) be compared. The latter two bullets have no corresponding operation in a row-store because row-stores do not employ Vertica's double-buffering insert mechanism, which is described in Section 2.6.3. The following sections discuss each of these bullet points and draw parallels to a row-store setting when possible.

5.3.1 Cost of the Insert

The immediate cost incurred by the insert is the cost of incorporating the new data into the physical design. This operation requires that DBMS performs the necessary joins (for all pre-joined MVs, if any) and incorporates the new rows into the updated MVs. In Vertica, the new data is appended to the WOS buffer of the MVs in the memory; thus, the dominant



(a) Cost of insert in Vertica



(b) Cost of insert in DBMS-X

Figure 5.7: **Immediate Insert Cost vs. Disk Size** – Normalized performance of 10K inserts in Vertica and DBMS-X

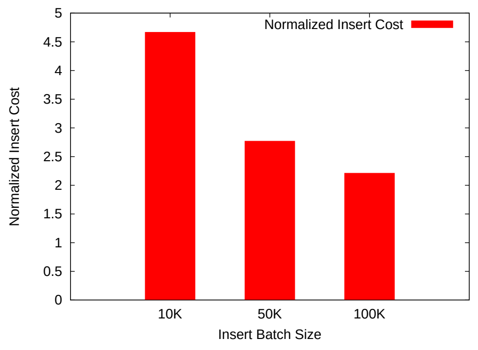
part of the insert cost is the cost to perform the required joins. Section 2.6.3 and Figure 2.7 outlined the update architecture.

As previously, we use the MVs defined at the beginning of this chapter. Unless otherwise noted, the experiments assume batches of 10K rows (we also perform at least three independent measurements, usually more, and remove outliers to achieve stable results). Figure 5.7 shows that the immediate cost of the insert is not proportional to the on-disk size of the MV. For example, in Figure 5.7(a), Vertica's performance shows that MV_3 is more than four times more expensive than is MV_{1f} when performing an insert of 10K rows, despite the fact that MV_{1f} is 25% larger than MV_3 is. By constrast, Figure 5.7(b) shows the linear relationship between the size of the equivalent MVs in DBMS-X and the cost of inserting 10K row batch. Note that although Figures 5.7 and 5.6 look similar, they are not the same.

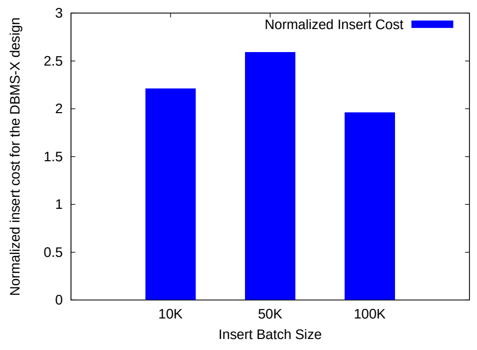
Figure 5.6 shows the total cost of inserting a row batch. Meanwhile, the performance disparity in Figure 5.7(a) is caused by the cost of the join, which does not depend on the size of the MV. As a result, the highest insert cost is MV_3, despite it being relatively small. This high insert cost is imposed by the required joins (making it 3–4 times as expensive as inserts into other MVs that perform fewer joins). In contrast to MV_3, MV_{1f} requires no joins and, therefore, cheaper to maintain, despite being the largest MV. By constrast, in DBMS-X the insert cost is directly proportional to the size of the MV. Note that the largest MV in DBMS-X is actually MV_3 and not MV_{1f}, as is the case in Vertica.

Inserting new rows one by one is not cost-effective and is unlikely to happen in a data warehouse; inserts are always batched to amortize the per row cost. The amount of insert batching depends on the particular application. Here, we only discuss the trade-offs involved, whereas in Section 7.6 we demonstrate the net effect of batched inserts on overall design performance. Our assumption is that in addition to the expected insert rate (rows/sec; unless otherwise specified, we will assume 1000 rows/sec), the user also specifies the anticipated level of batching. Figure 5.8 shows the insert overheads for a design containing all four example MVs. As before, the insert cost is normalized by the insert cost of the default design (just the base tables). We vary the insert batch between the a 10K batch (10 seconds worth of inserts at 1000 rows/sec) and the a 100K batch (16 minutes and 40 seconds worth of data). We observe that different batching produces different relative performances in Vertica compared with those in DBMS-X. In Vertica, Figure 5.8(a) shows that increasing the batch size reduces the average insert cost because a significant proportion of the cost is shared when a larger insert batch is executed. By constrast, in DBMS-X, Figure 5.8(b) demonstrates that these savings are insignificant because the amount of shared work is negligible.

The difference in performance is based on the cost composition. In both DBMSs, the MV pre-join is performed once for every batch and thereby amortized by the number of rows in the batch. However, in DBMS-X this cost is dwarfed by the cost of updating the clustering index of the MV. The inserted rows are likely to touch different disk pages (the same issue was described in Section 2.5.1). Therefore, in DBMS-X the cost of updating the indexes is roughly proportional to the amount of inserted data; the cost of the join is virtually negligible in comparison. In contrast to DBMS-X, Vertica does not need to update MV sort orders at the insert time. The cost of maintaining the MV sort orders is divided between the cost to *move-out* the data to disk and the cost to merge mini-ROS fragments. Thus, the cost of the pre-join comprises most of the total insert cost, and the insert cost can be reduced by increasing the batch size.



(a) Cost of insert in Vertica



(b) Cost of insert in DBMS-X

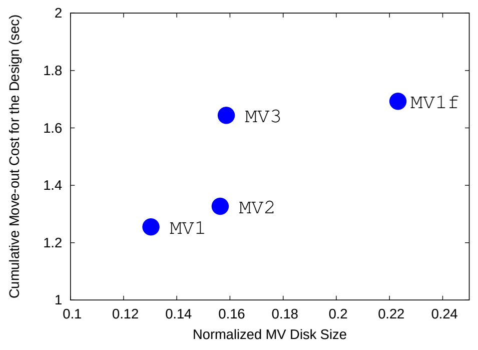
Figure 5.8: **Cost of Insert vs. Batching** – Insert Overhead in Vertica and DBMS-X

Next, we discuss the proportion of system resources spent on maintenance. Note that the double-buffering mechanism in Vertica (see Section 2.6.3) significantly reduces the cost of inserts; as a result, Vertica is able to sustain very high insert rates. In our example, DBMS-X is not able to sustain an insert rate of 1000 rows/sec; thus, we only present Vertica's throughput costs here. Table 5.1 shows the expected amount of system resources (as a percentage of the total time between inserts) required to sustain inserts at different batching levels. The designs are shown in the left column of Table 5.1.

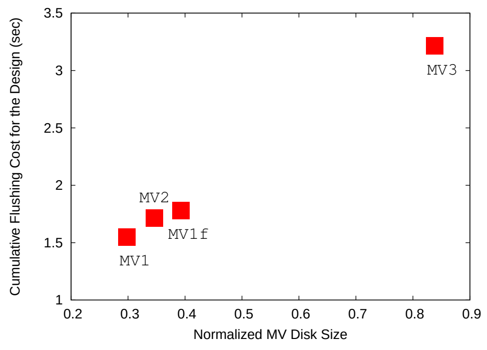
Design Batch Size	10K	50K	100K
$[MV_1, MV_2, MV_3, MV_{1f}]$	18.2%	4.3%	2.3%
$[MV_2, MV_3, MV_{1f}]$	17.9%	4.0%	2.2%
$[MV_1, MV_2, MV_{1f}]$	9.6%	2.3%	1.2%

Table 5.1: **Maintenance Required for Inserts** – The percentage of system resources required at different batching levels

As expected, a higher batching level reduces the insert overheads; by moving between 10K and 100K batching, the insert cost is reduced by a factor of 7–8. Hence, from the expected insert rate and the amount of insert batching, we can estimate the insert cost. In addition to the benefit of insert batching, we can also observe the *incremental* maintenance cost of adding an MV into the design, as shown in Table 5.1. The second row in the table contains the same design as that in the first row, except for the absence of MV$_1$. Further, the insert overheads are essentially the same for the two rows (with only a difference of



(a) Cost of move-out in Vertica



(b) Cost of flushing in DBMS-X

Figure 5.9: **Move-out Cost vs. Disk Size** – Normalized performance of 100K move-out in Vertica and buffer flushing in DBMS-X

approximately 5%). The third row in the table contains three MV design, but here MV_3 is the missing MV. In this case, the difference between the maintenance costs (line one and line three) is changed by a factor of two instead. This discrepancy is caused by the underlying joins; removing MV_1 does not change the overall pre-join required to update the design, while removing MV_3 eliminates some of the joins from the insert cost. In Section 5.4, we discuss estimating this cost, and in Section 7.6 we present an insert-aware physical design.

5.3.2 Cost of Creating Mini-ROSs

As the WOS buffer fills up, we need to *move out* its contents to free up space in the memory. This operation consists of sorting the MV rows in WOS and writing them out to disk. Each such on-disk snapshot is called a *mini-ROS* (see Section 2.6.3), which uses the sorting and encoding used of the MV. Every move-out operation on an MV creates a single mini-ROS. Unlike the insert cost discussed in the previous section, the cost of a move-out is typically proportional to the size of the MV.

Once again, we compare single-MV designs using the example MVs introduced at the beginning of the chapter. Note that DBMS-X does not have move-out functionality; however, we can measure the performance of a similar operation. DBMS-X permits us to issue a manual buffer flush that forces all cached pages (i.e., modified but not yet written back to disk) to be written to disk. We also use the same command to measure *cold-start* (i.e., repeatable query that cannot randomly benefit from page caching) runtimes in our experiments. Figure 5.9 shows that the buffer flushing costs of DBMS-X are, once again, proportional to the size of the MV. Vertica exhibits a slightly different behavior, although the costs are more linear than are those shown in Figure 5.7(a). To confirm, in Vertica the cost of the move-out operations is a sum of the sorting cost (which depends on the sort order) and the cost of writing the mini-ROS to disk (which is based on the size of the mini-ROS). Therefore, the shape of the move-out curve is closely connected to the size of the underlying MV. However, there is a noticeable difference between MV_2 and MV_3 because despite being similar in size, MV_3 is wider (and has string columns in the sort order), MV_3 is also more expensive to move out.

5.3.3 Cost of Merging Mini-ROSs

As discussed in the previous section, Vertica buffers the inserts in memory to amortize the insert costs. Caching or buffering is frequently used to improve the performance of DBMS, although typically at the disk page level. Furthermore, it is also worth noting that MyISAM [MyI] storage engine for MySQL implements an in-memory buffer in a manner similar to Vertica's approach. MyISAM delays the inserts to perform inserts in bulk. However, to our knowledge, no DBMS amortizes the insert cost by creating separate disk fragments (mini-ROSs in Vertica). As discussed in Section 2.6.3, this mechanism provides double buffering. First, data is buffered in the WOS and then the contents of the WOS are written to disk as a mini-ROS, further delaying the need to merge the new data with the old. In this section, we discuss the costs associated with performing this mini-ROS merge, namely then the mini-ROS fragments are merged into larger fragments to reduce MV fragmentation.

It should be noted that these costs are unique to Vertica; without a similar buffering mechanism, DBMS-X incurs no such costs. Thus, to model the cost of merging the mini-ROSs and the associated *SELECT* query penalty (see Section 5.3.4), we need to assume an existing mini-ROS merging policy.

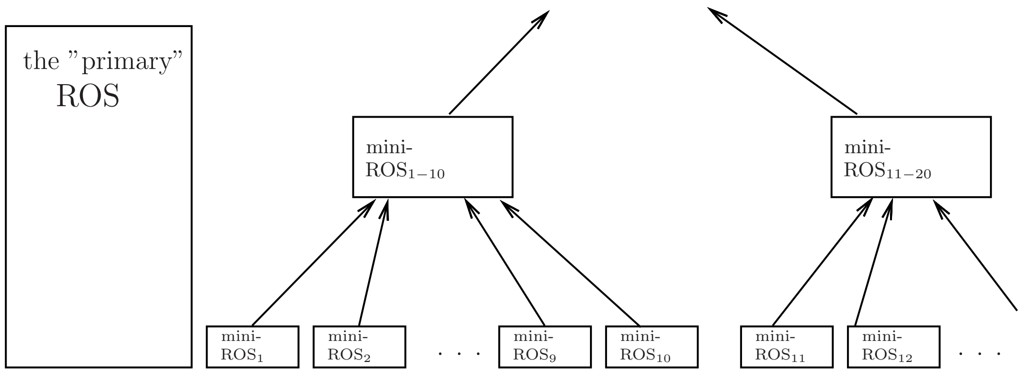


Figure 5.10: **Mini-ROS Merging Policy** – A Fixed Hierarchical Merging, Assuming 10 mini-ROSs per Merge Pass

For now, we assume that a constant hierarchical merging model will be used. In general, the selected merging policy has to balance the costs of performing data merges or leaving MVs fragmented and increasing the *SELECT* query penalty. We also assume the existence of a default policy, which merges whenever 10 mini-ROSs of the same size (i.e., same number of rows) are formed. Every merge produces a mini-ROS with 10 times the rows (Figure 5.10).

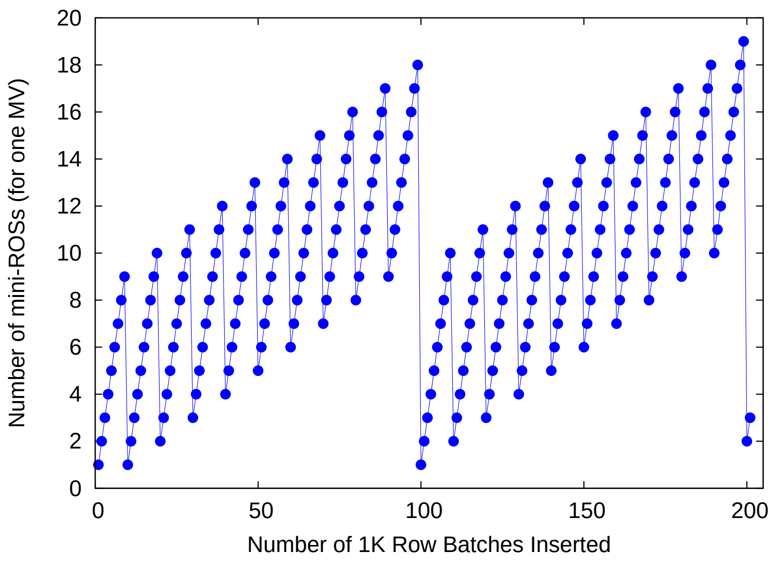


Figure 5.11: **Mini-ROS Count** – The Expected Number of mini-ROSs, Assuming a 10-merge Strategy

Figure 5.11 shows the per MV number of mini-ROS fragments for our assumed policy. After the 10th mini-ROS is created, the mini-ROSs are merged into a single larger mini-ROS. After the 20th mini-ROS is created, another 10 mini-ROSs are merged, and the MV contains two merged mini-ROSs. This process continues until 10 merged mini-ROSs are created and merged into an even larger single mini-ROS.

The cost of merging mini-ROSs is proportional to the amount of data merged. Merging 10 mini-ROSs requires reading all the mini-ROSs, merging them and writing the newly merged mini-ROS back to disk. No data sorting is required because the mini-ROSs have already all been sorted (thus, we can use merge-sort algorithm [SKS02]).

5.3.4 Mini-ROSs and *SELECT* queries

The mini-ROS fragmentation described in Section 5.3.3 affects the performances of all *SELECT* queries. As discussed earlier, every *SELECT* query has to read all the mini-ROS fragments of the MV it is using. Thus, the mini-ROS merging policy (which determines the amount of mini-ROS fragments for an MV) will determine the current penalty imposed on *SELECT* queries. See Figure 5.12 for an overview of Query$_i$ accessing MV$_3$.

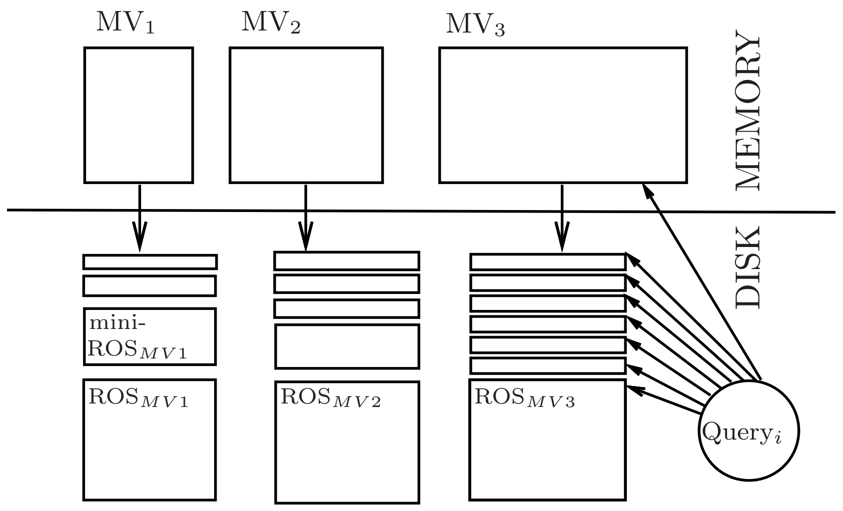


Figure 5.12: **The WOS–ROS Split** – The Contents of the WOS are Moved to Disk, then Mini-ROSs are Merged and Accessed by *SELECT* queries

Every new MV added into the design increases the pressure on the WOS buffer and, thus, on the frequency of the move-out (this will be further discussed in Section 5.3.5). The increased frequency of the move-out, in turn, affects the number of mini-ROSs for every MV. Consider MV_1, defined earlier in this chapter, and a simple read query that reads two of MV_1's columns. We vary the frequency of the move-out to determine the expected penalty associated with the different sized mini-ROSs. We also assume a constant hierarchical merging policy (i.e., 10 at a time), as described in Section 5.3.3. We then estimate the performance of the *SELECT* query, incorporating the penalty associated with different frequencies of move-out; the three available options are to make initial mini-ROSs with 1K, 10K or 100K rows. Figure 5.13 shows the expected performance of the query as mini-ROS merges occur in the background (the merging cost is ignored here). The insert rate is fixed; thus, for a 100K mini-ROS move-out, the *SELECT* query has no additional penalties until the first mini-ROS is moved out to disk. The cost of accessing the data in WOS is negligible (Figure 5.12). The query runtime shown in Figure 5.13 corresponds to the mini-ROS count (shown in Figure 5.11). The runtime spikes correspond to the accumulated mini-ROS fragments, and every drop corresponds to a mini-ROS merge. Note that the query runtime in Figure 5.13 is normalized (i.e., the runtime starts at one).

It should be noted that the x-axis in Figure 5.13 represents the number of inserted 1K batches. The exact time-line depends on the insert rate (e.g., 1000 rows could be inserted every second, as we have assumed here, or every five seconds). The query curve is an abstract estimate computed without knowing the insert rate. Next, we discuss the relationship between different designs and the memory budget.

5.3.5 WOS Memory Resource

In this section, we discuss the relationship between the WOS budget and the penalties discussed in Sections 5.3.2 and 5.3.3. Recall that in DBMS-X the size of the design is proportional to the size of the MV and thereby this can be used to approximate the costs of inserting new rows. In the absence of a more complex buffering mechanism, the insert

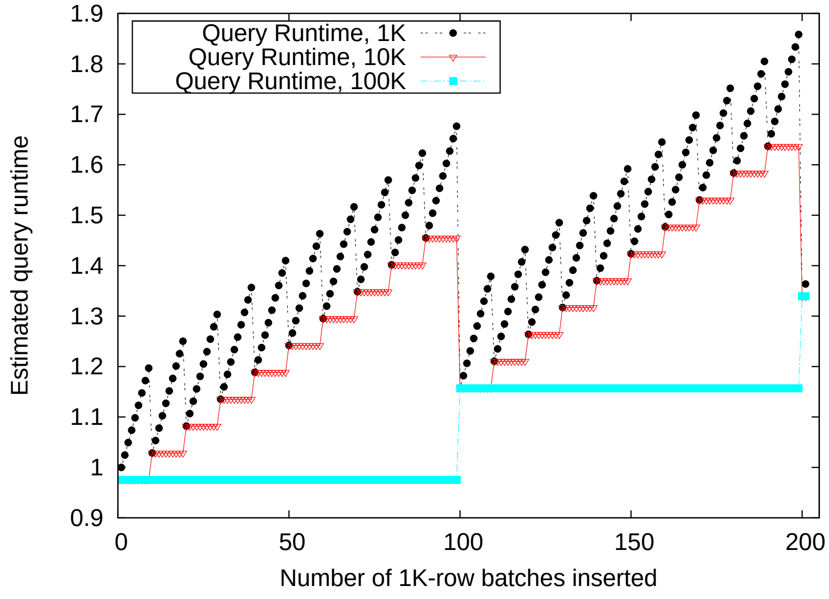


Figure 5.13: **Query Cost with Varying Mini-ROSs** – Penalties Imposed by Mini-ROSs in Vertica

penalties in DBMS-X depend on the cost of the insert and the additional maintenance required to move the contents of the memory buffer to disk. There is no merging maintenance cost in a row-store. Moreover, *SELECT* queries can benefit (but are never penalized) from inserts; in a row-store, executing inserts caches some of the pages in the memory, potentially reducing the cost of a query read. Increasing the stress on the memory buffer in a row-store increases the maintenance costs, but these costs are proportional to the number of pages affected, which, in turn, is proportional to the amount of data in the design.

The following example explains how how number and width of MVs present in the design affect the usage of the WOS buffer, and thereby influence the resulting overheads in Vertica. Unless otherwise noted, we assume a WOS buffer of 500 MB in this example. This is a realistic number; on a production server we might expect WOS to have a RAM allotment of 1000 MB or more. However, K-safety replication (see Section 2.6.4) reduces this buffer by a factor of K. We also still assume a default average insert rate of 1000 rows/second. Although not all data warehouses have a high update rate, some modern applications may require a higher insert rate. In particular, the LSST database [LSS] produces an estimated

20 TB of new data every night. When designing for a data warehouse with a low update rate, the design tool may ignore the update overheads discussed in this chapter.

The frequency of the move-out that is required to keep up with the insert rate depends on the memory footprint of the design (i.e., set of MVs) and the WOS budget. Given a 500 MB WOS budget, the move-out will initiate every time buffer use approaches 500 MB. Based on the SSB benchmark, the smallest amount of data inserted for each new row is approximately 100 bytes (the in-memory width of the fact SSB table, *lineorder*). Therefore, the 500 MB WOS can hold up to 83 minutes (5000 seconds) of data when using a default design (i.e., with no additional MVs). The following example demonstrates that the move-out frequency increases based on the square of the number of MVs in the design.

For example, suppose that our design contains 10 different MVs (of 100 bytes each); thus, for a total of 1000 bytes inserted for every new row. As a result, each newly inserted row requires 1000 bytes in the WOS buffer (instead of the 100 bytes in the default design). Every inserted row is duplicated for each MV in the design, effectively resulting in 10 inserted rows (one for each MV). Under these conditions, the same 500 MB WOS can now hold approximately 8 minutes (500 seconds) of data before it is full. Note that each MV in the design is moved out independently. Therefore, assuming that the move-out of each MV occurs uniformly, the move-out of an MV will occur (on average) every 50 seconds (10 MVs over 500 seconds).

Figure 5.14 shows the associated overhead costs normalized by the smallest design. The x-axis shows the average design width. For example, 1K on this axis implies that the combined total width of all MVs is approximately 1000 bytes; thus, they average 100 bytes for the 10-MV design and 50 bytes for the 20-MV design. The smallest design is 1K and 10 MVs. We only plot the move-out overheads (the third entry in components listed at the beginning of Section 5.3). As shown in Figure 5.14, increasing either the number of MVs or their width increases the move-out overheads. Doubling the number of MVs (from 10 to 20) increases the move-out overheads by a factor of two, whereas doubling the memory budget of the design increases the overheads by a factor of 1.6–1.7. The second bar in Figure 5.14

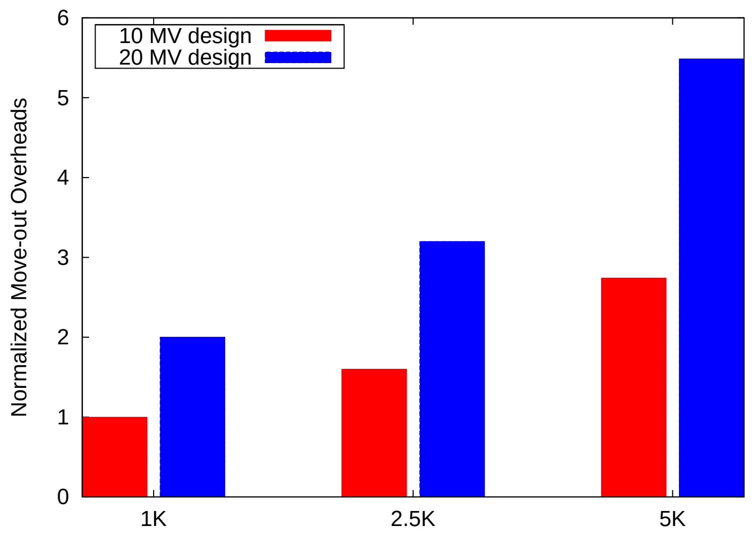


Figure 5.14: **The Cumulative Move-out Cost** – Move-out Overheads for Different Designs

contains narrower MVs (100 bytes vs. 50 bytes for an average width of 1K, etc.). Thus, if we were to start with a 10-MV, 1K design and add 10 more MVs to it, we would get a 20-MV, 2K design (each MV still has 100 bytes) Therefore, we can expect that doubling the number of MVs will quadruple the move-out overheads. The effect of doubling the design is twofold: we increase the number of MVs (move-outs have to occur twice as often) and we increase the pressure on the WOS buffer (the design is twice as wide).

5.3.6 WOS and ROS Budgets

The WOS buffer keeps newly inserted rows in the memory until they are moved out to disk in bulk. Recall that a design with multiple MVs often duplicates some attribute data with columns appearing in several MVs (potentially sorted by different clustered keys). Thus, for inserted rows some values appear in multiple locations, one for each WOS MV buffer (see Figure 5.12). As discussed in Section 5.3.5, the WOS buffer is a limited resource, but designs do not explicitly run out of WOS space as they might run out of disk space. Let's return to the example defined at the beginning of this chapter. The respective sizes of the MVs are DiskSize(MV_1) = 115 MB, DiskSize(MV_2) = 138 MB, DiskSize(MV_3) = 140 MB, and DiskSize(MV_{1f}) = 197 MB. Therefore, if the disk budget were 300 MB, any two of the first three example MVs or only MV_{1f} would fit into that budget. Ignoring the question of which is better (which depends on the queries), these are the *feasible* designs. As stated, when considering the WOS memory buffer, designs do not run out of memory as they might run out of disk space. Instead, each new MV places additional stress on the memory buffer, and this additional stress is proportional to both the number and width of the MVs in that design (Section 5.3.5).

An interesting observation is that the WOS cost is often disproportional to the disk size of the same MV because of the aggressive compression in Vertica. In our example MVs, each row of MV_3 occupies almost four times as much RAM as a row in MV_1 does and approximately twice as much RAM as a single row in MV_{1f} does. However, MV_3 is actually 40% smaller than is MV_{1f} with regard to disk size. This disparity is primarily from the use of RLE compression, which masks the differences between wide string attributes (such as *p_type* from the SSB benchmark) and simple integer attributes by compressing the data. However, dictionary compression can also produce a similar disparate effect, and, as explained in Section 5.1.2, LZO compression can produce varying compression rates, even in unsorted columns.

5.4 Cost Model

Before discussing our cost model, we should revisit our assumptions, particularly the properties inherent to OLAP. First, we rely on the relative simplicity of SQL queries. Second, we are not trying to build a new query optimizer (which is a well researched field [ML86, GM93, MLR03, MPK00] in itself) but rather model enough functionality to produce a good physical design in our setting. It would be possible to substitute another cost model into our framework.

The cost model presented here is analytic in that it estimates the *real* runtimes of the queries, or the wall-clock time which is the time from when the query is issued to when the result is returned to the user. It is important to measure the proportional cost of the query – we need to understand not only whether Q_1 is faster than Q_2 is, but *by how much* faster. Query optimizer cost models utilize a heuristic cost model, which measures relative runtimes (using internal cost units) [SKS02]. Such measurements do not always preserve the cost proportions between different estimates. But, next we argue why such models are sufficient for the purposes of a query optimizer even though they might fall short for the needs of a database design tool.

5.4.1 Building an Analytic Cost Model

It is well established that using the internal cost model incorporated in the query optimizer in DBMSs has a number of advantages [CN98a]. The most notable advantage is consistency; if the optimizer cost model does consider MV_i to be a good candidate, then DBMS will not use that MV, even if it actually is a good choice. Therefore, it pays to be only as accurate (no more, no less) as the built-in cost model, instead of developing an independent component. We hope that future DBMSs will include an analytic or hybrid cost model alongside the standard cost model that, in addition to making the same choices, would also model the underlying hardware (i.e., disk seeks and reads) to achieve a more accurate number for the database design phase.

Initially, we developed the cost model component because the Vertica prototype (at the time) did not have a *What-If* [CN98a] module available. However, we believe that there are important advantages inherent to using an analytic cost model like ours. This is not an argument for making such a cost model more precise or more sophisticated per se. Cost models can be arbitrarily complicated by introducing more and more parameters and variables that can affect the runtime of the query. Our argument is more fundamental. We believe that the optimizer cost model has a different goal when compared with a database design tool, and this goal is already being met by the typical cost models available in DBMSs. The goal of the optimizer is to *choose the best available MV for the query* or, in practice, to *avoid choosing a bad MV for the query.* In other words, when Q_a is issued by the user and MV_1 and MV_2 are available in the physical design, the optimizer needs to choose which MV will result in a faster Q_a runtime. As long as the query optimizer chooses MV_2 and that choice results in a faster runtime for Q_a, then the optimizer has done its job perfectly well. It does not benefit the optimizer to know *how much* faster MV_2 really is (e.g., it is irrelevant whether MV_1 is 10% slower or 10 times slower). Nor does it matter how accurate the cost estimate is as long as MV_2 is faster. What ultimately matters for an optimizer, is the relative *rankings* of the available MVs in the design. Moreover, in practice, the answer is sufficient even if MV_2 is, in fact, slower than MV_1 but not significantly so.

When creating a physical design, the accuracy of the query estimate becomes more important. Although estimating the MV cost (disk or memory) is not always precise, estimating the cost of the MV is easier and more accurate when compared with the task of estimating the query runtime. Although estimating the size of the MV is tricky compared with a clustered column-store (see Section 2.6.1), it is still simpler than correctly estimating the query runtime. For example, we may already know that that MV_2 is three times the size of MV_1. Once we know the relative sizes, it is crucial to know if Q_a will run twice as fast when using MV_2 or 10 times faster, because we already know that we will pay three times as much for MV_2.

Consider the following list of different MV candidates (the details of how they were created are in Chapter 6). These are all MV candidates for Query #4.3 from the SSB workload. Table 5.2 lists the normalized runtimes (normalized by the fastest candidate MV runtime in the respective column) using our cost model, and the available optimizer cost model along with the actual Vertica system runtime.

Let's begin by discussing the ordering of the candidates. All 15 candidates are assigned a unique ID that lets us compare their orders of preference if all 15 were available for query execution. In other words, each ordering column sorts the corresponding candidates by their cost model estimates. The "Actual" runtimes are the query runtimes measured in the Vertica DBMS, which is our optimum answer. Note that the optimizer cost model accurately estimates its ordering. Although it reorders #1 and #2 (there are also a few other

Our CM	Our Order	Actual Time	Actual Order	Opt CM	Opt Order
1	#11	1	#2	1	#1
1.008333	#8	1.163673	#1	128.4888	#2
1.038889	#10	1.203593	#5	229.7803	#5
1.061111	#7	1.343313	#3	330.8565	#10
1.080556	#6	1.403194	#6	457.1704	#3
1.333333	#5	1.46507	#10	1157.283	#6
1.336111	#4	1.532934	#11	1274.547	#11
1.372222	#3	1.882236	#4	1562.529	#4
1.422222	#2	2.147705	#7	1663.691	#7
1.575	#12	2.329341	#8	1937.516	#12
1.655556	#9	2.491018	#12	2191.897	#13
1.747222	#1	2.816367	#13	2607.296	#8
1.938889	#13	3.542914	#9	3270.26	#9
4.113889	#15	6.542914	#15	5333.184	#14
6.541667	#14	7.399202	#14	5548.161	#15

Table 5.2: **Measuring Query Runtime with Different Cost Models**

discrepancies), most queries are exactly where they should be or within two positions of the correct ("Actual") answer. Therefore, we conclude that a subset of these 15 candidates is available within the database, and that the query optimizer cost model is extremely accurate in choosing the best candidate for the query (which is its task). We also note that our cost model is somewhat less accurate at predicting the fastest candidate, particularly in terms of candidates #1 and #2, which are the best in reality and approximately 50% slower than the best in our estimates.

Now, let's observe the normalized cost estimates and consider how to select which MV candidate to add to the physical design. Naturally, each candidate has a specific size, and considering that space budget is limited, our aim is to select the most *beneficial* candidate, which will provide the highest query improvement per byte of disk space.

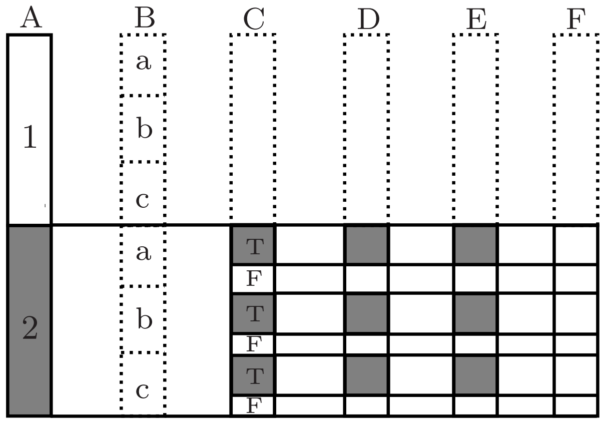
In such a case, the optimizer cost model is suddenly less informative. It suggests that candidate #1 is 128 times more expensive than #2 is (whereas, in reality, the difference is 16%). It also indicates that the improvement in query runtimes between the worst (#14 or #15) candidate and any of the top three is a factor of 25, 45, and 5500 respectively (even though the largest difference in real runtime between the best and worst candidate is approximately a factor of seven). Our cost model, by constrast, correctly predicts that all but the final two candidates are within a small factor of each other (we predict within a factor of two, when, in reality, they are within a factor of three). It also accurately predicts that the worst choice is approximately a factor of 6.5 (in reality a factor of 7.4) slower than is the best choice. Thus, once we consider that each candidate has a cost of X MB, our cost model, can select the one that provides the highest improvement per byte. If we had used the optimizer cost model, we might have been tempted to spend 5000 times the disk budget for candidate #1 instead of candidate #15, because it seems that candidate #1 is more than 5500 times faster than is candidate #15.

Numerous sources of inaccuracy can affect the estimates generated by the cost model. For example, query execution sharing or other forms of caching can speed up the query in an unpredictable manner. Moreover, it is not possible to model every parameter involved in query execution. However, we argue that the priorities and goals of the optimizer cost model and the database design tool cost model are different. The optimizer must correctly rank the available (i.e., already created in the DBMS) physical structures and select the best one. Whether the cost estimates are proportional to the actual query runtimes is irrelevant. By contrast, the database design tool needs to correctly estimate the relative benefits of adding an auxiliary structure, since it will be paying in terms of disk space for each chosen structure. The correct order (i.e., knowing which MV is faster if they offer similar performances) is less important than knowing the scale of the trade-offs. The database design tool cannot produce cost estimates that are incorrect by an order of magnitude, even if the ranking is correct because the MVs are selected during the design phase based on the relationship between the query improvement and MV cost.

5.4.2 The Cost Model and Disk Read Pattern

We conclude the chapter by discussing modeling of the I/O costs for reading an MV. The sort order of the MV determines both the amount of data that the query has to read and the amount of data that need to be processed (i.e., predicates applied, aggregate functions computed, joins performed, etc.). As shown throughout this chapter, the I/O cost and processing cost of the compressed data both play an important role in determining the overall query runtime. Thus, when evaluating the sort order selections, we first estimate the approximate read pattern, namely the sections of column data that are to be scanned. Then, we compute the expected cost for every column in the MV based on the required sequence of disk reads and seeks. Both these values (read and seek) can be determined experimentally or by using the hardware specifications of the disk.

Consider the example MV (A B C D E F | A B C), shown in Figure 5.15. Column A splits the MV rows into two *buckets*. Column B splits each of the two buckets into three buckets, resulting in six buckets. Finally, column C brings the total bucket count to 12. The figure demonstrates 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, then the combined cardinality and, thus, the number of buckets may be smaller.



```
Q:SELECT SUM(D), MAX(E)
  FROM MV
  WHERE A = 2,  AND C = 'T';
```

Figure 5.15: **Cost Model** – MV Access Calculation Example

Query predicates determine which of these 12 buckets need to be accessed (three in this case). Query Q in Figure 5.15 selects the second half of column A, which corresponds to the final six buckets. The predicate in column C selects three of these six buckets. We refer to the pattern of buckets to read as a *read pattern*. The read pattern determines the cost of accessing all subsequent columns such as columns D and E in Figure 5.15.

Once the read pattern has been determined, the cost of reading the subsequent columns accessed by query Q is a function of the size of the physical column on disk. Note that Vertica operates on large disk pages. Thus, in our example if column D takes four disk pages to store 12 similar sized buckets then each page would contain three buckets. Query Q would then have to read two of these disk pages (or 50%). However, the same column D with a size of 40 pages under the same assumptions would require reading approximately 10 pages (or 25%) with two seeks to relocate between buckets. In general, the cost model computes the cost of the sequence of reads and seeks as applied to each accessed column. Keep in mind that it is important to consider the seek time; in a column-store, the number of seeks needed to read the relevant part of each column is multiplied by the number of columns accessed by the query. Thus, seeks may become a dominant part of the overall cost. We use physical disk parameters (e.g., seek time, I/O time) to combine the read cost and seek cost into a single estimated time.

RB	Disk Read Pattern.
P	The size of column T (pages).
B	Bucket size in pages ($\frac{P}{\#Buckets}$).
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 pattern. (e.g. RUNS$_1$([1,1,0,0,0,1,1,1,1,0,0,1]) = (2,4,1)
DT	The set of dimension tables joined by an insert.

Table 5.3: Cost Model Variables

Table 5.3 lists the variables used to determine the cost of accessing column T, as expressed in Equation 5.2. For every consecutive run of 1s we estimate the amount of data that the query will scan. For every sequence of zeros, we compare the cost of seeking with the cost of reading the unneeded data and select the lower cost. Note that the seek cost is roughly an order of magnitude higher than the cost of reading a single page. For example, a read pattern that accesses one in every 10 pages will have a similar cost to a column scan.

$$ColumnCost(T) = \sum_{x \in RUNS_1(RB)} max(x * B, 1) * C_{read} \tag{5.1}$$

$$+ \sum_{y \in RUNS_0(RB)} min(y * B * C_{read}, C_{seek}) \tag{5.2}$$

The cost of a query Q is then:

$$Cost(Q) = \sum_{T \in Columns(Q)} ColumnCost(T) \tag{5.3}$$

We reiterate that inserts are handled by placing new tuples with the appropriate pre-join in the WOS. This requires accessing each dimension table in the pre-join. Thus, the cost of a single insert batch is equivalent to the cost of reading these dimension tables, as shown in Equation 5.4.

$$Cost(I_b) = \sum_{D \in DT} (\sum_{T \in Columns(D)} ColumnCost(T)) \tag{5.4}$$

Chapter 6

Building Materialized Views

In this chapter, we explain how materialized view (MV) candidates are generated. The most crucial step in the process of building the physical design is producing a suitable set of candidate MVs from which a good design can be built. MVs are designed to speed up queries; hence, this discussion only makes sense in the context of a query workload. We start by describing the issues involved in designing MVs for a single training query and then we discuss the concept of *shared* MVs (i.e., MVs that serve multiple queries). In addition to the example queries used throughout this chapter and the SSB benchmark queries, we use queries generated by a query generator (as described in Section 2.4.2). We supplement the discussion of each of the issues involved in designing good MVs in a clustered column-store with relevant examples and point out the significant distinctions (and similarities) compared with a row-store DBMS.

At the end of this chapter, we present our design algorithms. In the chapter that follows (Chapter 7), we show how our design tool selects subsets from the available MV candidates to form a comprehensive physical design.

6.1 One-query MVs

The starting point of this discussion is the design of a *dedicated* or a perfect MV for a single SQL query. Although this topic is not covered in detail in the physical design literature [CN98a, BC05], there can be many different MVs that trade off resource usage and query performance – even for a single query. Section 3.2 described the four major steps involved in the process of generating a physical design. This section elaborates on the first step, namely generating individual structures for a single query.

As explained in Section 4.2, MVs consist of two components: a pre-join and a sort order, as explicated below.

6.1.1 Choosing a Pre-join

The most intuitive choice for the pre-join is to join all the tables that the query needs; however, only the tables that are connected via a primary key–foreign key (PK–FK) relationship allow pre-joins to be computed (see Section 1.2.1). In a star-schema (or a snowflake-schema), all tables are connected by a PK–FK relationship, and thus we can easily identify the *anchor* table (intuitively, the "central" table to which all other tables are joined), which, subsequently determines the number of rows in the pre-joined MVs. We briefly discussed handling non-star-schema queries in Section 2.5.4. The benefit of a fully pre-joined MV is that the query using it will not need to perform joins, which are expensive operations and should avoided if possible. The presence of a join operator in the query plan also introduces uncertainty, since the resource allocation for join operations significantly affects query runtimes.

There are, however, still some reasons to consider a *partial* pre-join, (i.e., the pre-join of some but not all of the tables that the query accesses). For example, a query Q_i might join tables T_f, T_1, and T_2, where T_f is the fact (anchor) table. A fully pre-joined MV will join all three tables ($T_f \bowtie T_1 \bowtie T_2$, where $\bowtie$ denotes the join operation), whereas a partially pre-joined MV may contain any table subset (i.e., $T_f \bowtie T_1$, $T_f \bowtie T_2$) or even just T_f (no pre-join at all). A partial pre-join might be preferred because of the scale of the overheads that joins impose on the insert rate in the query workload. As discussed in Section 5.3.1, the cost of inserts depends on the cost of joins that they trigger on arrival. A partially pre-joined MV may also yield a superior MV candidate, either because it is slower but much less expensive (in terms of disk or memory resources) or, in some cases, because it is faster than the fully pre-joined MV is.

A partially pre-joined MV naturally uses less RAM in the WOS memory buffer (see Section 2.6.3), because it is usually narrower than is the fully pre-joined MV: when joining a dimension table D_i used by Q_1 to the fact table to get MV_i, MV_i includes all D_i columns used by Q_1. Alternatively, if we choose not to pre-join, $MV_{1NoJoin}$ only includes the FK for D_i. Substituting a single column for multiple columns naturally makes the projection narrower (of course, if Q_1 only touches a single column in D_i, then one column is traded for another).

In addition to the budget considerations of the WOS, there are two other, less intuitive, possibilities. First, a partially pre-joined MV may take up less disk space than does a fully pre-joined MV. In fact, in a row-store such as DBMS-X, it is universally true that a narrower MV always takes up less disk space. However, in Vertica, the ubiquitous presence of compression (Section 2.6.1) can push a narrower MV in either direction: a narrower, partially pre-joined MV may or may not be smaller than is a fully pre-joined one. Second, a partially pre-joined MV may, in some cases, result in a faster query runtime. Although joins are expensive operations, pre-joining all dimension tables can increase the I/O cost of the query. Pre-joining denormalizes the schema (the normalization of the schema eliminates duplicates in the tables as explained in Section 1.2.1). The result is determined by the particular pre-join, column compression, and sort order chosen for the MV (see also Section 6.1.2).

The number of different pre-joins available depends on the number of dimension tables used by the query. The number of all possible pre-join permutations is $2^{\#ofDimTables}$, a power set of the query dimension tables. Here we can use an example query (a variation on one of the SSB queries) to illustrate the variety of feasible MV candidates for one query. Note that choosing a sort order for the MV will be covered in Section 6.1.2.

```
SELECT d_year, s_city, p_brand1, SUM(lo_revenue)
FROM dwdate, customer, supplier, part, lineorder
WHERE lo_custkey = c_custkey
AND lo_suppkey = s_suppkey
AND lo_partkey = p_partkey
AND lo_orderdate = d_datekey
AND c_region = 'AMERICA'
AND s_nation = 'UNITED_STATES'
AND d_year = 1997
AND p_category = 'MFGR#14'
GROUP BY d_year, s_city, p_brand1;
```

We select the following four feasible MV candidates to evaluate the trade-offs between fully and partially pre-joined MVs (as in other examples, FKs are underlined):

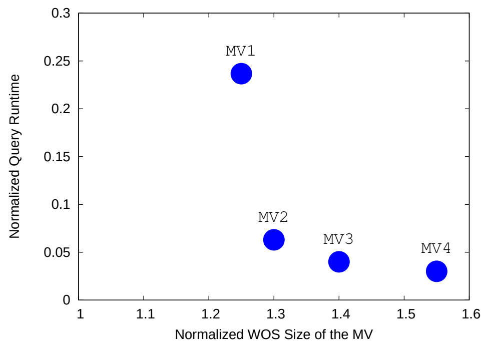
1. MV1: ($lo\_ordertotalprice$, $\underline{lo\_suppkey}$, $\underline{lo\_orderdate}$, $lo\_revenue$ | $lo\_ordertotalprice$)

2. MV2: ($d\_daynuminyear$, $lo\_ordertotalprice$, $\underline{lo\_suppkey}$, $lo\_revenue$ | $d\_daynuminyear$, $lo\_ordertotalprice$)

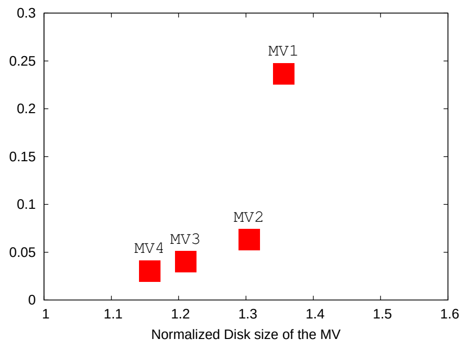
3. MV3: (s_region, $lo\_ordertotalprice$, $\underline{lo\_orderdate}$, $lo\_revenue$ | s_region, $lo\_ordertotalprice$)

4. MV4: (s_region, $d\_daynuminyear$, $lo\_ordertotalprice$, $lo\_revenue$ | s_region, $d\_daynuminyear$, $lo\_ordertotalprice$)

We will now assess the relative query runtime (i.e., speedup) provided by the example MV candidates above compared with the relative sizes of these MVs. All the values in Figure 6.1 are normalized by the default design values. As shown in Figure 6.1(b), the fully pre-joined MV (MV4) is, in fact, the smallest in terms of disk size because of the column compression in Vertica (we will later show cases where compression does not cause such unusual behavior). By contrast, to Figure 6.1(b), Figure 6.1(a) shows that the same MV4 is actually the largest in terms of WOS size (i.e., that is the widest MV of the four candidates). The more intuitive behavior in this case is because the data in the memory



(a) MV candidate performance compared with WOS (memory) size



(b) MV candidate performance compared with ROS (disk) size

Figure 6.1: **Resources vs. Runtimes for Pre-joins** – Normalized Query Runtimes in Vertica for Different MV Candidates

are not compressed. Importantly, such differences do not simply reflect the implementation choices in Vertica. As discussed in Section 2.6.1 and Section 5.1, column compression heavily relies on data clustering (i.e., sorting) as well as the immutability of the ROS files (see Section 2.6.3). The WOS buffer exists to make that possible, and it would be inefficient to keep the MV WOS data sorted as new rows arrive.

Overall, Figure 6.1(a) shows the typical diminishing returns curve (see Chapter 7) between additional memory and the resulting runtime. Figure 6.1(b) shows a less intuitive behavior, where wider MVs actually take up less disk space. MV1 produces the worst compression, because it contains two FKs (MV2 and MV3 contain one FK, and MV4 contains none). FKs typically compress very poorly because they have to supply enough values to reference every row in the target table (where they serve as PKs) and thus are many-valued.

6.1.2 Choosing an MV Sort Order

For any pre-join, we need to select a sort order to complete the definition of the MV. In the previous sections, we selected a sort order for all example MVs because otherwise they would be incomplete. Here, we describe the the methods for building a *dedicated* sort order for a single query, and then expand the discussion to cover generating sort order for multiple queries at once in Section 6.3.1. Recall that in a data warehouse setting, the query cost for accessing the MV is a combination of the I/O reading cost and some processing costs. Thus, the sort order should be designed to reduce both these components. First, we build a sort order *prefix* (i.e., the leading part of the sort order) to reduce the I/O cost component and then we extend the sort order in order to reduce the cost needed to process the remaining MV columns and improve column compression (Section 6.1.2). As a final step, we select an individual encoding for every column outside the sort order, choosing the most effective compression method for each column (see also Section 5.1.1).

Building the Sort Order Prefix

In any DBMS, the query engine may have to read extraneous values while executing a query. This occurs because the underlying physical storage has to read data in entire pages (a page size is often configurable, but it is at least 4 KB in DBMS-X and 64 KB in Vertica). In a clustered column store such as Vertica, a single disk page can contain tens of thousands of values when using the default LZO compression. This can be even more with other encoding techniques. The goal of minimizing the I/O cost is to avoid reading unnecessary values, and the goal of the database design tool is to create auxiliary structures and cluster them so that queries can avoid reading these unneeded data. The sort order prefix determines how many disk pages are read from the subsequent columns. We then select a good prefix by enumerating the promising combinations of columns and evaluating them by a mechanism similar to that of our cost model (described in Section 5.4).

The problem of finding a good sort order prefix in a clustered column-store is more challenging than the similar task of finding a good clustering index in a row store, owing to a number of factors including page size, compression type, column/row architecture and the selectivity of the query. In some cases, the right answer can be the same as that in a row-store. For example, if we query the total store revenue for *year* = 2001 (or any single year), the best clustering answer is the *year* column. However, for more complex cases (e.g. finding the revenue in some of the states and for multiple years), the answer becomes elusive in a clustered column-store. There are two main reasons why the answer remains trivial in a row-store:

1. When choosing a clustering index, predicate selectivity can guide the design tool towards the best answer.

2. The number of rows in the MV (i.e., the underlying table size) does not affect the *relative* performance of the auxiliary structure (index/MV).

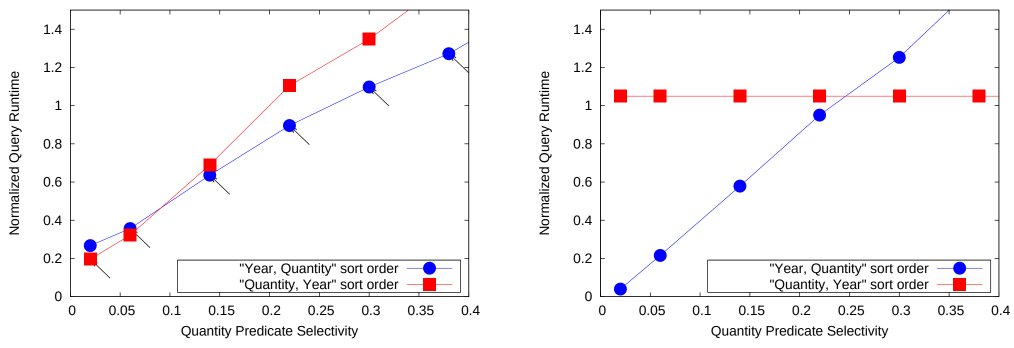
The runtime of a query in any DBMS will change as new rows are added to the underlying tables. However, in a row store, if any auxiliary structure S_a is chosen as the best one to speed up query Q, then S_a will remain the best choice even as the database grows or shrinks. In other words, the performance of all auxiliary structures changes in the same way in a row-store. By contrast, the best design in a clustered column store can change as the database size fluctuates (here, we assume significant size changes, a factor of two or more, rather than, say, 2%). Next, we will validate our claims through a number of examples, starting with claim #1.

When looking for a clustering index for an MV in a row-store, the accepted method is to rely on the selectivity-based ordering of query predicates [KHR+10, ACK+04]. In other words, the first attribute in the clustering key is that with the attribute with the lowest selectivity predicate (closest to zero), followed by the attribute with the second lowest selectivity and so on. The only exception to that rule is the equality predicate (A=c_1), which takes precedence over all other predicates. Equality predicates are still ordered according to their selectivity; thus, predicate selectivity is again decided by how clustering keys are built in a row-store. The following example query (which is a variation on one of the SSB queries) illustrates the difference:

```
SELECT SUM(lo_revenue), SUM(lo_extendedprice), SUM(lo_ordertotalprice)
FROM customer, dwdate, lineorder
WHERE customer.c_custkey = lineorder.lo_custkey
AND dwdate.d_datekey = lineorder.lo_orderdate
AND d_year > 1996
AND lo_quantity between [a] and [b];
```

There are two predicates in this example query: the *d_year* predicate has a fixed selectivity of approximately 0.24, and the quantity predicate range varies (by substituting constants instead of [a] and [b]) to achieve different selectivity values ranging between 0.02 (equality) and 0.40. For simplicity, we present the results using a fully pre-joined MV; thus, there are two sort order choices to consider: a clustering key (*d_year*, *lo_quantity*) or a clustering key (*lo_quantity*, *d_year*). Figure 6.2 shows the respective runtimes of the example query, and each runtime is normalized by the average query runtime taken at the selectivity intersection point. Figure 6.2(b) shows, as expected, that the performance of DBMS-X confirms the selectivity ordering rule: our example query runs faster using a clustering key leading with the column *d_year* as long as the corresponding *d_year* predicate has a lower selectivity. Once the selectivity of the *d_year* predicate is higher than that of the *lo_quantity*, the clustering key that leads with *lo_quantity* becomes a superior choice. Note that the performance of the (*lo_quantity*, *d_year*) clustering key remains flat at low selectivities of *lo_quantity*, because DBMS-X, as would be the case for any other row store, is limited in its ability to utilize composite clustering keys. If we were able to use techniques such as bitmap indexing [bit, Orac] or skip-scan [Orab], we could expect to see better performances as *lo_quantity* selectivity decreases. However, that would not alter the fact that the performance intersection point matches the selectivity intersection point in a row store.

In contrast to DBMS-X, Vertica exhibits a different query performance in Figure 6.2(a). The intersection of the two lines is nowhere near the conventional 0.24 selectivity; instead, the lines intersect around a selectivity of 0.06. By looking closely at the attributes of the sort order, we see that the attribute *cardinality* and the particular disk read pattern (see Section 5.4.2) outweigh the normal selectivity considerations when determining the relative query runtimes. In Figure 6.2(a), the intersection occurs at the point where the *lo_quantity* predicate selects a range of three values (with a selectivity of 0.06). This is the crossover point where the cost of extra disk seeks introduced by leading the sort order with *lo_quantity* negates the benefits of lower selectivity. Selecting a wider range of *lo_quantity* values further increases the gap in performance until it reaches approximately 30% at the



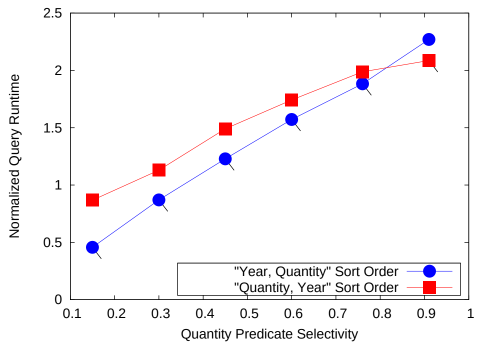
(a) Vertica sort order performance (b) DBMS-X clustered key performance

Figure 6.2: **Predicate Selectivity and Performance** – *d_year* Predicate Fixed at a Selectivity of 0.24

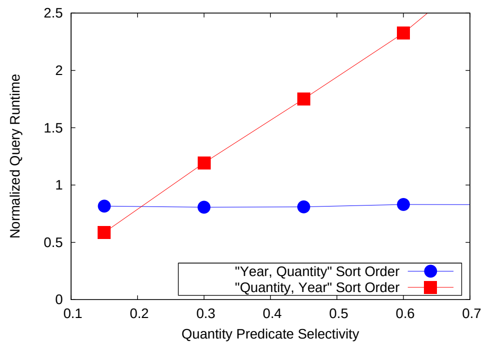
point of conventional intersection at a selectivity of 0.24. The arrows in Figure 6.2(a) mark the choices made by our cost model (Section 5.4) in this case and thereby confirm its accuracy.

To further clarify our argument, we use the same example query but fix the selectivity of the *lo_quantity* predicate instead. The *lo_quantity* selectivity is set at 0.22, and instead vary the selectivity of the *d_year* predicate in a similar manner. Figure 6.3 shows the normalized runtime achieved by our example query. Once again Figure 6.3(b) shows that the performance in DBMS-X intersects when the selectivities match (approximately 0.22), thereby confirming that conventional row-store DBMS wisdom results in the correct choice. As before, the (*d_year*, *lo_quantity*) clustering key remains flat instead of improving at the lowest selectivity of the secondary clustering column (*lo_quantity*). In this case, the Vertica crossover point in Figure 6.3(a) is close to a selectivity of 0.82 instead of 0.09 as it was in the previous example (while the selectivity crossover remains at 0.22 in both cases). The fragmentation caused by leading the sort order with *lo_quantity*, pushes the intersection point far to the right in this experiment. The arrows in Figure 6.3(a) once again show the (accurate) choices made by our cost model.

This demonstrates that even in a simple case with only two sort order choices, the crossover point is independent of where the selectivities cross. Although the intersection



(a) Vertica Sort Order Performance



(b) DBMS-X Clustered Key Performance

Figure 6.3: **Predicate Selectivity and Performance** – *lo_discount* Predicate Fixed at a 0.22 Selectivity

points may match, in a clustered column store (unlike in a row store), this is purely coincidental. To find the right answer in Vertica, we need to rely on our cost model, which takes other factors besides selectivity into account to provide the correct answer.

The intuition behind the observed disparity between Figures 6.3 and 6.2 is illustrated in Figure 6.4. In a clustered column store, the sort order prefix translates into a read pattern that depends both on the cardinality of the attributes and on the size of the column accessed by the query (in terms of pages). The overall cost of accessing the MV is a combination of reads and scans induced by the sort order. The final cost depends on the number of seeks and the reads that actually occur, which does not match the predicate selectivity when the query is forced to read extraneous values.

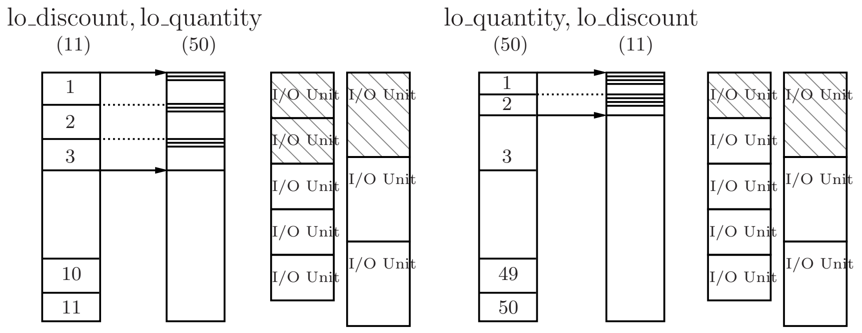


Figure 6.4: **Prefix I/O Pattern** – The Difference between Prefixes Caused by Sort Order Fragmentation

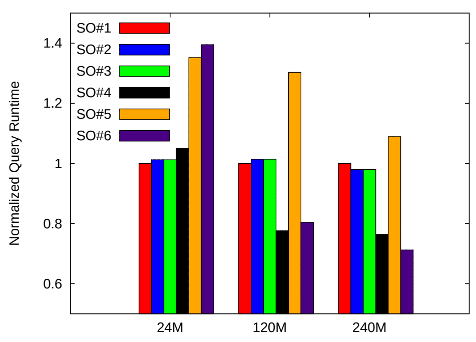
We now turn to the second statement made at the beginning of this section, which ties in to the illustration in Figure 6.4. The fragmentation induced by the sort order determines the cost; however, the amount of fragmentation (i.e., the number of pages read) depends on the column size as well as the sort order. Therefore, the size of the target column is a factor in determining the quality of the sort order. This feature is unique to a Vertica-like database. By contrast, in DBMS-X, once the best clustering index has been found (which can be easily done using predicate selectivity), it will remain the best choice no matter how much the MV size changes. To illustrate this point, consider the following example query:

```
SELECT SUM(lo_revenue)
FROM dwdate, lineorder
WHERE lo_orderdate = d_datekey
  AND d_DayNumInYear > 240
  AND lo_quantity > 24
  AND lo_discount > 2;
```

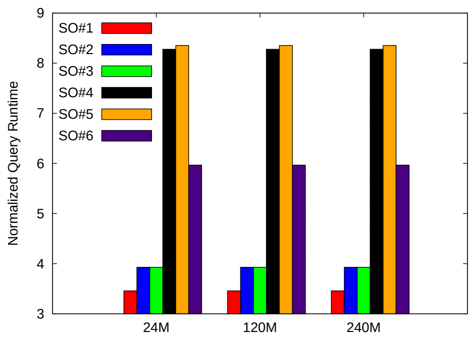
The cardinalities of *d_DayNumInYear*, *lo_quantity*, and *lo_discount* are 365, 50 and 11, while the selectivities are 0.3, 0.5, and 0.7, respectively. In this example, we generate six different fully pre-joined MVs that represent all feasible sort orders using a full pre-join. We then run the example against each MV while varying the SSB scale-X (# of rows) of the dataset. All six possible sort orders (the combinations of three predicated attributes) are enumerated below:

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 6.5 are normalized with respect to the MV that uses SO#1, the sort order based on predicate selectivity (i.e., the best choice for the row-store DBMS). Figure 6.5(b), once again, matches our expectations. SO#1 is indeed the best choice for DBMS-X because *d_DayNumInYear* performs best when it is in the first position in the MV clustering key owing to its lowest selectivity among query predicates. The number of rows



(a) Vertica Sort Order Merges



(b) DBMS-X Clustering Index Merges

Figure 6.5: **Different Index Merges** – Query Runtimes for Different SSB Scales

in the MV does not affect the performance of DBMS-X in the slightest (since the results are normalized). Next, we consider the query performance in Vertica in Figure 6.5(a). Here, the number of rows in the MV or, to be more precise, the size of the target *lo_revenue* column determines the best answer. At 24 million rows, *d_DayNumInYear* is actually a good leading choice, as in DBMS-X (sort orders #1 and #2 in Figure 6.5(a) are fast). However, as the number of rows increases, other sort orders begin to dominate. At 120 million rows, SO#3 and SO#5 are ahead by 20%, and by the time we reach 240 million rows, SO#6 is the best choice and SO#1 and SO#2 are the worst. A difference of 20-30% may not seem dramatic, but keep in mind that in this case every column except for the revenue column is RLE compressed. If the number of predicates were larger, some of them might not appear in the RLE portion of the sort order, thereby increasing the the gap between the runtimes.

Figure 6.6 shows how using the *skip-scan* technique (available in some row-store DBMSs [Orab]) in a row store causes DBMS-X to behave more like Vertica, thereby potentially benefitting from our MV generation approach. The skip-scan mechanism is a way to query an index with more than one range predicate in a row-store. Intuitively, this is performed by replacing the leading range predicate with a number of equality queries. For example, a query engine might explicitly use a predicate "(A=5 AND B>10) OR (A=6 AND B>10)" instead of "5≤A≤6 AND B>10." The bars on the left of Figure 6.6 show the same DBMS-X performance as that in Figure 6.5 and use the same sort orders as presented in the

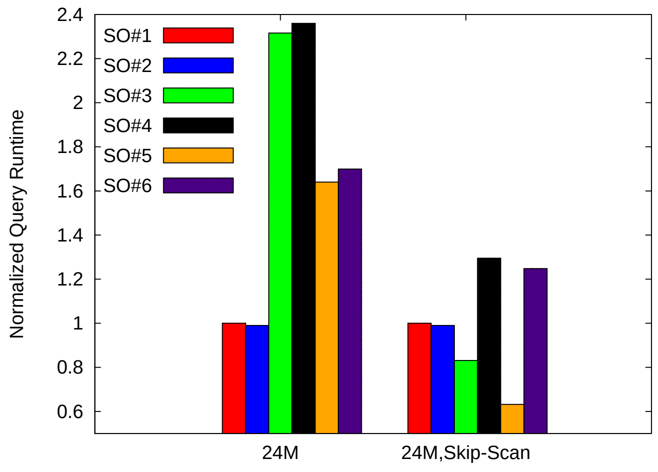


Figure 6.6: Covering Indexes with and without Skip-scan in DBMS-X

previous example. The bars on the right show the performance associated with the skip-scan technique on the same MVs. Although DBMS-X does not support skip-scan, we were able to manually rewrite the queries to achieve the same result. The observed improvement in performance on the right of Figure 6.6 is owing to the ability of skip-scan to take advantage of individual columns with the clustering key. The relative query performance approaches that of Vertica in Figure 6.5(a), meaning that a row store that supports such a technique may benefit from our design techniques.

Next, we discuss enumerating and evaluating the potential sort orders for an MV. To ascertain the size of the MV, we need to approximate the size of the target columns accessed by the query by making certain assumptions (such as assuming best-case compression and ignoring possible correlations between the sort order and target columns). Unfortunately, we cannot estimate the exact size of the MV before we have chosen a sort order. As demonstrated earlier in this chapter, the size of the target column affects the fragmentation induced by the sort order prefix. As discussed in Section 5.4, the prefix pattern translates into a disk read pattern, which, in turn, is applied to every accessed column. Having estimated the expected column size, we can then evaluate the cost of the disk read pattern of the candidate prefixes.

In order to enumerate the set of candidate prefixes, we exhaustively permute the predicate attributes in the query. We only consider prefixes up to a certain depth (to make this approach more tractable) based on the sizes of the target columns. Since the I/O read unit is relatively large (at least 64 KB, although it can be even larger owing to pre-fetching by the hard drive), even a large number of values in a column would produce a relatively small number of pages to be read. For example, the default scale-4 SSB dataset contains 24 million rows, and for the typical target column, we only need to consider sort order prefixes with a combined cardinality of up to 100. Thus, most candidate prefixes contain no more than two or three attributes. In addition, we apply filtering rules that eliminate some of the prefix candidates from consideration. We define the following filtering rule: attribute A supersedes attribute B if and only if: $Cardinality(A) < Cardinality(B)$ and $Selectivity(A) < Selectivity(B)$. Intuitively, if attribute A simultaneously has a lower selectivity (closest to zero) and lower cardinality than does attribute B, then it is a better choice than B and the superseded attribute is not considered.

Once we have generated a candidate set of prefixes, we evaluate the I/O cost of each prefix and select the best one. To estimate the I/O cost of a prefix, we compute the read pattern based on the predicates in the query. We do that by estimating which values will be chosen by the predicates. There are various ways to estimate the I/O cost of the prefix, balancing cost and accuracy of the estimate (and we have implemented all of them). The simplest approach is to assume that all attributes are independent and have a uniform distribution: thus, a prefix (d_year, $lo\_quantity$) would have $Cardinality(d\_year) \times Cardinality(lo\_quantity) = 77$ unique values, with each value corresponding to $\frac{1}{77}$th range of all rows. A more precise approach would be to use data histograms and correlation statistics (which have already been collected for estimating compression). Finally, it is possible to use a data sample to sort and evaluate the contents of each prefix to achieve the best accuracy. The prefix attributes have a small composite cardinality and thereby the computation based on the sample is sufficiently accurate. Our current prototype uses the latter approach.

To evaluate the approximate cost of the prefix, we convert the estimated sequence of reads and seeks into a time estimate by substituting the average time it takes to perform these operations on the hard drive. These values can be taken from the hard drive specifications (or initially estimated by the design tool, as is currently done in our prototype).

Once the best I/O prefix has been chosen, the approximate ranges of data that need to be read from the MV are nearly fixed (they can only undergo minor changes because of additional column compression). However, it is still important to correctly extend the sort order prefix with additional columns to improve the query-processing time.

Building the Sort Order Suffix

Although the sort prefix has already been selected, we can still reduce the cost of processing the remaining columns by extending the prefix. We build the *suffix* (i.e., the "rest" of the full sort order), by extending it to its maximum possible length. Processing column values includes several different operations that the query engine may apply to every value. These include the application of predicates, aggregates or joins. However, the overall query selectivity (i.e., the number of rows returned to the user) is fixed; thus, once the query finishes executing, every predicate has been applied. By choosing the RLE-ed sort order columns, we only alter the order and the time taken by query predicate execution. As we will show, choosing the best suffix is a trade-off between maximizing compression (all the columns in the sort order are RLE-compressed) and accelerating column processing, since Vertica is able to operate directly on compressed data.

As always, we must consider whether the row-store answer applies to our column-store setting. Adding predicates based on predicate selectivity makes sense because it is beneficial to eliminate most rows as early as possible and because the number of rows eliminated is proportional to (1 - the selectivity value). However, the actual answer is far more complex in a column-store because the (effective) length of the sort order in the MV is limited by the ratio of the composite cardinality of the sort order and the total number of rows in the MV (which, in turn, depends on the database size). The reason for this is due to the nature of RLE encoding, which records value runs by storing a pair of values (value, runLength). Therefore, the length of the maximum effective sort order is based on the average length of each run. In most cases, the value runs of three values or more will be effectively compressed by RLE (see Sections 2.6.1 and 5.1.1 for more details). The sort order can thus be extended as long as:

$$\frac{\#rows}{Cardinality(SortOrder)} > 3 \tag{6.1}$$

To choose the right sort order suffix, we order predicates using the following ranking function, instead of relying on selectivity alone:

$$AttrBenefit = \frac{(1 - Selectivity_{eff})}{Log(Attr_{eff})} \tag{6.2}$$

Here, $(1 - Selectivity_{eff})$ is the number of rows filtered out by the predicate (e.g. a predicate with a selectivity of 0.1 will eliminate 90% of the rows in the remainder of the columns from consideration by the query). This style of row-store ranking favors predicates with the lowest selectivity. However, we want the *effective* rather than the nominal selectivity of the predicate: for example, if the sort order prefix already contains the state attribute and the query has a predicate $state = `ME,`$ then the effective selectivity of a city predicate is

$Selectivity(city = `Portland`|state = `ME`)$

rather than the simple $Selectivity(city = Portland)$. That is, we only consider the rows from cities in Maine based on this particular prefix. In addition, we also use the effective cardinality of the attribute to determine the maximum depth of the sort order. For example, a *city* might have 1000 unique values; however, if preceded by *state* in the sort order, the composite cardinality of the sort order will only increase by a factor of 20 instead of by a factor of 1000 (assuming a perfect functional dependency; otherwise, it is computed using the $SoftFD(state, city)$ correlation strength measurement described in Section 2.1.2). We then divide the selectivity benefit by the logarithm of effective cardinality (Equation 6.2), because that cardinality determines how many attributes can fit into (i.e., be added to) a sort order. The depth of the sort order is limited by the average length of the value runs (Equation 6.1). Every time we add an attribute to the sort order, we decrease the average length of the value runs by the effective cardinality of that attribute. Intuitively, adding two columns of cardinality 10 increases the composite cardinality of the sort order by 100 (i.e., just like a single column of cardinality 100). Since the composite cardinality is, by nature, multiplicative, we apply the logarithm function shown in Equation 6.2.

The argument and ranking function presented above may seem counterintuitive at first glance; hence, we will illustrate it with the following example query:

```
SELECT SUM(lo_extendedprice)
FROM lineorder, part, customer
WHERE part.p_partkey = lineorder.lo_partkey
  AND customer.c_custkey = lineorder.lo_custkey
  AND p_brand1 between 'MFGR#15' and 'MFGR#30'
  AND lo_quantity between 10 and 30
  AND p_size between 20 and 40
  AND c_city between [a] and [b];
```

The selectivities of all predicates, with the exception of *c_city*, are fixed. We can vary the selectivity of *c_city* by substituting different values for [a] and [b]. Let us further assume that we have already opted to create a fully pre-joined MV and select the sort order prefix (*p_brand1*). We now have the task of extending the sort order using the remaining columns. The selectivity-based choice here makes sense, and we can compare this to the choices based on our ranking function.

Figure 6.7 shows that *lo_quantity* and *p_size* both have a fixed selectivity of approximately 0.40. We can vary the selectivity of the *c_city* predicate between 0.1 and 0.3. Now, we can consider the options of extending the sort order prefix with *c_city* or *lo_quantity*. Although the selectivity of *c_city* is always lower than that of the fixed selectivity of *lo_quantity*, the query performance of the two alternative sort orders intersects around *c_city* with a selectivity of 0.24 (and not at a selectivity of 0.40 as we might expect). This matches the intuition described earlier in this section, namely that the cardinality of the columns is as important as the predicate selectivity. Since *p_brand1* has a cardinality of 1000, it fragments the sort order more than do *lo_quantity* and *p_size* (cardinality of 50 each).

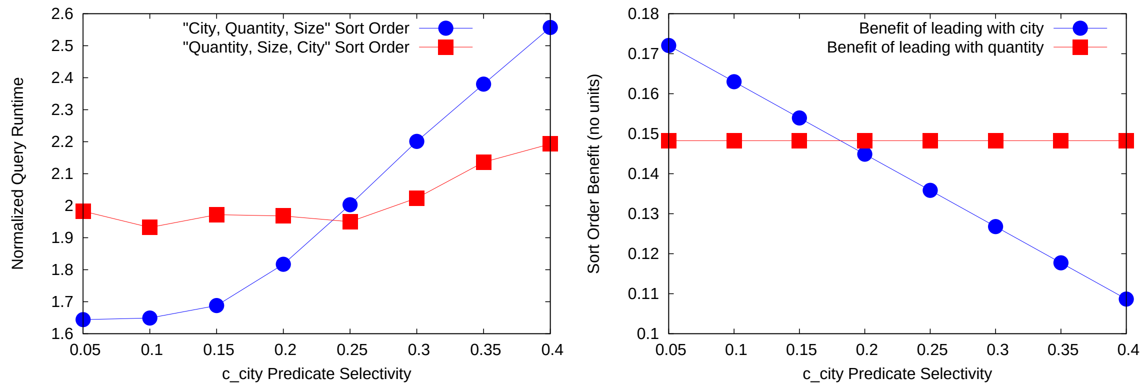


Figure 6.7: **Extending the Sort Order** – Evaluating Runtime and Estimated Benefit for Varying Predicate Selectivity

The second part of Figure 6.7 considers an evaluation based on Equation 6.2 for choosing the right suffix attributes. The ranking of *lo_quantity* and *p_size* is fixed at $\frac{.58}{log(50)} = 0.15$, since the selectivities do not change. The benefit of selecting *c_city* first changes as the selectivity of the underlying predicate changes. The right side of Figure 6.7 shows the values of the ranking Equation 6.2 for *c_city* and the alternative of *lo_quantity*. Note that Equation 6.2 is not an exact measurement and that the y-axis scale has no particular significance. Rather, Equation 6.2 is a heuristic that should capture the correct sort order preference (i.e., identify the fastest sort order) in which relying on the simple selectivity-based metric would result in a wrong answer. In this example, we correctly deduce that *c_city* should lead the sort order as long as it has a selectivity of 0.2 or less, after which, starting from a selectivity 0.3 (still lower than that of the other columns), *lo_quantity* should precede *c_city* in the suffix.

We selected this example query so that the size of the MV remains approximately the same for all sort orders in order to isolate the effect of suffix compression on the query processing time. Selecting different sort orders that have a significant effect on the overall size of the MV might have introduced additional considerations; for instance, the cost of the query might decrease if target columns achieved better compression.

In the end, if all the predicated attributes are already in the sort order, we may continue extending the sort order with the GROUP BY or ORDER BY columns until the maximum depth is reached. In that case, we cannot use Equation 6.2, because GROUP BY and ORDER BY attributes have a selectivity of one (and $log(1)$ is zero). Instead, we choose the attributes that provide the best compression rate once they are RLE-ed.

The sort order design techniques described in this section apply to both fully and partially pre-joined MVs. Once the pre-join has been chosen, the goal is to find the sort order that will minimize the cost of gathering the data from the MV, even if additional joins are subsequently necessary. Having described the ideas behind MV generation, we continue with a discussion of how we create *shared* or multi-query MVs.

6.2 Shared MVs

In some cases, we may have the resources to create one MV per query, in practice, the cost of such design is often prohibitively high. User queries can have similar predicates over the same attributes or the same target columns, and, thus, we look for an opportunity to create a single MV that serves multiple "similar" queries. In this section, we describe ways to do that, starting from row-store state-of-the-art wisdom [BC05, ACK+04], followed by an explanation of why that does not work in a clustered column-store before presenting our approach.

6.2.1 Merging MVs

One way to produce shared MVs [ACK+04] is to merge individual MVs that have already been designed for user queries. This problem is traditionally approached by considering pairwise merges of MVs [ABCN06] (we might also consider merging the already merged MVs further). Intuitively, for a pair of MVs, MV$_i$ and MV$_j$, created for queries Q$_i$ and Q$_j$, respectively, we consider creating a merged MV$_{ij}$ that is beneficial to both queries and requires less disk space. The *goodness* of this merge is based on the ratio of the query performance penalty (i.e., how much slower Q_i and Q_j become when using MV_{ij} instead of MV_i and MV_j, respectively) and the reduction in space used (the difference between the size of the resulting MV_{ij} and the sum of the sizes of MV_i and MV_j).

In a clustered column-store such as Vertica, this approach faces a number of problems. First, the issue of merging two sort orders in column-store is much more complex than it is in a row-store. In a row-store, index-merging is a relatively simple task: only the exact concatenations need to be considered [CN99]. For example, for indexes (A,B) and (C,D), we need only consider (A,B,C,D) and (C,D,A,B) as the two possible options. In contrast, the merging of two sort orders in Vertica introduces many viable possibilities to consider. Consider the two example queries Q_1 and Q_2 displayed in Figure 6.8. Assume that the best sort orders have already been generated (shown on the same figure). Now let us consider the problem of merging these two sort orders into a merged sort order that can benefit both Q_1 and Q_2. It is important to remember that in a column-store database, the query engine is able to use a subset of the sort order columns. Therefore, we have no choice but to consider possible interleaved sort orders: simply considering the two concatenations based on a row-store only makes available a tiny subset of all viable merges. Furthermore, the interleaving of sort orders can result in a better answer than simply concatenating. Indeed, we will show that in the presence of data correlations, the best answer could be the least intuitive one.

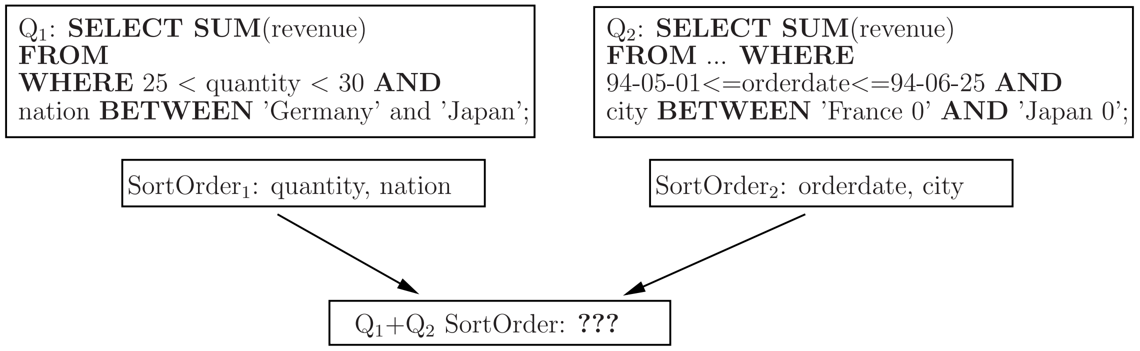

Q1: SELECT SUM(revenue) FROM WHERE 25 < quantity < 30 AND nation BETWEEN 'Germany' and 'Japan';
Q2: SELECT SUM(revenue) FROM ... WHERE 94-05-01<=orderdate<=94-06-25 AND city BETWEEN 'France 0' AND 'Japan 0';
SortOrder1: quantity, nation
SortOrder2: orderdate, city
Q1+Q2 SortOrder: ???

Figure 6.8: **The Sort Order Merging Problem** – Two Example Queries and their Sort Orders

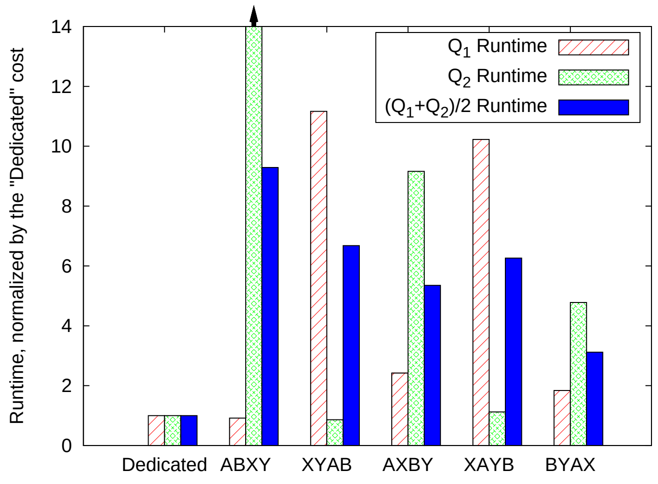


Figure 6.9: **Merging Sort Orders** – Concatenating and Interleaving Sort Orders in Vertica

To evaluate the performances of the feasible sort order permutations based on Figure 6.8, we built all possible permutations of the sort orders that were being merged. To save space in Figure 6.9, we used the following shorthand labels for attribute names: A=*quantity*, B=*nation*, X=*orderdate*, Y=*city*. The normalized runtimes of the two queries are shown in Figure 6.9.

Keep in mind that the first two merges (ABXY, XYAB) in Figure 6.9 correspond to what DBMS-X or any other row designer would consider. Moreover, the resulting runtimes in Figure 6.9 are unsurprising: after the concatenation, one of the queries (the one that "won") exhibits no slowdown, whereas the second query (the one that "lost") is now an order of magnitude slower. The two following indexes are simple (order-preserving) interleaves (AXBY, XYAB), and we can see that the average performance of both interleaves is better than that of the concatenated sort orders. When sort orders are interleaved, both queries run slower, but each is able to benefit from the combined index to some degree and the average query runtime is improved. Finally, we come to the most surprising result of all. Notice that the average performance of the index BYAX is much better than that those of all other indexes despite the fact that the leading column of the merge is the second column of the first parent followed by the second column of the second parent sort order. Intuitively, the best merge of the sort order is the one that first reverses each original sort order (which

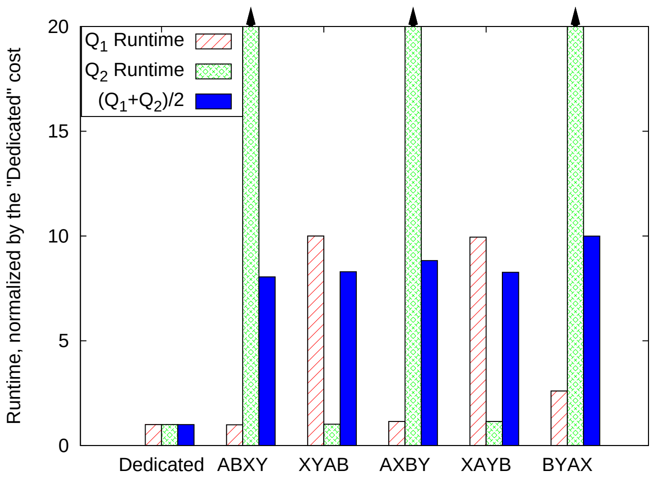


Figure 6.10: **Merging Clustering Indexes** – Concatenating and Interleaving Clustering Indexes in DMBS-X

would make it worse as a dedicated sort order) and then interleaves these reversed sort orders. This behavior can be explained by the existence of a correlation between B and Y (city→nation). This negates most of the penalties associated with fragmentation and lets Q_2 benefit from the second column of the sort order (*city*) almost as if it were the leading column in the index. In other words, the functional dependency between *city* and *nation* means that a query with a *city* predicate carries an *implied* predicate on *nation* (see Section 2.4.1).

Figure 6.10 confirms the state-of-the-art wisdom regarding index merging in a row store. Interleaving indexes is never a good idea in a row-store DBMS. The AXBY and XAYB interleavings only slow down the "winning" query by approximately 15%. However, such a merge does not benefit the second query in any way. The final interleaving of BYAX that wins in Vertica is the worst merge in DBMS-X.

Finally, we want to demonstrate another issue with merging MVs in Vertica. We cannot easily determine whether merging two MVs will result in saving disk space. Although the presence of an overlap between MV columns in a row-store means that the merged MV takes up less space than the two MV parents do, in Vertica the results are unpredictable. Note that the WOS (memory) size of the MV in Vertica works in the same way as it does for a row-store because there is no compression in the WOS. We will illustrate our point with a pair of simple MVs and their merge.

1. MV$_a$: *lo_discount*, *s_address*, *lo_extendedprice* | *lo_discount*, *s_address*
2. MV$_b$: *lo_discount*, *c_phone*, *lo_extendedprice* | *lo_discount*, *c_phone*
3. MV$_{ab}$: *lo_discount*, *s_address*, *c_phone*, *lo_extendedprice* | *lo_discount*, *s_address*

In terms of the WOS budget, MV$_{ab}$ is approximately 27% narrower than is the sum of MV$_a$ and MV$_b$. We expect such a reduction because the MVs overlap on two of their three columns. However, in terms of disk space, using the scale-4 SSB dataset results in MV$_a$ requiring 241 MB, and MV$_b$ requiring 238 MB of disk space. However, surprisingly, MV_{ab} requires 595 MB disk space, which is 22% *more* than the combined size of MV_a and MV_b. Thus, here it is actually cheaper to keep two dedicated MVs than it is to store their merge as one shared MV because neither *s_address* nor *c_phone* compress well using LZO. RLE does much better, but we cannot RLE both those columns in one MV. Thus, when we merge the MVs, one of these columns (*c_phone* in our example, because the *s_address* is RLE-ed) increases in size dramatically. As a result, the losses owing to lost compression outweigh all space savings achieved by eliminating (through the merge) the duplicate columns *lo_extendedprice* and *lo_discount*.

The goal of these examples has been to demonstrate that in a column-store DBMS, merging MVs is not guaranteed to produce useful new MVs for several reasons. First, when merging sort orders from two MVs, we need to consider many more merging permutations in Vertica. Preserving the parent sort orders (by concatenation or even interleaving) may not produce the best results. Second, owing to compression artifacts, a conventional similarity measure (e.g. the number and size of overlapping attributes) can estimate the WOS budget, but cannot estimate the disk size of the MV. Therefore, we now present an alternative approach that our design tool uses to generate shared MVs.

6.3 Multi-Query MVs

The idea of multi-query MV is to first merge the queries (e.g., Q_i and Q_j) into a group [Q_i, Q_j] and then use that group as a basis for the MV_{ij} candidate using the techniques for producing single-query candidates described in Section 6.1.2.

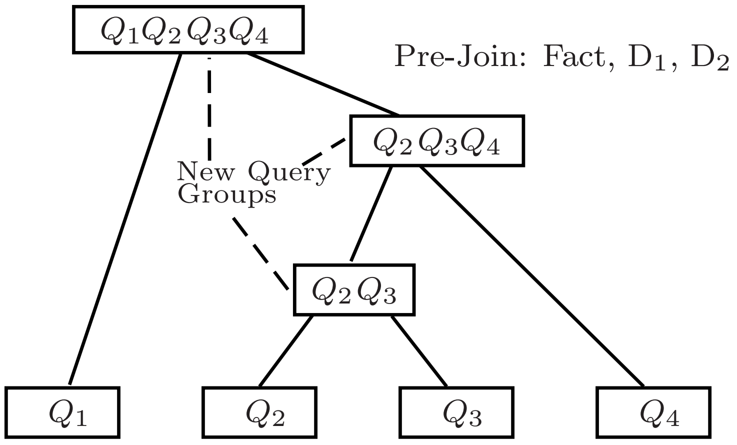


Figure 6.11: **Query Merging** – A Single Merge Pass

We begin with a known set of query groups (initially, single-query groups) and perform pairwise merging based on a distance function that we will describe shortly. A single *merge pass* (Figure 6.11) produces a fully merged tree. Algorithm 1 performs a new merge pass with all groups formed until it reaches the stopping condition. Figure 6.11 shows a hypothetical 4-query set that is being merged under the condition of 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., those most similar according to our distance function) is merged. To produce interesting new candidates, we avoid merging overlapping query groups. Section 6.5 discusses how to select the closest pair of queries to merge.

6.3.1 Building MV Candidates for a Query Group

The approach to building an MV that serves multiple queries at once is similar to that described above for a single query. Indeed, only a few modifications are necessary. We consider pre-joins with different dimensions exactly as we do for a single query. In fact, the number of possible pre-joins is determined by the schema rather than by the composition of the query group. The sort order prefix cost is computed in the same way as before, but this time it is averaged for all queries in a group and the prefix with the best average performance is selected. Note that such a computation assumes that all queries have the same weight (i.e., priority) in the query workload. It is also trivial to account for query weights by incorporating them into the average computation. We apply the same filtering technique to eliminate inferior prefixes. In addition, owing to the nature of the column-store DBMS, we can cache and reuse read pattern estimations, thereby avoiding recomputing the same prefix that has already appeared in a different computation.

Having selected the best sort order prefix, we extend it based on query predicate selectivity and cardinality, just as we did for a single-query MV. The attribute ranking function is adjusted to account for multiple queries in the group as follows (assuming that we have already chosen a prefix P_b):

$$Benefit(Attribute_A) = \Sigma_{\forall Q_i \in QGroup} \frac{(1 - Selectivity_{eff})}{Log(|A|_{eff})} \tag{6.3}$$

The attribute ranking function is the same as for a single query with one important modification: as in the previous case, we factor in the selectivity-based benefit of the attribute for each query Q_i in the query group, which is multiplied by the total selectivity of the prefix P_b using predicates in Q_i. This adjustment is necessary because a query predicate is only applied to the part of the column that is being read by the query. For example, a predicate with a selectivity of 0.5 applied to half the column will filter out more rows (25% of the rows) than will a predicate with a selectivity of 0.01 applied to one-tenth of the column (9.9% of the rows).

The benefit of additional columns in an RLE sort order is always affected by the prefix selectivity. However, when considering a single query (i.e., query group of size 1), prefix selectivity is a common factor for all calculations and does not need to be factored into the computations.

6.4 Replicated Materialized Views

As a reminder, replication in Vertica operates by segmenting MV pieces across multiple machines and keeping enough spare copies to recover lost data (see also Section 2.6.4). To build such MVs, we return to the idea of query grouping. Our goal here is to produce two sort orders instead of one (we will assume K= 1, but the approach extends to larger values of K) to further speed up the queries. We therefore split the candidate query group into two sub-groups and generate a separate MV sort order for each of them. Both the replicated MVs contain all the columns that the entire query group contains. Thus, instead of generating MV_{G1} for a query group G1 and replicating MV_{G1} to achieve 1-safety, we generate MV_{G1_a} and MV_{G1_b}, which contain the same columns as MV_{G1} and thereby serve as replicas. The advantage of using such 1-safe MVs is the ability to generate two distinct sort orders instead of generating one sort order twice. Interestingly, this means that we generate two entirely new sort orders instead of creating one additional sort order to add to that chosen by MV_{G1}.

There are several ways to divide the query group into multiple sub-groups. We could use K-Means clustering (similar to that used in [KHR+10]) or hierarchical clustering as described in Section 6.3. We chose to rely on hierarchical clustering by caching all MV pairs that had been merged during the design phase. As a result, for every query group under consideration, we already have a cached answer for dividing it into two sub-groups. Thus, we can use the approach described in Section 6.3.1 to build MVs and sort orders for each of these sub-groups.

6.5 The Query Distance Function

Recall that we need to organize user queries by grouping together similar queries (those that will likely benefit from the same MV). In this section, we present the details of the two distance functions used in our design tool.

6.5.1 Query Vectors

Previous work (e.g., [BC05]) has created designs by first generating candidates for a single query and then merging these candidates to reduce the total number of MVs in the design. In contrast, we group queries that are similar and then design MVs for each group. In Section 6.2.1, we explained why our approach is more effective in our particular problem setting. The first of our distance functions is similar to that described in [KHR+10].

Intuitively, since we are looking for queries that will be well served by a shared MV, finding queries with similar predicates is important. To detect similar queries, we represent each query with a *selectivity vector* (see Section 2.4.1). A 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-dimensional vector with N non-default entries (default = 1), one for each attribute's selectivity (between zero and one). All other dimensions will be set to one, although for brevity, we omit these default values in the following examples. Using the SSB [POX] Query #2.1:

SELECT SUM(lo_revenue), d_year, p_brand1
FROM lineorder, dwdate, part, supplier
WHERE ... 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:

$$QV_{2.1} = (p\_category : 0.025, s\_region : 0.2, d\_year : 1.0,$$
$$p\_brand1 : 1.0, lo\_revenue : 1.0)$$

As briefly discussed in in Section 2.1.2, by using selectivity propagation, we can make the following selectivity adjustments to QV$_{2.1}$ if there are two functional dependencies as shown.

$$\begin{aligned} QV_{2.1} = (&p\_category : 0.025, \rightarrow \mathbf{p\_mfgr : 0.2}, \\ &s\_region : 0.2, \rightarrow \mathbf{s\_nation : 0.2}, d\_year : 1.0, \\ &p\_brand1 : 1.0, lo\_revenue) \end{aligned}$$

As discussed, in some cases, a partially pre-joined MV may be a better design candidate compared with a full pre-join, either because it takes up less disk space or because it has a lower maintenance cost. The similarity between queries depends on the pre-join in the candidate MV under consideration. Intuitively, if query Q1 accesses dimension tables D1, D2, and D3, and Q2 accesses dimension tables D1, D4, and D5, then these queries are unlikely to be similar because they have only one dimension table in common. However, if we consider a partial pre-join of the fact table with D1, then the MV sort order would be built using attributes from the shared table D1, and in that context, Q1 and Q2 would be more similar. To capture this similarity, we introduce a projection operator that allows us to 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 by setting all other attributes in the vector to one. To use the same example query, a possible vector for Q2.1 is as follows:

$$QV_{2.1}[lineorder, dwdate] = (d\_year : 1.0, lo\_revenue : 1.0)$$

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

At every merge step, the closest pair of query groups (i.e., the most similar according to our distance function) is merged. To produce interesting new candidates, we avoid merging overlapping query groups. The distance between the 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 have not yet been generated. However, we use the LZO compression-based estimate for all attributes to approximate the size. For example, *d_year* compresses to approximately 1.00 byte per row, whereas *lo_revenue* compresses to 3.74 bytes per row. Section 5.1.1 provides a more detailed overview of this. Our 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 similar queries will have lower distances than do dissimilar ones. The idea of merging similar structures has already been established in [CN99, BC05]. Even though we merge queries instead of MVs (as in [BC05]), there are some conceptual similarities. In the case of [BC05], the basic similarity measure is the overall query penalty (i.e. performance deterioration) resulting from the MV merge. This similarity measure is normalized by the estimated size of the penalty resulting from that merge. In our algorithm, we choose to merge queries instead of MVs, since a great deal of information may be lost when a user query is "transformed" into a dedicated MV. Section 6.2.1 illustrated some of these problems with an example.

6.5.2 Jaccard Coefficient

The distance function presented in the previous section is a somewhat complex way to measure query similarity. We have tested a number of simpler distance functions; however, none of them consistently produces a good design. Here, we present a reasonably simple mechanism that will be evaluated in Section 7.3. We chose to base that mechanism on the *Jaccard coefficient* [Jac], because it is one of the most intuitive and it has been most frequently requested by other researchers.

The Jaccard coefficient measures the similarity between two sets. The value of the coefficient is the ratio of the intersection and the union of two sets. Thus, for sets A_1 and A_2, the coefficient is:

$$Jaccard(A_1, A_2) = \frac{|A_1 \cap A_2|}{|A_1 \cup A_2|} \tag{6.4}$$

The value of the coefficient ranges between zero (intersection is empty) and one (intersection matches the size of the union). The corresponding distance is therefore:

$$Distance_J(A_1, A_2) = 1 - Jaccard(A_1, A_2) = (A_1, A_2) = \frac{|A_1 \cap A_2|}{|A_1 \cup A_2|} \tag{6.5}$$

To apply this distance to SQL queries, we represent each query as a set of its attributes. The Jaccard distance will then compute the similarity measure based on the number of overlapping attributes. Note that this is the same intuition as that behind our query generator (see Section 2.4.2). Let us consider a few illustrative examples. Given two identical queries, the distance is Distance$_J$(Q$_1$,Q$_1$)= $1 - 1 = 0$. For two queries that share no attributes (an empty intersection set of attributes, set Q$_1$ $= (A, B)$ and Q$_2$ $= (C, D)$), the distance would be Distance$_J$(Q$_1$,Q$_2$)= $1 - 0 = 1$, which is the large possible distance. Finally, for two queries that share one of three attributes (Q$_1$ $= (A, B)$ and Q$_2$ $= (B, C)$), the distance would be Distance$_J(Q_1, Q_2) = 1 - \frac{1}{3} = \frac{2}{3}$.

6.6 Design Algorithms

Algorithm 1 covers the overall design process including all the components presented in this chapter.

First, we initialize the *queryGroupingType* based on the approach we plan to use to produce query groups. Our latest design tool defaults to hierarchical merging and the query vector distance function. We then initialize the stopping condition with a desired value: intuitively, once the candidates generated during the last pass of the algorithm do not improve the overall design curve by a certain desired percentage (here, we use the area under the curve as a quality measure), then the design process will stop. Lines *6* to *9* initialize the query group set with singletons.

Starting from line *10* is the main loop of the algorithm that continuously generates new query groups, builds candidate MVs for each generated query group and then generates a new design. The query-grouping process (line *11*) and the candidate-building process (line

14) are described in Section 6.3 and Section 6.3.1, respectively. In line *17*, we feed the current set of MV candidates to the LP Solver and receive the current design curve as an answer. Finally, the loop continues to iterate until the improvement in the overall design is deemed too small to continue (line *18*).

Algorithm 1 Main Candidate Design Algorithm

```
1: queryGroupingType ← [MergingType, DistanceFunction]
2: deltaIteration ← X {Stopping condition}
3: queryGroups ← {}
4: candidateMVPool ← {}
5: designCurve ← {} {Empty design curve}
6: for $Q_i$ in Queries do
7:    deltaIteration ← X {Stopping condition}
8:    {Initialize the set of query groups to single queries}
9:    queryGroups.append([$Q_i$])
10: end for
11: repeat
12:    newQueryGroups ← groupQueries(queryGroups, queryGroupingType)
13:    for $QGroup_i$ in newQueryGroups do
14:       {Add new candidates to the candidate pool}
15:       candidateMVPool ← buildCandidates($QGroup_i$)
16:    end for
17:    oldDesignCurve ← designCurve
18:    {Build design incorporating insert rate and produce the best design curve}
19:    designCurve ← BuildDesign(candidateMVPool)
20: until (oldDesignCurve.quality-designCurve.quality)<deltaIteration
```

Next, we elaborate on the query-grouping step in line *7* of Algorithm 1. The details are shown in Algorithm 2. We start by enumerating all possible joins in line *4* because we generate a single sort order for any particular pre-join, which is the maximum number of candidates a query group can have. For each chosen pre-join, we apply the query-grouping algorithm that has been specified. Each of the query-grouping methods uses the selectivity vector; however, the selectivity vector is designed according to the specified pre-join.

For the hierarchical query grouping, we perform (n-1) merges (where n is the number of input query groups), merging the “closest” pair every time (see Section 6.5). In lines *9* and *10*, we build query group vectors applying projection with the current pre-join, and in line *11*, we choose the best pairwise merge for the current merging step. The (n-1) pairings are recorded in newQueryGroups in line *14*. Finally, all new query groups are returned by Algorithm 2 in line *26*.

Next, we elaborate on the step in line *14* of Algorithm 1. We have already explained the ideas behind our approach of designing an MV candidate set for a query group. We use Algorithm 3 to generate MVs for a given query group. In line *4*, we adjust each query based on the particular pre-join (by removing the excluded dimensions and replacing them with foreign keys). Next, in line *5*, we enumerate all non-dominated prefixes exhaustively and evaluate each to find the prefix with the lowest cost in lines *6* to *12*. Once the lowest cost prefix has been selected, we extend the sort order by building a sort order suffix using Equation 6.3. Once the entire sort order has been selected (and the pre-join is already known), all that remains is to assign individual encodings to the columns that are outside of the sort order (see Sections 2.6.1 and 5.1). We assume that every attribute in the sort order is encoded using RLE.

Algorithm 2 Generate Additional Query Groups

```
1: Parameters: queryGroups, queryGroupingType
2: newQueryGroups ← {}
3: PreJoins$_{Qs}$ ← Enumerate all possible query pre-joins
4: for PreJoin$_{Dim_1 \ldots Dim_n}$ in PreJoins do
5:    if queryGroupingType=Hierarchical then
6:       for Step$_a$ in queryGroups.length-1 do
7:          for QG$_i$ in queryGroups do
8:             for QG$_j$ in queryGroups do
9:                Vector$_i$=Vector(QG$_i$, mod = PreJoin$_{Dim_1 \ldots Dim_n}$)
10:               Vector$_j$=Vector(QG$_j$, mod = PreJoin$_{Dim_1 \ldots Dim_n}$)
11:               {Identify the closest pair of query groups, normalized by LZO size estimate}
12:               bestPair = min(bestPair, Pair(Vector$_i$, Vector$_j$))
13:            end for
14:         end for
15:         newQueryGroups ← bestPair
16:      end for
17:   end if
18: end for
19: return newQueryGroups
```

Algorithm 3 MV Design Algorithm for a Query Group

```
1: Parameters: queryGroup
2: mvCandidates ← {}
3: for PreJoin$_{Dim_1...Dim_n}$ in PreJoins do
4:    modQueryGroup ← queryGroup MOD PreJoin$_{Dim_1...Dim_n}$
5:    Prefixes ← buildFilteredPrefixes(modQueryGroup)
6:    for prefix in Prefixes do
7:       cost = 0
8:       for Query$_i$ in modQueryGroup do
9:          cost += evalPrefix(query, prefix)
10:      end for
11:      bestPrefix ← lowest cost prefix
12:   end for
13:   suffix ← {}
14:   for attribute$_i$ in (modQueryGroups.attributes - bestPrefix.attributes) do
15:      RankedAttrs ← rank attributes using Equation 6.3
16:   end for
17:   while Cardinality(prefix+suffix) < $\frac{TotalRows}{3}$ do
18:      suffix.add(RankedAttrs.best())
19:   end while
20:   newMVCandidate ← PreJoin modQueryGroup sorted on [prefix,suffix]
21:   {Assign encodings to attributes outside of the sort order}
22:   for attribute$_j$ in (mvCandidate.attributes - [prefix+suffix]) do
23:      assignEncoding to attribute$_j$
24:   end for
25:   mvCandidates ← newMVCandidate
26: end for
```

Chapter 7

Evaluation of the Physical Design

This chapter comprehensively presents results of the design process by assessing the performance of Vertica DBMS. The preceding chapters described materialized views (MVs) (Chapter 4) that were used in the physical design, the resource considerations when adding MVs into the physical design (Chapter 5) and the proposed methods for building MV candidates based on a query workload (Chapter 6). Now, given the user inputs outlined in Section 1.2.3, in this chapter we describe the designs that have been generated by our design tool.

In Section 7.1, we introduce the experimental setup and measuring methodology. Then, in Section 7.2 we discuss ways of selecting a subset from the existing candidate MV pool. Section 7.3 is a discussion of the advantages of our techniques (as presented in earlier chapters) as well as the pitfalls of relying on row-store metrics in the column-store setting. We then present the SSB benchmark [POX] results and compare our designs to those produced by an expert database administrator (DBA) in Section 7.4. Following that, we proceed to evaluate the sensitivity of our designs to the skewness in the source data in Section 7.5. Finally, we end this chapter by presenting an update-aware design in Section 7.6 and a replicated 1-safe design in Section 7.7.

7.1 Experimental Setting

In this section, we reiterate some of the assumptions in our problem setting. As explained in Section 1.2.3, our design tool makes a certain number of assumptions that are based on a particular problem environment. Column-store DBMSs are primarily used for processing *Online Analytical Processing* (OLAP) [CD97] problems. Clustered column stores tend to perform particularly well in this setting; however, they are poorly suited to the *Online Transaction Processing* (OLTP) environment. Thus, we rely on the following assumptions based on the OLAP problem setting:

- The database contains a significant amount of data.
- The logical schema is (relatively) simple: it is either the *star*- or *snowflake*-schema described in Section 1.2.1 or a schema where queries are amenable to being split into star-compliant queries (as shown in [KHR+10, YYTM10]). Also note that queries may be star-compliant despite being defined within a non-star schema (see Section 2.5.4 for a further discussion).
- There are no (or very few) OLTP-style queries that request just a few rows or require an up-to-date result (i.e., that queries can tolerate staleness associated with insert batching). We generally assume that queries access a moderate fraction of the available data (i.e., more than one row but less than an entire table).
- Queries and query predicates are reasonably simple. Thus, we expect predicates that specify a range of data in a column. We do not handle complex predicates such as *User Defined Functions*, which, to our knowledge, is typical of data warehouse query predicates.
- The update workload consists primarily of inserts. Note that it is often assumed that data warehouses are not regularly updated (e.g., updates may be performed at night when the database is not otherwise in use). However, our design tool supports *INSERT* queries together with *SELECT* queries, although we still rely on insert batching.

The remainder of this section outlines the hardware, as well as some of the queries and the data set that we will use in subsequent experiments.

7.1.1 Hardware and Measurement Methodology

We ran our experiments using Vertica on machines running Fedora Core 6 with a single AMD64/3000+ processor, 2 GB of RAM and a 320 GB 7200RPM SATA II disk. We flushed the cache between runs by using the Linux /proc/sys/vm/drop_caches mechanism. In order to collect stable results, we averaged a series of runs, removing the outlier results to leave at least three stable runtimes.

In the majority of our experiments, we used a popular commercial row-store DBMS to illustrate the architectural differences between column stores and row stores. We refer to this row-store DBMS as DBMS-X in our experiments. The DBMS-X experiments were performed on a machine running Microsoft Windows 2003 Server x64 Edition. The machine had a 2.4 GHz Quad-core CPU, 4 GB RAM and 10K RPM SATA hard disk.

The runtimes of DBMS-X and Vertica were normalized with respect to the baseline performance of each DBMS for two main reasons. First, we ran the experiments using different hardware; thus, the runtimes could not have been compared easily. Second, the goal of our work is not to compare row-store and column-store performance directly (as [SAD+10]), but rather to illustrate the underlying architectural differences that need to be taken into account when generating a physical design for a clustered column-store DBMS. Thus, the normalized baselines vary by experiment, and this variation is specified in the analysis of each experiment.

7.1.2 SSB Benchmark

In several of our experiments, we use the SSB benchmark [POX, COO08], which is a star-schema revision of the TPCH benchmark [TPC]. Figure 7.1 (taken from [POX]) lists the tables that make up the SSB schema (Appendix A also lists all the SSB details).

The SSB benchmark comes bundled with a data generator that allows us to generate

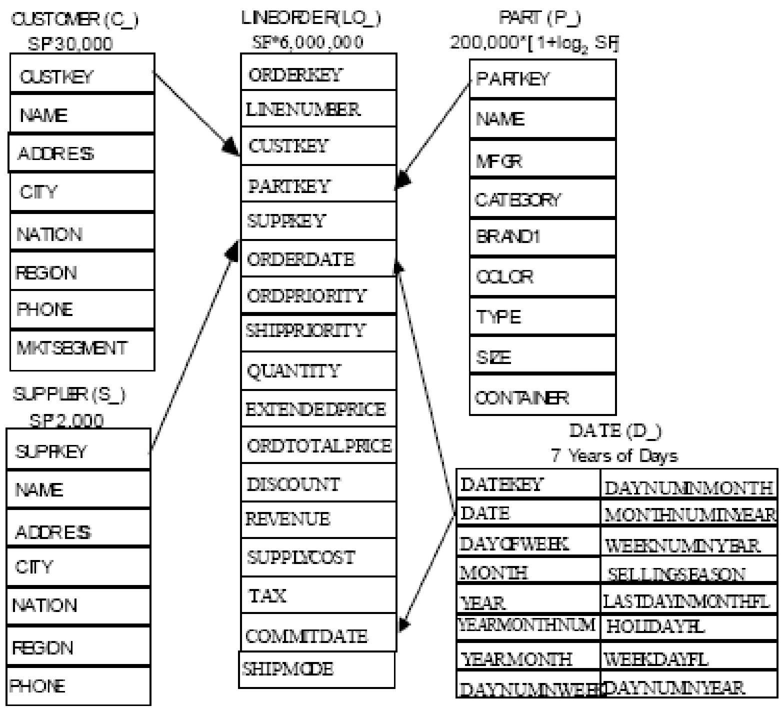


Figure 7.1: **SSB Benchmark Schema** – The tables and foreign keys connecting them

a set of an arbitrary size, while preserving the same data distribution and the existing correlation between the data columns (see Section 2.1.2). The amount of data to generate is specified by the *scale* factor, which is a multiplier of the number of rows in the fact table. Other tables are then scaled up accordingly by the data generator. The basic scale-1 size is 6 million rows – by default, we used a scale-4 dataset (24 million rows).

In Section 7.5, we apply manual changes to the data distribution within the SSB tables. The SSB benchmark also includes 13 representative data warehouse queries. We also use a number of altered SSB queries throughout this dissertation. In addition, we rely on our query generator (see Section 2.4.2) when we need a larger query set.

7.2 Selecting Candidates for the Design

This section describes the different approaches to choosing a design from the provided candidate pool. This task is relatively simple, and the greatest challenge lies in generating and improving the MV candidate pool. We discuss two different approaches: (a) the greedy solution, which is simpler to implement and scales to large candidate sets, and (b) the integer linear programming (ILP) solution, which can find the optimal answer to problems of a certain size [KHR+10].

7.2.1 Greedy Solution

The greedy solution algorithm is a practical and intuitive approach to design selection. However, it is naive in that the algorithm adds the *best* candidate at any given point (without thinking ahead). The quality of the MV candidate is measured as the ratio of the improvement in the runtime of the query workload (i.e., the difference in total query runtime before and after the MV is added) and the resource cost of adding the MV candidate to the design. The greedy mechanism is inefficient, particularly because it cannot undo any of the decisions made (although every MV is best at the time it is chosen, eventually it may become obsolete by subsequently adding further MVs). However, it often produces a "good enough" result.

The more general modification of the greedy algorithm is the *Greedy(m,k)* mechanism proposed by Surajit et al. [CD97]. This approach replaces a constant number of initial steps by running an exhaustive search. For example, setting $m = 2$ implies that the first two candidates are chosen by the exhaustive enumeration of all pairs and that the design population is then continued using the greedy method. Such a solution is not feasible for values of m beyond of two or three (because for C candidates, the number of exhaustive permutations is C^m), but it does improve the basic greedy algorithm.

7.2.2 The Optimal ILP Solution

As been proposed in [PA07] and [KHR+10], although there are costs associated with using an LP solver, introducing an optimal solution removes a potential source of inaccuracy. For example, if we had the resources to produce an exhaustive set of candidate MVs (a very expensive option, but one that we investigate in Section 7.3), we could guarantee that a better solution does not exist. In practice, provided that we supply a good set of candidate MVs, we can expect a good resulting design, because the designs will be exactly as good as the source MV candidates are.

As defined in [KHR+10, PA07], the physical design problem can be formulated as a general set of constraints for an LP solver. The following assumptions (that hold true for both row-store and column-store DBMSs) are made: (a) the design consists of a set of MV candidates, (b) each MV candidate provides a certain query runtime (which is beneficial if the current query runtime is slower), and (c) each MV candidate has a certain size on disk (MB) that uses up some of the budgeted resources. Table 7.1 lists the variables that were used to formulate the problem and produce the answer.

M	Set of MV candidates.
Q	Workload query set.
m	An MV candidate. $m = 1, 2, .., \|M\|$.
q	A workload query. $q = 1, 2, .., \|Q\|$.
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. $(r_1 \leq r_2 \Leftrightarrow t_{q,p_{q,r_1}} \leq t_{q,p_{q,r_2}})$. $t_{q,m}$ and $p_{q,r}$ are calculated by applying the cost model to all MVs.
$x_{q,m}$	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 7.1: LP Formulation Variables

Using the specified variables, the LP objective function is specified as follows:

$$\sum_{q} \left(t_{q,p_{q,1}} + \sum_{r=2...|M|} x_{q,p_{q,r}} (t_{q,p_{q,r}} - t_{q,p_{q,r-1}}) \right) \tag{7.1}$$

The objective function is minimized subject to the following constraints:

$$y_m \in \{0, 1\} \tag{7.2}$$

$$1 - \sum_{k=1}^{r-1} y_{p_{q,k}} \leq x_{q,p_{q,r}} \leq 1 \tag{7.3}$$

$$\sum_{m} s_m y_m \leq B \tag{7.4}$$

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 acquires its dedicated MV, the sum of penalties is zero. If an inferior MV is used owing 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 the (r)th best MV instead of the (r-1)st. Equation 7.3 constrains x$_{q,m}$ based on which MVs were chosen (y$_m$). Equation 7.2 ensures that y$_m$ is boolean, since MV$_m$ is either chosen or not. Finally, Equation 7.4 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 [cpl]). The values of y_m that are returned identify the ideal subset of MVs that will constitute the output design.

7.2.3 Handling Inserts

Many data warehouse workloads contain inserts, and the corresponding maintenance costs increase dramatically as the design size increases. In fact, the space budget is typically a metaphor for maintenance overheads. The cost of insert queries is incorporated into the total runtime, which we are trying to minimize. Note, as stated earlier, we assume that inserts arrive in batches and that the user can tell us the approximate size of these batches. In row stores, the insert cost is 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. In a clustered column store, such overhead costs are more pronounced (Section 4.1), which is why Vertica employs in-memory buffering (Section 2.6.3). Vertica amortizes the maintenance cost both by batching inserts and by writing these batches to disk (delaying the merge). As argued in Chapter 6, the size and number of MVs in the design is not a good predictor of the insert cost because this cost is amortized over many tuples. The bulk of the immediate penalty associated with the new inserts is the cost of the pre-joins utilized by the design MVs. Thus, we adapt our LP solution to account for the insert cost by partitioning the MV candidates based on which pre-joins they require.

We then run the LP solver to select the best design for each partition. For each of these results, we add the cost of the corresponding insert workload (based on the pre-join) to the cost of the query workload and select the best design. The cost of the insert depends on the size of the dimension tables that are read to produce the necessary pre-joins (Section 5.4). Thus, a design built from the candidate partition that avoids large dimension tables might be preferable.

In practice, we create candidate subsets by sequentially eliminating the most expensive table combination from the pre-joins in the candidate pool. Because the cost of the join is determined by the size of the dimension tables that need to be pre-joined with the inserted rows, by eliminating the single most expensive dimension table, we find the first candidate partition to be used as an insert-conscious design. To find the next partition, we eliminate the single most expensive pair of dimension tables to pre-join. Stopping the process of creating sub-partitions is a heuristic one. In practice, the stopping condition of 10% (i.e., when the set of tables to be removed next carries a total cost of less than 10% compared with the previous one) works well.

Inserts also incur move-out and merge-out penalties in the long run (Section 5.3.1). However, these costs are amortized and thereby negligible for most insert rates. For example, assuming a 500 MB WOS buffer (in-memory insert buffer, Section 2.6.3), most SSB designs would be able to buffer over 1 million rows before the first move-out had to be performed. Since a move-out occurs infrequently and does not take a significant amount of time (e.g., a few seconds every hour), we can ignore this in our cost formula.

7.3 Clustered Column-store Metrics

Throughout this dissertation, we have presented a number of techniques and algorithms to generate a physical design for a clustered column-store DBMS. We have argued that established row-store approaches produce unpredictable and often inferior designs. Thus, while a solution already exists for row-store DBMSs, column-store DBMSs present a new challenge. In this section, we evaluate the results for a range of different designs using query sets, including the SSB query set. There are only two ways in which we can fail to find the best possible design:

1. Failure to find a proportion of the good MV candidates (i.e., MVs that could, if they were available, improve the final design).

2. Failure to find the best design from the set of available MVs (i.e., not finding a better combination of MVs drawn from the available candidates).

When we use the LP solver to generate an optimal candidate subset, option 2 is eliminated as a source of error. Thus, assuming that the LP solution is feasible (which is the case for small to medium-sized problems since the LP solver can handle problems with several thousand MV candidates [KHR+10]), the quality of the final design is determined by which MV candidates were available for the LP solver to choose from. Therefore, expanding the set of candidate MVs can only improve the resulting physical design, which is why we incrementally add MVs to the candidate pool in Algorithm 1.

Although the design algorithm is geared to producing a single design based on the available budget, in many of our experiments, we present a *design curve* to evaluate the performance of our design tool. The design curve plots each design point (a design point is a single physical design containing a set of MVs corresponding to a pair (Workload runtime, Design Size)) as the overall performance of the workload (y-axis) versus the design budget (x-axis). The total runtime of the workload is computed using the relative query weights that are specified as part of the user input (see Section 1.2.3). In other words, the design curve is a plot of all possible designs a user could receive by trying different budgets.

We begin by discussing the merits of including partially pre-joined MVs in our candidate MV pool. The first design is a limited design case, in which we do not consider merging queries or MVs (in order to isolate the differences derived from using different pre-joins). In other words, in this experiment we generate no MVs that serve more than one query at a time.

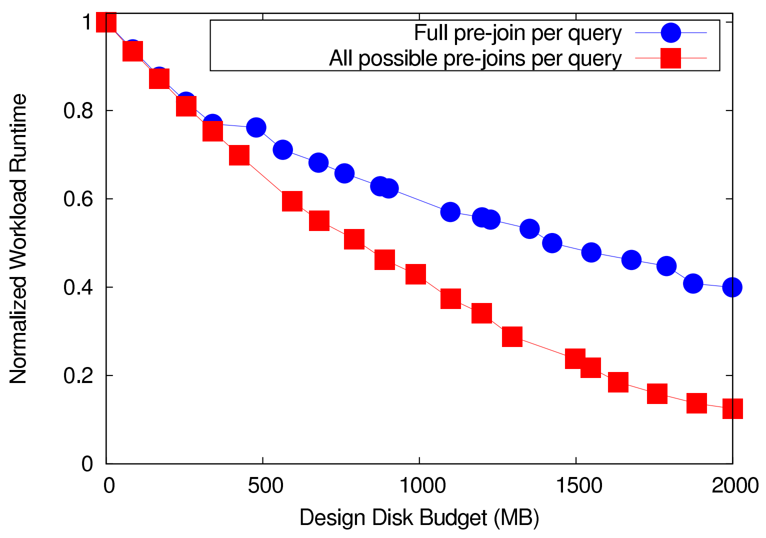


Figure 7.2: **Single-query Groups** – Adding Partial Pre-joins to the Generated MVs

Figure 7.2 plots the resulting two design curves: one that generates the dedicated MVs for every query (blue circles) and another that considers all partial pre-joins for each query in the training set (red squares). In both cases, we use the LP solver to find the optimal design points and normalize the results by using the runtime of the default design (no MVs added at a budget of zero). As shown in Figure 7.2, the design that uses partial pre-joins outperforms the design that relies exclusively on full pre-joins. Note that the gap between the two design curves is nonexistent at smaller budgets and becomes progressively wider at larger budgets. This is no coincidence, and there are several reasons for this outcome. First, as discussed in Section 6.1, the partially pre-joined MVs often resist compression. Therefore, our design tool needs more space to be able to add enough of them to achieve a good runtime (towards 1500–2000 MB the design runtime improves by 85–90%). Second, the initial candidates are derived from the first three queries (*flight-1*, queries numbered 1.X in Appendix A). These queries have a single join (*lineorder* ⋈ *dwdate*), and thus there are few differences between a partially and a fully pre-joined MV.

Next, we consider designs for the same query workload that use query grouping (see Section 6.3) to generate additional shared MVs. Figure 7.3 shows two curves: one that uses only fully pre-joined MVs and another that uses partially pre-joined MVs. However,

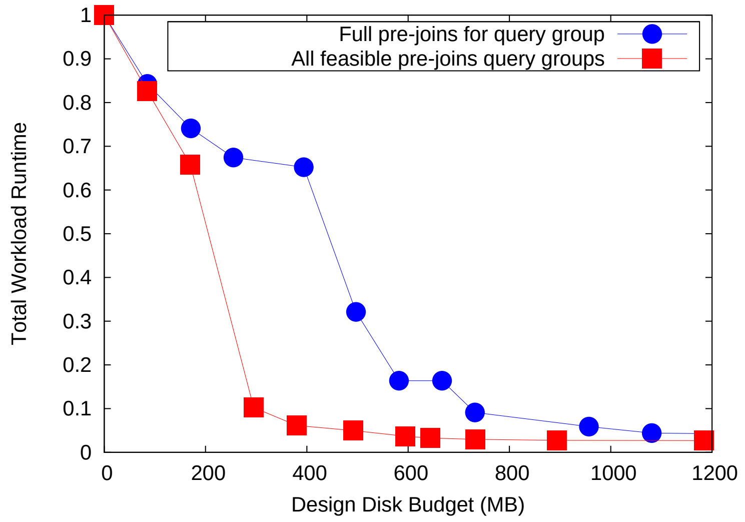


Figure 7.3: **Exhaustive Query Groups** – Adding Partial Pre-joins to the Generated MVs

instead of limiting ourselves to single-query MVs, we use multi-query MVs. To isolate the effects of pre-join use, we consider *all* query groups instead of using our merging algorithm. We reason that if we do not use the merging algorithm, then its accuracy will not affect the resulting designs. Given the 13 queries in the SSB workload, there are a maximum of 2^{13} or 8192 different query groups. We generate fully and partially pre-joined candidates for all these query groups in both cases. There are approximately 8000 unique candidate MVs generated for the first curve and approximately $47,000$ unique candidate MVs generated for the partially pre-joined case. The designs that include partially pre-joined MVs (red squares) perform much better than the ones in the second design curve. By contrast, the curves in Figure 7.2 both notably improved through the addition of shared MVs. The fully pre-joined design curve mostly benefits at larger budgets: although pre-joined MVs compress well, without MV sharing we need too many of them to reach a good overall performance. The design that uses partial pre-joins improves significantly at most budget ranges. The gap between the two curves in Figure 7.3 represents the benefit from using partial pre-joins in our design, since we generate the entire set of query groups that exist in each case. Keep in mind that the most interesting improvement is in the mid-sized budget range. It turns out that good design with very tight budgets (i.e., very little disk space) can be found easily, even by using simple greedy algorithms, because such designs contain very few MVs. A very large budget is also less interesting because it is easier to do well – given enough space, many MV pools will provide a good design. In fact, Figure 7.3 shows that both curves begin to converge at 1200 MB.

Although an exhaustive solution was feasible in the previous example, generating a design that relies on exhaustive query grouping is clearly impossible for larger query sets. For example, a 20-query workload has approximately 1 million unique query groups to consider (see Section 7.3). Recall that in the previous chapter we presented a hierarchical query merging mechanism that serves as a heuristic to find the most promising query groups to use as a basis for MV generation. Figure 7.4 compares the runtimes of (a) our group merging algorithm described in Section 6.6 with the stopping condition set to 1% and (b) the exhaustive design curve. Here, we use a modified workload SSB_EXT, which extends the SSB query set with additional columns (one additional column per dimension that the query accesses). In this experiment, the original SSB workload exhibits roughly the same behavior as does SSB_EXT. Note that the two designs shown in Figure 7.4 perform about the same (of course, nothing can improve on the exhaustive solution). However, the heuristic approach reaches nearly the same performance as does the exhaustive case after discovering only approximately 100 query groups.

Following on from the results of using the SSB workload, we continue with an evaluation of our algorithms with a larger query set. The SSB benchmark provides no more queries, so we rely on our query generator to produce larger query sets. The query generator is fully described in Section 2.4.2, but in short, it supports the following parameterizations: we can specify the attributes that the query will access (randomly chosen from a specified column set) and the selectivity of each predicated attribute by using a *query structure* that describes these values. The selectivity can be specified as a random variable with a one of several random distributions. We also have the option of generating multiple queries from the same query structure.

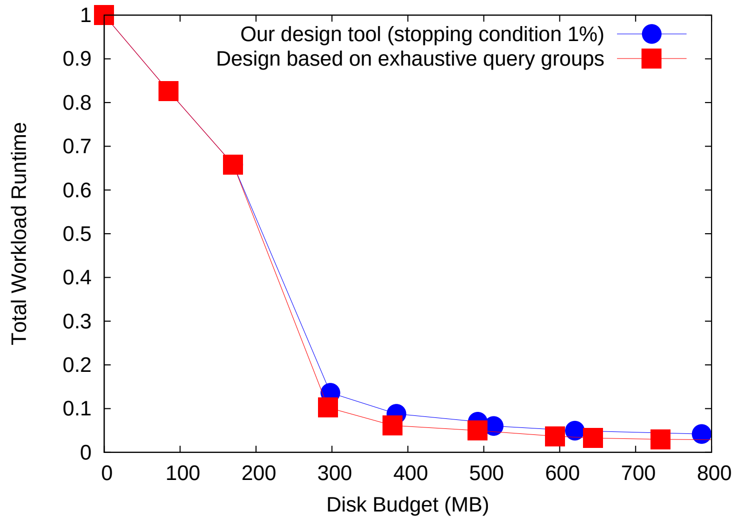


Figure 7.4: **Query Grouping vs. Exhaustive Enumeration** – The Query Merging Heuristic with SSB_EXT

In the next experiment, we generated a 30-query workload from 15 different query structures. We used between one and three random target columns and between two and five random predicates in each query structure. The selectivity of each predicate is drawn using a Gaussian distribution between 0.01 and 0.30. Figure 7.5 shows the runtimes for the fully and partially pre-joined designs.

Figure 7.5 highlights the relatively small performance gap between the two curves. This is because this workload is generated in a uniform manner. There are no *natural* query clusterings (i.e., no query sub-groups share an unusually large number of joins or predicates among themselves). Therefore, introducing partial pre-joins provides only a modest improvement. By contrast, the SSB workload has natural sub-groups that can be easily identified. Queries with similar numbering – for example 1.1, 1.2, 1.3 – are designed to be a natural group. Even more significantly, all the SSB queries use the $lineorder \bowtie dwdate$ join; however, in this experiment no join is shared among all queries. We thus observe that query grouping will find good candidates when they exist. However, if the query workload exhibits no natural grouping tendencies, then the benefit of query grouping and partial pre-joins is limited.

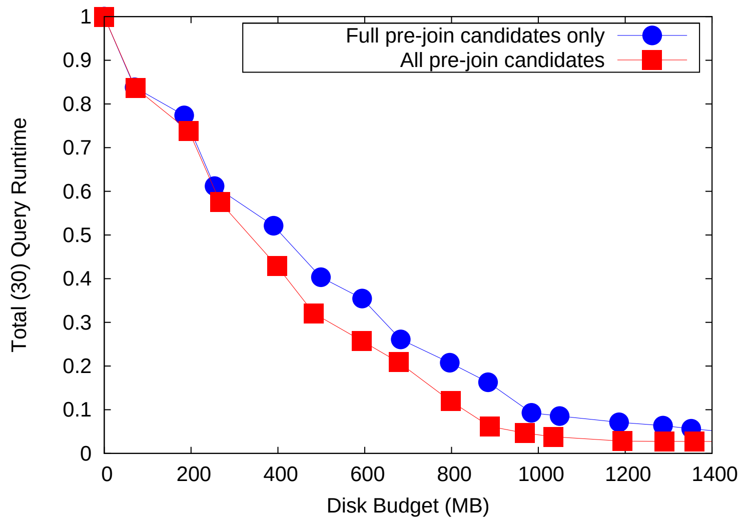


Figure 7.5: **Adding Partial Pre-joins to the Generated MVs** – A 30-Query Set Evaluation

Next, we discuss the difference between the sort order design approach introduced in Section 6.1.2 and the row-store approach to the same task. In this experiment, we use the SSB workload and generate two different design curves: one that uses our algorithm to generate sort orders, and another that picks selectivity-based sort orders for every MV. Note that the query merging is unchanged in both cases, and thereby the two approaches build MVs using identical query groups. Any differences in performance are because we are employing two different sort order design mechanisms. As shown in Figure 7.6, the row-store approach performs poorly. With very small budgets, the design curves match, but starting from 200 MB the curves diverge. Although the row-store approach improves at larger budgets, it clearly fails to find some of the better sort orders that our approach finds (hence, it seems to flatten out at approximately 25%).

Having confirmed that selectivity-based sort orders perform poorly in a clustered column-store, we next consider the other row-store method of generating shared MVs – merging through a concatenation [CN99]. For this experiment, we start from single-query candidates produced by our algorithm and generate shared MVs through concatenation of the parent MV sort orders. Merges are chosen based on the ratio of size change and query performance

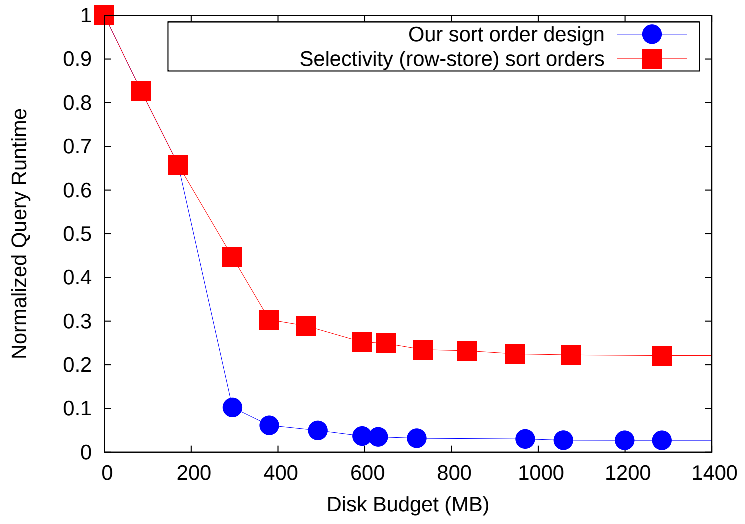


Figure 7.6: **Sort Order Design** – Selectivity-based vs. Column-store Method

penalty [BC05]. Thus, although we start with the same single-query MVs, the queries may be grouped differently as the hierarchical merge steps are performed.

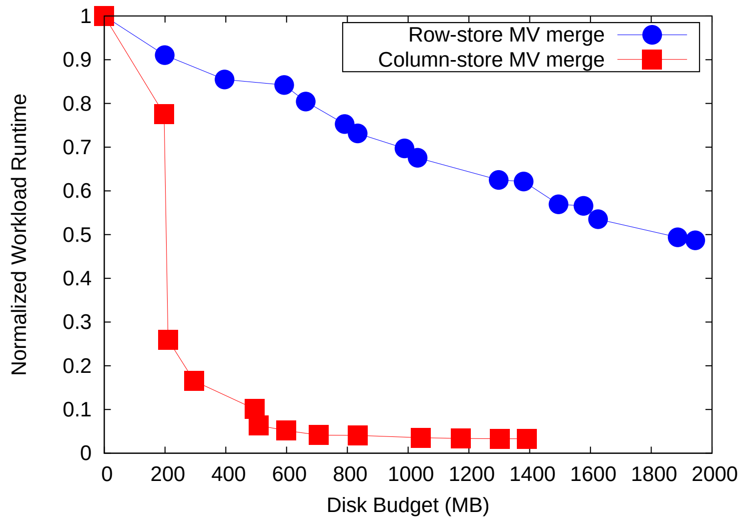


Figure 7.7: **Query Grouping Strategy** – Row-store vs. Column-store Method

Figure 7.7 demonstrates that merging MVs through sort order concatenation in a column store is a very bad idea. As explained in Section 6.2.1, an interleaving (or an interleaving of a reordering) of columns is important to find the right merged MV. A concatenation-based merge causes one query to "win" (i.e., keep its performance benefits), while another query "loses" (i.e., slows down by a factor of 10 or more). Thus, while in Figure 7.6 the sort order merge technique simply missed some of the better MVs, in this case the concatenation approach actively looks for bad MVs by refusing to compromise on shared MVs.

We now proceed to combining the two row-store techniques (selectivity sort order design and concatenation-based query merging) to see how a naive row-store algorithm adaptation might work in a column-store. We consider three combinations of possible row-store approaches in Figure 7.8: building sort orders using selectivity, merging MVs using the sort order concatenation method and using both simultaneously. Figure 7.8 clearly shows that the row-store merge and selectivity-based methods perform very poorly. As before, the selectivity-based sort order method performs moderately well, while the concatenation-based merge performs poorly. Thus, we conclude that none of the row store based approaches should be attempted in a clustered column-store DBMS.

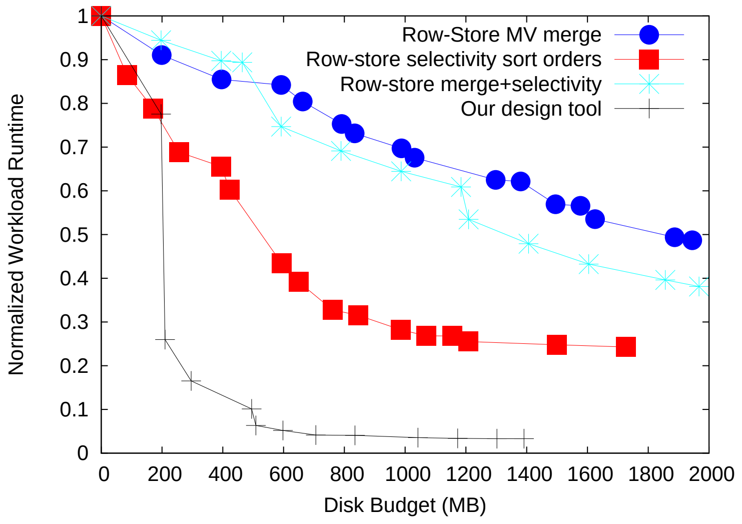


Figure 7.8: **Putting All Candidate Design Strategies Together** – Row-store vs. Column-store Techniques

Before comparing the performance of our design tool with those of the DBA's designs, we will present results that validate the use of our query distance measure. We have observed that our design tool performs well and that using merging adapted from the row-store approach is a bad idea. Recall that a distance function is used for grouping user queries.

Its goal is to identify *similar* queries that will be placed together in a group and that will serve as a foundation for generating additional candidate MVs for the design. We note that a query workload consisting of sub-groups of highly similar queries is more amenable to quick improvement (i.e., is more likely to show improvements at smaller budgets) than is a workload where queries are distributed more uniformly. For instance, consider the performance of the SSB workload in Figure 7.6 and that of a synthetically generated (uniformly distributed) workload in Figure 7.5. The SSB benchmark was designed to have a *natural* query grouping (see Appendix A) and, therefore, the design curve improves quickly (i.e., a steep curve). By contrast, the design performance in Figure 7.5 improves far more gradually because there are no query groups that have a high potential to improve the overall performance of the design (i.e., there are no query groups that allow us to generate MVs that significantly improve the performance of several queries simultaneously).

It is natural to ask whether our distance function (see Section 6.5) is the right way to look for similar queries and whether it could be replaced by simpler (and more intuitive) distance measures such as one based on the Jaccard coefficient (Section 6.5.2). The Jaccard-based distance measure is a natural distance measure that identifies queries that have the most overlapping attributes as being closest to each other. We proceed to generate a new 24-query workload (randomly chosen from two to three target attributes, two to five predicates and with a predicate selectivity drawn using a Gaussian distribution between 0.01 and 0.30). Note that we also evaluated the distance function using the SSB workload. However, the Jaccard-based distance measure performs almost as well as does our vector-based distance function. Recall that SSB queries are designed to group naturally; thus, a simple computation that looks at column overlap proved to be sufficient to identify good query groups.

Figure 7.9 shows two design curves, with each using a different distance function to merge queries. The design curve using the vector distance measure outperforms the Jaccard-based distance measure for most budgets. Not surprisingly, design performance is very similar at very small budgets, because too few MVs can be added to the design. Likewise, design

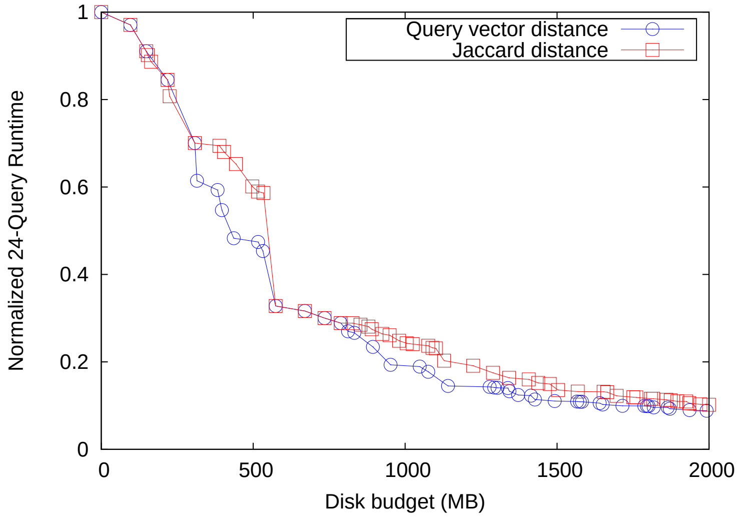


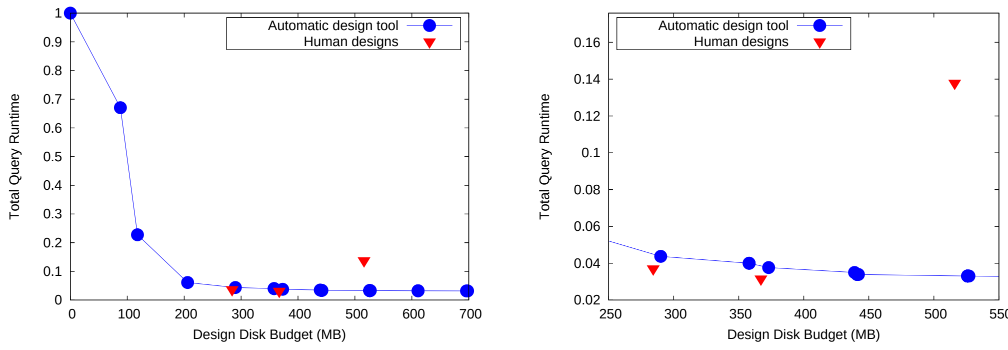
Figure 7.9: **Query Grouping Strategy** – Jaccard Distance vs. Vector-based Distance

performance is similar with very large budgets, since too many MVs can be added to the design. For the majority of mid-range disk budgets the vector distance measure is superior at identifying query groups, resulting in a better overall design. Interestingly, both these curves meet at one point (at approximately 500 MB). This means that at this budget there exists a good design with easily identifiable query groups – and in this case both query distance functions find the right MVs.

We conclude that our vector distance function is superior to simpler approaches because it captures not only the existing attribute overlap between the queries, but also the similarity of the query predicates. Attribute overlap alone is not always indicative of which queries may benefit from a shared MV. In the following section, we evaluate our designs by comparing them to manually generated designs.

7.4 DBA Design Evaluation

This section considers how our design tool compares to designs produced by an expert Database Administrator (DBA). We will use a physical design manually produced by an experienced DBA who is employed at Vertica. Note that the particular way this DBA



(a) Automatic designer curve compared with that of the DBA's

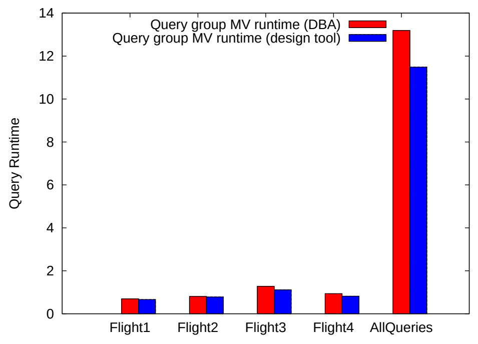
(b) Closer look at the DBA's designs

Figure 7.10: **Design Evaluation** – DBA vs. Automatic Design Tool

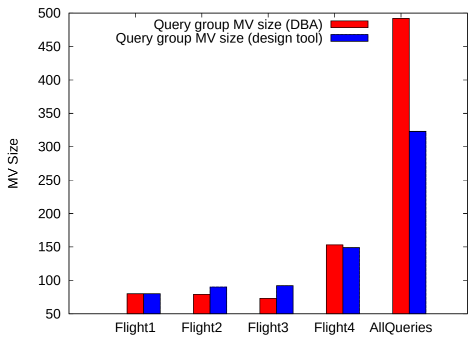
has produced the physical design was by partitioning the queries into a varying number of groups (one, three and four) and then manually creating a fully pre-joined MV for each query group. Figure 7.10 shows the overall design curve and a "zoomed-in" version for a closer look. The designs produced by the DBA are competitive with the output of our design tool (he is, after all, an experienced DBA). However, his DBA designs are not applicable for budgets lower than 275 MB and do not offer a solution that is faster than any of the three designs. Note that the reason one of the DBA's designs stands out as a "bad" design point (at 520 MB) is, once again, because of compression. The "bad" design point corresponds to a single MV (i.e., based on a single-query group incorporating all workload queries) and compresses relatively poorly.

Moreover, the DBA in question did not specify the size of the resulting design. The designs are very good, in part because the SSB workload has some intuitive groupings built into it. But if the DBA got the groupings wrong, the final designs would have been much worse. In fact, the performance of the single MV point in Figure 7.10 is an indication of this issue. Without a size estimate, the human DBA did not know whether this single MV would turn out to be much larger in size despite having fewer MVs than do other designs.

To complete the design analysis in this section, we compare the performances of the individual MVs that were used by the DBA for each query group to the output of our design tool. As mentioned earlier, the DBA partitioned the query workload into a number



(a) MV performance for each query group



(b) MV size for each query group

Figure 7.11: **Individual Flight Evaluation** – DBA vs. Automatic Design Tool

of groups and designed one MV for every group. We will compare the performance of each individual MV designed for every one of the four flights of the SSB workload and the performance of the MV designed for the whole set of queries (i.e., the single MV design). The SSB query flights are the similarly numbered queries (i.e., flight-1 = Q#1.1, Q#1.2 and Q#1.3). Figure 7.11 shows the normalized performances for every MV. In Figure 7.11(a), we plot the query runtimes corresponding to each query group and in Figure 7.11(b) we plot the sizes of every corresponding MV.

As shown in Figure 7.11, the performances of flight-1 through flight-4 are comparable between the DBA's designs and those produced by our design tool. The DBA's query runtime is slightly slower and our MVs are a little larger. The single MV design, however, performs better than that of the DBA by approximately 15%. This is not surprising since generating a single MV for the whole workload is a lot more complex than is generating MVs for a single-query flight. A good DBA may successfully analyze and design good MVs for a small number of queries (particularly when they are pre-grouped in the benchmark). However, as the number of queries increases and when no obvious query grouping exists, the DBA will need the help of an automated design tool.

In this section, we also evaluate the performance of our cost model and compare it to actual Vertica DBMS runtimes. Figure 7.12 shows the SSB workload performance compared with the cost model estimates produced by our designer. This highlights the fact that

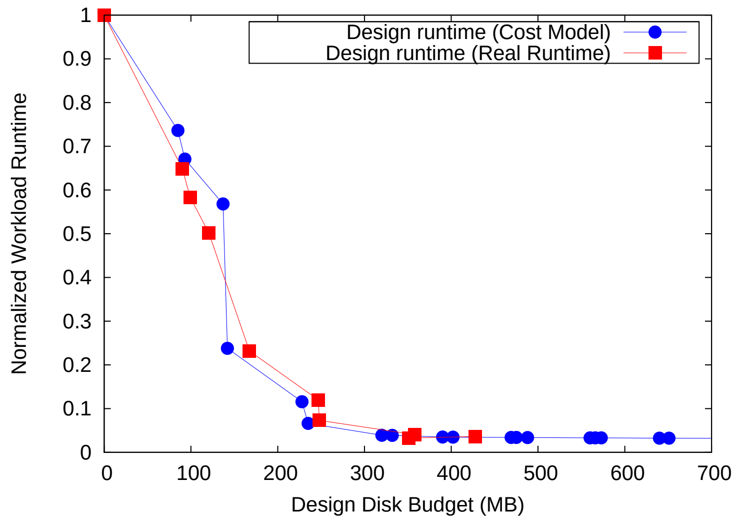


Figure 7.12: **Design Evaluation** – Cost Model Estimates vs. Actual MV Size and Query Runtime

cost model estimates reflect Vertica's actual performance throughout a range of different budgets. Note also that our cost model does not deviate significantly in either the cost or space estimates of the design. As discussed earlier, estimating column compression in Vertica is far more difficult than it is in DBMS-X; therefore, it is important for us to confirm that these estimates are accurate.

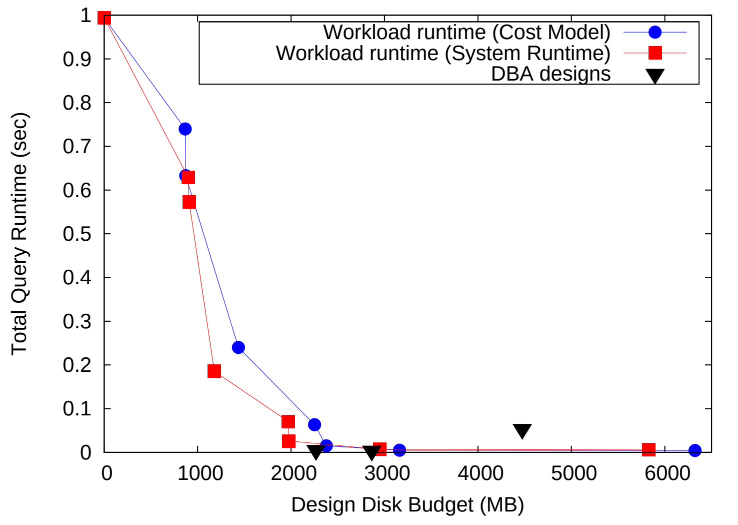


Figure 7.13: **Design Scale Evaluation** – Scale-40 DBA and Design Tool Runtimes

Before we proceed to the next section, we demonstrate how database designs scale in conjunction with the size of the database. Figure 7.13 shows the performance of the SSB design using a scale-40 (240 million rows) data set. Note that in general, performance remains similar for all designs, even though we increased the size of the database by a factor of 10 (from scale-4 to scale-40). Moreover, many of the same design points have slowed down by less than a factor of 10 (compared with Figure 7.10(a)). This is a consequence of column compression: first, RLE compression achieves a higher compression rate when it compresses more rows because the RLE-ed column size depends on the number of unique values rather than on the number of encoded rows. Thus, a column with a cardinality of 100 is reduced to the same disk size regardless of how many rows it represents. Second, with aggressive column compression in DMS, queries often read extraneous data: for example, a query may read a page where only half of the page values are needed. In that case, doubling the size of the database does not increase the page I/O cost incurred by the query.

7.5 Data Distribution

In this section, we discuss the potential effects of varying the distribution within the source data on the resulting physical design. The default data generator included with the SSB benchmark produces a roughly uniform distribution of values for the SSB columns at any scale value. The data generator also ensures that all the integrity constraints hold (e.g., PK-FK relationships between tables and correlations in the data). There is, however, no option to generate a data set that contains an uneven distribution of values in order to model the presence of skewness in the data. For example, we know that certain months are more popular among shoppers (e.g., December because of Christmas); thus, many more purchases are made during December compared with other months.

In order to generate skewed data, we built a data generator that can redistribute the rows of the SSB data before it is loaded into the database or sampled by our design tool. We used the Zipfian-based [ZIP] probability distribution because this is commonly used to approximate a real-world value distribution (such as language word frequencies or city population rankings). Thus, given a population of N elements, the function that lets us compute the frequency of the k-th element is:

$$f(k; s, N) = \frac{\frac{1}{k^s}}{\sum_{n=1}^{N} \frac{1}{n^s}} \tag{7.5}$$

where s describes the particular value distribution. Intuitively, the value of s determines how heavy the observed skewness is – a higher value corresponds to more pronounced skewness. After we generated the original data, we rebuilt the contents of the data tables using the probability distribution given in Equation 7.5. Note that here, we used the primary key of the table as the skewness key (i.e., for the *dwdate* table, we used the *d_date* column as the skewness key). We used several values of s to observe how sensitive the physical designs were to a particular distribution of the data because we wanted to observe whether the physical design needed to be changed substantially when the underlying data changes.

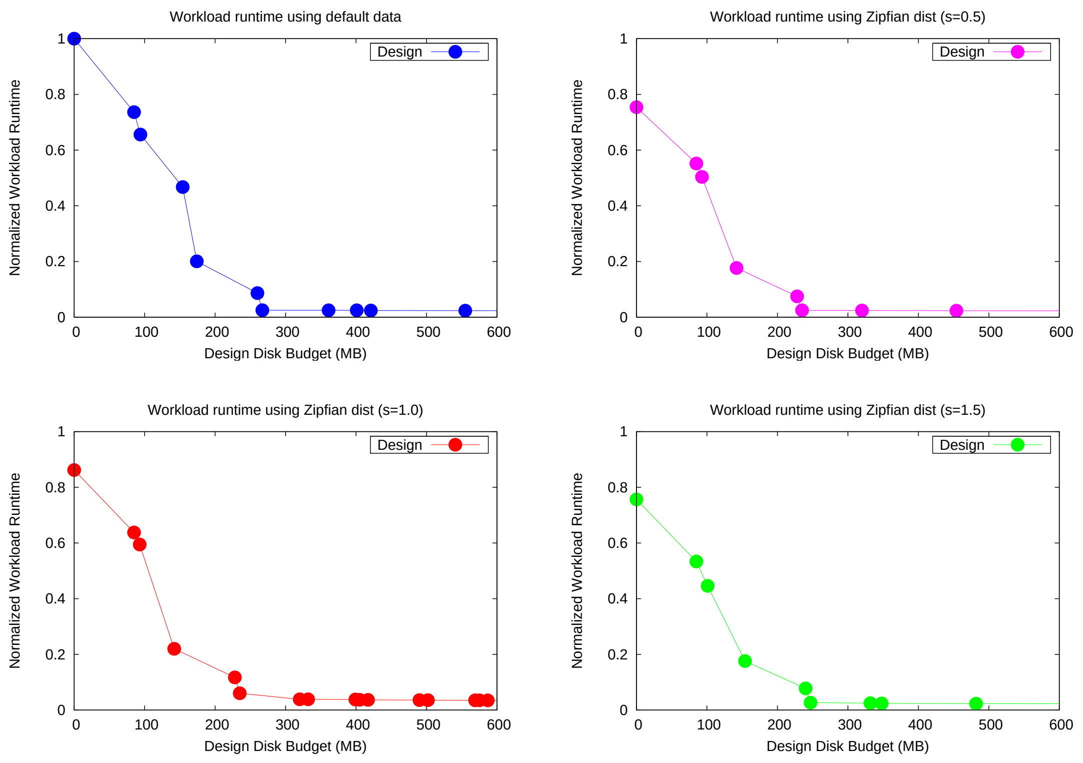


Figure 7.14: **Design Evaluation** – Changing the Skewness in the Data

Figure 7.14 presents the SSB workload curve using the original data set and the curves for three different values of the Zipfian parameter (s values of 0.5, 1.0 and 1.5), corresponding to different degree of skewness in the data (larger value implies heavier skewness). The curves in Figure 7.14 are consistent, exhibiting roughly the same performance improvement for all degrees of skewness. Note that we normalized the runtimes of all queries by the default design that uses the original data and that when the table data is skewed, the initial runtimes of the workload (at very low budgets) are somewhat faster (by approximately 15–20%) than are the original runtimes that use non-skewed data. This occurs because data skewness affects query selectivities and compression. Not all selectivities decrease, but the net effect is that the query workload selects fewer data rows and thereby runs slightly faster (some of the columns also compress better because of data skewness – primarily those encoded using dictionary compression). Finally, note that the difference caused by data skewness quickly shrinks and disappears for larger budgets, when all curves use more efficient designs.

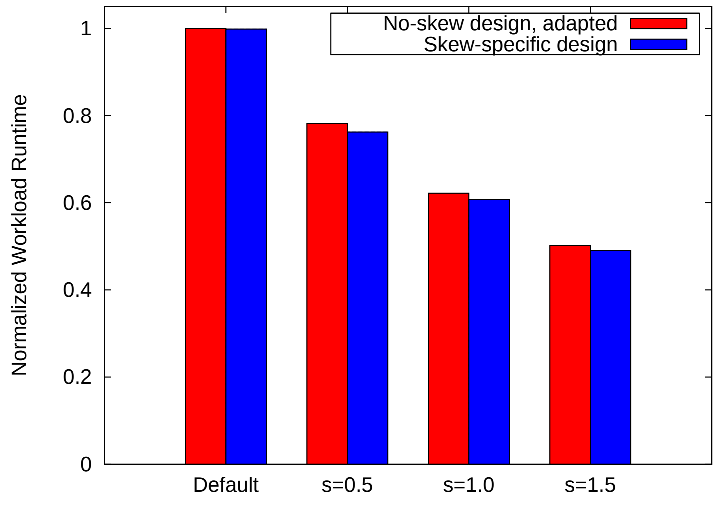


Figure 7.15: **One-MV Design Evaluation** – Skewness-specific vs. Default MV Design

Next, we evaluate the cross-skewness performance by picking a design point (we chose to measure the single MV design because it is the most complex) and examining the performance of the design built using the original data but used with the skewed data set. The goal of this experiment is to determine the sensitivity of the produced physical designs to skewness in the data. Figure 7.15 compares the normalized runtimes of the skewed and non-skewed designs. The left bar is the same one-MV design produced for the workload using the default data set; thus, the left design remains fixed in all cases (as the design tool was not made aware of the data skewness). As expected, the total workload runtime decreases as we increase skewness (this is caused by reduced selectivity and improved compression). The right set of bars in Figure 7.15 shows designs produced by using our tool, which has an awareness of the skewness. In that case, our design tool samples the data after it was skewed and recommends a design using accurate statistics. The first bar is the same MV design, since it still uses the default data. However, as we increase the value of s, our design tool recommends different MVs; not surprisingly, the skewness-aware designs perform a little better. However, the difference between the runtimes is almost negligible (we observed that the skewed MVs are also similar and that the leading columns in the sort order are identical). Therefore, these results suggest that our designs are insensitive to data skewness; however, there is one more kind of skewness to consider.

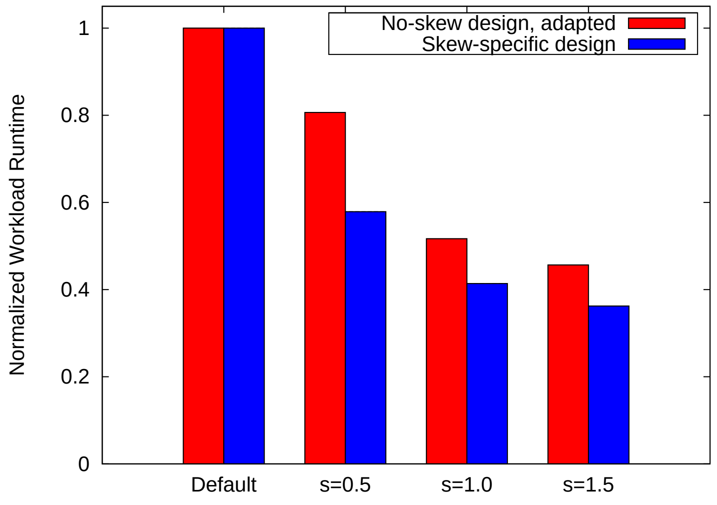


Figure 7.16: **One-MV Design Evaluation** – Skewness-specific and Default Designs (Targeted Skew)

Before we proceed to the next section, we will briefly revisit the question about the *kinds* of data skewness. Recall that in the previous experiment we used the table key as the skewness key. For example, a *dwdate* table was skewed according to the particular day. Such skewness changes the distribution of some, but not all, columns. For instance, the distribution of *d_year* or *d_daynuminyear* will change, because most values in the table will fall into the latter part of the original date (most dates would be within the last year or last couple of months of the original multi-year range). However, the distribution of a column such as *d_dayofweek* would not be affected because even if most dates fall within a couple of months, the distribution of weekdays remains unaffected. Thus, we introduce a more targeted skewness, specifically choosing the skewness keys using the columns that were selected for the original MV sort order (such as customer region).

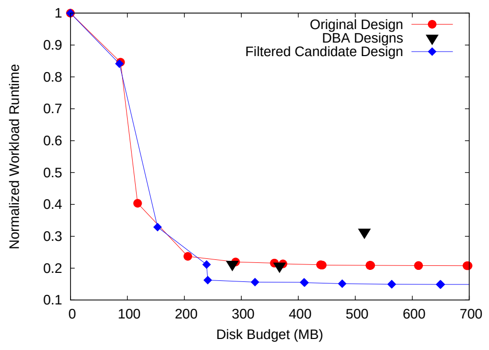
This new targeted skewness causes the design to change (sort orders are no longer similar to those in the previous experiment). We consider the same evaluation as that used in the previous experiment with the new skewed data. Thus, Figure 7.16 compares the original design and the designs produced for skewed data as in the previous experiment. In this case, we know that the difference between the MVs is more significant and that the performance difference is more pronounced, reaching differences of up to 40% between the original design and the skewness-specific design. Unsurprisingly, the resulting designs are more sensitive to targeted skewness, since it was chosen specifically by us to alter the workload performance. We can thus observe that designs are not sensitive to "generic" skewness (i.e., the kind that affects value distribution within some columns but not others, such as key-based skewness). However, if the skewness is specifically targeted to our queries (i.e., the data is skewed specifically towards the most popular query attributes), then the design is far more sensitive to skewness.

We conclude this section with an analysis of data skewness effect on distance measure between queries. We know that queries in the SSB workload were explicitly designed be easy to group. When we compared query grouping produced by vector-distance to the Jaccard-distance in an earlier experiment, we generated different (uniformly distributed, with no easily discernible sub-groups) query workloads. A new workload was necessary because SSB queries could be easily grouped using Jaccard-distance despite the fact that Jaccard-distance does not consider the underlying data at all (only query attributes are considered). In contrast to Jaccard-distance, the vector-distance measure is affected by the changes in the underlying data because the distance measure uses predicate selectivities. However, we make the following important observation: the improvement in the design (i.e., the steepness of the design curve) depends on how well the queries can be grouped. If we consider SSB performance (such as in Figure 7.12), we observe a sharp drop in the design curve. However, a design for a uniformly distributed query set that does not group well (Figure 7.9) or SSB workload that does not use query grouping (Figure 7.2) exhibits slow, gradual improvement. Whether we use skewed data or not, a query workload without natural groups will not result in rapidly improving designs. However, this is not the fault of the query grouping method; in the absence of large query sub-groups that share the same predicates or joins, there are no good MVs that can simultaneously speed up many queries (just like it is nearly impossible to speed up a query without predicates). Thus, a difficult-to-group query workload will not be affected by data skewness; in practice, such a query workload only results in small query groups that contribute to the design (i.e., we only find pairs or triplets of queries for which a good MV can be created). As a result, the task of the sort order designer is simplified, since the useful MV candidates are based on a small number of queries and relatively few predicates.

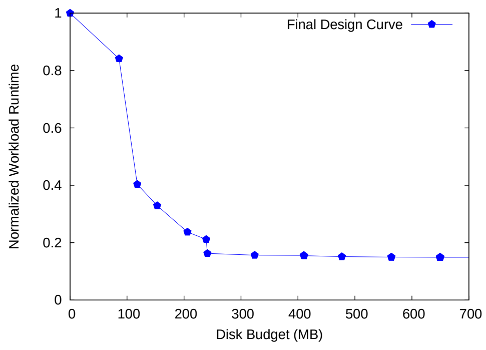
7.6 Update-aware Design

As explained earlier, when the query training set contains a significant amount of inserts, the design tool has to take potential costs into account. As discussed in Section 5.3, evaluating the cost of insert queries is a complex issue in a clustered column-store DBMS. Thus, special considerations are necessary in order to account for *INSERT* queries in the workload.

In Section 5.3, we detailed why the immediate cost of the insert comes from the cost incurred by performing the pre-joins needed by the candidate MVs and placing the new



(a) Design curves with different candidate filtering



(b) Combined design curve (best possible overall)

Figure 7.17: **Design Evaluation** – The Effect of the Insert Penalty on Overall Design

MV rows into the WOS. Therefore, the choice of design candidates will determine the insert penalties incurred by the design. In particular, the presence of certain pre-joins (e.g., those with larger dimension tables) increases the insert penalty and thereby the overall query workload runtime. Note that this is one of those places where we need to ascertain the query cycle frequency (see Section 1.2.3), which determines how often the query workload is executed. As stated in earlier sections, we assume an insert rate of 1000 rows per second and batching of 10K rows (i.e., every 10 seconds). We further assume that our *SELECT* query workload arrives every two minutes; thus, on average, 12 batches of 10K row batches arrive for a single execution of the *SELECT* query workload. Figure 7.17 outlines the runtimes of different design options. The first curve in Figure 7.17(a) shows the design curves of the insert-oblivious design (the design in Figure 7.10(a) that reflects the newly introduced insert penalty at every design point). For a point of reference, we also show the performance of the DBA's designs that incorporate the combined workload cost. The second curve in Figure 7.17(a) shows a design built using the same LP formulation, but chosen by considering the partitioned MV candidate pool as described in Section 7.2.3. Note that the second curve outperforms the original design (not surprisingly, as the original design is unaware of the insert overheads). In Figure 7.17(b), we show the combined curve that would be returned to the user (i.e., the curve that chooses the best design at every budget point).

The additional insert penalty depends on the insert rate. Thus, the original and the partitioned-MV-pool approach will differ accordingly. In this experiment, we specifically chose an insert batch frequency that keeps the design problem interesting. At a very low insert rate, the insert penalty can be safely be ignored. At a very high insert rate, we might be unable to sustain any pre-joined MVs, thereby eliminating the bulk of the MV candidate pool. In the following section, we discuss ways in which our design tool can exploit MV replication.

7.7 Replicated Design

In this section, we discuss the opportunities for improving the replicated physical design. As explained in Section 2.6.4, Vertica uses MV replication to ensure recovery in case of data loss. This approach is common in many DBMSs; most databases support recovery through a combination of logging and data replication. However, in Vertica the need for replicated MVs can be exploited to improve the physical design. We previously presented our approach to generating replicated MVs in Section 6.4.

Recall that the idea behind replicated design is to introduce an additional copy of every MV into the physical design so that lost data can be restored from that spare copy. In practice, replication is achieved by creating an additional copy of every MV for every increment of the K-safety factor (i.e., one additional copy for K=1, two additional copies for K=2, etc.). The easiest way to achieve this is to copy the original *0-safe* design produced by our design tool or by the DBA. In fact, this approach is used by DBAs and it reliably produces a replicated design. However, we can achieve better performance by generating two (or K+1) independent MVs, each with its own sort order. The sort orders can be created independently, because the hash-partitioning function that distributes MVs across multiple machines is orthogonal to the sort order of the MV.

First, let us briefly outline of our assumptions regarding building and using a K-safe physical design. In particular, we want to list the potential downsides of using custom K-safe MVs that we suggest. The following events occur when a failure is detected in Vertica (assuming K=1, but the idea can be easily extended to larger values):

- The DBMS begins recovering lost data from the surviving replica MVs. Note that because of MV fragmentation no single MV is entirely lost after failure, but rather, multiple fragments from different MVs are lost.
- While the recovery is in progress, some of the replicated fragments are unavailable. Therefore, one of the two copies is unavailable (either $\text{MV}_{Replica\#1}$ or $\text{MV}_{Replica\#2}$).
- As a result, the current K value drops by one until the recovery phase is complete (e.g., $K=1 \rightarrow K=0$).

The recovery process for any design suffers from the effects described above. However, there is an important implication that is unique to the custom K-safe design we are proposing. In our approach, the MV replicas are not identical because they have different sort orders. As a result, the design performance *during* the recovery phase may deteriorate significantly. Different replicas serve different subsets of the user query groups. Therefore, for example, if $\text{MV}_{Replica\#2}$ becomes unavailable for the duration of the recovery, then all user queries that relied on $\text{MV}_{Replica\#2}$ will have to use another auxiliary structure and thereby performance could deteriorate significantly. Thus, our approach should only be used when node failures occur infrequently. In an environment where machines fail often, the design tool would have to incorporate the failure-imposed query penalty into its design choices or rely on the conventionally replicated design, which does not suffer from the problem described earlier.

Next, we present the advantages of the 1-safe design. We also note that creating custom K-safe candidates does not guarantee an improvement in the resulting design. Fortunately, we can always fall back on the simple replicated design. The reason 1-safe candidates may not always be beneficial is, as always, compression. In Vertica, the total size of two different

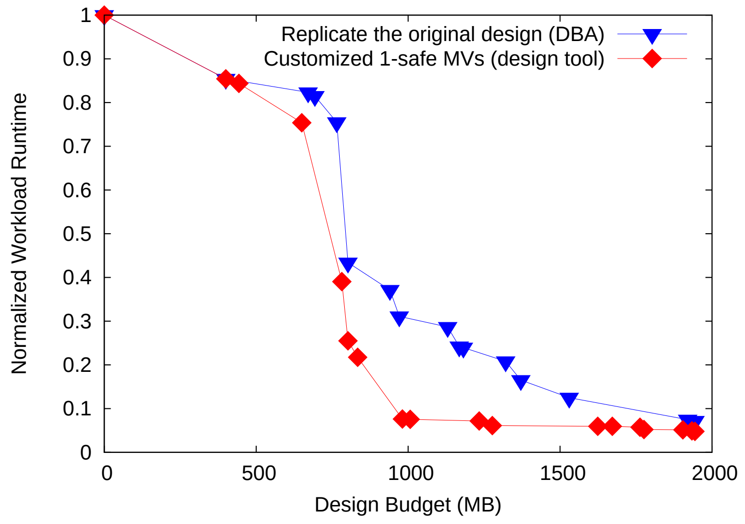


Figure 7.18: **Design Evaluation** – Simple Replicated (DBA) Design vs. Custom 1-safe Design

MV replicas may end up being larger than the simple duplication of the original 0-safe MV. In fact, we observed that there was little benefit in using this technique on the SSB workload for that reason. Therefore, we illustrate our replication approach by using SSB_EXT, which is the SSB query workload extended by additional columns.

Figure 7.18 shows two design curves: one (blue triangles) produced with a regular design that has been duplicated to achieve 1-safety and another (red squares) built using additional custom 1-safe MVs, which are described in Section 6.4. Unsurprisingly, there is no immediate benefit for very small budgets because the 1-safe MVs compress poorly (the sort orders are designed for query subgroups as discussed in Section 6.4, and, as a result, the remainder of the columns that belong to the second query subgroup compress poorly). Beginning from a medium-sized disk budget, the 1-safe candidates start to improve the overall workload runtime. Our 1-safe design curve performs much better than does the simple replicated design starting from a budget of 1000 MB.

In conclusion, it is possible to exploit the replication factor and generate better designs, as long as we understand the penalties that might be incurred during the DBMS recovery phase.

Chapter 8

Conclusions and Future Work

In this dissertation, we have described a comprehensive solution for automating the physical database design process in a clustered column store (i.e., a column store that supports clustered indexes). Despite the considerable body of work concerning various aspects of the problems of physical design in row-store DBMSs [LLZ02, ANY04, GM99, BC05, PA04], insufficient attention had been paid to the topic of physical design in the context of column stores. The majority of published research (which has primarily been based on different versions of MonetDB [Mon]) has studied the columnar database design problem in an environment where data clustering is not supported (with the exception of [HZN+10]). Excluding support for clustering keys from the DBMS simplifies the implementation of the DBMS engine. To support clustered auxiliary structures, DBMSs need to maintain data ordering in the presence of arriving updates. The majority of row-store DBMSs provide support for clustered indexes [sqlb, DB2, Orad, MyS], even though the costs of maintaining the design can quickly escalate because of high update rates [KHR+10]. Some row-store DBMSs have chosen to forego the extra work of maintaining data clustering. For example, PostreSQL [pos] only sorts data when the user explicitly requests it to do so. In a column-store setting, this situation is reversed: most column-store DBMSs do not support clustering indexes [Mon, Gre, Syb], whereas some provide clustered index feature [Ver].

The most significant advantage of not committing to keeping data clustered is the greatly simplified process of updating the auxiliary structures within the design. New data can simply be appended to the database storage as it arrives [pos, Mon]. However, in the absence of data clustering, DBMSs also forego a number of significant benefits. In any DBMS, clustering data allow speeds up user queries by collocating the queried data on the hard disk. Some row-store DBMSs have also used this as an opportunity to implement compression techniques that benefit from clustering [PP03]. A column store, however, provides many more opportunities to achieve high levels of compression, resulting in significantly improved query runtime and reduced auxiliary structure size. Although few of the column-store DBMSs support data clustering [Ver, Gre, HZN+10], we believe that providing clustered indexes in a column store is worth the effort necessary to implement them.

Throughout this dissertation we have demonstrated that clustered column stores provide many opportunities for aggressive data compression and result in significant acceleration of query runtimes. Moreover, although the presence of updates typically creates a maintenance challenge for data warehouses (particularly in column stores), we have shown how a database design tool can produce designs tailored to accommodate a significant insert rate.

The conceptual structure of the physical design process in a column-store DBMS is similar to the conventional approaches taken by row-store design tools. Briefly, the database design tool generates several candidate auxiliary structures (derived from the user-specified query workload). Then, the designer selects a subset of these structures with the aim of minimizing the query workload runtime without violating user budget constraints (i.e., disk space). Finally, there is an iterative improvement phase, based on either improving the the observed design improvement and limited by the time time allotted for generating the design. The majority of the analysis presented in this dissertation has focused on the issues of creating, evaluating and estimating the sizes of candidate MVs for the design. Although choosing the right subset from the available candidate MVs is important, the linear programming solution ([PA04], [KHR+10]) provides a way to select the optimal candidate subset in most cases. Therefore, the problem of producing a good physical design hinges on generating a suitable set of design candidates to supply accurate inputs to the LP Solver, from which a good design may be found.

The problem of generating design candidates requires a different approach in clustered column-store databases because they deviate from a number of the rules established in the row-store DBMS environment. Specifically, the problem of generating dedicated (single-query) MV candidates and merging them into shared MV candidates is well understood in a row store (described in [CN99]). As we have shown, the same problems in a clustered column store have to be approached in a different manner because of the fundamental architectural differences between row and column stores. Several major factors contribute to these differences. First, that the column-store storage environment both allows (and requires that) queries access data columns individually. However, even more significant is the presence of compression in the column-store DBMS engine. Compression permeates every decision: it allows us to reduce the sizes of the design candidates, thereby saving disk space and accelerating the response times of user queries. Vertica's capacity for operating directly on compressed data further increases the influence of compression on the performance of every MV.

Most design decisions in a clustered column store are affected by the (potential) presence of correlation in data. Although correlation has been recognized as an important influence on query execution (join size estimation in [IC91, IMH+04]) and for its potential to improve query execution in a row store (CMs [KHR+09] in CORADD [KHR+10]), accounting for data correlation is much more important in a clustered column store. Correlations have to be considered when selecting a sort order for the MV, when selecting individual column compression schemes, and when estimating the runtimes of user queries and the sizes of the MVs. As we have shown, ignoring correlations causes errors in all aspects of the cost estimation process during the design phase. In addition to correlation, we have also argued that generating good design candidates needs a query-grouping mechanism. The traditional row-store approach of merging MVs to produce additional candidate MVs is prone to errors in the clustered column store environment. Thus, we use query grouping to group similar queries and then use these groups as the basis for MV candidates. Representing queries as attribute vectors for the grouping phase allows us to incorporate correlations into the similarity metrics.

Moreover, the query-grouping process serves as a natural basis for producing improved K-safe candidate MVs. Replication designs rely on exploiting the (required) additional copies of the MVs, and the measurement of query distances can be used to partition a query group into sub-groups in order to generate customized K-safe MVs. More generally, the observed affinity among queries determines the overall quality of the design. As we have shown, a query workload where a “natural” close grouping exists results in “steep” design curves where the total workload runtime improves rapidly at relatively small disk budgets.

Finally, we have presented a framework for accurately incorporating the cost of updates incurred in this setting. The insert-amortizing mechanism used by Vertica (and recommended for use by other clustered column stores) almost eliminates the link between the insert cost and the design size. Thus, the insert cost instead primarily depends on the cost of producing the pre-joins of the MV. As a result we have altered the existing LP solver formulation (by partitioning MV candidates) to allow us to select the optimal design that correctly accounts for the inserts present in the workload.

This work can be extended in a number of ways. Although not fundamental, supporting other types of column encoding or additional indexing structures could further improve the performance of a clustered column store such as Vertica even further. Bitmap index support would be an interesting alternative to using delta and dictionary compression. Vertica processes columns by representing currently selected rows using a bitmap (and uses bitmaps to represents deletes), so a bitmap index can be easily incororated into all aspects of query processing. A clustered column store might also greatly benefit from implementing a CM-like index structure [KHR+09]. Note that these indexes, by definition, require support for clustering keys (since they map unsorted values into positions within the clustering key) and hence can only be used in a clustered DBMS.

Another promising direction for this work is the idea of *incremental design*. That is, instead of producing a design starting from a budget of zero, we could consider migrating from one design (with a non-zero budget) to another design through a sequence of "add an MV" and "delete an MV" operations. This problem is conceptually similar to the regular design problem because design process can ignore the existing (non-default) design (i.e., proceed to generate a design by only adding MVs to a design that is already deployed). However, we think that there may be promise in applying query grouping to identify a better solution (i.e., achieving a design that is either faster for the same disk budget or that can be deployed faster than can the conventional one) to the incremental design problem. The intuition for finding the right sequence of MVs to add or remove is to identify existing MVs that serve queries that are similar to queries that exhibit poor performance in the current design (such queries are targeted by the incremental design process for improvement). Thus, suppose that we have query Q_a that is served by MV_a and a query Q_s that runs very slowly. It is true that dropping MV_a would cause Q_a to run slower as well. It will also, however, free up the disk space that MV_a was using. Next, if Q_a and Q_s were similar, we could then design and deploy a new MV_{as} that would serve to speed up both Q_a and Q_s while not using little more disk space than MV_a originally used. It is precisely this kind of detection that query distance functions are already being used for in our work.

Appendix A

SSB Benchmark

For the reader's convinience, we include the tables and the SSB sample queries in the appendix.

```
create table part (
  p_partkey      integer        not null,
  p_name         varchar(22)    not null,
  p_mfgr         varchar(6)     not null,
  p_category     varchar(7)     not null,
  p_brand1       varchar(9)     not null,
  p_color        varchar(11)    not null,
  p_type         varchar(25)    not null,
  p_size         integer        not null,
  p_container    varchar(10)    not null
);

create table supplier (
  s_suppkey      integer        not null,
  s_name         varchar(25)    not null,
  s_address      varchar(25)    not null,
  s_city         varchar(10)    not null,
  s_nation       varchar(15)    not null,
  s_region       varchar(12)    not null,
  s_phone        varchar(15)    not null
);

create table customer (
  c_custkey      integer        not null,
  c_name         varchar(25)    not null,
  c_address      varchar(25)    not null,
  c_city         varchar(10)    not null,
  c_nation       varchar(15)    not null,
  c_region       varchar(12)    not null,
  c_phone         varchar(15)    not null,
  c_mktsegment varchar(10)    not null
);

create table dwdate (
  d_datekey              integer          not null,
  d_date                 varchar(19)      not null,
  d_dayofweek            varchar(10)      not null,
  d_month                varchar(10)      not null,
  d_year                 integer          not null,
  d_yearmonthnum         integer          not null,
  d_yearmonth            varchar(8)       not null,
  d_daynuminweek         integer          not null,
  d_daynuminmonth        integer          not null,
  d_daynuminyear         integer          not null,
  d_monthnuminyear       integer          not null,
  d_weeknuminyear        integer          not null,
  d_sellingseason        varchar(13)      not null,
  d_lastdayinweekfl      varchar(1)       not null,
  d_lastdayinmonthfl     varchar(1)       not null,
  d_holidayfl            varchar(1)       not null,
  d_weekdayfl            varchar(1)       not null
);

create table lineorder (
  lo_orderkey            integer         not null,
  lo_linenumber          integer         not null,
  lo_custkey             integer         not null,
  lo_partkey             integer         not null,
  lo_suppkey             integer         not null,
  lo_orderdate           integer         not null,
  lo_orderpriority       varchar(15)     not null,
  lo_shippriority        varchar(1)      not null,
  lo_quantity            integer         not null,
  lo_extendedprice       integer         not null,
  lo_ordertotalprice     integer         not null,
  lo_discount            integer         not null,
  lo_revenue             integer         not null,
  lo_supplycost          integer         not null,
  lo_tax                 integer         not null,
  lo_commitdate          integer         not null,
  lo_shipmode            varchar(10)     not null
);

alter table part add primary key (p_partkey);
alter table supplier add primary key (s_suppkey);

alter table customer add primary key (c_custkey);

alter table dwdate add primary key (d_datekey);

alter table lineorder
  add primary key (lo_orderkey, lo_linenumber);
alter table lineorder
  add constraint custconstr foreign key (lo_custkey)
  references customer(c_custkey);
alter table lineorder
  add constraint partconstr foreign key (lo_partkey)
  references part(p_partkey);
alter table lineorder
  add constraint suppconstr foreign key (lo_suppkey)
  references supplier(s_suppkey);
alter table lineorder
  add constraint dateconstr foreign key (lo_orderdate)
  references dwdate(d_datekey);

-- Q1.1
select sum(lo_extendedprice*lo_discount) as
revenue
from lineorder, dwdate
where lo_orderdate = d_datekey
and d_year = 1993
and lo_discount between 1 and 3
and lo_quantity < 25;

-- Q1.2
select sum(lo_extendedprice*lo_discount) as
revenue
from lineorder, dwdate
where lo_orderdate = d_datekey
and d_yearmonthnum = 199401
and lo_discount between 4 and 6
and lo_quantity between 26 and 35;

-- Q1.3
select sum(lo_extendedprice*lo_discount) as
revenue
from lineorder, dwdate
where lo_orderdate = d_datekey
```

```
and d_weeknuminyear = 6
and d_year = 1994
and lo_discount between 5 and 7
and lo_quantity between 26 and 35;

-- Q2.1
select sum(lo_revenue), d_year, p_brand1
from lineorder, dwdate, part, supplier
where lo_orderdate = d_datekey
and lo_partkey = p_partkey
and lo_suppkey = s_suppkey
and p_category = 'MFGR#12'
and s_region = 'AMERICA'
group by d_year, p_brand1
order by d_year, p_brand1;

-- Q2.2
select sum(lo_revenue), d_year, p_brand1
from lineorder, dwdate, part, supplier
where lo_orderdate = d_datekey
and lo_partkey = p_partkey
and lo_suppkey = s_suppkey
and p_brand1 between 'MFGR#2221' and 'MFGR#2228'
and s_region = 'ASIA'
group by d_year, p_brand1
order by d_year, p_brand1;

-- Q2.3
select sum(lo_revenue), d_year, p_brand1
from lineorder, dwdate, part, supplier
where lo_orderdate = d_datekey
and lo_partkey = p_partkey
and lo_suppkey = s_suppkey
and p_brand1= 'MFGR#2239'
and s_region = 'EUROPE'
group by d_year, p_brand1
order by d_year, p_brand1;

-- Q3.1
select c_nation, s_nation, d_year,
sum(lo_revenue) as revenue
from customer, lineorder, supplier, dwdate
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_orderdate = d_datekey
and c_region = 'ASIA'
and s_region = 'ASIA'
and d_year >= 1992 and d_year <= 1997
group by c_nation, s_nation, d_year
order by d_year asc, revenue desc;

-- Q3.2
select c_city , s_city , d_year, sum(lo_revenue)
as revenue
from customer, lineorder, supplier , dwdate
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_orderdate = d_datekey
and c_nation = 'UNITED_STATES'
and s_nation = 'UNITED_STATES'
and d_year >= 1992 and d_year <= 1997
group by c_city, s_city , d_year
order by d_year asc, revenue desc;

-- Q3.3
select c_city , s_city , d_year, sum(lo_revenue)
as revenue
from customer, lineorder, supplier , dwdate
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_orderdate = d_datekey
and (c_city='UNITED_KI1' or c_city='UNITED_KI5')
and (s_city='UNITED_KI1' or s_city='UNITED_KI5')
and d_year >= 1992 and d_year <= 1997
group by c_city, s_city , d_year
order by d_year asc, revenue desc;

-- Q3.4
select c_city , s_city , d_year, sum(lo_revenue)
as revenue
from customer, lineorder, supplier , dwdate
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_orderdate = d_datekey
and (c_city='UNITED_KI1' or c_city='UNITED_KI5')
and (s_city='UNITED_KI1' or s_city='UNITED_KI5')
and d_yearmonth = 'Dec1997'
group by c_city, s_city , d_year
order by d_year asc, revenue desc;
```

```
-- Q4.1
select d_year, c_nation,
sum(lo_revenue - lo_supplycost) as profit
from dwdate, customer, supplier, part, lineorder
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_partkey = p_partkey
and lo_orderdate = d_datekey
and c_region = 'AMERICA'
and s_region = 'AMERICA'
and (p_mfgr = 'MFGR#1' or p_mfgr = 'MFGR#2')
group by d_year, c_nation
order by d_year, c_nation;

-- Q4.2
select d_year, s_nation, p_category,
sum(lo_revenue - lo_supplycost) as profit
from dwdate, customer, supplier, part, lineorder
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_partkey = p_partkey
and lo_orderdate = d_datekey
and c_region = 'AMERICA'
and s_region = 'AMERICA'
and (d_year = 1997 or d_year = 1998)
and (p_mfgr = 'MFGR#1'
or p_mfgr = 'MFGR#2')
group by d_year, s_nation, p_category
order by d_year, s_nation, p_category;

-- Q4.3
select d_year, s_city , p_brand1,
sum(lo_revenue - lo_supplycost) as profit
from dwdate, customer, supplier, part, lineorder
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_partkey = p_partkey
and lo_orderdate = d_datekey
and s_nation = 'UNITED_STATES'
and (d_year = 1997 or d_year = 1998)
and p_category = 'MFGR#14'
group by d_year, s_city, p_brand1
order by d_year, s_city , p_brand1;
```

Bibliography

[AAD$^+$96] Sameet Agarwal, Rakesh Agrawal, Prasad Deshpande, Ashish Gupta, Jeffrey F. Naughton, Raghu Ramakrishnan, and Sunita Sarawagi. On the Computation of Multidimensional Aggregates. In *VLDB*, pages 506–521, 1996.

[ABCN06] Sanjay Agrawal, Nicolas Bruno, Surajit Chaudhuri, and Vivek R. Narasayya. AutoAdmin: Self-Tuning Database SystemsTechnology. *IEEE Data Eng. Bull*, 29(3):7–15, 2006.

[ACK$^+$04] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollár, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. Database Tuning Advisor for microsoft SQL Server 2005. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *VLDB*, pages 1110–1121. Morgan Kaufmann, 2004.

[ACN00] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors, *VLDB*, pages 496–505. Morgan Kaufmann, 2000.

[AMF06] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, editors, *SIGMOD Conference*, pages 671–682. ACM, 2006.

[AMH08] Daniel J. Abadi, Samuel Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In Jason Tsong-Li Wang, editor, *SIGMOD Conference*, pages 967–980. ACM, 2008.

[ANY04] Sanjay Agrawal, Vivek R. Narasayya, and Beverly Yang. Integrating Vertical and Horizontal Partitioning Into Automated Physical Database Design. In *SIGMOD Conference*, pages 359–370, 2004.

[Aut] AutoAdmin Project. http://research.microsoft.com/en-us/projects/autoadmin/defaul

[BC74] R. F. Boyce and D. D. Chamberlin. SEQUEL: A Structured English Query Language. In *ACM SIGMOD*, May 1974.

[BC05] Nicolas Bruno and Surajit Chaudhuri. Automatic Physical Database Tuning: A Relaxation-based Approach. In *SIGMOD Conference*, pages 227–238, 2005.

[BH] Paul Brown and Peter Haas. BHUNT: Automatic Discovery of Fuzzy Algebraic Constraints in Relational Data. In *Proc. of VLDB 2003*, Berlin, Germany.

[BH03] Paul Brown and Peter J. Haas. BHUNT: Automatic Discovery of Fuzzy Algebraic Constraints in Relational Data. In *VLDB*, pages 668–679, 2003.

[bit] Bitmap Index. http://en.wikipedia.org/wiki/Bitmap_index.

[BZN05] Peter A. Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, pages 225–237, 2005.

[CCMN00] Moses Charikar, Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. Towards Estimation Error Guarantees for Distinct Values. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS-00)*, pages 268–279, N. Y., May 15–17 2000. ACM Press.

[CD97] Surajit Chaudhuri and Umeshwar Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Record*, 26(1):65–74, 1997.

[CGN02] Surajit Chaudhuri, Ashish Kumar Gupta, and Vivek R. Narasayya. Compressing SQL workloads. In Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki, editors, *SIGMOD Conference*, pages 488–499. ACM, 2002.

[CN98a] Surajit Chaudhuri and Vivek Narasayya. AutoAdmin "what-if" index analysis utility. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 27(2):367–378, June 1998.

[CN98b] Surajit Chaudhuri and Vivek R. Narasayya. Microsoft Index Tuning Wizard for SQL Server 7.0. In *SIGMOD Conference*, pages 553–554, 1998.

[CN99] Surajit Chaudhuri and Vivek R. Narasayya. Index Merging. In *ICDE*, pages 296–303, 1999.

[Cod71] E. F. Codd. Further Normalization of the Data Base Relational Model. *IBM Research Report, San Jose, California*, RJ909, August 1971.

[Com79] Comer, D. The ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[COO08] Xuedong Chen, Patrick E. O'Neil, and Elizabeth J. O'Neil. Adjoined Dimension Column Clustering to Improve Data Warehouse Query Performance. In *ICDE*, pages 1409–1411. IEEE, 2008.

[cpl] IBM ILOG CPLEX Optimizer. http://www-01.ibm.com/software/integration/optimizat

[CS95] Surajit Chaudhuri and Kyuseok Shim. An Overview of Cost-based Optimization of Queries with Aggregates. *IEEE Data Eng. Bull.*, 18(3):3–9, 1995.

[CS96] Surajit Chaudhuri and Kyuseok Shim. Optimizing Queries with Aggregate Views. In *EDBT*, pages 167–182, 1996.

[CSt] C-Store project. http://db.csail.mit.edu/projects/cstore/.

[DB2] IBM DB2. http://www-01.ibm.com/software/data/db2/.

[DBB07] Jérôme Darmont, Fadila Bentayeb, and Omar Boussaid. Benchmarking data warehouses. *IJBIDM*, 2(1):79–104, 2007.

[GGZ] Parke Godfrey, Jarek Gryz, and Calisto Zuzarte. Exploiting Constraint-Like Data Characterizations in Query Optimization. pages 582–592.

[Gib] Phillip B. Gibbons. Distinct Sampling for Highly-Accurate Answers to Distinct Values Queries and Event Reports. In *Proc. of VLDB 2001*, Roma, Italy.

[GM93] Goetz Graefe and William J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *ICDE*, pages 209–218, 1993.

[GM99] Himanshu Gupta and Inderpal Singh Mumick. Selection of Views to Materialize Under a Maintenance Cost Constraint. In *ICDT*, pages 453–470, 1999.

[Gre] Greenplum. http://www.greenplum.com/.

[GSW98] Ashish Gupta, Oded Shmueli, and Jennifer Widom, editors. *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*. Morgan Kaufmann, 1998.

[GSZZ01] J. Gryz, B. Schiefer, J. Zheng, and C. Zuzarte. Discovery and Application of Check Constraints in DB2. In *17th International Conference on Data Engineering (ICDE' 01)*, pages 551–556, Washington - Brussels - Tokyo, April 2001. IEEE.

[HDD] Hard Disk Drive. http://en.wikipedia.org/wiki/Hard_disk_drive.

[HRU96] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing Data Cubes Efficiently. In *SIGMOD Conference*, pages 205–216, 1996.

[Hus] Ahsan Hussain. Query Generator. http://cs.brown.edu/research/pubs/theses/masters/2

[HZN+10] Sándor Héman, Marcin Zukowski, Niels J. Nes, Lefteris Sidirourgos, and Peter A. Boncz. Positional update handling in column stores. In *SIGMOD Conference*, pages 543–554, 2010.

[IC91] Yannis E. Ioannidis and Stavros Christodoulakis. On the Propagation of Errors in the Size of Join Results. In *SIGMOD Conference*, pages 268–277, 1991.

[IKM07a] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database Cracking. In *CIDR*, pages 68–78. www.crdrdb.org, 2007.

[IKM07b] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Updating a cracked database. In Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou, editors, *SIGMOD Conference*, pages 413–424. ACM, 2007.

[IMH+04] Ihab F. Ilyas, Volker Markl, Peter Haas, Paul Brown, and Ashraf Aboulnaga. Cords: Automatic discovery of correlations and soft functional dependencies. In *In SIGMOD*, pages 647–658, 2004.

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

[Jac] Jaccard Index. http://en.wikipedia.org/Jaccard_index.

[KHR+09] Hideaki Kimura, George Huo, Alexander Rasin, Samuel Madden, and Stanley B. Zdonik. Correlation Maps: A Compressed Access Method for Exploiting Soft Functional Dependencies. *PVLDB*, 2(1):1222–1233, 2009.

[KHR+10] Hideaki Kimura, George Huo, Alexander Rasin, Samuel Madden, and Stanley 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.

[KM05] Martin L. Kersten and Stefan Manegold. Cracking the Database Store. In *CIDR*, pages 213–224, 2005.

[LLZ02] Sam Lightstone, Guy M. Lohman, and Daniel C. Zilio. Toward Autonomic Computing with DB2 universal database. *SIGMOD Record*, 31(3):55–61, 2002.

[LSS] Large Synoptic Sky Telescope. http://www.lsst.org/lsst.

[ML86] Lothar F. Mackert and Guy M. Lohman. R* Optimizer Validation and Performance Evaluation for Local Queries. In *SIGMOD Conference*, pages 84–95, 1986.

[MLR03] Volker Markl, Guy M. Lohman, and Vijayshankar Raman. LEO: An autonomic query optimizer for DB2. *IBM Systems Journal*, 42(1):98–106, 2003.

[Mon] MonetDB. http://monetdb.cwi.nl/.

[MPK00] Stefan Manegold, Arjan Pellenkoft, and Martin L. Kersten. A Multi-query Optimizer for Monet. In *BNCOD*, pages 36–50, 2000.

[MS03] S. Muthukrishnan and Martin Strauss. Maintenance of Multidimensional Histograms. In *FSTTCS*, pages 352–362, 2003.

[MyI] MyISAM Storage Engine. http://dev.mysql.com/doc/refman/5.0/en/myisam-storage-eng

[MyS] MySQL Database. http://www.mysql.com/.

[Oraa] http://www.oracle.com/technology/obe/11gr1_db/perform/multistats/multicolstats

[Orab] http://download.oracle.com/docs/cd/B14117_01/server.101/b10752/optimops.htm#51

[Orac] http://download.oracle.com/docs/cd/B19306_01/server.102/b14200/statements_5010

[Orad] http://www.oracle.com/technetwork/database/features/index.html.

[PA04] Stratos Papadomanolakis and Anastassia Ailamaki. AutoPart: Automating Schema Design for Large Scientific Databases Using Data Partitioning. In *SSDBM*, pages 383–392. IEEE Computer Society, 2004.

[PA07] Stratos Papadomanolakis and Anastassia Ailamaki. An Integer Linear Programming Approach to Database Design. In *ICDE Workshops*, pages 442–449. IEEE Computer Society, 2007.

[pos] PostreSQL. http://www.postgresql.org/.

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

[PP03] Meikel Pöss and Dmitry Potapov. Data Compression in Oracle. In *VLDB*, pages 937–947, 2003.

[PR03] Paritosh K. Pandya and Jaikumar Radhakrishnan, editors. *FST TCS 2003: Foundations of Software Technology and Theoretical Computer Science, 23rd Conference, Mumbai, India, December 15-17, 2003, Proceedings*, volume 2914 of *Lecture Notes in Computer Science*. Springer, 2003.

[RAI] Redundant Array of Inexpensive Disks. http://en.wikipedia.org/wiki/RAID.

[SAB$^+$05] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-Store: A Column-oriented DBMS. In Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi, editors, *VLDB*, pages 553–564. ACM, 2005.

[SAD$^+$10] Michael Stonebraker, Daniel J. Abadi, David J. DeWitt, Samuel Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. MapReduce and parallel DBMSs: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.

[SDS] The Sloan Digital Sky Survey. http://www.sdss.org/.

[SKS02] A. Silberschatz, H.F. Korth, and S. Sudershan. *Database System Concepts*. McGraw-Hill, Inc. New York, NY, USA, 6th edition, 2002.

[SQLa] http://msdn.microsoft.com/en-us/library/ms177416(SQL.90).aspx.

[sqlb] Microsoft SQL Server 2008. http://www.microsoft.com/sqlserver/2008/en/us/.

[Syb] Sybase IQ. http://www.sybase.com/products/datawarehousing/sybaseiq.

[TPC] TPCH Benchmark. http://www.tpc.org/tpch/.

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

[VZZ$^+$00] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy M. Lohman, and Alan Skelley. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *ICDE*, pages 101–110, 2000.

[YYTM10] Christopher Yang, Christine Yen, Ceryen Tan, and Samuel Madden. Osprey: Implementing MapReduce-style fault tolerance in a shared-nothing distributed database. In *ICDE*, pages 657–668, 2010.

[ZIP] Zipfs Law. http://en.wikipedia.org/Zipf's_law.

[ZL77a] Ziv and Lempel. A Universal Algorithm for Sequential Data Compression. *IEEETIT: IEEE Transactions on Information Theory*, 23, 1977.

[ZL77b] Jacob Ziv and Abraham Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[ZRL$^+$04a] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy M. Lohman, Adam J. Storm, Christian Garcia-Arellano, and Scott Fadden. DB2 Design Advisor: Integrated Automatic Physical Database Design. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *VLDB*, pages 1087–1097. Morgan Kaufmann, 2004.

[ZRL$^+$04b] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy M. Lohman, Adam J. Storm, Christian Garcia-Arellano, and Scott Fadden. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *VLDB*, pages 1087–1097, 2004.

[ZZL$^+$04] Daniel C. Zilio, Calisto Zuzarte, Sam Lightstone, Wenbin Ma, Guy M. Lohman, Roberta Cochrane, Hamid Pirahesh, Latha S. Colby, Jarek Gryz, Eric Alton, Dongming Liang, and Gary Valentin. Recommending Materialized Views and Indexes with IBM DB2 Design Advisor. In *ICAC*, pages 180–188. IEEE Computer Society, 2004.