

(19) World Intellectual Property
Organization
International Bureau



(43) International Publication Date
27 January 2005 (27.01.2005)

PCT

(10) International Publication Number
WO 2005/008529 A2

(51) International Patent Classification⁷: **G06F 17/30**, 9/44

(21) International Application Number:
PCT/US2004/021672

(22) International Filing Date: 7 July 2004 (07.07.2004)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
60/485,321 7 July 2003 (07.07.2003) US
60/485,638 8 July 2003 (08.07.2003) US

(71) Applicant (for all designated States except US):
NETEZZA CORPORATION [US/US]; 200 Cross-
ing Boulevard, Framingham, MA 01702 (US).

(72) Inventors; and

(75) Inventors/Applicants (for US only): **ZANE, Barry, M.**
[US/US]; 4 Cobblestone Circle, Wayland, MA 01778
(US). **BALLARD, James, P.** [US/US]; 379 Fowler Road,
Northbridge, MA 01534 (US). **HINSHAW, Foster, D.**
[US/US]; 22 Campbell Park, Somerville, MA 02144 (US).

YERABOTHU, Premanand [IN/US]; 63 Latisquama
Road, Southborough, MA 01772 (US). **KIRKPATRICK,**
Dana, A. [US/US]; 146 Whitney Street, Northborough,
MA 01532 (US).

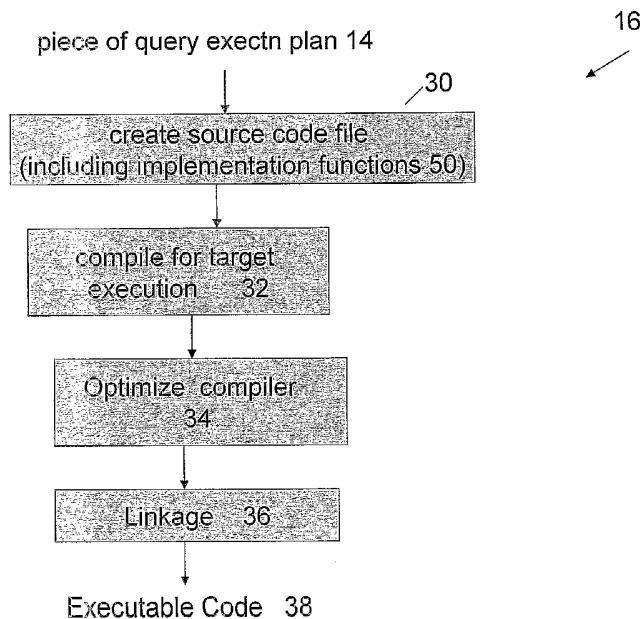
(74) Agents: **WAKIMURA, Mary Lou** et al.; Hamilton,
Brook, Smith & Reynolds, P.C., 530 Virginia Road, P.O.
Box 9133, Concord, MA 01742-9133 (US).

(81) Designated States (unless otherwise indicated, for every
kind of national protection available): AE, AG, AL, AM,
AT, AU, AZ, BA, BB, BG, BR, BW, BY, BZ, CA, CH, CN,
CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, EG, ES, FI,
GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE,
KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD,
MG, MK, MN, MW, MX, MZ, NA, NI, NO, NZ, OM, PG,
PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, SY, TJ, TM,
TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM,
ZW.

(84) Designated States (unless otherwise indicated, for every
kind of regional protection available): ARIPO (BW, GH,
GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM,
ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM),
European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI,

[Continued on next page]

(54) Title: OPTIMIZED SQL CODE GENERATION



(57) Abstract: This invention relates generally to a system for processing database queries, and more particularly to a method for generating high level language or machine code to implement query execution plans. The present invention provides a method for generating executable machine code for query execution plans, that is adaptive to dynamic runtime conditions, that is compiled just in time for execution and most importantly, that avoids the bounds checking, pointer indirection, materialization and other similar kinds of overhead that are typical in interpretive runtime execution engines.



FR, GB, GR, HU, IE, IT, LU, MC, NL, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

Published:

— *without international search report and to be republished upon receipt of that report*

OPTIMIZED SQL CODE GENERATION

RELATED APPLICATION

This application claims the benefit of U.S. Provisional Application No. 60/485,321, filed July 7, 2003 and 60/485,638, filed on July 8, 2003 the entire
5 teachings of which are incorporated herein by reference.

BACKGROUND OF THE INVENTION

When using a declarative query language like SQL, a database client enables generation of queries that say 'what to do' with a database, but not 'how to do it'. It is the task of a database management system (DBMS) to translate the client's
10 declarative specification (query) into an operational procedure. When a client specifies a query like:

```
select lastName, firstName from emp where dept = 'software'
```

the DBMS parses and analyzes the query to form a high-level execution plan. This plan might consists of several steps:

- 15 1. Scan the set of records in the "emp" table
2. Restrict the set to just those records whose "dept" column value is 'software'
3. Project the "lastName" and "firstName" columns from the resulting records
- 20 4. Return a set of <lastName, firstName> tuples

This sequence of steps will satisfy the client's SQL query in principle, but it is not executable code by itself. Many details have been left unspecified. The task of turning a query execution plan into executable code is typically accomplished in the interaction between two DBMS components: a code generator and a runtime
25 execution engine.

A code generator takes a high-level execution plan, fills in details, and produces code that the runtime engine can execute. In some DBMSs, the runtime engine is a powerful, high-level interpreter of execution plans. In these systems, the code generator might merely need to provide some annotations of the execution plan, such as field offsets or names. In other DBMSs, the code generator might produce executable machine code that is dynamically linked and called by the runtime engine. Many DBMSs opt for a middle ground, where the code generator produces instructions in an intermediate "p-code", which is interpreted by the runtime engine.

Systems where a code generator produces executable machine code for ad-hoc queries are very rare. One reason is that it is much easier to write and debug an interpreter. It is also true that DBMS source code is less "portable" to different computer platforms when it generates code targeted at a specific chip set. But a more compelling reason is that the performance advantages of executing compiled machine code in the runtime engine must be balanced against the time required to produce this code in the code generator. In environments where there are many small, short queries, the cost of producing executable code may be greater than the runtime benefit. In environments with I/O-bound queries (where the cost of evaluating the query is dominated by the time it takes to read records from disk), the advantage of saving a few CPU cycles by running machine code may be irrelevant, or too small to be worth the bother.

SUMMARY OF THE INVENTION

The present invention addresses the problems of the prior art and improves on the prior art systems for processing database queries. The present invention provides an optimized method for generating executable machine code to implement query execution plans.

A computer system (method and apparatus) for generating high level language source code (executable code) in response to a received query in a database management system is disclosed in related U.S. Patent Application No. _____ (Attorney Docket No. 3336.1024-001) and herein incorporated by reference. The present invention provides optimizations for such a computer

system. The optimizations reduce or minimize compilation time of the generated source code and reduce or minimize resulting execution time. These time savings enable the present invention to achieve high speed and hence high volume query processing throughput as heretofore unseen by the prior art.

5 In particular, the present invention involves several techniques for generating executable code to implement query execution plans in a way that eliminates interpretive overhead, generates only needed code and structures and adapts to dynamic conditions. Different degrees of optimization and hence different ones of the invention techniques are used for different expected execution times of subject
10 queries. The longer the expected execution time, the higher degree of optimization is used.

 In one embodiment, for each piece of a subject query execution plan the present invention generates source code as a function of data characteristics, current conditions and/or workload. The generated source code is then able to be compiled
15 in a manner that optimizes compilation time.

 The data characteristics include precision of a value existing in the database, scale of a value existing in the database, size of affected data in the database and/or data type.

 The present invention selects code generation techniques based on
20 intermediate results from earlier pieces of the subject plan. In this sense, the present invention code generation adapts to dynamic conditions.

 The present invention code generation generates source code for each piece of the plan on a need-only basis. For example, source code for only those relevant members of a declared structure are generated. In a preferred embodiment, the
25 present invention minimally defines relevant structures and classes. From those definitions (declarations), the present invention forms optimized "Include" statements in the generated source code which compile at reduced compilation times.

 In the case where a subject query includes a join operation, the present
30 invention represents the output stream of the join operation as local variables that reference current records in each input stream. The named fields of the records are

then projected. This minimizes intermediate materialization and the use of auxiliary structures.

In a preferred embodiment, the projecting of named record fields utilizes structure offset expressions in the generated source code. These expressions
5 compile into a faster form than interpreted offsets.

In another feature, the present invention adjusts widths of the variables in the generated source code to be only as large as the actual respective data values in the database. In the preferred embodiment, this adjustment is based on determined (tracked) range of values in the database tables.

10 In another technique of the present invention, each instance of a function call to a respective function, recited in the source code may be replaced by the source code implementing the respective function itself. First, the size of the data affected by the function call is determined. Next, based on this determination, the respective function code is in-lined in the generated source code. In some embodiments, the
15 determination may be made with respect to a threshold. The threshold may be user configurable.

Another aspect of the present invention concerns outer join operations. If the subject query includes an outer join, the present invention effectively overwrites field references in the outer join operation with Null values. Thus the generated
20 source code uses Null value indicators or the like, and enables compilation time to be optimized.

These and other features of the present invention will be further described later. Generally speaking, advantages of computer systems employing the present invention optimized DBMS are as follows.

Avoiding unnecessary computations and checks

An executable code generator is useful in environments where there are frequent long or computationally complex queries. In such cases it is valuable to
5 reduce computation overhead to a minimum, particularly so when it occurs in an “inner loop”. Query execution engines that interpret plans or p-code typically pay a performance penalty for extra checking, conservative assumptions about arithmetic, pointer indirections, and suboptimal memory usage. Code generation can avoid these kinds of overhead.

10 Checks can be generated only when there’s a possibility they may be necessary. When numeric fields have a suitable scale and precision, machine instructions that directly support that scale and precision can be used to advantage. Instead of referencing a field as an indirection from a base pointer plus a field offset, the code generator can place the address of the field directly in a register. Memory
15 management overhead can be reduced by placing working variables on the program stack, and by avoiding intermediate materialization. Specific examples of these cases are considered below.

Avoiding checks for null values

20 The code generation techniques used in the invention can eliminate unnecessary checking for null values. Databases typically allow record fields to have a “null” value – which means that no value for the field has been specified. Some systems implement the null value as a special data value; but it is more common to track null values in a special area, that is invisible to the user. One
25 common technique is to use an extra invisible field in each record that contains a bit-vector. If the n^{th} bit is set, that means that the n^{th} field is null. Whatever mechanism is used to represent null values, query execution plans must frequently check for the presence of null values. However sometimes it’s possible to know that a field cannot hold a null value. A SQL column definition may preclude the possibility of
30 null values. Even if a column definitionally allows null values, database statistics may show that there are no instances of null values for that column at the time of the

query. In such cases, the code generator avoids generating the code to test for a null value.

Using natively-supported arithmetic

5 Another situation of interest involves the precision and scale of arithmetic expressions and aggregations. Say the SQL implementation supports 128-bit math for a numeric data type having maximum declared precision of 38 decimal digits, but in one embodiment there is native support only for 16 and 32-bit arithmetic. Sometimes the declared precision and scale of column definitions allows the code
10 generator to “know” that it can safely use the natively supported arithmetic functions. Other times, the actual audited minimum and maximum values of columns imply that natively supported arithmetic can safely be used at the time the query is evaluated. In these situations CPU time can be save by avoiding overflow checking and by using natively supported instructions. Importantly, the code
15 generator tracks the precision and scale of the result of a numeric expression, so that compound and intermediate expressions are assigned a precision and scale derived from the precisions and scales of their components.

By way of an example involving aggregation, consider a large table that is horizontally partitioned across a thousand nodes, with each node holding 100,000
20 rows. Assume that the table has a numeric column of scale zero, whose maximum value is 250; and that a query wishes to compute the sum of that column. The query execution plan calls for each of the thousand nodes to compute a local sum, and then to aggregate these results into a final global summation. On each of the thousand nodes, the maximum value for the local summation can be computed using 32-bit
25 arithmetic (8 bits for holding each record value, and 17 bits for representing each row). But the global aggregation would require 35 bits in the worst case, and would therefore require 64-bit arithmetic.

By way of an example involving a scalar expression, consider a table having numeric column A of declared precision and scale (15,2), and numeric column B of
30 declared precision and scale (10,1). Say we are selecting the expression A * B (i.e. the product of columns A and B). Considering only the declared shapes of the components of the calculation, the result would have a precision and scale of (25,3)

and evaluation would require 128-bit arithmetic. But suppose the actual audited minimum and maximum values of columns A and B are such that those values would fit in precisions and scales as small as (5,2) and (3,1), respectively. The product of such values would have a precision and scale of (8,3), permitting
5 evaluation in narrow native arithmetic operations.

The above technique for the evaluation of expressions and aggregations of the numeric data type can be simply extended to include operations on integer data types by recognizing that an integer is no more than scale 0 numeric.

10 Compiled Field References

Two of the most frequent tasks of the runtime engine are to retrieve or modify the value of a field in a record or tuple. Essentially this involves computing the address of the field as an offset from the start of the record or tuple. In the invention, this computation occurs once at compile time (for fixed-length fields or
15 for the first varying field). In an interpretive runtime engine, the computation must typically be performed for every access. While the computation is very fast, when operating across millions of records, or joins whose Cartesian product is extremely large, this small overhead adds significantly to overall latency.

20 Avoiding intermediate materialization

Queries that contain a join of two or more tables logically form the cartesian product of the constituent records. Consider a simple query such as:

```
select emp.name, dept.name from emp, dept, where emp.dept_no = dept.id
```

The target response to the query is to return a two column result consisting of the
25 name of an employee in the emp table and the name of the employee's department. One way to conceptualize this procedure is to imagine a stream of records from the "emp" table being combined with another stream of records coming from the "dept" table, with an output stream consisting of records with two columns for the emp.name and the dept.name.

30 In some systems, the output stream literally consists of new records, residing in memory, consisting of the emp.name value followed by the dept.name value.

This is called a “materialization”. Because allocating memory and copying data is expensive, it is desirable to delay materialization for as long as possible.

The invention represents the output stream of the join operator as two local variables that reference current record in each stream, and projects out the name
5 fields of the records via structure offset expression, such as “currentRecordPtr->name”. This approach avoids the memory allocation and copying overhead of materialization until data values must be returned.

Adapting executable code to dynamic condition

10 When a query execution plan is divided into a number of pieces, it’s possible that the result of executing an earlier piece of the plan can impact the code generation choices for subsequent pieces of the plan. As previously discussed, it could be that the range of values returned by one piece is such that 32-bit arithmetic can be used in a subsequent piece. Or it could be that the number of results returned
15 by a piece of the plan is small enough that a subsequent piece can use an in-memory sort, rather than a disk-based sort. The ability to adapt generated code to the dynamic intermediate results of a query opens important avenues for performance enhancement.

20 BRIEF DESCRIPTION OF THE DRAWINGS

The foregoing and other objects, features and advantages of the invention will be apparent from the following more particular description of preferred embodiments of the invention, as illustrated in the accompanying drawings in which
25 like reference characters refer to the same parts throughout the different views. The drawings are not necessarily to scale, emphasis instead being placed upon illustrating the principles of the invention.

Fig. 1 is a block diagram of a DBMS implementing the present invention.

Fig. 2 is a schematic view of a sequence of processes of a high level query plan generated by a query optimizer component of the system of Fig. 1.

30 Fig. 3 is a block diagram of the code generator component of the system of Fig. 1.

Fig. 4 is a flow diagram of the method used by the code generator of Fig. 3 to generate source code in a preferred embodiment.

DETAILED DESCRIPTION OF THE INVENTION

5 Illustrated in Fig. 1 is a database management system (DBMS) implementing the principles of the present invention. An SQL query 12 is input (e.g., from a database client) to the DBMS 10. In response, the DBMS 10 initially parses and analyzes the input SQL query 12 and generates a respective high level execution plan 14 having a sequence of pieces and corresponding processes. This is
10 accomplished by a set of front-end components described below. A code generator 16 generates source code in a high level language for each piece (process) of the generated execution plan 14. The code generator 16 also compiles the source code for execution in a target environment and generates linkage information. The resulting executable machine code 38 is executed or run in a runtime execution
15 engine 18 to perform a search on and retrieve requested data from the database corresponding to the SQL query 12.

The front-end components of DBMS 10 include a query parser 13, a query analyzer 15 and a query optimizer component 17. The query parser component 13 takes a native SQL query 12 and produces a parse tree structure using known
20 techniques. The query analyzer component 15 takes the parse tree and produces an initial query execution plan. In turn, the query optimizer component 17 examines the initial query execution plan in conjunction with various runtime statistics and alters the plan to reduce its estimated compilation and execution costs. The result is an optimized high level execution plan 14 formed of a sequence of "pieces" which
25 are targeted at different execution locales.

For example, in a distributed database environment, the plan may be subdivided into several pieces, where each piece may be distributed to and executed upon a different computational platform.

Further the high-level plan 14 and its various pieces consists of a sequence of
30 one or more "processes", where each process describes a particular kind of processing (set of steps) to be performed. Fig. 2 is illustrative. Records are

processed by each process in sequence, and the results, if any, are passed on to the next process in the sequence.

In particular, the first process 20 of a query plan 14 involves a scan of a disk or a network or a memory cache for source records. The last process 24 of a query plan involves returning values derived from those source records. At the intermediate processes 22, there are a large number of different computational possibilities.

Returning to Fig. 1 and with reference to Fig. 3, code generation by code generator 16 occurs as each piece of a query plan 14 becomes ready to execute. As illustrated at 30 in Fig. 3, a source code file is created for a subject piece of the plan 14, and code that implements the processes 20, 22, 24 in that piece is written in a high-level language and stored in the source file. In one embodiment, code is written in the C++ language, but other compilable languages could be used as well. Producing assembler code would also be possible, but would limit execution to particular hardware platforms. Going to a high level language enables the present invention optimizations such as register tracking and other optimizations supplied by the compiler.

In a preferred embodiment, different code generation techniques are employed at 30 for different pieces of the plan depending on data size (and other characteristics), current conditions and/or workload. For example, code generator 30 selects code generation techniques based on intermediate results of earlier pieces of plan 14. As such, source code generation is adapted to dynamic conditions. This and other examples will be made clear below.

After the source code is written to the source code file at 30 (Fig. 3), the source code file is compiled 32 for the target execution environment. The degree or level of compiler optimization 34 enabled can be a function of the expected query execution time. Queries with minimal expected execution can use a lower level of compiler optimization. On the other hand, queries with a relatively long (Typically multiple seconds) expected execution time receive a higher level of compiler optimization at 34. If the target execution environment 18 uses a different computer architecture than does the compilation environment, then a cross compiler is used to perform the compilation 32.

Finally linkage information 36 is generated so that the compiled code can be dynamically linked into a running executable 38.

The Structure of a Source Code File 30

5 In the preferred embodiment, there is regularity to the structure of each source code file at 30 (Fig. 3). First, there is a comment that describes the piece of the plan that is implemented in the file. Next come preprocessor definitions and #include directives. Third are definitions describing the structure of tuples flowing into and out of each node of the piece of the plan 14. Fourth are static function
10 declarations for any "helper" functions. Fifth are one or more functions implementing the processes of the piece of the plan 14. In general, there is an implementation function for each process; but the implementations of some processes are combined into a single function. Last are the implementations of "helper" functions.

15

The Structure of an Implementation Function 50

There is regularity as well to the functions 50 that implement the processes of the plan 14. The signature of such functions 50 is always the same. They 50 take as inputs a plan object, a pointer to the start of an incoming record stream, a pointer
20 to the end of the record stream buffer, and a Boolean indicator of whether the function 50 is being called for the last time. The functions 50 return no values, but they typically modify the plan object passed in, so as to reference the values produced by the implementation of the plan process 20, 22, 24.

The body of the implementation functions 50 consists of four segments.
25 First is a header section containing variable declarations and initializations. Next comes a loop and whatever declarations, initializations and resource allocations are required to initialize the loop. The purpose of the loop is to iterate over the records in the input stream. Third comes the body of the loop. This is where individual records are processed according to the dictates of the particular process 20, 22, 24.
30 Finally, a footer section is generated to perform post-processing, release any

resources and update the plan 14 to return results or connect to the next process /
implementation-function 50 in the piece of the plan 14.

In one embodiment, multiple plan processes 20, 22, 24 can share the same
loop, which avoids materializing intermediate results sets in a plan 14. Most
5 typically, the loop is associated with the ScanNode. All processes 20, 22, 24 that
follow the ScanNode have their body in the same loop until a materializing process
occurs. In this embodiment, loops occur only for materializing processes.

The Approach Used for Code Generation

10 Code generation is driven by the processes of a piece of a plan 14. For every
type of plan process, there are generation functions that emit portions of the process
implementation functions 50 described above. Generating a process 20, 22, 24
implementation function 50 is accomplished by calling four functions in sequence:
(1) to generate the signature of the function and to emit stack-based variable
15 declarations, (2) to generate the for loop, (3) to generate the body of the for loop and
(4) to generate footer code that comes after the for loop. These four functions may
be implemented differently for different types of plan processes – so that the
generated code performs the function described by the process. In essence, the code
generation occurs in an object-oriented fashion, dispatching its implementation on
20 the type of the process that is generating code.

Next described are the methods or procedures employed in code generation
16 of Fig. 3.

25 The method used to create executable code 38 for a plan 14

The first time the procedure is called, it reads variables which determine the
target compiler to use, i.e., from the environment or from a configuration file. Next,
it creates file names to hold the generated source code at 30. The names consist of
the identifier of the plan, an indication of target locale of execution, and the
30 identifier of the piece of the plan being implemented. In the third step, the
procedure creates and initializes a CCodeGen structure. Next it performs (via
optimizer 34) a quick check to see if the desired code has already been recently

generated and compiled. If not, it calls the CCodeGen::genSrc function to generate the source code 30 for the subject piece of the plan 14. Then it calls a routine to compile 32 the source code, after which it generates linkage information 36.

5 The method used at 32 to compile generated source code

The first time the procedure is called, it reads variables from the environment or from a configuration file, which determine the target compilation flags to use as well as the -I include directives. Next, it logs the compilation attempt, and runs the compiler in a child process. Signals for abnormal termination of the compilation are
10 caught and handled.

Utility functions used at 30 to write generated code to the source file or buffer

The following functions are used to output various programming constructs in the high-level programming language. Most of these add appropriate indentation
15 and syntax specific to given programming constructs.

- Put (file, indentation level, format, ...) - writes an indented line to the file
- PutLine – indents formatted text, writes to buffer, adds newline
- PutComt – indents, adds “//” comment to buffer
- PutStmnt – indents, adds “;” at end, adds to buffer
- 20 • PutBlok – increments indentation level, adds “{” at end, adds to buffer
- PutCBlk – same as above, but adds a comment to the block: “{ //”
- PutQBlk – used for aligning “if” and “else” statements
- PutBlank – emit a blank line
- PutLabl – emit a label: used for a possible goto statement
- 25 • EndBlok – decrements indentation level, PutLine(“}”);
- EmitStruct – emits a “struct” declaration

While one embodiment uses C++ as the source code language, programming constructs like comments, statements, labels, blocks, structures and if-else statements are common to most procedural languages. The same utility functions
30 ideas can be used with other choices of a high-level language used for code generation.

The method used at 30, 32 to generate tuple structure declarations

The CGenCode::genStreamNodes procedure takes the first and last processes 20, 24 as input, and produces structure declarations for tuples in the associated
5 output streams. One of the performance advantages of the invention is that referencing fields via structure member references is compiled into a faster form than can be achieved by reference fields through interpreted offsets.

For example, if the subject query includes a join operation, code generator 30 represents the output stream of the join operation as local variables that reference
10 current records in each input stream. Then at compilation 32, named fields of the records are projected. Consequently intermediate materialization and auxiliary structures (such as a set of pointers) are minimized.

An example would be a simple join query that joins two tables on a character-string column and sums the integer field values of the records of one of the tables that
15 survives the join restriction – “select sum(salary) from emp, division where emp.deptname = division.deptnames”. The example code is generated code to support this query – the line that demonstrates the usage of a non-materialized field is “Aggr8->f0 += Scan0->f0”. Code lines not germane to the concept have been deleted and replaced with “...”.
20

```

DCC_PACKED struct TScan0 { // the table to be scanned
    TRecordNum rowNum;
    TXid txCreated;
    TXid txDeleted;
    uint16 size;
    uint8 nulls[1];
    uint8 pad1[1];
    int f0;
    char f1[12];
    char f2[12];
} GNUC_PACKED ;

DCC_PACKED struct TTemp5 { // the previously scanned inner table, cached in memory
    uint16 size;
    uint8 nulls[1];
    uint8 pad1[1];
    char f0[12];
} GNUC_PACKED ;

DCC_PACKED struct TAggr8 {
    uint16 size;
    uint8 nulls[1];

```



```

        uint8 pad1[1];
        CNumeric128 f0;
        } GNUC_PACKED ;

5    ...

        for ( Scan0 = (TScan0 *)bufStarts; Scan0 < (TScan0 *)bufEnds; Scan0++, curRowAddr++ ) {
            ...
            hash4 = NzHashCharBuf((uint8*)Scan0->f2,sizeof(Scan0->f2),0);
10         pTst5 = SELECT_HASH_LIST(tTemp5,hash4);
            while ( TRUE ) {
loop_5:
                // walk hash collision list, comparing all values
                for ( ; pTst5; pTst5 = pTst5->next ) {
15                 Temp5 = (TTemp5*)pTst5->recP;
                    if ( memcmp(Scan0->f2,Temp5->f0,sizeof(Scan0->f2)) == 0 ) { // got match?
                        pTst5 = pTst5->next; // update pTst5 here because we don't hit loop ender
                        goto joinIn5; // allow processing of this record-pair by subsequent steps
                    }
20             }
                // move onto next stream row
                goto endLoop0_0;
joinIn5:
                // Note that the aggregation uses the value directly from the scan record
                // rather than relying on an intermediate materialization or pointer set.
25                 Aggr8->f0 += Scan0->f0;
            }
            endLoop0_0;
30     }

```

To accomplish other minimization of auxillary structures in the preferred
 embodiment, the procedure 30 starts by allocating a TStreamInfo structure for each
 35 plan process. Then, if the process' target locale is the host processor and if the
 process is not a scan, the procedure generates a structure definition for the record as
 read off the network fabric. Next, it loops over the processes in the
 CGenCode::m_nodeArray, and switches on their type. For most process types, it
 40 emits a structure definition whose name consists of the type of the process followed
 by the number/position of the process in the m_nodeArray. For example if the first
 process was a ScanNode, the name of the structure describing the record being
 scanned would be TScan0; and if the 4th process was a SortNode, the name of the
 structure describing the output of the SortNode would be TSort3.

45 The procedure to emit a record structure definition is called
 CGenCode::emitStruct. It takes a description of the shape of the processes output,
 along with the type of the process, its position, the plan containing the process and a

Boolean value to indicate whether the process' input stream comes from the disk or from the network fabric. The procedure does two things. It writes a structure definition to the source code file (via the CGenCode buffer) and it creates a TStreamInfo structure as an internal representation. The structure definitions
5 include the relevant columns, and also some fields invisible to the query for tracking nulls (if necessary) and the size of the record (for processing variable length records).

In addition, the structures for different types of plan processes may specify additional fields. The records scanned from disk as the output of a ScanNode may
10 contain additional information such as a rowNumber and one or two transaction identifiers. The records resulting from a ReturnNode have an indicator of whether byte-swapping is required.

The processing for a RestrictNode involves creating a p-code representation of the restriction expression. Similarly, the processing for a ProjectNode also
15 involves creating a p-code representation of all project expressions. Furthermore, the shape of the input and output of the ProjectNode is analyzed and remembered.

The method used at 30 to generate source code

In the preferred embodiment, source code is generated by the CGenCode::
20 genSrc procedure 40 illustrated in Fig. 4. The CGenCode::genSrc procedure 40 takes four arguments: the first process 20 of a piece of a plan 14, the last process 24 of the piece of the plan 14, the name of a source code file and a query execution plan 14. The procedure 40 does not return a value, but it writes source code to procedurally implement the processes passed in as inputs, and it modifies the plan
25 object.

Initialization 42

In one embodiment, the first time the procedure 40 is called, it reads control variables from the environment or a configuration file and saves these in static
30 variables. These variables are a convenience, and not strictly necessary for operation of the invention.

Next, it creates a memory allocation pool, initializes the source code indentation level to a non-indented value and initializes a state variable to indicate that it is not processing the last request. Then it opens the named source code file for writing, and sets user permissions on the file to limit access as desired. In one
5 embodiment, access is limited to the user or the user's group, but not to the general public.

Emit comment 44 describing the purpose of the source code file

The first thing written to the source code file is a multi-line comment that
10 describes the current piece of the execution plan 14. The plan parameter has a function that dumps the current piece of the plan.

Emit pre-compiler directives

After the initial comment, the next step 46 is to write out precompiler
15 directives, which may include header files, define constants and macros, or do this conditionally depending on the state of certain configuration parameters.

Some of this is driven by the types of the processes in the piece of the plan. In order to more easily access the processes of the plan, the next step is to build an array of process pointers in the sequence they appear in the plan. The procedure
20 GenCode::buildNodeArray takes the first and last processes 20,24 as inputs, allocates memory for pointers to the first through the last process 20, 22, 24, initializes the array and stores it in the CGenCode m_nodeArray data field.

Next the procedure 40 loops over the processes in this new array, and emits a precompiler definition of the form "#define NEED_<NodeType> 1" for each
25 process type. For example, if the first process was of type TScanNode, then the precompiler directive would be "#define NEED_ScanNode 1"; and similarly for other types of processes. These definitions are used to pull in pieces of a subsequently included file.

Next the procedure 40 emits an include directive to pull in a file containing
30 structure definitions of process 20 types. Only those structures defined as NEEDED

above are declared at 46, in order to save compilation time. Likewise only relevant structure members are included in the generated source code.

Next the procedure 40 emits a definition of a CTable structure, and finally it emits an include directive to bring in a number of useful macro definitions. Finally,
5 the procedure switches the current buffer from the headerBuffer to the codeBuffer.

For example, in a preferred embodiment, the invention generates C++ code tailored to perform the function of a piece of the query's execution plan. This generated source code needs to access information in data structures that are also used by other portions of the generated source code. Some of these structures are
10 very large. For example, one important structure is the *CTable* object declaration, which contains a great deal of information about a user table.

Generated code accesses only a handful of the many members of the CTable object. In order to minimize compile time for generated code, the invention avoids doing an #include of the CTable declaration header in the generated code. Instead,
15 it inserts in-line into the generated code a local declaration of the CTable object defining only those CTable members that will be used by the generated code. This technique also speeds generated code compile time by avoiding pulling in additional headers that are #included by the CTable header itself.

These seemingly small reductions in the amount of code processed by the
20 compiler in the course of compiling the generated code may seem like splitting hairs - but to the user of the invention, the total query time includes the time spent compiling the generated code (assuming it is not in the cache), and that compilation time can be a significant fraction of the total query execution time.

The invention code that creates the generated code's in-line definition of the
25 CTable object is, itself, C++ code. It thus has access to the complete declaration of the CTable object and can programmatically emit the in-line CTable structure deriving it from the complete declaration of the structure.

For example, say that the generated source code needed only the 4th, and 7th members of a 10-member CTable structure. And say, for the purpose of this
30 example, the CTable structure were as simple as:
struct CTable

```

    {
        int member1;
        int member2;
        int member3;
5       int member4;
        int member5;
        int member6;
        int member7;
        int member8;
10      int member9;
        int member10;
    }

```

The inline CTable structure in the generated code would then be:

```

struct CTable
15  {
        char pad1[12];
        int member4;
        char pad2[8];
        int member7;
20      char pad3[12]
    }

```

The generated source code that produces this abbreviated CTable declaration does so by computing the size of each required padN character array. It can do so via code of this form:

```

25      CTable *pTb1 = Null;
        int padLen;
        int sizeSoFar = 0;

        printf("struct Ctable\n{\n");

30      padLen = (char *) (pTb1->member4) - (char *) pTb1;
        printf("    charpad1[%d];\n", padLen);

```

```

    printf("  int member4;");
    sizeSoFar += padLen + sizeof(pTb1 ->member4);

    padLen = (char *) (pTb1->member7) - (char *) pTb1->member4);
5    printf("  charpad2[%d];\n", padLen);
    printf("  int member7;");
    sizeSoFar += padLen + sizeof(pTb1 ->member7);

    padLen = sizeof (CTable) - sizeSoFar;
10    printf("  char pad3[%d];\n", padLen);

```

By this means, a minimal sturcture declaration is generated and compile times are reduced.

In another example, the following demonstrates two forms of generated code for the same query. The first example includes null checks for the joined

15 fields. The second example demonstrates the suppression of the null-checks. The code generation software relies on the table definition in the database schema to determine whether nulls checks are required. If either table is the intermediate result of an earlier part of the query, the code generation software relies on the result set static and/or dynamic statistics to determine whether null-checks are required.

```

20    /***** Null checks required in table join *****/

    DCC_PACKED struct TScan0 {
        TRecordNum rowNum;
25        TXid txCreated;
        TXid txDeleted;
        uint16 size;
        uint8 nulls[1];
        uint8 pad1[1];
30    int f0;
        char f1[12];
        char f2[12];
    } GNUC_PACKED ;

35    DCC_PACKED struct TTemp5 {
        uint16 size;
        uint8 nulls[1];
        uint8 pad1[1];
40    int f0;
        char f1[12];
        vlen_t f2[0];
    } GNUC_PACKED ;

```

```

...

    for ( Scan0 = (TScan0 *)bufStarts; Scan0 < (TScan0 *)bufEnds; Scan0++, curRowAddr++ ) {
5      { // Hasher:
        hash4 = *((int32*)&Scan0->f0);
      }
      // node 05 (THashJoinNode)
      pTst5 = SELECT_HASH_LIST(tTemp5,hash4);
10      while ( TRUE ) {
loop_5:
        for ( ; pTst5; pTst5 = pTst5->next ) {
          Temp5 = (TTemp5*)pTst5->recP;
          // PERFORM THE INNER LOOP JOIN COMPARE HERE
15          if ( !(Scan0->nulls[0] & 0x1) && !(Temp5->nulls[0] & 0x1) && (Scan0->f0 == Temp5->f0) ) {
            // update pTst5 here because we don't hit loop ender
            pTst5 = pTst5->next;
            goto joinIn5;
          }
20        }
        // move onto next stream row
        goto endLoop0_0;
      joinIn5:
        // logical record used in subsequent steps of the query
25        ...
      }
    }
    endLoop0_0;
30  }

  /***** Null checks not required in table join *****/

  DCC_PACKED struct TScan0 {
35      TRecordNum rowNum;
      TXid txCreated;
      TXid txDeleted;
      uint16 size;
      uint8 nulls[1];
40      uint8 pad1[1];

      int f0;
      char f1[12];
      char f2[12];
    } GNUC_PACKED ;

45  DCC_PACKED struct TTemp5 {
      uint16 size;
      uint8 nulls[1];
      uint8 pad1[1];
50  int f0;
      char f1[12];
      vlen_t f2[0];
    } GNUC_PACKED ;

55  ...

    for ( Scan0 = (TScan0 *)bufStarts; Scan0 < (TScan0 *)bufEnds; Scan0++, curRowAddr++ ) {
      { // Hasher:

```

```

        hash4 = *((int32*)&Scan0->f0);
    }
    // node 05 (THashJoinNode)
    pTst5 = SELECT_HASH_LIST(tTemp5,hash4);
5   while ( TRUE ) {
loop_5:
        for ( ; pTst5; pTst5 = pTst5->next ) {
            Temp5 = (TTemp5*)pTst5->recP;
            // PERFORM THE INNER LOOP JOIN COMPARE HERE
10         if ( Scan0->f0 == Temp5->f0 ) {
                // update pTst5 here because we don't hit loop ender
                pTst5 = pTst5->next;
                goto joinIn5;
            }
15     }
    // move onto next stream row
    goto endLoop0_0;
joinIn5:
    // logical record used in subsequents steps of the query
20     ...
    }
}
endLoop0_0;
25 }

```

Emit declarations of src/target records

30 The next 48 phase is to emit structure definitions for tuples in the output streams of each process 20, 22, 24. This is largely accomplished by the procedure called CGenCode::genStreamNodes which is described above. Before calling genStreamNodes, the CGenCode::genSrc procedure initializes the expression generator. After calling genStreamNodes to define the output record formats of each

35 process, CGenCode::genSrc emits expression parameter structure definitions if necessary.

Emit code 41

40 The next phase 41 produces the code for procedures which implement each plan process. In general, there is one procedure per process 20, 22, 24 (however ScanNode, RestrictNode and ProjectNode are frequently combined into one procedure, and the ReturnNode processing is generally included with the implementation of the penultimate process 22).

The code implementing each function is produced in four steps. The first step, called CGenCode::emitLoopHdrStuff, emits the function signature, variable declarations and initializations and forward references to statics. Then, CGenCode::emitLoopHdr emits the setup code for looping over each record in the process's input stream. The third step, CGenCode::emitLoopBodyStuff, emits the code for processing the next record in the input stream. The final step, called CGenCode::emitLoopFtr, performs post-processing and cleanup.

These steps are executed in a loop as follows:

```

    for (int nextNode=0; nextNode < m_sizeStreamInfo;
10  nextNode=followingNode)
        m_endLoopNestingLevel = 1;
        followingNode = chooseNextNode( nextNode );
        emitLoopHdrStuff( nextNode, followingNode );    // define function
        emitLoopHdr( nextNode, followingNode );        // emit "for" loop
15  emitLoopBodyStuff( nextNode, followingNode );    // emit loop body
        labelPos = m_streamInfo[nextNode]->nodePos;
        while (--m_endLoopNestingLevel >= 0) {
            PutLabl("endLoop%d_%d:", labelPos,
                    m_endLoopNestingLevel);
20  EndBlok();
        }
        PutLabl( "endLoop_%s:", m_streamInfo[nextNode]->recName;
        emitLoopFtr( nextNode, followingNode );        // close braces
    etc.
25  }    // end for each node

```

Emit the code to define functions for each plant process

In step 43, CGenCode::emitLoopHdrStuff performs the following:

- o Emits a comment indicating the start of code

- Calls a function to emit forward references to static helper functions used for sorting and aggregation. For example, a comparison function used by a sorter would be declared in advance of its definition.
- Emits forward references to continuation functions, that implement subsequent processes 22, 24 in the plan 14.
- Emits a the signature of the function, which is always of the form:


```
void GenPlan<local><piece#>_<node#>(Cplan* plan, char*
      bufStarts, char* bufEnds, bool lastCall ).
```
- Emit a statement that initializes an integer bufSize variable = bufEnds-
bufStarts
- Potentially emit other variable declarations depending on the process 20, 22, 24 type

Emit the Loop Header (step 45)

- CGenCode::emitLoopHdr has the job of producing the “for” loop that iterates over each record in the input stream. There are special cases for ScanNodes for records that have nulls and ScanNodes for records that don’t; ScanNodes for records that have variable length fields and those that don’t; ScanNodes that read records from disk; scanNodes that read records from the network fabric; and ScanNodes that read records from memory or a temporary file (for the inner loop of a join). There are also special cases for SortNodes and Aggregate Nodes, and MaterializingNodes. The basic pattern for a fixed-length disk-based scan that allows null values is the following:

```
char *recP, nextRecP;          // pointers to current & next record in stream
char* pPrev;
char* pRecordLength = bufEnds - sizeof(unsigned short );
int recordLength = 0;
if ( bufStarts ) recordLength = *(unsigned short*) pRecordLength;
else          bufEnds = bufStarts;
for ( recP = bufStart; recP < bufEnds; recP += sizeof( TScan0 ) ) {
    Scan0 = (TScan0*) recP;
    nullsScan0P = (char*) & (Scan0->nulls);
```

```
if ( plan->m_interruptRequested ) GencAbortPlan( plan );
```

The basic idea of this code fragment is to loop over each record in the input stream, casting it to the structure definition (TScan0) emitted earlier in step 48. This allows the subsequent code in the loop body to access fields of the record efficiently.

- 5 The test in the last line is for external interruptions and transaction aborts so that the system can prematurely halt execution of a plan each time through the loop.

Emit the Loop Body

- This is the routine 47 that emits the main bulk of the code that implements the intent of a plan process or a sequence of processes 20, 22, 24. The routine 47 takes start and end process 20, 24 identifiers as parameters, and loops from the start process 20 up to but not including the end process 24. Each time through the loop, the routine 47 dispatches based on the type of the next process, calling a routine (examples below) to emit the loop body for that specific type of process.

- 15 The following example illustrates the generated code for a query that joins two tables and returns a sorted result set. It includes two loops – the first to scan, join and materialize into the pre-sort table. The second to scan the pre-sort table materializing into a return set. While not the case for this example, the return set may include different fields than the preceding sort materialization. The FirstPhase routine will be called repeatedly until all scanned data is processed. The
- 20 SecondPhase routine is called only once - after the last call to the FirstPhase.

```

25 DCC_PACKED struct TScan0 {
    TRecordNum rowNum;
    TXid txCreated;
    TXid txDeleted;
    uint16 size;
    uint8 nulls[1];
    uint8 pad1[1];
30 int f0;
    char f1[12];
    vlen_t f2[0];
} GNUC_PACKED;

35 DCC_PACKED struct TTemp5 {
    uint16 size;
    uint8 nulls[1];
    uint8 pad1[1];

```

```

    int f0;
    } GNUC_PACKED ;

    DCC_PACKED struct TSort8 {
5         uint16 size;
           uint8 nulls[1];
           uint8 pad1[1];

    int f0;
    char f1[12];
10    } GNUC_PACKED ;

    DCC_PACKED struct TReturn9 {
           uint16 size;
           uint8 nulls[1];
15         uint8 pad1[1];

    int f0;
    char f1[12];
    } GNUC_PACKED ;

20    void FirstPhase(CPlan *plan, char *bufStarts, char *bufEnds, bool lastCall) {

        // Join and add records to materialization.
        // Sort when all done

25        for ( Scan0 = (TScan0 *)bufStarts; Scan0 < (TScan0 *)bufEnds; Scan0=nextScan0, curRowAddr++ ) {

            { // Hasher:
                hash4 = *((int32*)&Scan0->f0);
30            }
            // node 05 (THashJoinNode)
            pTst5 = SELECT_HASH_LIST(tTemp5, hash4);
            while ( TRUE ) {
                loop_5:
35                for ( ; pTst5; pTst5 = pTst5->next ) {
                    Temp5 = (TTemp5*)pTst5->recP;
                    if ( !(Scan0->nulls[0] & 0x1) && !(Temp5->nulls[0] & 0x1) && (Scan0->f0 == Temp5->f0) ) {
                        // update pTst5 here because we don't hit loop ender
                        pTst5 = pTst5->next;
40                        goto joinIn5;
                    }
                }
                // move onto next stream row
                goto endLoop0_0;
45    joinIn5:

                {
                    // For SortNode:
                    Sort8 = RecordPositionInLastBlock(recStore8);
50                    tSort8->m_records[tSort8->m_numRecords++] = (record_t*)Sort8;
                    // materialize record into *Sort8
                    Sort8->size = 20;
                    char *Sort8Pos = (char *)Sort8 + 20;
                    Sort8->nulls[0] = 0;
55                    uint16 lenScan0_p0;
                    if ( (Temp5->nulls[0] & 0x1) ) {
                        Sort8->nulls[0] |= 0x1;
                        Sort8->f0 = 0;

```

```

        }
        else
            Sort8->f0 = Temp5->f0;
            uint16 lenScan0_p1;
5         if ( (Scan0->>nulls[0] & 0x2) ) {
            Sort8->>nulls[0] |= 0x2;
            BytewiseMemClear(Sort8->f1,sizeof(Sort8->f1));
        }
        else
10         BytewiseMemcpy(Sort8->f1,Scan0->f1,sizeof(Scan0->f1));
            Sort8->size = ALIGN_RECORDS((uint8 *)Sort8Pos - (uint8*)Sort8 - 0);
        }
    endLoop0_1;;
}
15 endLoop0_0;;
}
endLoop_Scan0:
    // for SortNode
    if ( !lastCall )
20     return;
    node8->m_sortInfo.qsortTable();
}

void SecondPhase(CPlan *plan, char *bufStarts,char *bufEnds, bool lastCall) {
25
    // Loop thru the sorted records, add them (or a subset of the fields)
    // to the return-set materialization.

    rdex ixSort8;
30     for ( ixSort8 = 0; true; ixSort8++ ) {
        if ( ixSort8 >= tSort8->m_numRecords )
            break;
        Sort8 = (TSort8*)tSort8->m_records[ixSort8];
        {
35         // For ReturnNode:
        // The RecordPositionForReturns() call is a caching layer
        // that transmits previously materialized records as needed.
        Return9 = RecordPositionForReturns(recStore9);
        tReturn9->m_numRecords++;
40         // materialize record into *Return9
        Return9->size = 20;
        char *Return9Pos = (char *)Return9 + 20;
        Return9->>nulls[0] = 0;
        uint16 lenSort8_p0;
45         if ( (Sort8->>nulls[0] & 0x1) ) {
            Return9->>nulls[0] |= 0x1;
            Return9->f0 = 0;
        }
        else
50         Return9->f0 = Sort8->f0;
        uint16 lenSort8_p1;
        if ( (Sort8->>nulls[0] & 0x2) ) {
            Return9->>nulls[0] |= 0x2;
            BytewiseMemClear(Return9->f1,sizeof(Return9->f1));
55         }
        else
            BytewiseMemcpy(Return9->f1,Sort8->f1,sizeof(Sort8->f1));
        Return9->size = ALIGN_RECORDS((uint8 *)Return9Pos - (uint8*)Return9 - 0);
    }
}

```

```

    }
    endLoop9_0;;
    }
    endLoop_Return9:
5   }

```

In the preferred embodiment, if the main code being generated has a function call to a respective function, routine 478 considers "in-lining" the function code (i.e. copying lines of code for implementing the respective function and replacing the function call with those copied lines.) Routine 47 determines whether to in-line the respective function code based on size of affected data (i.e. records referenced in the function call). A user configurable threshold may be used.

15 Emit Loop Footer Code

At step 49, loop footer code is emitted. The loop footer code performs cleanup and loop wrap up functions. It first emits code to reset the expression pool and to update node statistics. Then it loops through the processes passed in as parameters, and dispatches to the variant of emitLoopFtr code for the specific type of process.

The CGenCode::loopFtrReturnNode checks to see if the routine 49 has been called for the last time, and if so, calls the TReturnNode's returnNewBlock routine (to return the last buffer of results).

25 Wrapup CGenCode::genSrc

After having written the source code to process the input stream for each process in the plan or piece of the plan, this procedure (step 51) performs the following wrap-up work:

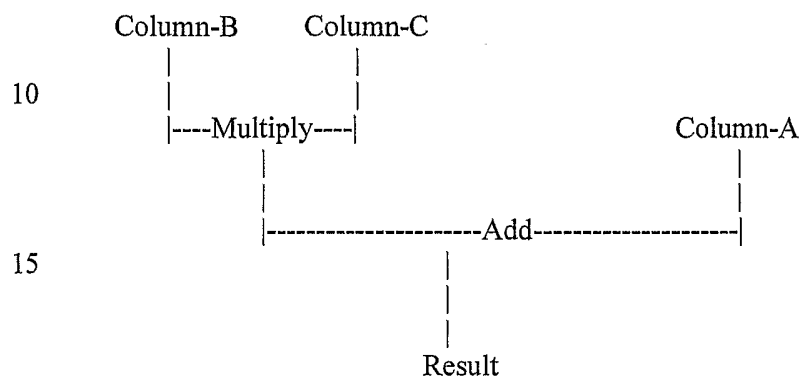
- Close the final function definition with a "}"
- 30 ○ Emit the definitions for all static helper functions (sort & aggregate comparisons)
- Write the headerBuffer to the source file
- Write the sourceBuffer to the source file
- Close the source file

- Release the memory pool

Annotating precision and scale for natively-supported arithmetic

The method used to utilize natively-supported arithmetic is now described.

- 5 Expressions are represented by a typical “expression tree.” For example, the expression $A + (B * C)$ is represented by a tree of the form:



- 20 The method for choosing narrow native arithmetic operations for expression evaluation calls for each node of an expression tree to be annotated with precision and scale information. The user of the invention has declared the three columns referenced by the example expression to have particular precisions and scales. Call these the “declared precision and scale.” In addition, the system audits the actual
- 25 values in each table, keeping track of column minimum and maximum values, among other statistics. Thus, each column has a secondary precision and scale, based on the minimum precision needed to hold the actual values existing in the table at the time of query evaluation. Call these the column’s “domain precision and scale.”

- 30 A technique for the evaluation of the result of an expression involves recursively processing each node of the tree. In the example tree above, such an evaluation might proceed as follows: Evaluate the Add node by first evaluating each of its two arguments and then adding them together. To evaluate the first argument to Add, recourse and evaluate the Multiply node. The arguments to the Multiply
- 35 node are two simple column references, so simply fetch the two column’s values and

multiply them together. This returns us to the evaluation of the Add node with its first argument value in hand. The second argument to the Add node is a simple column reference, so add the value of that column to the result of the multiplication to produce the result of the addition. This produces the result of the entire
5 expression.

To annotate the expression tree with precision and scale information, a process parallel to the above outlined evaluation of the expression's result is performed. But rather than evaluating each node's value based on its arguments, each node's precision and scale are evaluated based on the precision and scale of the
10 node's arguments. Evaluation of precisions and scales is done using natural rules. For example, the precision and scale of the result of adding arguments having precisions and scales (P1, S1) and (P2, S2) is given by:

Scale: $\max(S1, S2)$

15 Precision: $\max(S1, S2) + \max(P1-S1, P2-S2) + 1$

Note that (P1-S1) gives the number of integer places in the first argument. Thus the formulas given above formally represent the "natural" behavior that the addition result can have one more integer digit than has its argument having the larger number of integer places, and the result scale is the larger of the two argument
20 scales.

Similar natural rules can be formalized for the four arithmetic operations. For example, the precision and scale of the result of the multiplication of the example arguments are:

Scale: $S1 + S2$

25 Precision: $P1 + P2 + 1$

So, using such precision and scale evaluation rules and a standard recursive expression tree evaluation method, each node of an expression tree can be annotated with both the declared and domain precisions and scales.

Using expression tree precision and scale annotations

In the generation of C code to evaluate an expression, each time a numeric argument needs to be manipulated (as in the example expression of the preceding section), the annotated domain precisions are used (at 34, Fig. 3) to determine the size of the object actually manipulated by the C code produced. For example, consider two columns of precision and scale (15,2) that are multiplied together. While each value might be stored on disk as a 64-bit integer, domain precisions might be small enough to permit only the low-order 32 bits of each column value to be picked up and multiplied, thus giving a 64-bit result, narrower than the worst-case 128-bit result. This use of narrow native arithmetic operations provides a significant performance savings over doing worst-case evaluations in software emulation of 128-bit or even 64-bit arithmetic.

Further in the generation of the source code, optimizer 34 may include a review or determination of the range of values stored on disk (i.e., widths of data). According to the determined range of database table values, optimizer 34 adjusts width of variables in the source code being generated.

Example

The following SQL Statements create and populate a table:

```
create table ps (n numeric(20,2) not null, m numeric(20,4) not null);
insert into ps values (25.10, 130.3000);
```

And this statement retrieves a computed value from the table:

```
select (n+m)+2*(n+m) from ps;
```

The two columns, n and m, have declared precisions of 20 decimal digits and are stored by the invention as 128-bit values. So, a C++ code fragment showing the record layout would be:

```
// Input data record structure.
struct TScan 0 {
    CNumeric128 f1;    // column n
    CNumeric 128 f2;   // column m
```

```
};
```

But, by virtue of the invention's tracking of actual column values, one knows that the data in column n requires only 4 decimal digits, and column m only 7 decimal digits. So, this is a good example to show the advantage of the invention.

- 5 First, let's look at some C++ code fragments in the absence of this feature of the invention:

```

// Select result row, with wide 128-bit result column f0.
struct TReturn 3 {
10   CNumeric128 f0;    // wide expression result
};

// Expression evaluation code.
{
15   // Wide 128-bit expression result temp.
   CNumeric128 var1;

   // tmp0 holds value of common subexpression (n+m) in the example
   // and tmp0 is 128 bits wide.
20   CNumeric128 tmp0;

   // Evaluate the common subexpression (n+m).
   // Note that here we manipulate the full width of each column.
   // Scan0 points to the data's TScan 0 structure.
25   tmp) = (((100*(Scan0->f1))+((Scan0->f2))))); // f1 scaled for
alignment

   // Evaluate the expression result into the wide result temp.
   var1 = tmp0+((0x2*tmp0));
30

   // Move the result to the output record.
   // Return3 points to the return data's TReturn3 structure

```

```

        Return3 ->f0 = var1;
    }

```

As one can see, the above code manipulates 128-bit values throughout, even though the actual data values could be manipulated in narrow variables using the processor's native arithmetic operations.

Here is the same expression evaluation in C++ code generated by the invention using the precision and scale annotations as described:

```

10 //Select result row, with narrow 64-bit result column f0
   struct TReturn3{
   CNumeric64 f0;          // narrow expression result
   };
15 //Expression evaluation code.
   {
       // Narrow 64-bit expression result temp
       CNumeric64 var1;
20 //tmp0 holds value of common subexpression (n+m) in the example
       // and tmp0 can be narrow, 32 bits wide.
       int32 tmp0;
25 // Evaluate the common subexpression (n+m).
       // Note that although columns f1 and f2 are declared wide in the
       // record layout above, we know we can manipulate only the bottom
       // 32 bits of each column.
       tmp0 = (((100*(Scan0->f1.getI32()))+((Scan0->f2.getI32()))));
30 // Evaluate the expression result into the 64 bit result temp.
       var1 = tmp0+(((int64)0x2*tmp0)); // Native 32-bit arithmetic

```

```
        // move the result to the output record
        Return3->f0 = var1;
    }
```

5

In the above C++ fragments, one sees instances of objects of classes *CNumeric64* and *CNumeric128*. These are classes used by the invention to hold 64- and 128-bit integer values. They implement very efficient arithmetic operations via optimal C-language inline assembler code, taking advantage of native machine operations. In addition, they implement methods like the *getI32()* method used above to efficiently extract the low order 32 bits of a wider value.

The examples also show how variables are sized appropriate to the range of values they hold. You can see this in the declarations of *var1*, *tmp0*, and member *f0* of structure *TReturn3*, all of which are sized narrow in the example of the optimized generated source code.

Service Routines

There are a number of subroutines that are called by many of the main code emitters, to do specialized checking or to code emitting.

20

CGenCode::fieldNullAllowed

This procedure returns true if a specified field of a tuple can possibly be null, and returns false if the field cannot be null. The procedure is used in a number of contexts by the Code Generator component 16 to determine whether it is necessary to emit code to check for the presence of NULL values.

The procedure performs a number of checks. If the supplied output stream has a flag indicating that its tuples can have null fields, the procedure returns true. If the field is a computed expression, a check is made to see if the result cannot be null. Finally, the description of the specified field is checked to see if null values have been declaratively precluded.

30

In one embodiment, statistics are kept on the actual existence of null values for a field of a given table. Even if the field declaratively allows null values, if the statistics show that there are no null values currently, the fieldNullAllowed routine could return false.

5

CCodeGen::emitNullSetStr

This procedure emits the source code necessary to set a field to a null value. The code it generates must be tied to the underlying DBMSs representation of null values. In one embodiment, a DBMS may represent null values as a bit vector which is an extra field of a record called "nulls"; and where the value of the n^{th} bit indicates whether the n^{th} field of the record is null. In such an embodiment, the code emitted by this procedure to set the 5^{th} field to null would have the form:

recordPtrVariableName->nulls[0] |= (0x1 << 5);
which would set the 5^{th} bit of the null vector of the record to 1. Different
15 representations of null values are possible and this procedure would then generate different code.

In a preferred embodiment, code generator 16 may use "nulls" to overwrite field references in an outer join operation. Null value indicators and the like may be used here.

20 The following code fragment demonstrates the use of 'useNulls' Flag. The code shown below materializes the results of a left outer join into the return set to be returned to the client.

Note the use of the variable 'useNullsTemp5' below. useNullsTemp5 is false when materializing a matching row. useNullsTemp5 is set to true when
25 materializing a non-matched row from the left table. useNullsTemp5 is needed because during a materialization phase of a left outer join the pointer to record from the right is NULL. With the use of the overriding useNullsTemp5 flag one is able to use the same code structure for inner and outer joins. Note that for inner joins the 'if' statement doesn't even test for useNulls.

30

```
// node 08 (TReturnNode)
{
```

```

// For ReturnNode:
TMemblockInfo *block8 = recStore8 ->m_lastBlock;
VERIFY_FSPACE (Return8, TReturn8, tReturn8, block8, 2008,
node8,
5   returnNewBlock);
    block8->m_numRecords ++;
    tReturn8 ->m_numRecords ++;
    // materialize record into *Return8
    Return8 ->size = 8;
10   Char *Return8Pos = (char *) Return8 + 8;
    Return8 ->nulls [0] = 0;
    uint16 lenScan0_p0;
    if ( (useNullsTemp5 || (Temp5->nulls [0] & 0x1)) ) {
        Return8 ->nulls [0] |= 0x1;
15   Return8 ->f0 = 0;
    }
    else
        SWAP_INT32 (&Return8->f0, Temp5->f0);
    Return8->size = ALIGN_RECORDS ((UInt8 *)Return8Pos -
20   (*uint8*)Return8 - 0);
    //lock record in
    UPDATE_WRITE_ADDR(block8, Return8 ->size);
    SWAP_UINT16(&Return8->size, Return8 ->size);
}

```

25

CGenCode::getNullTestStr

This procedure emits the source code necessary to test whether a field is null. The code generator calls it only if a field could possibly be null. If a field is declared “not null” in a SQL column definition, or if the DBMS can determine that

30 there are no null values for the field, then the code generator does not emit the code to (needlessly) perform the test.

The code that is generated to test whether a field is null must be tied to the underlying DBMSs representation of null values. In one embodiment, a DBMS may represent null values as a bit vector which is an extra field of a record called "nulls"; and where the value of the n^{th} bit indicates whether the n^{th} field of the record is null.

- 5 In such an embodiment, the code emitted by this procedure to test the 5th field would have the form:

```
if ((recordPtrVariableName->nulls[0] & (0x1 << 5) != 0)
```

which would determine whether the 5th bit of the null vector of the record was 1.

Different representations of null values are possible and this procedure would then

- 10 generate different code.

CCodeGen::loadFields

This procedure emits a declaration of an array of pointers to field values, and initializes their actual values. It is used by the RestrictNode to specify the field

- 15 values used in a restriction qualification.

The procedure takes a TStreamInfo parameter and a String to be the name of the array of fields. First, it finds the number of fields in the stream, and emits a declaration of the array as follows:

```
void* ArrayName[numFields];
```

- 20 Next, the procedure loops over the fields in the stream, and retrieves the value of the next field of the current record in the stream. Then it calls the CCodeGen::fieldNullAllowed function described above to determine whether the field can possibly hold null values. If not, the procedure emits an assignment of the next field value as follows:

- 25 `ArrayName[nextFieldIndex] = nextFieldValue;`

If the field could possibly be null, this assignment is modified to preserve null values:

```
ArrayName[nextFieldIndex] = (<field-dependent test for null> )  
? NULL : nextFieldValue;
```

- 30

One embodiment of this procedure is shown below.

```

void CGenCode::loadFields (TStreamInfo *flowInfo, char *arrName /*="fields"*/)
{
    int numFields = flowInfo->numFields;
5   char fieldAt[100], nullStr[100];

    PutStmnt("void *%s[%d]",arrName,numFields);
    for ( int ix = 0; ix < numFields; ix++ ) {
        getFieldAt(fieldAt,flowInfo,ix);
10   if ( fieldNullAllowed(flowInfo,ix) ) {
            getNullTestStr(nullStr,flowInfo,ix);
            PutStmnt("%s[%d] = (%s ? NULL : %s)",arrName,ix,nullStr,fieldAt);
        }
        else
15   PutStmnt("%s[%d] = %s",arrName,ix,fieldAt);
    }
}

```

CGenCode::getFieldStr

20 This procedure generates names of variables and structure field references for use in generated code. It takes three parameters: a pointer to a string that will contain the resulting name (called fieldStr), a pointer to a TStreamInfo structure (called sP), and an integer index (called dex) of a field in the sP stream.

25 The procedure first accesses the TStreamField description associated with the dexth field in the sP stream. It then retrieves the name of the variable used to refer to the current record in the stream. If this name is null, then the field referenced by 'dex' is a computed column, and the procedure returns a value of the form "var#" into 'fieldStr', where the number ('#') is specified in the TStreamField (its 'varNum' member).

30 If the requested field does have a name, the procedure then checks whether the field has a fixed size or if not, whether it is the first variable size field. In either

of these cases, the field is at a known offset, and the procedure returns a string of the form:

recordVariableName->fieldName

into the 'fieldStr' parameter, where the 'recordVariableName' comes from the

- 5 TStreamField's 'recName' member, and the 'fieldName' is of the form "f#" (meaning the nth field).

Finally, if the field is not at a known offset, the procedure returns a string of the form:

- 10 GetVarFieldPtr((record_t*) recordVariableName, offset,
 firstVariableOffset)

into its 'fieldStr' parameter. The function call above, when compiled and executed, will locate a pointer to the field. In one embodiment, it operates by chaining down each of the variable length fields of the record indicated by recordVariableName.

One embodiment of this procedure follows:

- ```

15 void CGenCode::getFieldStr(
 char *fieldStr,
 TStreamInfo *sP,
 int dex)
{
20 TStreamField *fP = &sP->flds[dex];
 char *nm = fP->recName;
 int srcCol = fP->srcDex;

 if (!nm) { // computed column?
25 sprintf(fieldStr,"var%d",fP->varNum);
 return;
 }
 int offset = fP->table->m_fields[srcCol].offset;

30 // if a fixed length field or the first variable length, just point to it

```

```

 if (fP->isFixedSize || (offset == 0))
 sprintf(fieldStr,"%s->f%d",nm,fP->srcDex);

 // no, is a var fields (and not the first of them)
5 else
 sprintf(fieldStr,"GetVarFieldPtr((record_t*)%s,%d,%d)",
 nm,offset,fP->stream->firstVarOffset);
}

```

#### 10 CGenCode::emitCopyRecord

This procedure generates code that copies a whole structure from a source location to a destination location. If the structure is fixed size, the generated code uses the compiler support for the copy. If the structure is variable size, it generates code to copy an integer at a time.

15 The procedure takes four parameters: a destination memory pointer, a source location memory pointer, the name of a type of structure (called typeName), and a CTable pointer (called table) that describes the records being copied. The first step is to test the CTable pointer to see if the records in the table have variable length fields. If there are none, then a line of code is emitted to perform a structure-level

20 copy as follows:

```
“*((typeName) destination) = *((typeName) source)”
```

where the “(typeName)” casts the destination and source pointers to their proper structure name, so that the compiler can generate the most efficient machine code to perform the copy. If there are variable length fields, the procedure emits a function

25 call that performs an integer-by-integer copy:

```
“IntwiseMemcpy(destination, source, *((int*) source))”
```

where the first word of the source pointer is presumed to contain the length (in integers) of the structure to be copied. One embodiment of the procedure follows:

```

void CGenCode::emitCopyRecord (char *dstName, char *srcName, char
*typeName, CTable *table)
{
 if (!table->m_numVaryingFields)
5 PutStmt("*((%s*)%s) =
(((%s)(%s))", typeName, dstName, typeName, srcName);
 else

PutStmt("IntwiseMemcpy(%s,%s,*((uint16*)(%s)))", dstName, srcName, srcName);
10 }

```

#### CGenCode::emitCopyField

15 This procedure generates code that copies a field from a source record into a corresponding field in a destination record. If the field is variable length, it also emits a declaration of a character pointer to track where the next varchar, if any, should go. The procedure also emits code to handle copying null values and to perform byte swapping if necessary.

20 In the first step, the procedure calls the CGenCode::getFieldStr utility function to get back the strings used to reference the source and destination fields in the generated code. These strings will either be variable names (for computed fields), or of the form "structure->fieldMember" (for references to the columns of a record).

25 In the next step, the procedure checks whether the fields are fixed length. If so, it next checks whether null values are allowed in either the source field or the destination field. If nulls are not allowed, the procedure then checks whether byte swapping is necessary. If byte swapping is not necessary, then the procedure emits a simple assignment:

```

destinationRecordPtr->destinationField = sourceRecordPtr->sourceField
30 where the strings "destinationRecordPtr->destinationField" and
"sourceRecordPtr->sourceField" were produced in the first step. Otherwise, byte
swapping is necessary, and the procedure calls the procedure

```

CGenCode::emitSwapAndAssign, passing in the TStreamField pointers that describe the source and destination fields, as well as the strings used to reference those fields, as produced by the first step. The emitSwapAndAssign function emits code that (1) swaps bytes of the source field value if necessary, depending on its  
 5 datatype; and then (2) assigns the swapped value to the destination field.

If nulls are allowed (CGenField::fieldNullAllowed), the procedure emits code that (1) checks whether the source field is null (CGenField::getNullTestStr), and if it is, then (2) to set the destination field to be null (CGenField::emitNullSetStr); else if it (the source field) isn't null, then to emit the  
 10 same code described above for depending on whether byte-swapping necessary or not.

In one embodiment, this processing of fixed length field copying is effected as:

```

 if (fieldNullAllowed(srcP,srcCol) && dstTable->m_nullsAllowed) {
15 PutBlok("if (%s)",getNullTestStr(nullStr,srcP,srcCol));
 emitNullSetStr(dstP,dstCol);
 PutStmt("%s = 0",dstStr);
 EndBlok();
 PutQblk("else");
20 //BYTE-SWAP
 if (dstP->m_byteSwap)
 emitSwapAndAssign(dstFldP, dstStr, srcFldP, srcStr);
 else
 PutStmt("%s = %s",dstStr,srcStr);
25 } else {
 //BYTE-SWAP
 if (dstP->m_byteSwap)
 emitSwapAndAssign(dstFldP, dstStr, srcFldP, srcStr);
 else
30 PutStmt("%s = %s",dstStr,srcStr);
 }

```

### GenCode::materializeRecord

This procedure is used to construct a record in an output stream from fields in an input stream. In one embodiment it takes three parameters: the two streams, and a callback function to be used to extend the amount of space (technically the  
5 recordStore) of the output stream.

The first step is to emit a comment announcing that the following code is materializing a particular type of record:

```
PutComt("materialize record into *%s", outputStream->recName);
```

Then, a test is performed to determine if the data in the input stream has the  
10 same "shape" as the records in the output stream. Records have the same shape if they have the same number of fields of the same type in the same order. If so, the procedure attempts to emit code to do an efficient structure-level copy from input to output, by calling the emitCopyRecord service routine, and passing in the name of the variable used to indicate the location of the end of the output stream, the name of  
15 the variable used to indicate the location of the current record in the input stream, the name of the type of the record in the input stream, and a pointer to the CTable describing the input stream records. In this case of same shape records, this materializeRecord procedure is now done, and returns 0.

Next, if nulls are allowed in the destination table in the output stream, then  
20 code is emitted to clear all null values in the destination record. In an embodiment where a DBMS represents nulls as bits in a vector, this generated code would set all the bits in this vector to 0.

Next, the procedure emits code to perform a field-by-field copy from the current record of the input stream to the current location in the output stream, calling  
25 CGenCode::emitCopyField to accomplish this.

CGenCode::EmitVerifySpace

This procedure's main purpose is to emit a MACRO call to ensure that there is sufficient space to fit a record in a recordStore. If there isn't enough space, the MACRO invokes a callback function to extend the recordStore. This callback  
30 extension function may have interesting and useful side effects, such as executing a phase of a sort, or returning a partial set of results back to a client.

The format of the output is:

TMemblockInfo \*block = last block of the recordStore of the table of the out stream

```

 VERIFY_FSPACE(RecordPtr, TypeOfRecord, table, block, size, node,
5 callback)
 tTableDescriptor->m_numRecords++;

```

The first line above locates the last block of the record store associated with the output stream. The next line invokes the verification macro. If there isn't enough room for the RecordPtr within the last block the supplied callback function  
10 is invoked to create more room. The last line increments the number of records written to the target table.

The procedure takes a destination stream, and a callback extender function. In one embodiment it also takes a variety of other boolean and integer parameters to handle special cases which are not of interest here.

15 One embodiment of this procedure is as below:

```

void CGenCode::emitVerifySpace (
 TStreamInfo *sP, // destination stream info
 char *extendFunc, // run this when extending recordstore
)
20 {
 char *nm = sP->recName;
 int nid = sP->nodePos;;
 CTable *tbl = sP->table;
 PutStmt("TMemblockInfo *block = recStore%d->m_lastBlock",nid);
25
 int maxSize = tbl->m_maxRecordSize;

 PutStmt("VERIFY_FSPACE(%s,T%s,t%s,block,%d,node%d,%s)",
 nm,nm,nm,maxSize,nid,extendFunc);
30 PutStmt("t%s->m_numRecords++",nm);
}

```

## Restrict

The RestrictNode follows a ScanNode and filters out records from the stream that don't meet a boolean criterion. This is accomplished by generating a test within  
 5 the loop body that looks like:

```
if (!ExecRestrict(node1->qual, plan, tScan1, fields))
 continue;
```

The meaning of this code is that if the field values specified in 'fields'  
 10 parameter, do not satisfy the qualification specified by the first parameter, then filter out the record and continue processing the next record in the input stream.

The code above is emitted by the CGenCode::loopBodyRestrictNode method, which takes an input stream and an output stream as parameters.

In one embodiment, the procedure emits a comment announcing that the  
 15 following code is "// For RestrictNode", and then emits code that starts a block ("{" in C). Next, a test is performed to see if there are any special considerations in generating the code for the restriction. If not, the procedure emits a declaration of an array of field values and emits initializations of field values for every field in the current record in the input stream (conventionally called 'flowInfo'). Next, the  
 20 procedure emits the code for the "if" and "continue" statements illustrated above, and finally closes the block ("}" in C). An example implementation follows:

```
loopBodyRestrictNode(TStreamInfo* sP, TStreamInfo* flowInfo)
 char fieldStr[100];
 int nid = sP->nodePos;
 25
 TRestrictNode *pNode = (TRestrictNode*)sP->pNode;

 PutComt("For RestrictNode:");

 30 PutBlok("");
 if (usePGexpressions ||
```

```

 !pNode->m_plan->createCcodeForNode((TRestrictNode*)pNode, sP,
flowInfo, 0, this))
 {
 int nid = sP->nodePos;
5 // can't generate c code for filter - pump out call to ExecRestrict
 loadFields(flowInfo);
 PutQblk("if (!ExecRestrict(node%d->qual, plan, t%s, fields))", nid, flowInfo-
>recName);
 PutStmt("continue");
10 }

 EndBlok();

```

Return

15       The ReturnNode materializes the information in its input stream and produces records of a requested shape on its output stream. It is used to return both final and intermediate results from one processing locale to another. In a distributed or massively parallel DBMS, where a query plan is divided up into pieces that are distributed to multiple processors, the ReturnNode is called at the end of a

20       distributed piece of work to send partial results back to a requestor.

      The main work performed by the code that implements the ReturnNode is to copy field values from various locations in its input stream into records in its output stream. In one embodiment, its output stream consists of a memory buffer that is iteratively filled with records. When the buffer becomes full, its contents are

25       transmitted back to the processing locale that issued the requested execution plan, possibly across a network. When the computer chip architecture of the requesting locale differs from the chip architecture used to process the ReturnNode, the ReturnNode code performs byte swapping, as part of copying values from its input stream to its output stream.

30       The main code generation occurs in a procedure called

      CGenCode::loopBodyRecordSetNode



which takes the name of a node type ("Return" in this case), a pointer to an output stream (TStreamInfo\* sP), a pointer to an input stream (TStreamInfo\* flowInfo), and the name of a callback function ("returnNewBlock" in this case) which is called to extent the outputStream if necessary. The procedure performs the following code

5 generation:

```

 PutBlok("");
 PutComt("For %sNode:",type);
 emitVerifySpace(sP,"returnNewBlock");
10 materializeRecord(sP,flowInfo, extendFunc);
 EndBlok();

```

This emits a block scope and a comment. Then it calls a procedure (emitVerifySpace, described above) that generates code to check that there is enough space in the outputStream to fit another record; and if not, to call the

15 function named "returnNewBlock". This returnNewBlock function operates by flushing the last block of the output stream's record buffer back to the caller, and then allocating a new last block to replace it. Finally, the procedure generates a call to the CCodeGen::materializeRecord function to copy fields from the source input stream (flowInfo) to the output stream (sP); and closes the block scope.

20 While the "returnNewBlock" function does most of the work of transmitting the results of the ReturnNode back to the caller, it may leave a partially filled last block in the output stream. To finish up and send this last partial block back to the caller, there is a CGenCode::loopFtrReturnNode which performs the following:

```

 PutStmt("if (lastCall) node%d->returnNewBlock(lastCall)",sP->nodePos);
25

```

This emits a source code statement that checks to see if this is the last call to the ReturnNode routine, and if so, to call the "returnNewBlock" procedure one last time to flush any remaining records in the last buffer of the output stream back to the caller.

## AggregateNode

Aggregate functions, like the SQL “min”, “max” and “sum” functions operate on an input stream and accumulate a resulting value. The arithmetic operators used to accumulate this resulting value depend on the datatype of the field in the input stream. Simply put, floating point addition and comparisons are different than integer addition and comparisons. In a DBMS whose runtime execution engine is interpretive, there must be runtime checks to determine which kind of data type and which kinds of arithmetic operator to use. In contrast, the invention determines the datatype of the fields being aggregated at code generation time 16, and generates precisely the right kind of variable declarations and arithmetic operators to perform the calculations. In essence the time required to check the data type of the field is moved from “for every record during runtime” to “once at code generation time”. The header, body and footer methods below are one embodiment of this technique.

```

15 loopHdrAggNode(..)
 TAggregateNode *pNode = (TAggregateNode *)sP->pNode;
 char *nm = sP->recName;
 int nid = sP->nodePos;
 char recSize[100];
20 pNode->m_gencHelperStream = cloneWithNewRecName(sP, "curRec");

 DEBUGCMT(" ***** loopAggregateNode ***** ");

 //////////////////////////////////////
25 // // For AggregateNode:
 // TAggregateNode *node4 = (TAggregateNode*)plan->m_nodeArray[4];
 // CTable *tAggr4 = node4->m_result;
 // TAggr4 *Aggr4 = BADPTR(TAggr4*);
 //////////////////////////////////////
30 PutComt("For AggregateNode:");
 PutStmt("CTable *t%s = node%d->m_result",nm,nid);

```

```

// Set Aggr* to 1 in hopes of trapping if used before set. (Was getting compiler
warning.)
PutStmt("T%s *%s = BADPTR(T%s*)",nm,nm,nm);

5 if (pNode->m_numGroups != 0) {
 //
 // // Aggr grouping hash table info
 // THashInfo *pTst4;
 // THashInfo *groupHashRec4;
10 // TMemblockInfo *groupHashBlock4 = tAggr4->m_hashStore->m_lastBlock;
 // if (tAggr4->m_hashStore)
 // groupHashBlock4 = tAggr4->m_hashStore->m_lastBlock;
 // CRecordStore *recStore4 = tAggr4->m_recStore;
 //
15 // grouping, emit hash table handling data
 PutComt("Aggr grouping hash table info");
 PutStmt("THashInfo *pTst%d",nid);
 if (pNode->HasGrpRecId())
 PutStmt("TGroupHashInfo *groupHashRec%d",nid);
20 else
 PutStmt("THashInfo *groupHashRec%d",nid);
 PutStmt("TMemblockInfo *block%d = t%s->m_recStore-
>m_lastBlock",nid,nm);
 PutStmt("TMemblockInfo *groupHashBlock%d",nid);
25 PutQblk("if (t%s->m_hashStore)",nm);
 PutStmt("groupHashBlock%d = t%s->m_hashStore->m_lastBlock",nid,nm);
 PutStmt("CRecordStore *recStore%d = t%s->m_recStore",nid,nm);
 char funcStr[300];
 PutBlok("if (!node%d->m_bInitialized)", nid);
30 PutStmt("node%d->InitializeSorters()", nid);
 PutStmt("node%d->m_sortInfo.setCmpRecFnForSort(%s,t%s)",
 nid, getCompareFuncStr(funcStr, nid), nm);

```

```

 if (pNode->m_flags & NfReclaim)
 initAggrVals(sP, flowInfo);
 EndBlok();
 char *helperNm = pNode->m_gencHelperStream->recName;
5 PutStmt("T%s *%s = (T%s*)node%d->m_curRec", nm, helperNm, nm, nid);
 // following "tcurRecAggrN" def needed for potential assignment to dest
 curRec below
 PutStmt("CTable *t%s = t%s", helperNm, nm);
 }
10 else { // single record, find it here
 ///
 // // Aggr record created during node init
 // Aggr3 = (TAggr3*)tAggr3->m_recStore->m_lastBlock->bytes;
 ///
15 PutComt("Aggr record created during node init");
 PutStmt("%s = (T%s*)t%s->m_recStore->m_lastBlock->bytes", nm, nm, nm);
 PutBlok("if (!node%d->m_bInitialized)", nid);
 PutStmt("node%d->InitializeSorters()", nid);
 if (pNode->m_flags & NfReclaim)
20 initAggrVals(sP, flowInfo);
 EndBlok();
 }

 TAggrField *aggrP;
25 TStreamInfo *aggrsP;
 int fdex;

30 for (int aggrDex = 0; aggrDex < pNode->m_numAggrs; aggrDex++) {
 aggrP = &pNode->m_aggrs[aggrDex];
 if (aggrP->limitDistinct) {

```

```

 PutStmt("int hitLimit%d_%d = -1",nid,aggrDex);
 PutStmt("int recNum%d_%d = 0",nid,aggrDex);
 }
 if (aggrP->distinct) {
5 aggrsP = sP->aggrs[aggrDex];

 PutStmt("TAggrField *aggr%d_%d = &node%d->m_aggrs[%d]",
 nid, aggrDex, nid, aggrDex);
 PutStmt("CTable *tDistinctAggr%d_%d = aggr%d_%d->table",
10 nid, aggrDex, nid, aggrDex);
 PutStmt("THashInfo* pTst%d_%d", nid, aggrDex);
 PutStmt("T%s *%s", aggrsP->recName, aggrsP->recName);
 PutStmt("uint32 distinctHash%d_%d",nid, aggrDex);

15 // !FIX-jpb don't need this at each aggr loop hdr. just first?
 PutBlok("if (!node%d->m_aggrs[%d].m_sortInfo.m_cmpRecFn)", nid,
aggrDex);
 PutStmt("node%d-
>m_aggrs[%d].m_sortInfo.setCmpRecFnForSort(GenPlan_RecCmp%d_%d_w,t%s)
20 ", nid, aggrDex, nid, aggrDex, nm);
 PutStmt("node%d-
>m_aggrs[%d].setCmpRecFnForNarrowCmp(GenPlan_RecCmp%d_%d_n,t%s)",
nid, aggrDex, nid, aggrDex, nm);
 PutStmt("node%d-
25 >m_aggrs[%d].setWidenRecFn(GenPlan_DistincterWidener%d_%d)", nid,
aggrDex, nid, aggrDex);
 EndBlok();
 }

30 }

 // Emit the variables & setup used for express stats generation

```

```

 if (pNode->m_forExpressStats)
 loopHdrExpressStats(sP,flowInfo);

loopBodyAggNode(..)
5 TAggregateNode *pNode = (TAggregateNode *)sP->pNode;
 int nid = sP->nodePos;

 DEBUGCMT(" ***** loopBodyAggregateNode *****");

10 if ((flowInfo->flowType != FlowRecord) && (flowInfo->flowType !=
 FlowProjected))
 THROW_ERROR(NZ_ERROR_IMPL,NULL,flowInfo->flowType);

 PutComt("For AggregateNode");

15 // Emit the per-loop setup used for express stats generation
 if (pNode->m_forExpressStats)
 loopBodyExpressStats(sP,flowInfo);

20 if (pNode->m_numGroups == 0) // no grouping?
 loadAggrVals(sP,flowInfo, false);
 else if (isMergedAggregation(pNode,nid))
 emitAggregateMergeEngine(sP,flowInfo);
 else
25 emitAggregateHashedEngine(sP,flowInfo);

loopFtAggNode
 TAggregateNode *pNode = (TAggregateNode *)sP->pNode;
 int nid = sP->nodePos;
 char *nm = sP->recName;

30 if (isMergedAggregation((TAggregateNode *)pNode,nid))

```

```

 return; // no loop footer for merging!

 //////////////////////////////////////
 // //for aggregate node
5 // if (!lastCall)
 // return;
 // LogMaterializeNode(plan,node3,lastCall);
 //
 // node3->lastAggregateNewBlock();
10 //////////////////////////////////////

 PutComt("For AggregateNode");

 // For ZONE MAPS
15 if (NZCFG->system->getInt("zoneMapTableSizeThreshold") > 0) {
 TProjectNode *prev = (TProjectNode *)pNode->m_prev;
 if (pNode->m_plan->i_forGenStats() && !m_forHost &&
 !isFabricScan(getScanNode())) {
 int fdex, colDex;
20 bool lockIn, lastColumn;

 PutComt("Add the block-based min-max values to the zone map");

 // Since we do not get the block header in the gen'd code, we keep a
25 // counter of our own. This works only because the vacuumAnalyze scans
 // the whole table from block0, and the generated code is called for
 // every block read (even empties). Very sleazy!
 // !FIX-bk Better way to get block number
 PutQblk("if (!lastCall)");
30 PutStmnt("node%d->m_numBlocks++",nid);

 int numFields;

```

```

TScanNode* pScanNode = getScanNode();
if (Nz->realFpga && !isRawScan(pScanNode))
 numFields = flowInfo->table->i_numFields();
else
5 numFields = prev->numFields;
 for (fdex = 0; fdex < numFields; fdex++) {
 // If the FPGA is not involved, then the flow is a full width table scan.
 // Otherwise, find the true base table column id from the projection list.
 colDex = prev->fieldsList[fdex];
10 lockIn = isColumnForZoneMap(flowInfo,fdex,pScanNode-
 >tblId,colDex,&lastColumn);
 if (lockIn) {
 if (lastColumn)

15 PutStmt("ExecLockInBlockRange(node%d,%d,%d,blkMax%d,blkMin%d,TRUE,las
 tCall)",
 nid,pScanNode->tblId,colDex,fdex,fdex);
 else

20 PutStmt("ExecLockInBlockRange(node%d,%d,%d,blkMax%d,blkMin%d,FALSE,l
 astCall)",
 nid,pScanNode->tblId,colDex,fdex,fdex);
 }
 }
25 }
}

// Emit the variable-saves & count extractor for express stats generation
30 if (pNode->m_forExpressStats)
 loopFtrExpressStats(sP,flowInfo);

```



```

PutQblk("if (!lastCall)");
PutStmnt("return");
PutStmnt("LogMaterializeNode(plan,node%d,lastCall)",nid);

5 if (pNode->HasDistinctAggrs()) {
 int ix;
 TAggrField *aggrP;
 PutComt("do distincters first and then grouping aggregate");
 for (ix = 0; ix < pNode->m_numAggrs; ix++) {
10 aggrP = &pNode->m_aggrs[ix];

 if (aggrP->distinct) {
 TStreamInfo *aggrsP = sP->aggrs[ix];
 TStreamInfo *aggrWidesP = sP->aggrsWide[ix];
15 TStreamInfo *aggrFinalsP = sP->aggrsFinal[ix];
 TStreamInfo *helperVar = aggrP->m_gencHelperStream;
 char *helperNm = helperVar->recName;
 char buf[200];
 char *nm = aggrFinalsP->recName;
20 bool bWiden = pNode->m_numGroups != 0;

 PutBlok("");
 PutComt("final distincting for aggr%d_%d",nid,ix);
 PutStmnt("TAggrField *aggr%d_%d = &node%-
25 >m_aggrs[%d]",nid,ix,nid,ix);
 PutStmnt("T%s *%s", aggrFinalsP->recName, aggrFinalsP->recName);
 PutStmnt("T%s *%s", aggrWidesP->recName, aggrWidesP->recName);
 PutStmnt("T%s *%s", aggrsP->recName, aggrsP->recName);
 PutStmnt("CTable *tFinalDistinctAggr%d_%d = aggr%d_%d-
30 >finalTable",nid,ix,nid,ix);
 PutStmnt("CTable *tWideDistinctAggr%d_%d = aggr%d_%d-
>wideTable",nid,ix,nid,ix);

```

```

 PutStmt("CTable *tDistinctAggr%d_%d = aggr%d_%d-
>table",nid,ix,nid,ix);
 PutStmt("T%s *%s = NULL",aggrWidesP->recName,helperNm);
 PutStmt("rdex ix%d_%d",nid,ix);
5 PutQblk("if (aggr%d_%d->m_sortInfo.m_diskBased)",nid,ix);
 PutStmt("aggr%d_%d->distincterNewBlock(true)",nid,ix);

 PutBlok("for (ix%d_%d = 0; true; ix%d_%d++)",nid,ix,nid,ix);
 PutStmt("TMemblockInfo *block = t%s->m_recStore-
10 >m_lastBlock",nm);
 PutBlok("if (aggr%d_%d->m_sortInfo.m_diskBased)",nid,ix);
 if (pNode->m_detailStats)
 PutStmt("aggr%d_%d->m_sortInfo.m_sortTime.Start()",nid,ix);
 PutStmt("T%s = (T%s*)aggr%d_%d-
15 >m_sortInfo.mergeStep((record_t**)&%s)",
 aggrWidesP->recName,aggrWidesP->recName,nid,ix,helperNm);
 if (collectNodeStats(pNode))
 PutStmt("aggr%d_%d->m_sortInfo.m_sortTime.Finish()",nid,ix);
 PutQblk("if (!%s)",aggrWidesP->recName);
20 PutStmt("break");
 // emit the distincter compare
 PutQblk("if (!%s)",helperNm);
 PutStmt("goto NewUnique%d_%d",nid,ix);
 for (int jx=0; jx <= pNode->m_numGroups; jx++) {
25 char compareStr[1000];
 getCompareStr(compareStr,helperVar,jx,aggrWidesP,jx,true);
 PutQblk("if (!(%s))",compareStr);
 PutStmt("goto NewUnique%d_%d",nid,ix);
 }
30 // Use record N in next iteration (N+1) "same?" test
 // mergeStep() is much happier that way.
 PutStmt("T%s = %s",helperNm,aggrWidesP->recName);

```

```

 PutStmt("continue");
 PutLabl("NewUnique%d_%d:",nid,ix);
 PutComt("new unique value");
 PutStmt("%s = %s",helperNm,aggrWidesP->recName);
5 if (bWiden) {
 getRecSizeBuf(buf,aggrWidesP);
 PutStmt("VERIFY_SPACE(%s,T%s,t%s,block,%s)",nm,nm,nm,buf);
 // !FIX-jpb need to set m_record[n] ??
 sprintf(buf,"T%s",aggrWidesP->recName);
10 emitCopyRecord(nm,helperNm,buf,aggrWidesP->table);
 }
 EndBlok();
 PutBlok("else");
 PutQblk("if (ix%d_%d >= t%s->m_numRecords)",nid,ix,aggrsP-
15 >recName);
 PutStmt("break");
 if (!bWiden) {
 PutStmt("%s = (T%s*)t%s-
>m_records[ix%d_%d]",helperNm,aggrWidesP->recName,
20 aggrsP->recName,nid,ix);
 } else {
 PutStmt("VERIFY_SPACE(%s,T%s,t%s,block,t%s-
>m_maxRecordSize)",nm,nm,nm,
 aggrFinalsP->recName);
25 getWidenerFuncStr(buf, nid, ix);
 PutStmt("%s(t%s->m_records[ix%d_%d], (record_t *)%s, t%s)",
 buf,aggrsP->recName,nid,ix,aggrFinalsP->recName, sP-
>recName);
 }
30 EndBlok();
 if (!bWiden) {
 /* hack - we have no Txxx struct for the helper stream

```

```

so, if fixed size rec, we use wide table struct for sizeof
otherwise, use helper as source of instanct size
*/
if (helperVar->table->m_sizeWord)
5 getRecSizeBuf(buf,helperVar);
else
 getRecSizeBuf(buf,aggrWidesP);

 PutStmt("VERIFY_SPACE(%s,T%s,t%s,block,%s)",nm,nm,nm,buf);
10 // !FIX-jpb need to set m_record[n] ??
 sprintf(buf,"T%s",aggrWidesP->recName);
 emitCopyRecord(nm,helperNm,buf,aggrWidesP->table);
 }
 PutStmt("t%s->m_numRecords++",nm);
15 PutComt("lock record in");
 getRecSizeBuf(buf,aggrFinalsP);
 PutStmt("UPDATE_WRITE_ADDR(block,%s)",buf);
 EndBlok();
 EndBlok();
20 }
 }
 }

 if (pNode->m_numGroups != 0) { // grouping?
25 PutStmt("node%d->lastAggregateNewBlock()",nid);
 }

 emitCallChain(sP);

```

30

While this invention has been particularly shown and described with references to preferred embodiments thereof, it will be understood by those skilled

in the art that various changes in form and details may be made therein without departing from the scope of the invention encompassed by the appended claims.

For example the DBMS 10 illustrated in Fig. 1 is typically provided at a host server in a client-server architecture. The server may be formed of one or multiple  
5 digital processors. Data processing may be in parallel processes or use distributed or other processing techniques. Data storage may be redundant or otherwise.

One example of suitable computer system is disclosed in U.S. Patent Application No. 10/667,127 filed September 18, 2003 herein incorporated by reference. Other computer system architectures and configurations are suitable.

## CLAIMS

What is claimed is:

1. A method for generating machine executable code for implementing a query of  
5 a database, the database having tables and records of data, comprising the steps of:  
receiving a subject query;  
forming an execution plan corresponding to the subject query, the execution  
plan having a sequence of pieces and corresponding processes for implementing the  
pieces; and  
10 for each piece of the plan, (a) generating source code using different code  
generation techniques as a function of any combination of data characteristics,  
current conditions and workload, and (b) compiling the generated source code to  
form machine executable code for implementing the subject query, said compiling  
being in a manner that optimizes total query processing time, including,  
15 compilation time and execution time.
2. A method as claimed in Claim 1 wherein the data characteristics include any  
combination of precision of a value existing in the database, scale of a value existing  
in the database, size of affected data existing in the database and data type.  
20
3. A method as claimed in Claim 1 wherein the step of generating further includes  
selecting code generation techniques based on intermediate results from earlier  
pieces, such that source code generation is effectively adapted to dynamic  
conditions.  
25
4. A method as claimed in Claim 1 wherein:  
the subject query is in a language native to the database; and  
the generated source code is in a high level language.

30

5. A method as claimed in Claim 1 wherein the step of generating source code includes generating source code for each process of each piece of the plan on a need-only basis, such that a savings on compilation time is achieved.
- 5 6. A method as claimed in Claim 5 wherein the need only basis provides generating source code for only those relevant members of a declared structure.
7. A method as claimed in Claim 1 wherein the subject query includes a join operation; and
- 10 the step of generating source code includes (a) representing output stream of the join operation as local variables that reference current records in each input stream, and (b) projecting named fields of the records, such that use of intermediate materialization and auxiliary structures are minimized.
- 15 8. A method as claimed in Claim 7 wherein the step of projecting includes utilizing structure offset expressions in the generated source code.
9. A method as claimed in Claim 1 wherein the step of generating source code includes: (a) minimally defining relevant structures and classes, and (b) forming
- 20 therefrom optimized include statements in the source code, the optimized include statements enabling reduced compilation time.
10. A method as claimed in Claim 1 wherein the step of generating source code includes, adjusting variables in the source code to have widths no larger than widths
- 25 of actual data values of respective data in the database.
11. A method as claimed in Claim 1 wherein the step of generating source code uses different combinations of the code generation techniques to provide different degrees of optimization in compiling.
- 30 12. A method as claimed in Claim 1 wherein the step of generating source code further includes for each instance of a function call to a respective function recited in

the source code, (i) determining size of data affected by the function call, and (ii) based on determined data size, replacing the instance of the function call in the source code with source code for implementing the respective function, such that the respective function itself is coded in-line.

5

13. A method as claimed in claim 12 wherein the step of replacing is performed as a function of determined data size relative to a threshold.

14. A method as claimed in Claim 1 wherein the subject query includes an outer  
10 join operation; and

the step of generating source code includes effectively overwriting field references in the outer join operation with null values.

15. A method as claimed in Claim 14 wherein the step of generating source code  
15 includes using a null value indicator for overwriting field references in the outer join operation.

16. A method for generating machine executable code for implementing a query  
20 of a database, the database having tables and records of data, comprising the steps of:

receiving a subject query;

forming an execution plan corresponding to the subject query, the execution plan having a sequence of pieces and corresponding processing nodes for implementing the pieces;

25 generating high level language source code for each piece of the plan including (a) minimally defining relevant structures and classes, and (b) forming therefrom optimized Include statements in the source code; and

compiling the generated source code to form machine executable code for implementing the subject query, the formed optimized Include statements in the  
30 source code enabling relatively reduced compilation time.



17. A method as claimed in Claim 16 wherein the step of forming optimized  
Include statements in the source code includes generating source code for only these  
relevant members of a declared structure.
- 5 18. A method as claimed in Claim 16 wherein the step of minimally defining  
includes inserting in-line into the generated source code respective local declarations  
of a relevant structures and classes defining only those class members and structure  
members that will be used by the generated source code.
- 10 19. A method as claimed in Claim 16 wherein:  
the high level language is C++; and  
the step of generating source code includes omitting a traditional #Include  
statement of a header of a structure declaration in the generated source code, where  
only certain members of the structure are relevant to the subject query.
- 15 20. A method for generating machine executable code for implementing a query  
of a database, the database having tables and records of data, comprising the steps  
of:  
receiving a subject query;  
20 forming an execution plan corresponding to the subject query, the execution  
plan having a sequence of pieces and corresponding processing nodes for  
implementing the pieces;  
for each piece of the plan, generating high level language source code  
including adjusting variables in the source code to have widths equivalent to widths  
25 of respective actual data values existing in the database, such that optimized  
compilation is enabled; and  
compiling the generated source code to form machine executable code for  
implementing the subject query, said compiling being done in optimal compile time.
- 30 21. A method as claimed in Claim 20 wherein the step of adjusting variables is  
based on determined ranges of data values in the database.

22. A method as claimed in Claim 21 wherein the step of adjusting variables further includes tracking ranges of data values in the database.
23. A method for generating machine executable code for implementing a query of  
5 a database, the database having tables and records of data, comprising the steps of:  
receiving a subject query;  
forming an execution plan corresponding to the subject query, the execution  
plan having a sequence of pieces and corresponding processing nodes for  
implementing the pieces;  
10 for each piece of the plan, generating high level language source code;  
optimizing the source code for compilation, said optimizing being a function  
of expected query execution time; and  
compiling the source code to form machine executable code for  
implementing the subject query.  
15
24. A method as in claimed in Claim 20 wherein step of optimizing includes  
applying a higher degree of optimization for longer expected execution time.

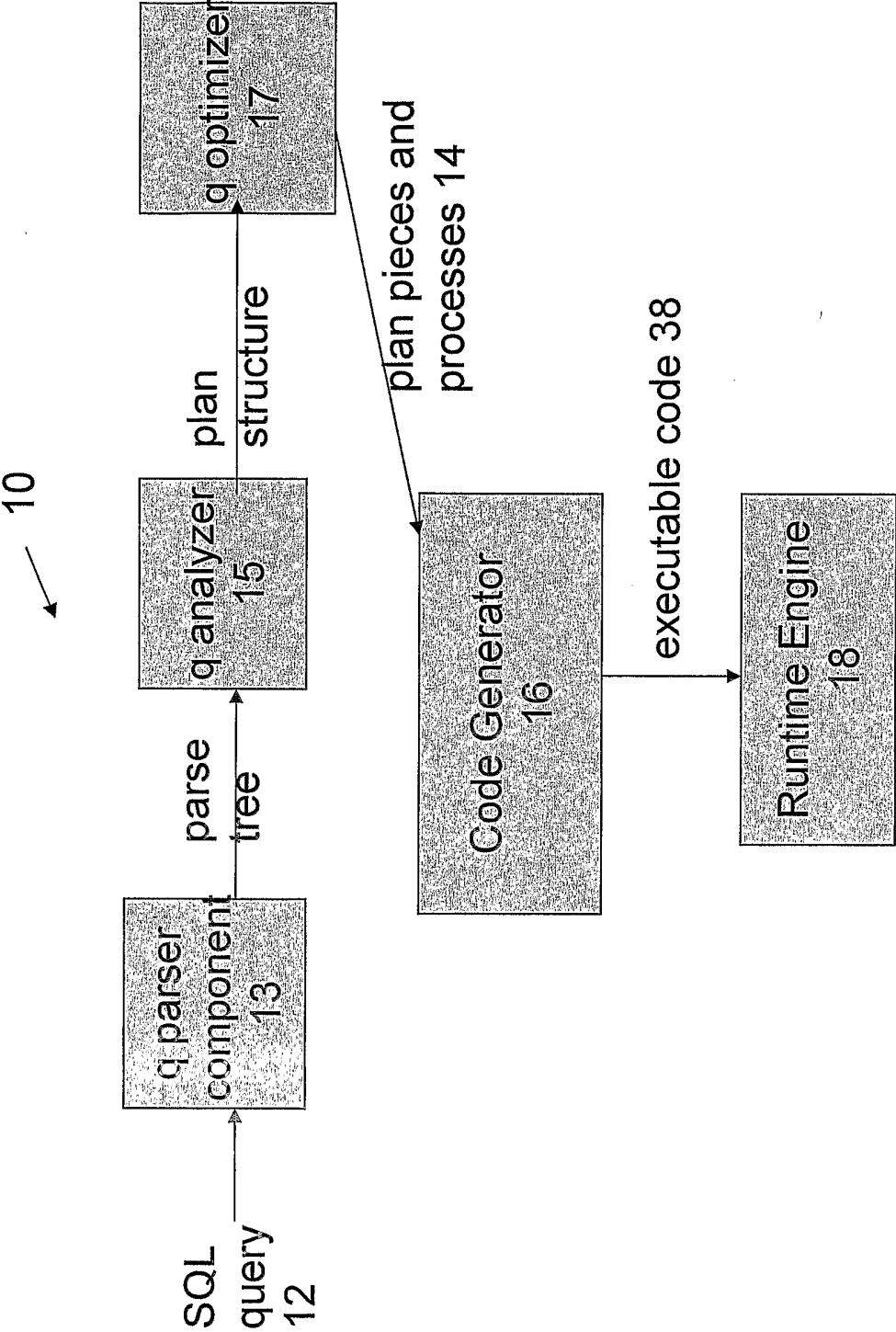


Figure 1

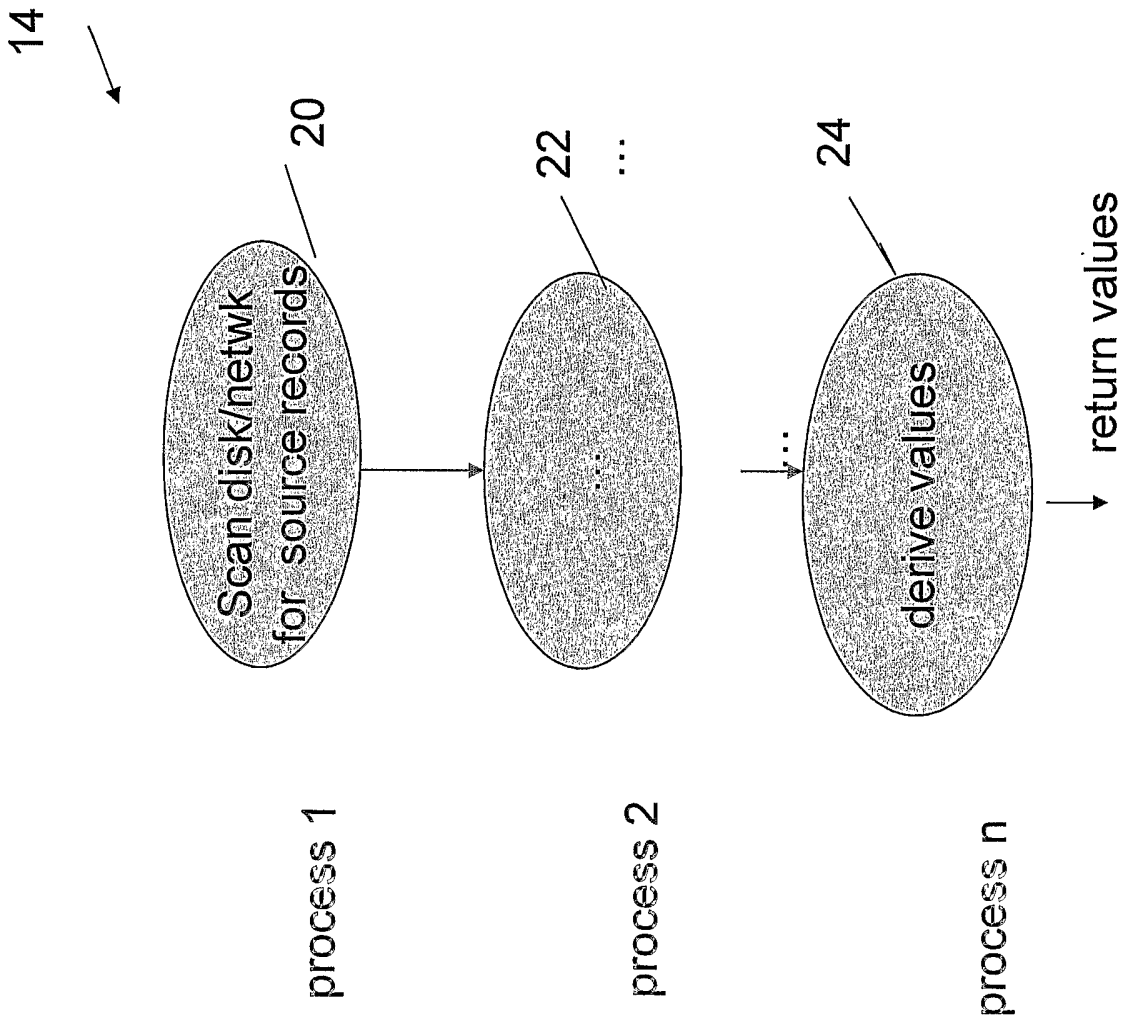


Figure 2

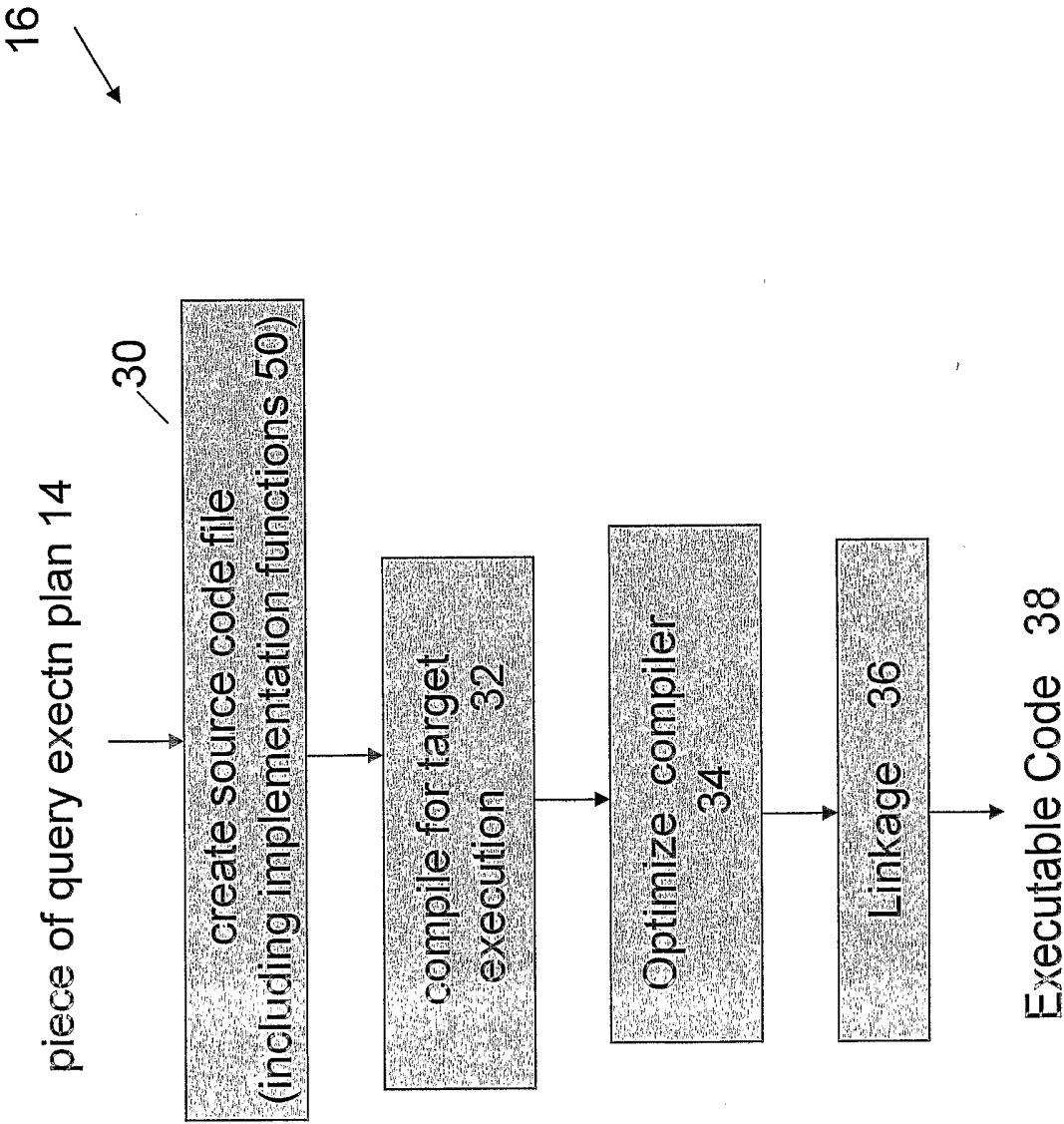


Figure 3

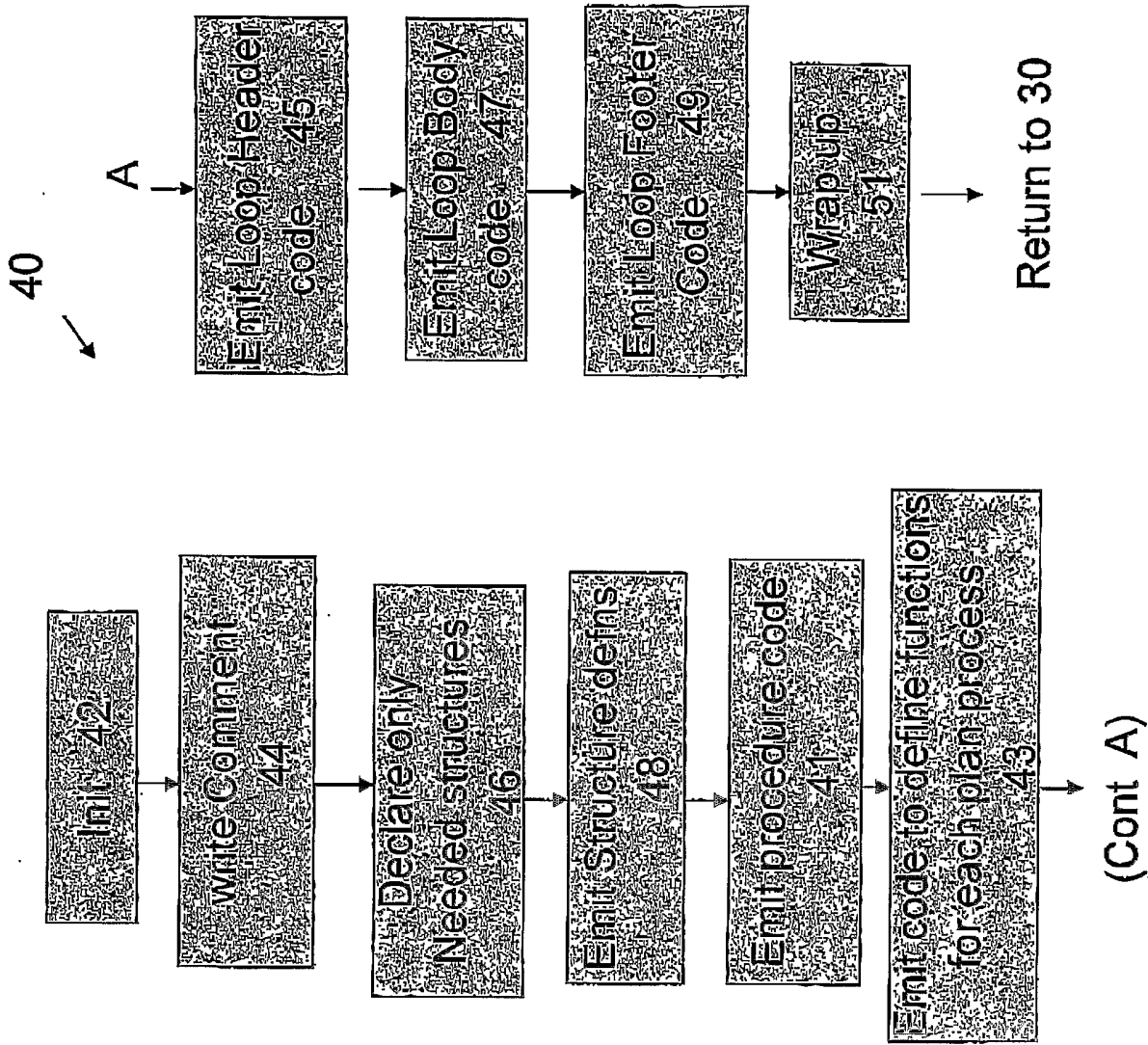


Figure 4