Integration of VectorWise with Ingres

Doug Inkster, Actian Corporation
Marcin Zukowski, Actian Netherlands B.V.
Peter Boncz, CWI

ABSTRACT

Actian Corporation recently entered into a cooperative relationship with VectorWise BV to integrate its VectorWise technology into the Ingres RDBMS server. The resulting commercial product has already achieved phenomenal performance results with the TPC-H industry standard benchmark, and has been well received in the analytical RDBMS market. This paper describes the integration of the VectorWise technology with Ingres, some of the design decisions made as part of the integration project, and the problems that had to be solved in the process.

1. INTRODUCTION

The Ingres project of the 1970's at UC Berkeley was one of the first attempts to practically apply the theory of the relational model to solving problems of data storage and access. The project led to much research and many published papers [6], some of which were fundamental to the early development of relational database systems. Berkeley Ingres subsequently formed the basis of numerous commercial ventures including the Ingres RDBMS introduced by Relational Technology Inc., formed by several members of the original Ingres research team in 1980. While its market share is not what it once was, Ingres has been continually developed since its original market introduction. After a succession of acquisitions and following its release into open source in 2004 by CA Technologies, Ingres was reintroduced as an independent RDBMS by the newly founded Ingres Corporation in 2005, and its development and promotion has been pursued even more aggressively since then. In 2008, Ingres Corporation (renamed as Actian Corporation in September 2011) entered into a commercial agreement with VectorWise BV, a spin-off company from the Dutch research institute CWI (Centrum voor Wiskunde en Informatica). VectorWise was created to bring to market the technology that was developed by the MonetDB/X100 project [1, 2, 10]. The goal of the agreement was to integrate the VectorWise technology with the Ingres database server to produce a viable commercial RDBMS that provided the accepted components of enterprise class database management together with the extreme performance of the VectorWise engine as demonstrated in the numerous research papers generated from the MonetDB/X100 project.

2. VECTORWISE

The MonetDB/X100 research project was initiated at CWI to extend the ideas behind its open source MonetDB column store database to workloads where the data set size is significantly larger than memory. This led to the *vectorized query execution* paradigm proposed in the Ph. D. thesis of Marcin Zukowskix100phd, that conserves and even improves the raw computational efficiency of MonetDB, but avoids its policy of full intermediate result materialization, hence improving scalability. The focus on large disk-resident workloads led to further innovations in highly efficient compression methods [10] and Cooperative Scans [9] to to reduce I/O bandwidth pressure. Like its ancestor MonetDB, X100 is a column store database server designed for read-mostly applications performing complex analytic queries on large volumes of data. That said, it also employs sophisticated updating mechanisms based on Positional Delta Trees (PDTs) that give it acceptable performance in update applications from moderate to heavy [2].

The design goals of X100 were to build upon the lessons learnt in the MonetDB project[1] and to exploit features of modern computer hardware architectures to deliver improved retrieval performance to relational queries. In addition to the benefit of reduced disk access inherent in column store databases, the X100 design takes a holistic view of disk, main memory, CPU cache and CPU to produce a database server that balances resource con-

[1]See `http://www.monetdb.org`

sumption from disk bandwidth, to memory bandwidth to CPU instruction cycles. Among the more novel ideas incorporated into the X100 architecture is a lightweight compression scheme which uses different techniques depending on the data type and value distribution. The compression algorithms are simple enough to avoid diverting CPU resources from actual query execution, yet effective enough to reduce the need for disk space and increase the effective bandwidth of disk transfer. Moreover, the data is left in its compressed form even in main memory. Only when it is ready to be used by the actual operators of the executing query plan is it decompressed. This technique allows better utilization of the scarce bandwidth between CPU cache and memory.

The primary source of the improved performance realized by the VectorWise engine is its server architecture. Data flows passed through algebraic X100 operators are kept in columnar form in an effort to reduce the execution overhead of row store database engines. Moreover, the columns are processed in *vectors* of a size designed to maximize the performance achieved in CPU cache. The server functions that implement the operators are written to exhibit strong locality of reference and to take advantage of such machine features as instruction pipelining and compiler generation of SIMD (single instruction, multiple data) instructions. The vectorizing of operators to use SIMD instructions allows operations to be executed on many data items simultaneously and greatly reduces the machine cycles consumed for each tuple processed by the query engine. During the process of integration with the Ingres server, the VectorWise engine has been enhanced with other features typically associated with high performance analytical query execution. Parallel execution of complex queries using the Volcano model, revamped sorting, disk overflow hashing and many more scalar functions are just some of the improvements that have been made to VectorWise during the integration process.

The interface to the VectorWise engine is a low level, relational algebra language. Operators are defined for session management, DDL and DML (samples shown in Table 1). It is easily compiled into the code form used for execution and depends on queries being already optimized to get the best performance. Though the VectorWise compilation process does incorporate a rewriter to perform localized optimization, the generation of optimal queries is primarily the responsibility of the coder of the query. Figure 1 shows a simple example of an SQL query and the corresponding VectorWise algebra

Operators Producing a Flow of Tuples
Project(Flow input, Expr* calc)
Select(Flow input, Expr cond)
Sort(Flow input, Expr* orderby)
TopN(Flow input, Expr* orderby, int N)
Aggr(Flow input, Expr* groupby, Expr* aggregates)
HashJoin1(Flow in1, Expr* k1, Flow in2, Expr* k2)
HashJoin01(Flow in1, Expr* k1, Flow in2, Expr* k2)
HashJoinN(Flow in1, Expr* k1, Flow in2, Expr* k2)
MergeJoin1N(Flow in1,Expr* k1,Flow in2,Expr* k2)
HashSemiJoin(Flow in1,Expr* k1,Flow in2,Expr* k2, Expr cond
HashRevSemiJoin(Flow in1,Expr* k1,Flow in2,Expr* k2, Expr cond
HashAntiJoin(Flow in1,Expr* k1,Flow in2,Expr* k2, Expr cond
HashRevAntiJoin(Flow in1,Expr* k1,Flow in2,Expr* k2, Expr cond
DDL and DML Operators
CreateTable(str table, str* colspec)
Insert(str table, Flow input)
Delete(str table, Flow input, Expr* key)
Modify(str table, Flow input, Expr* key)
Start()
Commit()
Abort()

Table 1: Sample X100 Algebra Operators

query. The `MScan` operator creates a data flow containing the `sno` and `qty` columns from the `sp` base table. `MScan` stands for Merge-Scan, as it scans columnar data from disk while merging in possible committed updates present in the PDTs [2]. `Aggr` performs a hash aggregation grouping the input flow on `sno` and computing the sum of `qty` values for each group. The `Project` operator in VectorWise is used to compute new columns using input columns, functions and simple expressions – but it does not eliminate duplicates in the manner of pure relational projection. In this example, the `Project` just selects the two relevant output columns from the aggregation flow.

3. INTEGRATION WITH INGRES

The first question asked about this project was whether it was even feasible to integrate the type of leading edge database technology as embodied in VectorWise with a mature, row-based RDBMS server such as Ingres. The VectorWise server, regardless of its implementation, still provides a relational interface to the users of the data it maintains. Such is the continuing strength of the relational model that, even though user interface languages

SQL:

```
SELECT sno, SUM(qty) FROM sp GROUP BY sno;
```

VectorWise:

```
Project(
    Aggr(
        MScan('sp', ['sno', 'qty']
        ), [sp.sno], [col2 = sum(sp.qty)]
    ), [sno, col2]
)
```

Figure 1: Simple SQL query in X100 Algebra

may differ, access to contained data is achieved with analogous mechanisms. Different relational languages can almost always be compiled into compatible database interfaces. So queries, both DDL and DML, can be coded in SQL, given to the Ingres server to be processed as necessary, and then passed seamlessly by Ingres to the VectorWise engine for execution. In fact, the original Quel language of Berkeley and commercial Ingres can also be used for expressing queries on VectorWise databases.

3.1 Adding VectorWise Tables to Ingres

Two primary mechanisms were available for defining VectorWise tables to the Ingres server. Ingres has long included a gateway capability that allows tables defined in federated RDBMS', and even flat file systems, to be "registered" in the Ingres information schema. Heterogeneous queries on such tables are compiled in Ingres, then passed to the database or file servers in question for processing. Wrapper functions are required to be written for each such interface to allow exchange of definitions, data and transaction processing commands. To develop such an interface to allow access to data stored in a VectorWise database would have been a natural use of the gateway facility, one which would have been quite straightforward to implement.

The ease of defining a gateway interface from Ingres to VectorWise was offset, however, by several factors. Possibly the very first design goal of the entire integration project was to avoid burdening the VectorWise processing technology with the overheads of the Ingres row store architecture as much as possible. The gateway approach delivers rows back to the Ingres server one at a time to be returned to the user and this was deemed an unacceptable bottleneck. Moreover, for a better user experience, it was determined that the interface from Ingres to VectorWise should be as seamless as possible and putting VectorWise tables in the same category as tables created in other RDBMS' would impede our ability to provide, among other things, an integrated transactional model for updates.

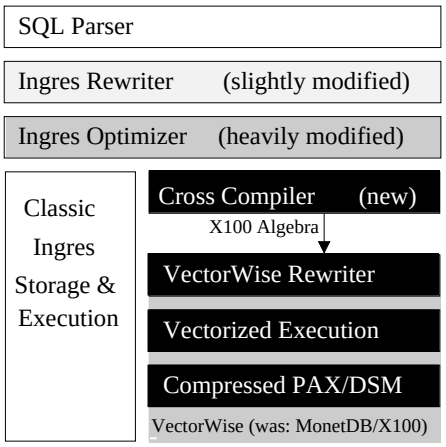

Figure 2: Architecture of Ingres VectorWise.

The decision to fully integrate VectorWise tables was consistent with the fact that Ingres already supported several table storage options, specified when tables are created. Rows of "HEAP" tables are simply stored sequentially as they arrive in the database, kept in a storage heap that extends automatically as rows are added. "ISAM" and "BTREE" are indexed sequential options and "HASH" stores rows scatters in the storage space, based on a hash of their key column values. VectorWise tables are simply defined by a new Ingres table storage option using the "structure=" parameter of the Ingres `CREATE TABLE` statement, analogous to the HEAP, ISAM, BTREE and HASH options already supported.

Various Ingres DDL statements have been extended to support the definition of VectorWise tables and indexes. Definitions of VectorWise tables and columns are recorded first in the Ingres information schema so that Ingres can recognize references to VectorWise tables and compile queries accordingly. The DDL is then passed to VectorWise so that definitions can also be recorded in local VectorWise catalogs for VectorWise engine use. Most importantly though, from the user's perspective the same interface is used to define and maintain VectorWise tables as those of native Ingres. Figure 2 depicts the final architecture of the integrated system, in which dark boxes correspond to (heavily) modified components. In the following, we discuss these modifications in more depth.

3.2 Optimizing VectorWise Queries

While the notion of integrating different table organizations into an existing database server is quite reasonable, generating optimal query plans for table structures as radically different from classic row

stores as the VectorWise column store takes a bit more of a leap of faith. However, arguably the most critical function of query optimization is the estimation of base table cardinalities in the presence of restriction predicates and intermediate result cardinalities in the presence of joins and aggregations. The remaining activities of a query optimizer are the enumeration of all possible, or at least reasonable, query plans and the association of cost estimates with each. These are primarily mechanical processes that can easily be extended to accommodate new data organizations.

Ingres was the first commercial RDBMS to use histograms as a method for representing value distributions of the data in a column, histograms being but one of numerous novel techniques introduced to the Ingres optimizer from the Ph. D. research of Robert Kooi[4]. As such, estimations of predicate selectivity and resulting cardinality have always been strengths of its query optimizer. Moreover, Ingres histograms are built by a utility that simply executes SQL queries on the underlying data. Row counts for each distinct value are accumulated in the utility itself, or by the Ingres server using aggregate queries, depending on the ratio of distinct values to rows in a table. Basic SQL retrieval support for VectorWise data allowed histograms to be built for VectorWise tables very early in the integration process and provided a basis for using the existing Ingres optimizer to generate acceptable query plans for VectorWise queries.

Therefore, the initial creation of query plans by Ingres for use in the VectorWise engine simply used the same cost algorithms as are used for queries on native Ingres tables. Certain Ingres-only join techniques were removed from the set of target strategies, but the remaining operations such as hash or merge joins were compiled identically for VectorWise and native Ingres queries. The resulting plans were quite promising, though certain characteristics of the VectorWise execution model definitely required additional consideration. Some of the changes to address these characteristics are described later in Section 3.2.2.

3.2.1 Changes to the Ingres Rewriter

Several categories of change were implemented in the rewriter component of the Ingres query optimizer. Not all functions supported by the Ingres dialect of SQL are supported in syntax by the VectorWise engine. Examples include statistical aggregate functions such as `stddev_pop`, `stddev_samp`, `var_pop`, `var_samp`, `corr`,
`regr_slope`, `regr_intercept`, and so forth. These functions all have well known expansions into more basic operators such as multiplication, division, exponentiation, etc. and are defined that way in the SQL standard[3]. So the parse tree fragments constructed by the Ingres parser for statistical aggregates are detected and rewritten by the optimizer rewriter as more primitive sequences of functions and operators that are supported by the VectorWise engine.

Like all commercial implementations of SQL, the Ingres RDBMS rewrites many forms of `WHERE` clause subqueries into flattened joins with the containing queries. This allows more efficient execution strategies to be developed. However, some forms of subquery are very difficult to flatten and native Ingres relies on a nested loop "subquery" join operator to handle such cases. For each row of the containing query, the subquery is effectively executed with the current set of correlation variables (with some optimizations to avoid continual reevaluation), to determine the truth value of the subquery predicate. Subqueries that are connected to a `WHERE` clause by an `OR` are examples of the Ingres use of the subquery join, rather than some complex flattening strategy.

The VectorWise engine does not support any nested loop join operators. This has required the development of additional novel flattening strategies in the Ingres rewriter to handle subqueries in VectorWise queries that were not flattened for classic Ingres. These new subquery flattening strategies are generic in that they are also applicable to native Ingres queries and will deliver performance benefits to existing users of Ingres. There are numerous such examples of the symbiosis between the VectorWise integration project and native Ingres.

Figure 3 shows a `>= ALL` subquery and how it can be flattened to compute the `MAX` and `COUNT` of the comparand value, as well as `COUNT(*)`, then restricts the containing query to rows for which the subquery has either no result rows or result rows that all contain `null`, and for which either all subquery rows are `null` or the maximum subquery value is less than or equal the containing row comparand value. VectorWise has the ability to re-use the results of any operation (joins, projections, etc.) to be reintroduced later in the execution of the same query, such that proper graph-shaped query plans are possible. The Ingres rewriter already has the ability to identify some common table-level expressions so that their results can be computed once, and cached (typically in a lightweight temporary table) for reuse in the same query. This technique is used, for example, when the same aggregate view is referenced more than once in a single query. For

SQL:

```
SELECT * FROM p WHERE pno >=
 ALL (SELECT pno FROM sp WHERE QTY = 100)
```

VectorWise:

```
Project (
 Select (
  Select (
   CartProd(
    MScan (
     'p', [ 'pno', 'pname', 'color', 'weight', 'city']
    ) [ 'est_card' = '6' ] ,
   Aggr (
    Select (
     MScan (
      'sp', [ 'qty', 'pno']
     ) [ 'est_card' = '12' ] , ==(sp.qty, sint('100'))
    )  [ 'est_card' = '1' ] ,
    [] ,
    [TRSDM_3 = max(sp.pno),_TRSDM_4 = count(*),
     jTRSDM_5 = count(sp.pno)] , 200
    ), 1
   ), || (==(TRSDM_4,TRSDM_5), ==(TRSDM_4, sint('0')))
  ), || (==(TRSDM_4, sint('0')), >=(p.pno,TRSDM_3))
 )  [ 'est_card' = '3' ] ,
 [p.pno, p.pname, p.color, p.weight, p.city])
```

Figure 3: Flattened <compop> ALL subquery.

VectorWise queries, however, the Ingres rewriter has been extended to perform a more general search for common expressions. Resulting query fragments are then executed once with their results cached by the VectorWise engine. The cached result may then be referenced elsewhere in the same query without the need to rematerialize the rows from scratch.

3.2.2 *Optimizer Changes*

While the Ingres query optimizer generated acceptable plans for VectorWise queries almost immediately, there are certain capabilities that are specific to the VectorWise engine that required additional properties to be tracked during query plan enumeration. Ingres has long supported referential and primary key/unique constraints; however its optimizer has never exploited the presence of constraints while compiling queries. Accurate histogram-based cardinality estimates of restrictions and joins largely superseded the need to consider the implications of join predicates that map to referential relationships between tables in a query. In contrast, VectorWise has several features that can only be targeted with knowledge of the constraints that underlie the tables referenced in a query. As a result, a new information schema table was defined in Ingres to define more compactly the mapping of columns in a referential relationship. Rows are added to this table during the execution of the DDL statements that define referential constraints. Corresponding code was introduced to the Ingres optimizer to identify join predicates in a query that map to referential relationships.

The first benefit of such information is that the optimizer can more easily identify and estimate costs for joins that can take advantage of the join indexes that VectorWise will optionally create to support referential relationships. Merge joins between a join index in a referencing table and the tuple ID of the corresponding referenced table are the most efficient joins that VectorWise can perform.

Referential relationships and unique/primary key constraints are also the basis (along with the grouping columns of intermediate aggregate results) of the functional dependency property tracked for each node of a query plan for a VectorWise query. Functional dependencies allow the leveraging of a feature of the VectorWise aggregation operators. Those operators include an operand list describing the aggregate results to be computed and an operand list describing the grouping columns upon which the aggregation is to be performed. Columns in the grouping list whose values are functionally dependent on one or more other columns in the list do not logically contribute to the determination of distinct groups over which the aggregation is performed. As such, VectorWise allows them to be designated with the `DERIVED` attribute, in which case they are left out of the grouping operation. The requirement to include all columns of an SQL select-list in either an aggregate function or the group by clause of the query leads to many queries that include large text columns in grouping lists, simply to allow them to be included in the result row. Use of the `DERIVED` attribute allows such columns in the grouping column list without imposing a potentially significant overhead on the hashing and comparison operations used to group the data. Figure 4 shows an aggregation on `sno` and `sname`. By virtue of the unique constraint on `sno`, `sname` is functionally dependent on it and can be flagged as `DERIVED` in the aggregation, not needing to be part of the actual grouping operation taking place. Like most relational query optimizers, the Ingres optimizer tracks ordering properties of nodes of a query plan. This allows it to eliminate sorts at various points in the query plan and to use the much more efficient ordered aggregation operator to perform grouping operations, rather than relying on hash techniques. However, ordered grouping only requires its input to be clustered on the grouping column values. That is, all of the rows with the same values in the grouping

SQL:

```
  SELECT s.sno, sname, SUM(qty)
    FROM s, sp
   WHERE s.sno = sp.sno
GROUP BY s.sno, sname;
```

VectorWise:

```
Project (
 Aggr (
  HashJoin01 (
   MScan (
    'sp', [ 'sno', 'qty']
   ) [ 'est_card' = '12' ] , [ sp.sno ],
   MScan (
    's', [ 'sno', 'sname']
   ) [ 'est_card' = '5' ]  , [ s.sno ]
  ) [ 'est_card' = '5' ] ,
  [s.sname DERIVED, s.sno] ,
  [col3_3 = sum(sp.qty)
 ) [sno_1 = s.sno, sname_2 = sname, col3_3]
)
```

Figure 4: DERIVED designator in aggregation

columns must be contiguous. Clustering is a weaker property than ordering, and the Ingres optimizer has been extended to track it in addition to ordering to enable the identification of grouping operations that can be performed with the more efficient VectorWise `OrdAggr` operator, rather than the hash-based `Aggr` operator.

3.3 Query Plan Cross Compiler

The key component in the integration of Ingres and VectorWise is a cross compiler which generates a VectorWise query from an optimized Ingres query plan. Ordinarily, the Ingres query compiler transforms the optimized query plan into a code form executable by the Ingres query execution engine. For a query on a VectorWise database, the Ingres code generator invokes the cross compiler to transform the optimized plan into query syntax to be executed by the VectorWise engine. The cross compiler traverses the optimized plan using recursive descent, much like the code generator does for native Ingres code generation. But as the plan tree is descended, syntax is emitted for each of the corresponding operators of the VectorWise relational algebra, and on return from the descent the columns, expressions, literals and other parameters of the operators are emitted. The result is a query which builds data flows from base table access and successively refines them through the execution of the nested operators such as joins, aggregations and unions.

One source of difficulty within the cross compiler is the fact that the VectorWise query references database entities (tables and columns) by name, whereas Ingres query plans compile into buffer references and offsets. In fact, entity names are effectively removed from the query when it is turned into a parse tree – before query optimization takes place. The cross compiler goes to great lengths to avoid name ambiguities and scope errors by generating distinct table and column names and making heavy use of table name qualifiers in column references. Different instances of the same table and column names are modified to assure distinctness, sometimes at the expense of readability of the resulting syntax. A simple exception table is used to identify Ingres expression operators that do not have an exact equivalent in the VectorWise algebra. For example, VectorWise supports `is_null` and `like` operators, but not the negated `is_not_null` or `not_like` operators. The exception table has entries for the Ingres `is not null` and `not like` operators that result in the `!` (not) operator being prepended to the supported VectorWise `is null` and `like` operators.

There are also numerous examples of implementation effort being divided between the VectorWise engine and the cross compiler. VectorWise does not have native support for windowed functions such as `rank()`, `dense_rank()`, `ntile()` and so forth. To relieve the VectorWise development team of the entire burden of implementing these complex functions, the cross compiler took on the responsibility of generating syntax to perform the partitioning and ordering, as well as the joining together of results of window functions using differing window specifications. The VectorWise engine supplies more primitive functions to first partition the input rows, then to identify peer groups within each partition based on the requested window ordering. The results of those functions are then used to compute the various windowed functions supported by Ingres/VectorWise.

Example 5 shows a query that computes the `rank()` function over two distinct window specifications. The cross compiler computes the first rank value and a sequentially assigned row number for each row in the first pass of the result set. This initial result is cached using the VectorWise intermediate result re-use mechanisms described in Section 3.2.1, which surfaces here in the assignment into the `IIWINSQNAME1` identifier of a subquery result, and the subsequent re-use of this identifier. The cached result is re-sorted to generate the second rank value and the results of the two computations are joined back together on the computed row number.

SQL:

```
SELECT sno, pno,
   RANK() OVER (PARTITION BY sno ORDER BY qty)
        AS srank,
   RANK() OVER (PARTITION BY pno ORDER BY qty)
        AS prank
FROM sp;
```

VectorWise:

```
Project (
   HashJoin01(
       IIWINSQNAME1 =
      Project (
         Sort (
            MScan (
                'sp', [ 'qty', 'sno', 'pno']
            ) [ 'est_card' = '12' ] , [sp.pno,
                          sp.qty]
        ), [TRSDM_0 = diff(sp.pno),
          TRSDM_1 = rediff(TRSDM_0,sp.qty),
          IIWINRNUM10  = rowid(uidx('0')),
          sp.qty, sp.sno, sp.pno, prank =
                  sqlrank(TRSDM_0,TRSDM_1)]
      ), [IIWINRNUM10],
      Project(
         Sort(
            Project(
               IIWINSQNAME1, [IIWINRNUM11 =
                   IIWINRNUM10, sp.sno, sp.qty]
            ), [sp.sno, sp.qty]
         ), [TRSDM_3 = diff(sp.sno),
           TRSDM_4 = rediff(TRSDM_3,sp.qty),
           srank =sqlrank(TRSDM_3,TRSDM_4),
           IIWINRNUM11]
      ), [IIWINRNUM11]
   ), [sp.sno, sp.pno, srank, prank]
)
```

Figure 5: Windowed functions with differing window specifications

3.4 VectorWise Rewriter

The X100 Algebra plans created by the cross compiler lack certain details that were felt not relevant during the (join-order) query optimization phase. These typically concern highly VectorWise-specific features. For instance, VectorWise represents decimal and date/time data types using simple integers of different widths (1, 2, 4, 8 and 16 bytes) that are carefully selected to be of the minimal width needed to prevent overflow. This selection is based on min/max statistics kept for all columns. This mapping of logical SQL data type to physical type is performed by a VectorWise rewriter that operates on the already optimized algebraic query plan, as a post-processing step.

Similarly, VectorWise represents `null`-able columns as a pair of a value- and boolean-column, where the boolean column indicates whether the tuple value `is_null`. This representation and resulting processing model keeps functions null-oblivious and avoids the need for null checks deep inside the execution primitives. This absence of checks in turn conserves the efficiency of the vectorized model, maximizing e.g. SIMD opportunities. The extra boolean `null` columns, and the propagation of these, are also handled by a phase in the VectorWise rewriter.

For this new VectorWise rewriter we used a pattern-matching tool called Tom[2] that can be embedded in C/C++ code to aid with the formulation of rewrite rules. The presence of a separate, column-oriented rewriter has aided the project as certain functionality can be engineered with much less impact to the main Ingres optimizer. Besides the mentioned physical typing and null optimizations, there are various other rules implemented and many opportunities for new rules. Notably, this column-oriented rewriter was also used to implement multi-core query parallelization. Thus, parallelism is planned posterior to main join-order query optimization. The same approach is taken by many other systems, but may miss possibilities an integrated optimization method could potentially spot. Thus, the flexibility of having a post-processing VectorWise rewriter also leads to certain design trade-offs of the scope of query optimization versus the complexity of development.

3.5 Query Execution

The VectorWise server runs in a separate process from the Ingres server. It is initiated by a VectorWise interface layer that exists in the Ingres server. This does not occur until the first user request to Ingres is received that actually requires access to the VectorWise server. Execution by Ingres of queries destined for the VectorWise server is a relatively straightforward process, primarily following the paradigm of making the interface as simple as possible to take full advantage of the performance of the VectorWise engine. As mentioned in Section 3.1, DDL statements that affect the Ingres information schema are executed first in Ingres to perform necessary information schema changes, then passed to the VectorWise interface where they are converted to X100 algebra for delivery to the VectorWise engine where an analogous process records information in the VectorWise catalog. Transaction synchronization is performed to assure that errors in either the Ingres server or the VectorWise server result in rollback of the effects of the statement in both servers.

Queries are optimized in Ingres and transformed to X100 syntax by the cross compiler as described in Section 3.3. The syntax is then delivered to the

[2]See `http://tom.loria.fr`

VectorWise interface, along with the address of a vector of row buffers to expedite the return of retrieved result rows from the VectorWise server to the user. The size of the buffer array is configurable. As rows are returned to the VectorWise interface in the Ingres server, they are reformatted to match the data types expected by the user and contained in a tuple descriptor that is passed to the interface along with the query.

`UPDATE` and `DELETE` statements are executed in the same manner, though `INSERT` statements go through the usual Ingres insert processing and are diverted to the VectorWise interface before the row is written. A corresponding VectorWise `append` operation is prepared there and delivered to the VectorWise engine for execution. `INSERT .. SELECT` and `CREATE TABLE .. AS SELECT` statements differ in that the retrievals that provide the row images to be inserted may be executed on native Ingres tables or on VectorWise tables. This allows a path for converting Ingres tables to VectorWise tables for existing users. When the `SELECT` references Ingres tables, row images are built from the result rows of the retrieval in the Ingres query executor and delivered in arrays to a bulk load interface to the VectorWise engine, analogous to that used for the Ingres `COPY` statement. If the `SELECT` only references VectorWise tables, a VectorWise statement is composed by the cross compiler to stream the result of the retrieval into a VectorWise `append` operator that stores the newly formatted rows. Session and transaction management details are exchanged between the servers to assure recovery from any type of statement failure.

4. FUTURE ENHANCEMENTS

The short term goals of Ingres/VectorWise include functionality enhancement and further integration with Ingres and native Ingres tables. Currently, queries may reference all VectorWise or all native Ingres tables, but not a combination of the two (with the exception of `INSERT .. SELECT` and `CREATE TABLE .. AS SELECT` statements as described in Section 3.5). For the benefit of existing users of Ingres, it is desired to permit heterogeneous queries that combine results from both native Ingres and VectorWise tables. The Ingres query execution model certainly permits such sophisticated query plans, but a significant amount of work is required both to optimize heterogeneous plans and to handle appropriate interfacing between the two engines as the queries are executed.

Another future project will be the enhancement of the Ingres query optimizer to include more specific knowledge of VectorWise cost parameters in the building of query plans. The strategy in the initial release of the product has been to track certain VectorWise-specific properties during the building of a query plan and take advantage of them when generating the corresponding VectorWise query. This enhancement will associate costs with the properties being tracked so that the presence of such properties can influence the selection of query plans.

Column store engines offer an opportunity to implement finer granularity query optimization. Traditional query optimizers focus on strategies for table access and joining. All columns from a given table that are required to solve a query are typically retrieved once and carried through the remainder of the execution of the query. Column stores offer the opportunity to defer retrieval of some columns from a table until the evaluation of restrictions on other columns from the same table have significantly reduced the cardinality of intermediate results. Longer text columns may then be retrieved later in query execution and joined to the intermediate results based on their ordinal positions in the table. This type of optimization is under consideration for both the Ingres optimizer and the VectorWise rewriter.

The VectorWise team maintains a working relationship with CWI, including an active intern program. Research projects include cooperative scans, advanced compression techniques and "just in time" compilation [5]. Because of its academic roots, the VectorWise/X100 project is widely known in the research community. Numerous university research facilities in Europe and North America have entered the academic licensing program.

A variety of projects are underway at several of these schools, including fine-grained multi-table clustering structures and XML storage using the Pathfinder [7] XQuery compiler. An obvious benefit to Ingres is the leveraging of this research into future enhancements to the Ingres/VectorWise commercial product.

5. SUMMARY

Considerable risk was involved in attempting the integration of leading edge technology such as that embodied by the VectorWise engine with a mature RDBMS such as Ingres, much of whose architecture was developed over 15 years ago. In particular, the mixing of column store concepts with row store concepts was not an intuitive thing to do. Due largely to the robustness of the relational model, this integration project has seen considerable success already. The integrated product has been publicly available since July 2010 and already has been well

accepted by the user community. In the first half of 2011 TPC-H results where published for VectorWise on the 100GB, 300GB and 1TB sizes, all using single servers. The 100GB result (QphH 251,561.7), is quite comparable with a Microsoft SQL Server 2008 R2 Enterprise Edition result (QphH 73,974) as both tests where performed on a dual-socket HP DL380 server with 144GB RAM and in total 12 Intel Xeon X5680 cores. In addition to the record breaking results for non-clustered systems in these TPC-H benchmarks, Ingres/VectorWise has been certified by numerous application vendors, with more on the way.

6. ACKNOWLEDGEMENTS

We would like to recognize the contributions of the late Bob Kooi to the commercial Ingres optimizer, many of which survive in Ingres to this day.

7. REFERENCES

[1] Peter Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, 2005.

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

[3] International Standards Organization. *(ANSI/ISO/IEC) 9075-2,* SQL *Foundation*, 2008.

[4] R. Kooi. *The Optimization of Queries in Relational Database Systems*. PhD thesis, Ph. D. Thesis, Case Western Reserve University, 1980.

[5] J. Sompolski, M. Zukowski, and P. Boncz. Vectorization vs. compilation in query execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, pages 33–40. ACM, 2011.

[6] M. Stonebraker. *The INGRES Papers: Anatomy of a Relational Database System*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[7] J. Teubner et al. Pathfinder: Xquery compilation techniques for relational database targets. *Technical University of Munich, Munich, PhD thesis*, 2006.

[8] M. Zukowski. Balancing vectorized query execution with bandwidth-optimized storage. 2009.

[9] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Cooperative scans: Dynamic bandwidth sharing in a dbms. In *Proceedings of VLDB*, pages 723–734. VLDB Endowment, 2007.

[10] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-Scalar RAM-CPU Cache Compression. In *ICDE*, 2006.