The Vertica Query Optimizer: The Case for Specialized Query Optimizers

Nga Tran [#1], Andrew Lamb [*2], Lakshmikant Shrinivas [*3], Sreenath Bodagala [#4], Jaimin Dave [#5]

[#] *HP Vertica*
Cambirdge, Massachusetts, USA
[1] ntran@vertica.com
[4] sbodagala@vertica.com
[5] jdave@vertica.com

[*] *Nutonian*
Cambirdge, Massachusetts, USA
[2] andrew@nerdnetworks.org
[3] lshrinivas@nutonian.com

***Abstract*—The Vertica SQL Query Optimizer was written from the ground up for the Vertica Analytic Database. Its design and the tradeoffs we encountered during its implementation argue that the full power of novel database systems can only be realized with a carefully crafted custom Query Optimizer written specifically for the system in which it operates.**

I. Introduction

The Vertica Analytic Database (Vertica) [20] is a modern, commercially successful RDBMS. It contains a SQL query optimizer, written from scratch purposely for the Vertica Storage System and Execution Engine. We purposely chose to write our own optimizer despite a countervailing trend in the industry to reuse or wrap existing optimizers [7], [25] in new database systems.

The choice to write an optimizer from scratch was not taken lightly and it delayed our product's initial introduction to the market. However, as the authors of the C-Store[22] system envisioned, writing a custom optimizer was the only way to take full advantage of Vertica's columnar storage system and distributed execution engine. We believe our experience and success in the marketplace validated our decision. Furthermore, we believe choices made while implementing our query optimizer makes it unique among other products on the market, though this belief cannot be empirically evaluated given the scant literature on other industrial optimizers currently available.

This paper describes the Vertica SQL Query Optimizer (Vertica Optimizer) and some of the optimizations and techniques it employs. While none of these techniques individually are new to the research community, we hope that by enumerating the combination we found useful in practice for our distributed column store optimized for analytic workloads, we can help guide future efforts and validate past work. The Vertica Optimizer is covered by granted U.S. patents 8,086,598 and 8,214,352 as well as several additional pending applications.

The main contributions of this paper are:

1) Argue that a query optimizer should be built for the system in which it resides rather than reused from a system with a different set of constraints.
2) Describe the structure and implementation of a modern, industrial SQL query optimizer.
3) Describe the rationale and customer experience that influenced our design and should be considered in future research.

The rest of this paper is organized as follows: Section II reviews Vertica Data Model, and summarizes what a query optimizer is, and describes how our experience with earlier query optimizers at Vertica informed the design of the optimizer described in this paper. Section III briefly describes the possible plans which our system can execute and thus the optimizer must choose from. Section IV describes the implementation of our optimizer, Section V presents experimental evaluation and Section VI summarizes related work.

II. Background

In this section, we firstly sumarize data model of Vertica, focusing on how data is physically stored. The full details of Vertica Data Model are covered in [20]. We then review the general tasks of a query optimizer to highlight portions that can be highly customized to specific RDBMS such as Vertica.

A. Vertica Data Model

Like all SQL based systems, Vertica models user data as tables of columns (attributes), though the data is not physically arranged in this manner. Vertica supports the full range of standard INSERT, UPDATE, DELETE constructs for logically inserting and modifying data as well as a bulk loader and full SQL support for querying.

Projections and Super Projection: Vertica physically organizes table data into *projections*, which are sorted subsets of the attributes of a table. Any number of projections with different sort orders and subsets of the table columns are allowed. Because Vertica is a column store and has been optimized so heavily for performance, it is **NOT** required to have one projection for each predicate that a user might restrict. In practice, most customers have one *super projection*,

which contains all columns of the anchor table, and between zero and three narrow, non-super projections.

Each projection has a specific sort order on which the data is totally sorted as shown in Figure 1. Projections may be thought of as a restricted form of materialized view [4], [27]. They differ from standard materialized views because they are the only physical data structure in Vertica, rather than auxiliary indexes.

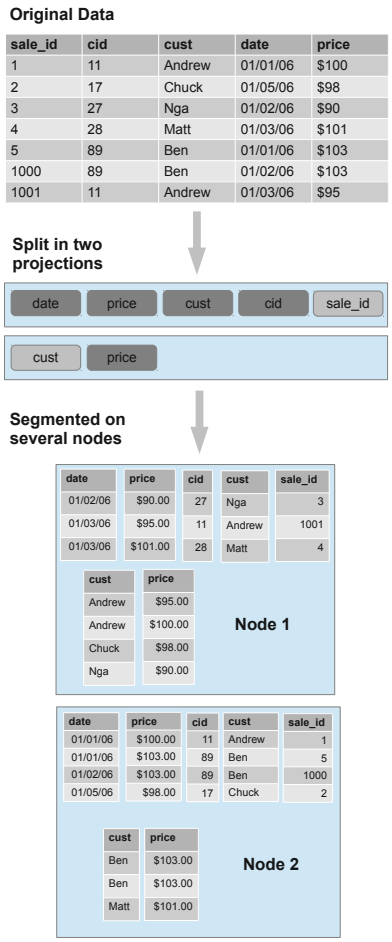

Fig. 1. Relationship between tables and projections. The *sales* tables has 2 projections: (1) A super projection, sorted by date, segmented by $HASH(sale\_id)$ and (2) A non-super projection containing only $(cust, price)$ attributes, sorted by $cust$, segmented by $HASH(cust)$.

Prejoin Projections: Vertica supports *prejoin* projections which permit joining the projection's anchor table with any number of dimension tables via N:1 joins. This permits a normalized logical schema, while allowing the physical storage to be denormalized. The cost of storing physically denormalized data is much less than in traditional systems because of the available encoding and compression.

Encoding and Compression: Each column in each projection has a specific encoding scheme. Different columns in a projection may have different encodings, and the same column may have a different encoding in each projection in which it appears. The same encoding schemes in Vertica are often far more effective than in other systems because of Vertica's sorted physical storage.

Encoding Types:

1) **Auto**: The system automatically picks the most advantageous encoding type based on properties of the data itself. This type is the default and is used when insufficient usage examples are known.
2) **RLE**: Replaces sequences of identical values with a single pair that contains the value and number of occurrences. This type is best for low cardinality columns that are sorted.
3) **Delta Value**: Data is recorded as a difference from the smallest value in a data block. This type is best used for many-valued, unsorted integer or integer-based columns.
4) **Block Dictionary**: Within a data block, distinct column values are stored in a dictionary and actual values are replaced with references to the dictionary. This type is best for few-valued, unsorted columns such as stock prices.
5) **Compressed Delta Range**: Stores each value as a delta from the previous one. This type is ideal for many-valued float columns that are either sorted or confined to a range.
6) **Compressed Common Delta**: Builds a dictionary of all the deltas in the block and then stores indexes into the dictionary using entropy coding. This type is best for sorted data with predictable sequences and occasional sequence breaks. For example, timestamps recorded at periodic intervals or primary keys.

Partitioning and Segmentation: We use *partitioning* and *segmentattion* to represent the intra-node and inter-node horizontal partitionings respectively.

1) **Partitioning**: Vertica provides a way to keep data segregated in physical structures in a single node based on value through a simple syntax:
   `CREATE TABLE ... PARTITION BY <expr>.`
   This instructs Vertica to maintain physical storage so that all tuples within a storage container evaluate to the same distinct value of the partition expression. Partition expressions are most often date related such as extracting the month and year from a timestamp.
2) **Segmentation - Cluster Distribution**: Segmentation is a way to split tuples among nodes. Unlike *partitioning*, which is specified at table level, segmentation is specified for each projection, which can be (and most often) different from the sort order. Projection segmentation provides a deterministic mapping of tuple value to node and thus enables many important optimizations. For example, Vertica uses segmentation to perform fully local distributed joins and efficient distributed aggregations, which is particularly effective for the computation of high-cardinality distinct aggregates.
   Projections can either be *replicated* or *segmented* on some or all cluster nodes. As the name implies, a replicated projection stores a copy of each tuple on every projection node. Segmented projections store

each tuple on exactly one specific projection node. The node on which the tuple is stored is determined by a segmentation clause in the projection definition: `CREATE PROJECTION ... SEGMENTED BY <expr>` where `<expr>` is an arbitrary [1] integral expression.

B. Query Optimizer

In this section, we review the typical tasks of a query optimizer in a relational database system in order to highlight portions which can potentially be reused and which portions must be highly customized to the RDBMS in which they run. We refer readers to one of the many excellent query processing surveys such as [5] for more detail.

In contrast to *procedural* programming and data management systems such as MapReduce[9], users pose queries to an RDBMS using the *declarative* SQL language. The query specifies **what** the user wants computed, not **how** it is to be computed. It is the optimizer's job to translate the user request into a specific set of computation instructions that answers their query in the most efficient manner given the system's available data structures and algorithms. The abstraction barrier between query declaration and its execution imposed by a declarative query language significantly improves application developer productivity and application portability amongst RDBM systems. In the following list, we enumerate the choices an optimizer must make and also highlight how tightly coupled they are to the execution and storage system.

Query Transformations. There are well known query transformations and simplifications which are (almost) always beneficial regardless of RDBMS system and thus are typically always applied. Example transformations include OUTER to INNER join conversions[11], and pushing predicates into subquery blocks. These transformations are applicable regardless of the RDBMS storage and execution architecture because they can be implemented as SQL rewrites and tend to reduce the size of intermediate results flowing within the query.

Join Order Selection. As joins are often used for filtering and their selectivity varies dramatically, the choice of in which order to perform them is critical to performance. The right join order is often the difference in a plan that takes a few seconds and a plan which for all practical purposes will never finish. The optimizer is also often constrained in join order choice by the semantics of SQL, available algorithms and quirks of the execution engine and storage layer. Some parts of join order selection such as ordering based on selectivity are not greatly influenced by the overall RDBMS system. Other parts of the join order selection **are** very specific to the RDBMS such as tracking intermediate distribution across clusters of machines, the importance of sortedness, and information to enable late materialization [3]. Most optimizers choose a join order using a guided search based on heuristics which are tightly coupled and highly tuned to the runtime constraints of their particular RDBMS. It is unlikely that such heuristics can be easily reused across different systems.

Physical Structure Selection. The optimizer determines which data structure the execution engine should use when searching for the rows required to answer the query. This choice involves tradeoffs based on how effective particular structures when evaluating certain types filters, the I/O cost to fetch the structures from disk, the CPU required to use such structure, and implications for other operators later in the query plan. Physical structure selection is almost entirely dependent on the structures available in the data-base system, and the algorithms to choose them must be tightly coupled with the physical system. For example, traditional RDBMS systems must choose between a variety of secondary index structures, whereas the Vertica Optimizer simply chooses which *projection*[2] to use.

Operator Algorithm Selection. The optimizer also determines which execution algorithm to use from those supported by the RDBMS execution engine. The choice of algorithms has a dramatic effect on performance and is constrained by various requirements of the implementation. For example, choosing between a Hash-Join, a Sort-Merge-Join, and a Merge-Join at minimum depends on estimating the sizes of inputs, the cost of sorting input(s), and if the data is already sorted. This choice is highly dependent on the operators available in the execution engine and thus the algorithms to choose between the various operators are not easily reused between systems.

Mechanical Translation. In order to actually produce the results requested by a particular query, the optimizer builds an explicit set of instructions, often called a *query plan* for the execution engine. This plan contains numerous explicit details such as what predicates to evaluate where, what form the intermediate data flowing through the plan looks like, and many other largely mechanical (but critical) details. This part of a query optimizer must, by necessity, be tightly coupled to the representation of the plan in the system in which it will be run.

Of the preceding list of optimizer tasks, only query transformations are general enough to be significantly reused across different database systems. Some have argued that much of the join order enumeration may also be reused. However, our experience, as described in Section II-C, was that join order enumeration is highly specialized and led us to write the Vertica Optimizer from scratch using customized algorithms for the choice of physical structure, join order, and operator algorithms available within the Vertica Analytic Database.

C. Design History

The C-Store research prototype did not contain a query optimizer implementation, but the design called for a Selinger-style[24] optimizer that used cost-based estimation for plan construction. The C-Store authors noted that their optimizer would need to be aware of compression and its effects on the CPU and I/O cost of database operations, and they listed the

[1] While it is possible to manually specify segmentation, most users let the Database Designer determine an appropriate segmentation expression for projections.

[2] projection is defined in section III-A

major optimizer decisions as 1) which projections to use for a given query, 2) how to prune the plan search space to a feasible size, and 3) at what points in the plan they should materialize columns. The Vertica Query Optimizer addresses all of these decisions and many more, but the final design did not spring forth fully formed on our first attempt. The Query Optimizer used in Vertica is actually our third version, and the history of how we arrived at the current design highlights some the challenges faced by industrial optimizer implementers.

StarOpt. The initial Vertica Optimizer was Kimball-style query optimizer[19] that assumed any interesting customer schema and query could be modeled as a star or snowflake. When this design assumption met the real world, it became clear that most customer data did not exactly conform to the star ideal. For a variety of technical and non-technical reasons we couldn't expect customers to change their schemas, so we needed a different optimizer with a different set of assumptions. StarOpt also applied distribution and sort-algorithm decisions **after** the join order had been chosen, which significantly affected its ability to minimize data movement for complex plans running against complicated distributed schemas. As we learned, an industrial optimizer must handle all the vagaries of messy schemas and data however nonsensical they may seem to the implementors.

StarifiedOpt. The second generation optimizer was a modication to StarOpt which internally forced non-star queries to look like a star solely for the purpose of applying the existing StarOpt algorithms. This approach was far more effective for non-star queries than we could have reasonably hoped, and it bought us sufficient time to design and implement the third generation optimizer: the custom built, but poorly named, V2Opt.

V2Opt. When we refer to the Vertica Query Optimizer in the rest of this paper, we are referring to *V2Opt*. This optimizer, designed in collaboration with Mitch Cherniak at Brandeis University, is completely aware of distribution, sortedness, and non-star schemas in all its decisions. It has been the default Vertica Optimizer since version Vertica 4.0 and has served us well at thousands of customer installations and planned hundreds of millions of queries.

Another specific example of a significant optimizer change between StarOpt and V2Opt directly driven by our experiences is that V2Opt does **not** specify the order in which to apply predicates during a projection column scan operation. Instead, the predicate evaluation order is left to the execution engine as the most efficient order is often dependent on conditions not known at plan time. We found that choosing the obvious predicate evaluation order of decreasing selectivity was sometimes far from optimal due to compression and clustering effects in our storage system. The execution engine was in a much better position to change the predicate evaluation order dynamically and in retrospect moving where this decision was made had a significantly positive impact on our query plans.

III. Query Plan Search Space

A. System Review

This section reiterates some material from the detailed system description of the Vertica Analytic Database [20] which are relevant to the choices the Vertica Optimizer makes.

Vertica physically organizes table data into *projections*, which are sorted subsets of the attributes of a table. Any number of projections with different sort orders and subsets of the table columns are allowed. Each projection has a specific sort order on which the data is totally sorted and the system maintains this sortedness through the tenure of the data within the system. Each column in each projection has a specific encoding scheme. Different columns in a projection may have different encodings, and the same column may have a different encoding in each projection in which it appears. Vertica's distributed storage system assigns tuples to specific nodes via a horizontal partitioning scheme called *segmentation*. Segmentation is specified individually for each projection, and the scheme we use provides a deterministic mapping of tuple value to node, thus enabling many important distributed optimizations.

Vertica's execution engine provides operators which perform the actual computation on user data. The execution engine provides several different algorithmic implementations of the same logical operation such as Merge Join and Hash Join algorithms. Merge Join is faster and requires less memory if the input is pre-sorted, but Hash Join can be used regardless of the incoming data's sortedness. Operators also have specializations to operate directly on encoded data, which is especially important for scans, joins and certain low level aggregation operations.

1) Supported SQL Constructs: The Vertica Optimizer implements the SQL-99[16] specification along with portions of SQL-2003[17] which our customers have found useful. We next list some of SQL features that Vertica supports to illustrate the breadth and depth of features needed for a commercial analytic database product at the time of writing. Since the order in which we implemented SQL functionality was driven entirely by customer demand, we have ordered the following list in approximate implementation order to give readers an idea of the relative importance of certain features to our customers.

1) **Expressions and Filtering** for the full range of SQL predicates, functions, date manipulations, string manipulations, regular expressions, and user defined functions.
2) **Aggregation** and aggregations of distinct values. Vertica supports all SQL-99 aggregate functions, plus DISTINCT and multiple DISTINCT aggregates and full HAVING clause support.
3) **Set Operations** such as UNION, UNION ALL, INTERSECT, EXCEPT.
4) **Basic Joins**, implemented in decreasing order of usefulness: INNER JOIN, LEFT OUTER JOIN, single and multi column equality predicates, ON clause predicates and their subtly different semantics compared to

WHERE clause predicates, and specialized range join predicates.

5) **Basic Subqueries** used as table expressions and correlated predicates in the FROM and WHERE clause.
6) **Advanced Joins** such as RIGHT OUTER JOIN, FULL OUTER JOIN, SEMI and ANTI joins (introduced via subquery rewrites), multi-column variations of ANTI joins, along with subqueries correlated via non-equality predicates.
7) **SQL-99 Analytic Functions**, including moving windows. Vertica also supports several custom SQL extensions designed for analytics such as event series joins designed for timeseries analysis, pattern matching across rows within a partition via regular expression, and timeseries normalization functionality.
8) **Advanced Subqueries** correlated and non-correlated subqueries in the HAVING clause, under OR predicates, NOT IN, used as scalar values and other esoteric functions.

2) Optimizer Plan Space: We are now in the position to describe the theoretical set of possible plans that can be chosen by the Vertica Optimizer. The optimizer does not fully search this space, and the mechanism to restrict the search and choose a plan is described in the following section.

- **Table Access**. The optimizer must choose exactly one projection that contains all columns required for each table referenced in the query. As there is no such thing as an index in Vertica, the only choice is which projection to use with the execution engine's highly optimized and tuned column "scan" operation.
- **Join Order**. The optimizer may perform the table joins in the query in any order as long as that order returns the same result as would the query as phrased by the user. The Vertica Optimizer does not limit its search space to so called *left-deep* plans like the System-R[24] optimizer – it can and does consider so-called *bushy* plans.
- **Algorithms**. Each plan operation may be executed with one of several different algorithms. There is typically a sort based algorithm, and a hash based algorithm available to choose between. Due to the lack of auxiliary indexes, the choice of algorithm in Vertica is comparatively smaller than in other systems as we do not consider index-assisted algorithms such as index-nested-loops join.
- **Column Materialization**. Because Vertica is a native column store, columns from projections may be fetched on demand at various stages during plan execution at no additional runtime cost. The Vertica Optimizer must account for this optimization as a first class citizen *during* the join enumeration and pruning as it has dramatic effects on the overall plan's performance.
- **Redistribution**. The optimizer must specify where in the plan to *resegment* (redistribute data according to some new segmentation scheme) or *broadcast* (ensure that all nodes get a copy) data. Some redistribution operations *must* be done for correctness and some redistribution operations *may* be done to improve performance.
- **Predicate Placement**. A predicate may be specified as part of a scan operation, or applied as a separate subsequent filtering step. The optimizer chooses where to place the predicate in the plan to ensure correctness as well as optimal performance. The optimizer also must place synthetic predicates such as *sideways information passing* predicates [26].

Note that we include column materialization as part of our plan space, unlike other typical optimizer search space descriptions. This is because the Vertica Analytic Database is a native column store and late materialization is critical for performance. Typically, native row stores (even those which have retrofitted columnar storage) treat optimizations which are the most similar to late materialization as special cases of some indexed join types. It is hard to account for such cases in core join order enumeration algorithms, given the already inherent complexity, unless those cases are as core to the system architecture as materialization is to Vertica. Similarly, Data redistribution, like column materialization, is a core architectural feature of the Vertica Analytic Database, and thus must be at the core of the Vertica Optimizer's choices. We believe that retrofitting distribution choices into an algorithm originally designed to create serial plans is inherently inferior to building an optimizer's core algorithms around such choices. Both of these points further argue the case for purpose built optimizers.

IV. The Vertica Query Optimizer

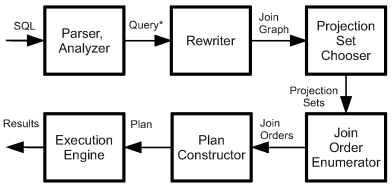

Fig. 2. Query plan creation within the Vertica Query Optimizer.

Figure 2 shows a block diagram depiction of how the Vertica Optimizer creates an execution plan. The tasks within each box of the figure are discussed in more detail in the following sections

A. Inputs

The input to the Vertica Optimizer is an SQL query and optional statistical summaries of the tables and available projections. These inputs are similar to other academic and industrial query optimizers and we highlight them only briefly below. Vertica uses the parser and semantic analysis engine from the PostgreSQL[1] system to verify the syntax and semantics of the query. We did not write a custom SQL parser because SQL is a standard language, and thus no value is created by writing a custom parser.[3] The Vertica Analytic Database

[3]Custom parsers also invariably create slightly incompatible dialects of SQL which is bad for application developers.

provides the optimizer with the full set of standard state of the art data statistics such as automatically maintained row count, minimum and maximum values for each table column, and equi height per column histograms with both value count and distinct value count for each histogram bucket.[4] The statistics also contain physical characteristics for each column in each projection such as total size on disk and number of disk pages.

B. Join Graph

In the Vertica Optimizer, the parsed `Query*` tree from the PostgreSQL parser is first transformed into an equivalent structure called a *Join Graph* similar to the one described in Starburst[14]. A Join Graph does not represent or store any planning decisions (e.g. Merge Join vs Hash Join), rather it conveniently represents the parsed query with quick access to important relationships. Join Graphs are more convenient to manipulate than the raw `Query*`for two reasons: 1) Many of our algorithms are more natural to express as walks over graphs and 2) The `Query*` is a C structure and Join Graphs is a C++ class.

```
SELECT A.a1, A.a2, C.c, D.d, SUM(B.b)
FROM   A, B, C, D
WHERE  A.x = B.x AND B.y = C.y AND B.z = D.z
   AND A.a1 = 100 AND C.c LIKE 'Spring'
GROUP BY A.a1, A.a2, C.c, D.d;
```

Fig. 3. Query 1: 4 Tables, 3 Joins

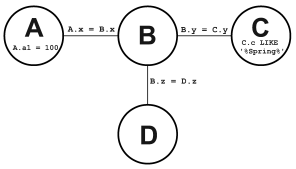

Fig. 4. Join Graph 1, representing Query 1 in Figure 3

The nodes in a Join Graph represent the relations in the `FROM` clause of an SQL query, typically tables or subqueries. An edge between nodes in a Join Graph represents joins between two tables. Both nodes and edges store properties about the relation and join respectively. Figure 4 illustrates the Join Graph of the query from Figure 3. There are four vertices, one for each of the four tables A, B, C, and D in the `FROM` clause. The three edges represent the three joins, `A.x = B.x`, `B.y = C.y` and `B.z = D.z` respectively. The single column selection predicates `A.a1 = 100` and `C.c LIKE '%Spring%'` are stored on the vertices. Because Vertica is a native column store, our Join Graph also stores information for column store specific optimizations such as late materialization. One example of such information is the list of attributes required for computing quantities after all joins. Example properties stored on edges are the type of join (e.g. `INNER`, `OUTER`, `SEMI`) being performed and if the join is along a Foreign Key - Primary Key relationship. Information such as the `SELECT` list (`A.a1, A.a2, C.c, D.d, SUM(B.b)`) and the `GROUP BY` list (`A.a1, A.a2, C.c, D.d`) are stored as Join Graph wide properties.

[4]We use the Smoothed Jackknife algorithm [15] to estimate the distinct value count with a maximum of 100 buckets.

```
SELECT A.a
FROM   A
WHERE  A.x IN (SELECT B.x
               FROM   B, C
               WHERE  B.y = C.y
               AND C.c LIKE 'Spring');
```

Fig. 5. Query 2, with Subquery

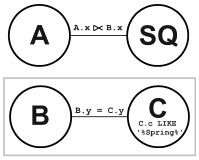

Fig. 6. Join Graph for Query 2 in Figure 5

Subqueries are represented as a vertex in a Join Graph with a pointer to another Join Graph instance representing the sub query block, as illustrated in Figure 6 for the query in Figure 5. Similarly to subqueries, the Join Graph also represents restrictions on join orderings required to maintain semantic equality using special node types. This is necessary, for example, because it is not possible to arbitrarily reorder certain combination of `INNER` and `LEFT OUTER` joins. A full description of our restriction representation is outside the scope of this paper, but we highly encourage any reader who wishes to create an industrial optimizer to study research literature such as [23].

C. Rewriter

As previously mentioned, the raw SQL query is parsed and semantically analyzed by code derived from the PostgreSQL system. The Rewriter step applies a series of query-level transformations to produce a query which is simpler and easier for later stages to optimize. The first type of Rewriter transformations are the standard SQL transformations from PostgreSQL such as `OUTER JOIN` to `INNER JOIN` conversion, predicate push down into subqueries, and unnesting subquery blocks when possible. We have significantly extended these transformation passes as customer needs have arisen, and we are able to quickly incorporate newly added transformations from the PostgreSQL codebase. The second type of SQL transformations done by the Rewriter are those which we found to be relevant to customer queries and did not have suitable implementations in Postgres such as subquery and view de-correlation and flattening [18], adding transitive predicates based on join keys, and pushing grouping and top-K filtering into subqueries and sub portions of the plan. The output of this chain of components is a Join Graph structure, as described in Section IV-B, which is ready to be optimized.

For implementation expedience, we initially assumed that fully general, restartable query blocks for correlated subqueries were not important to support within the context of a distributed analytic database. We instead focused on automatically rewriting all subqueries into joins instead. Our experience in the market has validated this initial assumption: it is very rare to encounter a customer query with subqueries which can not be automatically flattened into `SEMI JOIN` or `ANTI JOIN` [5]. Due to the distributed architecture within which it is running, the Vertica Optimizer has a much larger library of subquery to join transformations than systems which treat such rewrites as an optimization. This is an example of a design choice made in for the Vertica Optimizer directly driven by the system for which it was written.

D. Projection Set Chooser

The input of the Projection Set Chooser is a Join Graph and the list of projections which are available for query processing. The output is a compact representation of one or more candidate *Projection Sets*. A Projection Set specifies, for each table node in the JoinGraph, which projection should be used to fetch the data for that particular table. Only projections that *cover* the table (i.e., contain all the table's columns used in the query) are eligible. The total space of candidate Projection Sets consists of all possible combinations of covering projections for the tables in the query, and it is the job of the Projection Set Chooser to produce candidates which are not provably suboptimal. For example, if a projection P_1 has the same sort order and segmentation as projection P_2, but is more expensive to scan because of different encoding types, then any projection set containing P_1 can be eliminated. The intuition for elimination is that the same projection set with P_2 substituted for P_1 will be better for all queries and thus P_1 can be safely excluded without missing an optimal plan. The Projection Set Chooser also performs two other operations of note.

First, it determines how to access each table's data when some cluster nodes have failed and their data is not accessible. Because this information is encoded into the Projection Set, the rest of the Vertica Optimizer's algorithms are aware of and influenced by the potential change of distribution, sort and cost when nodes are down in the cluster.

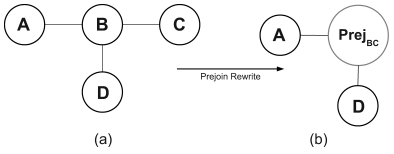

Fig. 7. Prejoin Rewrite. The nodes and edge representing selecting from tables A and B and the join between them is replaced with a single new node representing a prejoin projection.

Secondly, the Projection Set Chooser performs *prejoin rewriting*. A prejoin projection contains multiple tables joined via a foreign key - primary key relationship during data load. If a prejoin projection could be used in place of actually performing one or more joins during query execution, the Projection Set Chooser will create a new Join Graph for possible optimization by rewriting the input Join Graph. For example, Figure 7(a) shows a Join Graph of a query that joins 4 tables, A, B, C, and D, and Figure 7(b) illustrates the new Join Graph which is constructed after rewriting the join between B and C with a prejoin projection.

[5]Our `SEMI JOIN` and `ANTI JOIN` implementations include several variants which are necessary for full semantically equivalent rewrites.

E. Join Order Enumerator

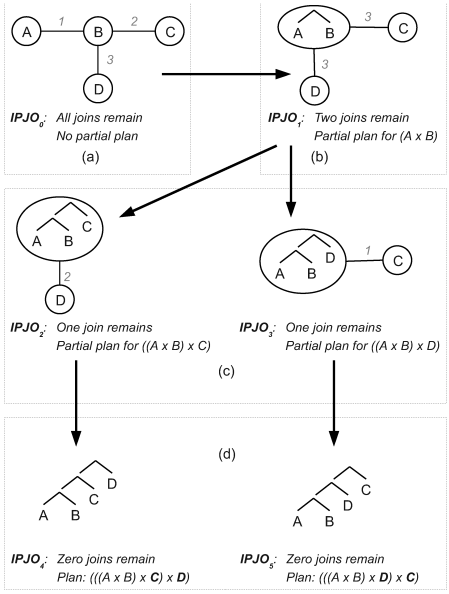

Fig. 8. Join Order Enumeration via Join Ranking, described in section IV-E

The *Join Order Enumerator* heuristically explores potential join orderings using a bottom-up enumeration algorithm based on ranking joins using a *Join Ranker*, as described in Section IV-E1. The Join Order Enumerator algorithm is a work list algorithm that operates on partial plans called *In Progress Join Orders* (IPJOs). Each IPJO represents a partial execution plan with one or more joins not yet performed. The Join Order Enumerator proceeds as follows:

1) Initialize the worklist with a single IPJO for each Covering Projection Set
2) While the worklist is not empty, choose the next least complete IPJO from the worklist
   a) If the IPJO is complete, add it to the output
   b) Otherwise, rank all joins not yet completed in that IPJO (note that the rankings of the same join can and will be different in different IPJOs)
   c) Determine the minimum rank value R of any join
   d) For each join J with the rank value of R
      i) Choose potential join methods for J
      ii) Form new IPJOs for each join method
      iii) Add the newly created IPJOs to the worklist
3) Remove non-interesting items from the worklist using the Join Pruning algorithm described in Section IV-E2.

4) The output of the Join Order Enumerator is a set of complete plans. The optimizer chooses the plan with the lowest estimated cost using the cost model described in Section IV-F.

Figure 8 illustrates the join enumeration algorithm. Figure 8(a) illustrates the initial $IPJO_0$ for a query with four tables, A, B, C, and D [6]. The $IPJO_0$ has no partial plan, and each remaining join is annotated with its rank, in this case 1, 2, or 3. The numerical join ranks were determined by a Join Ranker, described in section IV-E1, using query attributes not shown in this example. The join between A and B has the smallest rank, 1. $IPJO_1$ is formed in the next step by joining A with B to form the partial plan shown in Figure 8(b). As the join between A and B has different properties than either the tables A or B individually, the ranks of the two remaining joins in $IPJO_1$ are different than their ranks in $IPJO_0$. In this case they are both assigned the rank 3 and thus are both considered as candidates as the next possible join to perform. Exploring both of these possibilities creates the two new IPOJs, $IPJO_2$ and $IPJO_3$ shown in Figure 8(c). $IPJO_2$ and $IPJO_3$ each have only a single join remaining, which results in the two final join orders $IPJO_4$ and $IPJO_5$ shown in 8(d).

1) Join Ranking: The *Join Ranker* encapsulates the heuristics used to explore the join order search space via a numerical rank for each remaining join in an In Progress Join Order (IPJO). For any join with a particular rank value, the join will be performed **after** all joins with lower rank values and **before** all joins with higher rank values. The Join Ranking system allows our Optimizer to find good join orders without exploring the entire plan space. Our code is structured to accept a Join Ranker which satisfies the following interface:

- **Input**: A single IPJO, comprised of: 1) a Join Graph, 2) a Covering Projection Set, 3) a partial plan which joins some of the tables in a particular order and method, and 4) a list of remaining joins which have not yet been performed.
- **Output**: A mapping of remaining join to an integral *rank*. Any particular rank value is definitively better or worse than any other rank (the ranks form a total order), but multiple joins can be assigned the same rank.

It is possible to use different Join Rankers with the Vertica Optimizer to change the join order search space the optimizer explores. In reality, we have one Join Ranker which is used to plan all customer queries, and the primary benefit to splitting the Join Ranker from the Join Order Enumerator is a clean separation between software modules which allowed decoupled development, testing and validation of the components. Our production Join Ranker ranks joins based on various physical properties of the partial results being joined such as:

- **Selectivity** What percent of the data will be eliminated after going through the join?
- **Cardinality** What is the relative estimated size of the join inputs?
- **Co-location** Are the two inputs segmented such that a join can be performed without redistributing the intermediate results?
- **Sortedness** Is the input data sorted in advantageous ways such as allowing merge join or upstream pipelined grouping?
- **Constraints** Are there any applicable schema constraints (e.g. Primary Keys)?

Further details of the Join Ranking formula, its inputs, and how we pick between the possible join methods such as algorithm (e.g. Hash or Merge) and input order (e.g. which table to hash and which to probe) are beyond the scope of this paper. Such details are not intellectually interesting but do represent some of the hard won knowledge required to build a practical and successful industrial optimizer.

2) Join Pruning: As the Join Order Enumerator explores the join order search space, it periodically applies a Join Pruning algorithm to the elements in the work list:

1) Group IPJOs together which represent the same partial results and the same joins remaining to perform.
2) Within each group, keep only those IPJOs which have the lowest estimated cost for *some* particular Interesting Property and discard the rest.

The Vertica Optimizer defines *Interesting Properties* for each column or set of columns which might allow cheaper operations upstream (such as group by or join columns). Interesting properties exist for several dimensions such as sortedness and segmentation. This pruning strategy is an extension of the System R[24] approach of keeping only access paths which have "Interesting Orders" because interesting properties include not just sortedness but cluster distribution and other Vertica specific properties. The inclusion of distribution information *during* the join enumeration algorithms was a key design goal of our optimizer and we believe it is another compelling reason for writing a custom optimizer.

Taken together, the Join Ranker and Join Pruning algorithms encode the heuristics used to limit the query plan search space. Like all heuristics, they are not perfect and in pessimal cases may not reduce the search space to a feasible size. In order to prevent run away planning time, the Vertica Optimizer imposes a fixed size on the memory it devotes to storing intermediate results. If this cap is hit, the optimizer goes into a *hurry up* mode where only the single best partial plan is kept at each stage. Of course hurry up mode sacrifices potentially optimal plans, but it provides two nice features: 1) Testability (the search space considered is simply a function of query and physical design, *not* of the power of the system running the optimizer, and 2) a simple way to avoid resource explosion (in addition to the time explosion) when our heuristics are not limiting the space sufficiently.

F. Cost Model

When exploring the plan space, the optimizer needs a way to compare different plans and eliminate sub-optimal plans. The ultimate goal is to find the plan with minimum run time, but it is notoriously hard to estimate the actual run

[6]While the actual enumeration algorithm runs in terms of projections, we use tables in our example for expository simplicity.

time of a query plan due to the sheer number of factors that affect query execution. Furthermore, we placed significant value on predictable behavior to improve the user experience. Thus, rather than building a complex model that attempted to predict query run-times directly, we chose to model predicted data flow through the plan. We heuristically assume that minimizing the data flow through the plan is a good proxy for minimizing the query execution time and it seems to work well in practice.

The advantages of a data flow based cost model are:

- **Simplicity** In production environments, debuggability and predictability are very important features. Having a cost model that is easy to reason about is often more important than one which captures many nuances.
- **Robustness** Modern hardware architectures have many sophisticated features that affect query run times, such as non-uniform memory access, pipelining, and hyper-threading. Furthermore, variables such as Disk I/O performance, network bandwidth and latency, and CPU significantly affect query performance but vary widely both across deployment environments and over time in a particular environment. By modeling these variables via the amount of data processed, our cost model is less sensitive to changes in the deployment environment resulting in more robust behavior.

1) Cost Aspects: The data flowing through an operator/node in the query plan is classified into one of four categories – Disk, CPU, Network or Memory termed *cost aspects*. The intuition behind distinguishing between different aspects is that the cost of fetching a certain number of bytes from disk is different from the cost of processing the same amount of data in the CPU or sending it over the network. For example, we model a hash join operator using the following aspects:

- **CPU** Estimated sizes of outer and inner relations of the join, since both relations need to be hashed on the join keys.
- **Memory** Estimated size of the hash table needed to store the inner relation.
- **Disk** Estimated amount of data that will spill to disk.
- **Network** If the join requires resegmentation or broadcast, then the amount of data sent over the network.
- **Parallelism** The *number* of cluster nodes the join will execute on.

The distinction between the CPU and memory aspects is purely for development convenience – data that is being copied is technically also processed by the CPU, so the Memory and CPU aspects could be merged. However, keeping them separate encourages developers to pay attention to both aspects of a query operator, reducing the likelihood of some aspect being missed. The estimated data sizes are computed as the product of the estimated cardinality (discussed in Section IV-F2) and the tuple width. Because Vertica is natively a column store, the tuple width at each operator is often significantly different, since typically only the columns needed for the operation are fed to the operator. Using cost aspects we can also model execution engine operations acting directly on compressed data, which dramatically affect the disk, memory, and CPU requirements in a way that a cost model based solely on I/O or rows processed can not.

2) Cardinality Estimation: Estimating the cardinality (the number of rows) at intermediate points of a query plan is needed to calculate the cost aspects, as the number of rows processed is directly proportional to the number of bytes that flow through the plan. The optimizer must estimate for what percentage of rows a given predicate will evaluate to true, as well as estimating the output cardinality of grouping and join operations. These calculations are notoriously hard to get right[5], especially for joins.

The cardinality estimator within the Vertica Optimizer is based on histograms of table column values. Vertica contains a partial expression evaluation system, known as *ExprAn* which calculates expression range information. Specifically, given an arbitrary expression and the range of values for each variable (i.e., column reference) ExprAn gives the tightest possible minimum and maximum values for the expression's output. ExprAn is used by several other subsystems such as the execution engine which skips fetching entire partitions or disk blocks when they are guaranteed to hold no row that will satisfy the predicate.

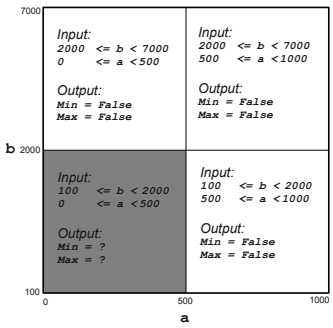

Fig. 9. A hypercube used to estimate the selectivity of $a + b <= 100$. The expression evaluates to false for the entire ranges of a and b depicted in the three white quadrants, and the expression evaluates to either true or false for the ranges in the lower left hand gray quadrant.

For each predicate, the Vertica Optimizer builds a multi-dimensional *hypercube*, where each dimension represents the value distribution of a specific column referenced in the expression. For an expression with n column references, the optimizer builds an n-dimensional hypercube. The value distribution is directly taken from the column histograms described previously. Figure 9 shows the two-dimensional hypercube – i.e., a square – built for the expression $a+b <= 100$. Each cell in the hypercube corresponds to one bucket from the histogram of each column reference in the predicate.

A region within the hypercube represents a set of ranges for each column reference. For example, the upper right corner of Figure 9 represents the range $[500, 1000]$ for a and $[2000, 7000]$ for b. Using a divide and conquer technique, the optimizer explores the regions within the hypercube, using ExprAn to determine the output range of the predicate. It then

tallies row counts of regions known to be true, regions known to be false, and regions which could be either, to form a final estimate of the overall selectivity of the predicate. We use a similar approach for join predicate selectivity and another cardinality estimation technique for grouping operations based on distinct value calculation. Like other industrial optimizers, we assume column value independence, and column correlation is the most significant source of selectivity estimation error we see in customer cases. Our estimates are highly skew tolerant due to the use of equi height histograms which captures skewed value distributions well.

3) Cost Estimate: We form the overall cost for a plan by forming the total estimated data movement as follows:

1) Calculate the cost aspects for each operator in the query plan using a per operator model and input and output cardinality estimates.
2) Calculate the total cost of each operator as a weighted combination of the aspects divided by the degree of parallelism (the number of nodes in the cluster).
3) Calculate the cost of the entire query by summing the cost of each operator.

Incorporating the degree of parallelism makes the cost of a plan proportional to the amount of data processed per machine. This property makes the model invariant to the number of machines in the database cluster – for example, the cost of a plan running on 4 nodes is the same as the cost of the plan running on 8 nodes for the same amount of data per node. This normalization avoids unwanted plan changes when the number of nodes in a cluster is scaled up to handle increasing data sizes.

The specific value of the weights given to the linear combination of the cost aspects and the cardinality estimates cannot be changed, tweaked or otherwise tuned by users. This approach is different than other industrial optimizers[2] which allow users to fine tune internal weights and measures. We believe the simplest and most effective way for users to override the optimizer's choice of plan is not through indirect knobs, but rather through direct control of the desired plan features. We provide users this level of control via a mode known *syntactic optimizer*, in which users specify exactly the join orders and projections they desire via query hints or session properties. We have found this approach very effective in the field.

G. Plan Construction

Once the join order, algorithms, and projections have been chosen, the plan construction phase mechanically creates an operator-annotated (executable) plan. By mechanical we mean easy to predict, not devoid of value. The construction phase includes transformations and operations critical for performance and correctness such as local parallelism annotations, partition elimination, and localized storage optimizations. In addition, there are several features mechanically added in the Vertica Optimizer's plan construction phase which are often chosen during query optimization via cost model in other systems. Such operations with Vertica are enabled and disabled selectively based on actual data flowing through the execution engine once the plan is running.

One example of the close collaboration between the optimizer and the execution engine is *group by pushdown*. Group by push down[6] is an optimization technique that performs grouping operations before join operations in the hopes of reducing cardinality of join inputs and thereby reducing query runtime and resource usage. However, depending on the actual data and predicates in the query, the pre join grouping operation may actually cost more in resources than is gained by reduction in cardinality. Rather than attempting to distinguish between these two cases at plan time, The Vertica Optimizer *always* places specially annotated grouping operations below joins when it is correct to do so. At run time, the execution engine evaluates the actual effectiveness of ongoing grouping operations, and disables the grouping when its cost is not justified by its current benefits. Other examples of Optimizer and Execution Engine collaboration are changing predicate evaluation order at runtime, delaying column decompression, and switching join algorithms when memory is exhausted and externalization is required. Further details of our plan structure and the steps to create it are beyond the scope of this paper.

V. Experimental Results

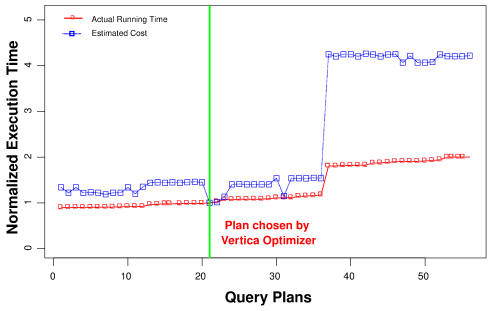

Fig. 10. Cost model estimates versus actual runtime for multiple plans to compute the results of Q8 of the TPCH benchmark. The blue line is the optimizer's normalized cost estimates and the red line is the normalized actual run time. The vertical green line signifies the plan chosen by the optimizer.

A. Cost Model Evaluation

In order to gauge the "quality" of our cost model, we measured the runtime performance and the estimated cost of *all* the plans produced by the join order enumeration algorithm for Q8 from the TPCH benchmark suite. While the plans do not represent the entire space of possible plans, we manually verified that the plans considered included the optimal join patterns for this query. For this experiment, we used a slightly modified version of the Vertica Analytic Database 6.0 where we could override the plan chosen by the optimizer. We did not change any other settings or parameters. We performed these experiments on the TPCH data set on a 3 node cluster of HP ProLiant BL460c G7 blade systems.

The results of our experiment are plotted in Figure 10. The Vertica Optimizer's cost estimates are plotted in blue and the actual runtimes are plotted in red. The plan automatically

TABLE I
STATISTICS (QUERIES PLANNED, AVERAGE MEMORY AND TIME REQUIRED FOR PLANNING) FOR THE VERTICA QUERY OPTIMIZER IN 14 INDIVIDUAL CUSTOMER SYSTEMS. THESE NUMBERS REFLECT A SINGLE WINDOW IN TIME, NOT THE ENTIRE LIFETIME OF THE SYSTEMS.

Queries Planned	Avg Planning Memory (MB)	Avg Planning Time (ms)
9,683,440	0.88	23.88
8,902,349	0.67	32.87
8,257,351	13.16	124.59
4,967,392	0.75	27.73
4,664,993	11.35	2.97
4,300,289	0.15	30.72
3,500,384	0.48	22.06
2,983,020	0.55	38.11
2,747,417	3.37	98.01
2,718,460	3.65	271.96
2,437,333	0.26	14.45
2,302,822	11.07	222.02
2,085,196	0.17	2.53
...	...	...
97,974,340	3.71	53.11

chosen by the optimizer, designated by a vertical green line, is the one with the lowest estimated cost. The runtime of the chosen plan, while slightly longer than the minimal runtime plan, is quite close to the optimal. The figure shows that the cost model accurately predicts the jump in execution time which occurs around query plan 37, and that the optimizer correctly picks a plan in the faster regime. The plan with the actual lowest runtime is not chosen due to the data flow (which the model predicts) not being perfectly aligned with runtime for some particular plan pattern. This misalignment leads to the dip in estimated cost for plan 21 without a corresponding dip in actual runtime. It is, of course, impossible to draw general conclusions on the quality of a query optimizer from the analysis of a single query, but we believe this result is representative of most customer experiences. Similarly to any query optimizer design which meets the real world, there are times when pathological data or queries conspire to prevent an optimal plan and require manual intervention. However, it has been our experience that the Vertica Analytic Database produces good plans for the vast majority of customer queries.

B. Customer Statistics

At the time of writing Vertica has well over a thousand customers in production. We have a database which has workload information for some of these production clusters across 3 major and 8 minor software releases of the Vertica Analytic Database. Due to various technical reasons, the data in this database is both incomplete and contains a biased record of Vertica's use by customers, but nonetheless the information is still insightful and we share some selected excerpts blow.

We have optimizer runtime and memory usage statistics for almost 98 million queries run on real customer databases. Across these queries, the Vertica Query Optimizer requires on average 55 ms and consumes 3.71 MB of memory to create a query plan. Table I shows optimizer statistics broken per individual customers if we had information on at least two million queries. The number of queries reported is for user requests only, and does not include system tasks such as plans run by the Tuple Mover and furthermore reflect a particular window of time, not the lifetime workload of these systems. We believe that the significant differences in average time and memory consumption required for different customers demonstrates the real world workload heterogeneity which the Vertica Query Optimizer supports.

TABLE II
THE TOTAL ON DISK SIZE (INCLUDING COMPRESSION AND COPIES REQUIRED FOR HIGH AVAILABILITY), IN TERABYTES, ROW COUNTS (IN MILLIONS), AND NUMBERS OF QUERIES WHICH REFER TO THOSE TABLES IN SYSTEMS FOR WHICH WE HAVE DATA.

On Disk Size	Rows Stored	Query Count
184 TB	10,273,815 M	2,151,734
45 TB	702,860 M	1,573,309
32 TB	1,020,209 M	9,397
32 TB	1,372,130 M	266,398
32 TB	992,785 M	331,442
31 TB	1,309,403 M	177,977
29 TB	1,253,861 M	123,351
24 TB	774,069 M	8,703
21 TB	1,043,664 M	183,409
19 TB	594,814 M	8,252

Table II shows statistics for some of the largest tables in our customer database against which our optimizer plans queries. The on disk size is the total of all projections anchored on that table, and thus includes all compression effects and redundancy necessary for high availability. The statistics in Table II reflect significantly different workloads than the optimizers in transaction processing systems were designed to handle. The data size is much larger by comparison, and the number of queries executed per second is smaller. The difference in workload is one of the reasons designing the Vertica Optimizer from scratch made more sense rather than reusing an optimizer from an existing system.

Our optimizer routinely handles large, complex queries, generated directly by user as well as automated tools. Table III shows the distribution of the number of tables referenced in queries for which we have detailed information. The number reported is the total distinct number of tables referenced in the query, including all subquery blocks. Due to how it is captured, our data counts multiple references to the same table a single time, and thus the numbers are actually the *lower bound* on the actual number of joins in each query. The information in Table III shows that the Vertica Optimizer's algorithms can and regularly does handle complex queries, though the majority of queries do not contain an excessive number of joins. More than half the customers for which we have data regularly use queries which join seven or more tables. We hope that by sharing this distribution of join complexities in an industrial setting, we can help future database optimizer researchers focus on relevant industrial problems.

VI. RELATED WORK

Historically, the idea of a distribution aware query optimizer and execution was first introduced in the R* system [8], [21].

TABLE III
DISTRIBUTION OF NUMBER OF TABLE REFERENCES (A LOWER BOUND ON THE NUMBER OF JOINS) PER QUERY, AND THE PERCENTAGE OF CUSTOMERS WHOSE WORKLOAD CONTAINED QUERIES WITH THAT MANY TABLE REFERENCES. THIS DATA REFLECTS THE SMALL PERCENTAGE OF OUR OVERALL CUSTOMER BASE FOR WHICH WE HAVE DETAILED USAGE DATA.

Tables Per Query	Customers (Percent)
100+	3 %
50-100	7%
25-49	15%
15-25	37%
10-15	52%
9	49%
8	47%
7	58%
6	68%
5	81%
4	90%
3	95%
2	98%
1	100%
Total	100%

The idea of extensible query optimizers and optimizer frameworks was first pioneered in the Volcano[12] and Cascades[13] projects. The Starburst[14] project was the first to describe the use of a JoinGraph like structure (Query Join Graph) for query optimization.

Several new commercial database products optimize queries using a wrapper or other abstraction around an existing SQL optimizer to retrofitting knowledge of new structures into it. Typically this approach is taken to reduce implementation complexity and decrease time to market. For example, ParAccel extends the PostgresSQL [7] Optimizer to handle its columnar format and SQL Server Parallel Data Warehouse product reuses the single node SQL Server Optimizer[25] for distributed query planning.

SQL Query Optimzation still generates significant research and industrial interest and is a field rich in needed research. We are particularly encouraged by recent work such as [10] which has focused in particular on the needs of join order enumeration.

VII. CONCLUSIONS

This paper presents the case for custom query optimizers in novel database systems and described the architecture of such an optimizer, the Vertica Query Optimizer which is part of the Vertica Analytic Database. While all of the individual techniques used in the Vertica Query Optimizer have roots in the research literature, we hope by enumerating the combination we used helps guide future efforts and validate past work. While the wisdom of writing a totally new query optimizer for the Vertica Analytic Database was not clear eight years ago when we began, our experience has shown it was well worth the effort.

ACKNOWLEDGMENTS

Mitch Cherniack was instrumental in helping us architect and design the Vertica Query Optimizer. Shilpa Lawande, Priya Arun, Hongmin Fan, Matt Fuller, and Mingsheng Hong, while not authors of this paper, designed and wrote significant portions of the Vertica Query Optimizer. Goetz Grafe contributed useful and insightful comments to this paper and its argument. Wentian Lu performed the cost model experiments, and Matthew Fay created the internal database from which some of the results section was drawn.

REFERENCES

[1] PostgreSQL. http://www.postgresql.org/.
[2] Oracle Database Performance Tuning Guide, 10g Release 2 (10.2), part number b14211-03. Section 13.3 Enabling and Controlling Query Optimizer Features, 2009.
[3] D. J. Abadi, D. S. Myers, D. J. Dewitt, and S. R. Madden. Materialization Strategies in a Column-Oriented DBMS. In *ICDE*, 2007.
[4] S. Ceri and J. Widom. Deriving Production Rules for Incremental View Maintenance. In *VLDB*, 1991.
[5] S. Chaudhuri. An Overview of Query Optimization in Relational Systems. In *In PODS*, 1998.
[6] S. Chaudhuri and K. Shim. Including Group-By in Query Optimization. In *VLDB*, 1994.
[7] Y. Chen, R. L. Cole, W. J. McKenna, S. Perfilov, A. Sinha, and E. Szedenits, Jr. Partial Join Order Optimization in the Paraccel Analytic Database. In *SIGMOD*, 2009.
[8] D. Daniels, P. G. Selinger, L. M. Haas, B. G. Lindsay, C. Mohan, A. Walker, and P. F. Wilms. An Introduction to Distributed Query Compilation in R*. In *DDB*, 1982.
[9] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.
[10] P. Fender, G. Moerkotte, T. Neumann, and V. Leis. Effective and Robust Pruning for Top-Down Join Enumeration Algorithms. In *ICDE*, 2012.
[11] C. Galindo-Legaria and A. Rosenthal. Outerjoin Simplification and Reordering for Query Optimization. *ACM Trans. Database Syst.*, 22(1), Mar. 1997.
[12] W. J. M. Goetz Graefe. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *ICDE*, 1993.
[13] G. Graefe. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.*, 18(3), 1995.
[14] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible Query Processing in Starburst. *SIGMOD Rec.*, 18(2), June 1989.
[15] P. J. Haas, J. F. Naughton, S. Seshadri, and L. Stokes. Sampling-Based Estimation of the Number of Distinct Values of an Attribute. In *VLDB*, 1995.
[16] ISO/IEC 9075-1:1999. Information Technology – Database Languages – SQL – Part 1: Framework (SQL/Framework), 1999.
[17] ISO/IEC 9075-1:2003. Information Technology – Database Languages – SQL – Part 1: Framework (SQL/Framework), 2003.
[18] W. Kim. On Optimizing a SQL-like Nested Query. *ACM TODS*, 7(3), 1982.
[19] R. Kimball and M. Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*. Wiley, John & Sons, Inc., 2002.
[20] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The Vertica Analytic Database: C-store 7 Years Later. In *VLDB*, 2012.
[21] G. M. Lohman, C. Mohan, L. M. Haas, D. Daniels, B. G. Lindsay, P. G. Selinger, and P. F. Wilms. Query Processing in R*. In *Query Processing in Database Systems*. Springer, 1985.
[22] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden and E. J. O'Neil et.al. C-Store: A Column-oriented DBMS. In *VLDB*, 2005.
[23] A. Rosenthal and C. Galindo-Legaria. Query Graphs, Implementing Trees, and Freely-Reorderable Outerjoins. In *ICDE*, SIGMOD '90, 1990.
[24] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *SIGMOD*, 1979.
[25] S. Shankar, R. Nehme, J. Aguilar-Saborit, A. Chung, M. Elhemali, A. Halverson, E. Robinson, M. S. Subramanian, D. DeWitt, and C. Galindo-Legaria. Query Optimization in Microsoft SQL Server PDW. In *SIGMOD*. ACM, 2012.
[26] L. Shrinivas, S. Bodagala, R. Varadarajan, A. Cary, V. Bharathan, and C. Bear. Materialization Strategies in the Vertica Analytic Database: Lessons Learned . In *ICDE*, 2013.
[27] M. Staudt and M. Jarke. Incremental Maintenance of Externally Materialized Views. In *VLDB*, 1996.