Ingres Optimizer Facility Overview

1. Introduction

The Optimizer Facility (OPF) is one of the major components of the Ingres database server. It is responsible for transforming a parsed query into an executable query plan. This transformation includes the phases one would expect from a state of the art commercial query optimizer including query rewrite and cost based plan generation. This document discusses the architecture of OPF in detail sufficient to enable a dbms practitioner to understand its major control structures and flow of control.

Note that this is a work-in-progress. Numerous sections have been laid out but not yet filled in. More will be added to this document over time.

2. Overview

2.1 Server Architecture

The flow of control diagram below shows the main components of the Ingres server and how they interact. Queries arrive in the server from Embedded SQL, the terminal monitor or some other query source through the generalized communication facility (GCF). GCF calls the system control facility (SCF) to pass the query to an existing connection with the user session, or to start a new session to receive the query. SCF places the query syntax into query storage facility (QSF) cache and calls the parser facility (PSF) to retrieve the syntax and parse it. PSF performs lexical, syntactic and semantic analysis of the query text. Depending on the statement, it may retrieve data base definitions from the Ingres catalog tables using the relation description facility (RDF). RDF also maintains a cache to avoid excessive access to the catalog tables, but may have to call the disk management facility (DMF) to retrieve rows from the catalogs. If the syntax is valid, PSF may then (again depending on the statement) build an internal form of the syntax known as the parse tree. The parse tree is augmented as needed with information from the catalogs. If it is a statement that needs to be compiled by OPF (most frequently, insert, update, delete and select statements, and database procedure statements, with several less obvious additions), PSF passes the parse tree to QSF to be stored in the query cache. PSF returns to SCF which then calls OPF to compile the parse tree into an executable query plan. OPF calls RDF for additional catalog information, compiles the query plan and places it into QSF cache. OPF then returns to SCF which calls QEF to execute the query plan. The plan is located in QSF cache and is then executed by QEF. Any data base access is achieved by calling DMF. Results are returned to the user through SCF via GCF.

The only remaining component is the abstract data facility (ADF) which is the expression evaluation
engine of Ingres. It is used by PSF to perform data type resolution of constants, columns and expressions coded in a query. It is used by OPF to generate scalar expressions for evaluating the contents of select lists, where clauses and any other syntax that involves column level expressions. And finally it is used by QEF to execute the scalar expressions compiled by OPF into the query plan.

2.2 OPF Architecture

OPF is a classic cost based relational database query optimizer. It reads an internal parse tree form of the original database query and by means of successive phases of refinement, it turns it into an optimized executable code form. Ingres supports two distinct query languages - QUEL and SQL. By the time the parse tree is generated, it is effectively independent of the original source query language. So while there are certain language features of each of QUEL and SQL that require attention by OPF (QUEL aggregate handling is one example, SQL 1992 outer join syntax is another), OPF is also independent of the query language.

2.2.1 Query Rewrite

OPF processes a typical query through 3 main phases - query rewrite, enumeration/plan generation and code generation. Query rewrite modifies the parse tree to generate a simplified form of the query that is both easier to optimize and is in a form that incorporates more optimization potential. Rewrite involves some complex transformations to the query to perform such operations as replacing view references with the parse tree of the view itself, flattening subselect where clauses into joins with the containing query and performing detailed analysis of aggregate functions to maximize optimization potential. This is important for QUEL queries which can contain aggregate constructs that are more complex than those that can be coded in SQL.

GCF
ESQL, TM, …
SCF
PSF
OPF
QEF
RDF
DMF
QSF
1. Ingres server flow of control.

ADF

In addition to major rewrite strategies such as those described above, OPF also incorporates a plethora of relatively simple rewrite heuristics designed on a smaller scale to achieve the same goals of simplifying the query and enhancing its optimization potential. Amongst these are the application of DeMorgan’s laws to eliminate “not”s from Boolean expressions (in the on, where and having clauses), the use of polynomial transformations to turn Boolean expressions into conjunctive normal form (all ORs are nested inside all ANDs) and a novel flattening transformation in Ingres that turns NOT EXISTS/NOT IN/<=> ANY subselects into outer joins with the containing query.

The rewrite phase of Ingres is also responsible for splitting a query into the individual pieces that will
subsequently be optimized. These are (somewhat confusingly) called subqueries inside OPF - not to be confused with syntactic subselects in the query. An obvious example of subqueries are the selects of a union query. The selects are turned into subqueries by the rewrite phase and each is then independently optimized by OPF.

2.2.2 Enumeration/Plan Generation

Enumeration takes each subquery generated by query rewrite and performs an exhaustive search to identify the best plan for executing it as determined by a cost model that associates costs with the atomic operations performed by the execution engine. As enumeration proceeds, Ingres retains the plan that has produced the lowest cost estimate to date, replacing it only when a subsequent plan is enumerated with an even lower cost estimate. It is this process that produces an “optimized” query plan for each query.

Ingres has always supported bushy join trees (the ability to have a join of 2 other joins), rather than the left or right deep join trees of other dbms’. Ingres also uses a novel technique for analyzing the use of secondary indexes in a query plan. Indexes are treated as distinct base tables (which they are, structurally speaking) which can be joined to the corresponding base table with a TID join. So the search space of possible query plans for an Ingres query is vast. It includes all structurally unique binary tree shapes of degree n (where n is the number of tables and/or indexes to be attached to the leaf nodes of the tree), all combinations of zero or more secondary indexes with the original base tables of the query and for each tree shape, table/index combination, all permutations of those tables and indexes.

Once a potential plan has been assembled (a tree shape, a set of tables and secondary indexes and a permutation of the tables and indexes assigned to the leaf nodes of the tree), the cost analysis is invoked. It performs a recursive descent of the plan tree, accumulating information as it goes. On the way out of the recursion, it computes costs of each of the table accesses and joins and appends them to the tree nodes. Upon return to the root node of the plan tree, the cost estimate for the whole plan can be completed.

Numerous heuristics are applied to limit the plan tree search space as much as possible. OPF also caches intermediate enumeration results – the structure and estimated cost of partial plans – to avoid as much as possible the recomputation of plan costs. So at each stage of cost analysis, it checks first to see if the plan fragment it is currently positioned on has already been analyzed and is in the plan cache. If so, it can simply use that cost, otherwise the analysis proceeds and those partial results are also cached. Moreover OPF incorporates a “short circuit” or timeout strategy in which it will stop searching for more plans if the estimated cost of the current best query plan is less than the amount of time spent in compilation.

Despite these heuristics Ingres can still take a very long time to compile queries with large numbers of tables and potentially useful secondary indexes (thus producing a large search space) and in which the table sizes are quite large (assuring that the cost of any plan is high and thus delaying the use of the timeout strategy). For these circumstances, a new enumeration algorithm was recently added that performs a localized or “greedy” cost analysis, building plan fragments from 3 tables at a time from the original query, rather than building a whole plan at a time. This changes the cost of enumeration from $O(n!)$ (the number of permutations of the tables and indexes all at once) to $O(nC_3)$ (the number of combinations of tables and indexes taken 3 at a time).
2.3 Code Generation

The final stage of query optimization in Ingres is code generation. It takes the optimal plan as determined by enumeration and generates the code form used by QEF to actually execute the query. This compiled query plan is stored in QSF cache (actually, the memory used to build the plan is acquired from QSF cache).

The major structures in an Ingres query plan are query trees, consisting of query node headers, and action headers. Query trees are generated from each subquery of the original query and consist of base table/index access, join mechanisms for joining rows together, sorts and links from one subquery to another. Action headers perform operations on the data returned from query trees. These operations can be as simple as projection of result columns/expressions in the select list of a query, to aggregate computations, to coordination of row retrieval and subsequent update/delete for update/delete statements.

Code generation involves recursive descent of the plan tree generated by OPF enumeration. As table and join operations are encountered in the plan tree, the code generator constructs the corresponding query tree nodes and incorporates table/index control blocks as needed for QEF to perform the operations. Information in the OPF subquery structures allows the code generator to build the action headers required to tie the query trees together.

In addition to the query tree nodes and action headers, the code generator compiles ADF expression code (CXs) to compute expression results at various points in the query plan. Each query tree node and most action headers have specific CXs that must be computed as part of their execution. For example, sort nodes have expression code to project the intermediate row to be sorted containing only the columns required above the sort. This allows the sort to be performed on a minimum of data, reducing the cost of plan execution. The different join algorithms all have specific expression requirements, including the comparisons that determine joining rows and the contents of outer joins.

Code generation is a largely mechanical operation, following the guideline of the enumerated plan tree. However there are a few instances of optimization implemented at this stage of OPF processing. When CXs are generated for projecting result rows, ADF will check for contiguous source columns being projected into a contiguous result location and will generate a single move operation for as many bytes as it can. When a query node is constructed to perform a keyed access to an index structured table, some effort is expended to assure that the most efficient possible use of the index is made.

3. External Interface to OPF

There is only one external interface to OPF and that is through the function opf_call. Other than 2 obscure calls from PSF, SCF is the only facility to call OPF. The following are the function codes with which OPF is called:

OPF_STARTUP – initializes OPF for processing when an Ingres server is first started up. It allocates and formats a server wide OPF control block which contains a variety of session default settings and values that can vary from Ingres installation to installation. It also calls ULM to initialize the OPF server wide memory pool and RDF to start itself up. A semaphore is initialized to control access to OPF.
OPF_SHUTDOWN – shuts down RDF and releases the semaphore. This indicates that OPF is no longer active.

OPF_BGN_SESSION – prepares an Ingres session for calling OPF. A session level control block is allocated with a copy of the server defaults for OPF. Exception handling is also initialized for the OPF session.

OPF_END_SESSION – is effectively a no-op.

OPF_ALTER – processes various statements that alter OPF session parameters. For example, there are statements that control whether OPF will attempt to flatten subselects, generate parallel query plans, perform various forms of debug processing and so forth. All such statements are parsed by PSF, then send to OPF by SCF using the OPF_ALTER function.

OPF_CREATE_QEP – performs the optimization of an executable statement.

4. Code Directories

As with most Ingres facilities, the code for OPF is split over a set of directories roughly categorizing the contained modules. There is also a header directory containing header files that define structures local to OPF. Structures shared between multiple facilities of the Ingres server are stored in back(hdr or common/.hdr. The remainder of this section outlines the OPF module directories and briefly describes the modules therein.

4.1 opa

The opa subdirectory contains modules that perform the lion’s share of the rewrite phase of OPF. Input to the opa phase is the parse tree as it was produced by PSF. The result of the opa phase is a set of OPF subqueries, one for each separately optimizeable piece of the original query, each addressing a modified piece of the original parse tree incorporating the result of the rewrite operations.

The “a” of opa is for the aggregate analysis that was originally the chief component of rewrite processing. However, these modules now also perform such widely disparate operations as view and subselect flattening, application of deMorgan’s laws and the pushing of predicates through unions, aggregates and views to the underlying base tables.

opaaggr.c is the entry module to the opa code. It coordinates the different aggregate analysis and rewrite logic of OPF. All DML statements pass through this module.

opabyprj.c analyzes aggregate functions to determine the need for projecting columns other than the by-list for computing aggregates. This is largely required by the complexity of QUEL aggregates.

opacompat.c contains functions that analyze aggregate functions to determine if they are compatible and can be executed together. This is a vestige of QUEL since it allows a single query to contain aggregate functions each with its own by-list. Optimizations are performed to group aggregates according to their by-lists, so that those with common by-lists can be executed together.

opafinal.c performs the final pass over a query, completing the changes made possible by the aggregate analysis phase.

opagen.c coordinates most of the flattening logic. It recursively descends the parse tree and performs operations depending on the parse tree nodes encountered. As view column references are encountered,
it executes logic to incorporate the view definition in the main query. Likewise, when subselects are encountered it attempts to flatten them into joins with the containing query. Aside from the aggregate analysis, most of the rewrite logic is contained here.

opainitsq.c allocates and formats the internal opf OPS_SUBQUERY structure for each subquery that is to be spun off from the main query.

opalink.c, opaoptim.c, opaprime.c perform other aspects of aggregate analysis. Most of this logic pertains specifically to expressions that can only be coded in QUEL. However, it may be possible to simulate some of the conditions addressed by these modules using nested SQL aggregate views.

opasuboq.c contains the logic that implements the OPF algorithm that flattens NOT EXISTS/NOT IN/<> ANY subselects into outer joins with the containing query.

4.2 opb

opb modules process Boolean factors.

The opb modules perform operations on Boolean expressions to prepare them for later phases of OPF execution. The Boolean expressions may originate in on clauses, where clauses or having clauses.

opbbfkget.c locates an internalized comparison constant (or list of constants) that appear in some Boolean restrictions. It is used to prepare keyed access through index structures.

opbcreate.c performs detailed analysis of Boolean expressions. It separates conjunctive factors into separate structures making it easier to perform later key access analysis. It builds an internal data structure that likewise makes it easier to compile index access strategies.

opbjkeyl.c builds a data structure indicating which columns of an index structure are matched by Boolean factors in the query. This is also part of the logic to identify potential keying strategies.

opbmbf.c checks for matching Boolean factors on an indexed key structure.

opbpmbf.c builds data structures for indexed tables showing keyed access potential.

opbpqbf.c builds data structures showing potential for partition qualifying access to partitioned tables. Partition qualifying predicates allow only qualified partitions to be accessed in executing the query.

opbsfind.c checks if an equivalence class in a Boolean factor is “sargable”; i.e., it is compared to a constant value (part of enumeration processing).

4.3 opc

opc modules perform OPF code generation.

opcadf.c contains functions that interface with ADF to build ADF compiled expression (CX) code.

opcagg.c contains the functions that prepare aggregate processing action headers. The bulk of the code is responsible for creating the various ADF CXs that accumulate and compute the aggregate values.

opcahd.c is responsible for building action header nodes and coordinating the logic that fills them in.

opcentry.c is the main entry into the code generation phase. It performs various initialization operations and calls other opc functions to build the query plan.
opcxch.c builds exchange nodes into a query plan for the execution of parallel queries.
opcjcommon.c contains functions common to different join algorithms that are called to compile join nodes into a query plan.
opcjoins.c builds the different join nodes supported by Ingres.
opckey.c contains functions to perform optimized keyed access to a structured base table or secondary index.
opcorig.c builds the orig node into a query plan to retrieve rows from a base table or secondary index.
opcprtree.c contains functions used to display tree shapes in various forms of debug output.
opcqen.c contains the top level function for building query tree nodes and numerous common service routines for code generation.
opcqpr.c contains functions to fill in background information in the root query plan control block (QEF_QP_CB).
opcqtree.c contains service routines that aid in identifying Boolean restrictions that can be used in conjunction with index structures.
opcqual.c builds ADF CXs for Boolean expressions in where, on and having clauses, and for projections the final result row of a query.
opcran.c builds the qp node that links the result of an action header to another segment of a query plan (as in a union select, for example).
opcsejoin.c builds subselect join nodes for subselects that were not flattened during query rewrite.
opcsorts.c builds sort nodes into a query plan.
opctuple.c contains functions to materialize intermediate results in a query plan.
opcomp.c contains functions to perform a detailed debug display of a compiled query plan.
opftoc.c contains functions to perform debug displays triggered by various OPF debug trace requests.

4.4 opd

opdcost.c contains cost analysis functions specific to distributed (STAR) queries.

4.5 ope

ope modules involve the processing of equivalence classes.
opeaddeq.c adds an attribute (column or expression) to an existing equivalence class.
opaseqcls.c scan a parse tree fragment looking for PST_VAR (column) nodes. Assign each to an equivalence class.
opcreate.c allocates and formats root structure that contains equivalence class definitions.
opfindj.c searches for join predicates, allocating and populating equivalence classes for all equijoin predicates and removing equijoin predicates from where clause parse tree fragments.
opemerge.c merge two attributes into same equivalence class, merging their respective equivalence classes in the process.

openeweq.c allocate and format a new equivalence class.

opetwidth.c compute length of a “row” comprised of attributes contained in an equivalence class map.

opetype.c return a data type from the attribute map of an equivalence class.

4.6 oph

oph modules perform the processing of histograms. They load the histograms necessary for the compilation of a query and perform the estimation of row counts from table scans and joins based on column value distributions and join and restriction predicates in a query.

opcbmem.c, ophccmem.c, opdcbme.c, opdccme.c, opdmemory.c and ophmemory.h perform allocation/deallocation of memory for histograms built during query compilation.

ophcreate.c creates the basic structure of a histogram when a dynamic histogram is required to reflect the value distributions of a column in the result of a join.

ophhistos.c loads histograms required to compile a subquery. Histograms are effectively required for each column referenced in a where clause. RDF loads the requested histograms if they exist, otherwise a skeleton histogram is created. If constants are referenced in restriction predicates for histogrammed columns, a check is made for the existence of the constant in a histogram cell. If there is no cell corresponding to the constant value, cells are inserted into the histogram to store the constant value and interpolation is used to rearrange value frequencies in the cells around it.

ophinter.c performs interpolation to determine the proportion of rows that correspond to a supplied predicate.

ophjselect.c computes the estimated number of rows resulting from the join of 2 tables with equijoin predicates represented by a supplied list of equivalence classes. The estimate is done by merging the histograms that correspond to the attributes identified in the equivalence class list. It may also produce result histograms showing the distribution of values in the join result.

ophmaxptr.c, ophminptr.c returns the greater or lesser of 2 histogram values.

ophpdf.c manipulate the proportions in a histogram so that they sum to 1.0.

ophrelse.c computes the estimated number of rows resulting from the application of a set of Boolean factors to a table. It may also build replacement histograms showing the distribution of values of various columns after the restrictions have been applied.

ophupupdate.c normalizes a histogram so that the sum of the proportions recorded in its cells is the same as an input selectivity value.

4.7 opj

opj modules post process the result of the rewrite phase performing a few more rewrite operations. Once the subqueries return from enumeration, the plan fragments are further processed to add information needed during code generation.

opjjoinop.c is the main module of the opj directory and coordinates both the post-processing of the
rewritten subqueries and the enumerated query plans. It adorns the subquery structures with information such as tables, equivalence classes, Boolean factors, columns/expressions and outer joins. After enumeration is complete, it flags each query plan (CO) join node with the join type (hash, merge, etc.) and sets various bit maps required for code generation. It also performs the analysis to parallelize the query plan.

opjnorm.c performs the conjunctive normal form transformation of Boolean expressions (until recently the Ingres expression evaluation engine could only execute expressions in CNF). Because this is a polynomial transformation, the resulting expressions may be quite a bit larger than the original. So this module makes a dry run, estimating the number of terms and operators in the transformed expression. If the number exceeds a threshold level, the expression is left unconverted and code generation produces ADF CXs that are not in CNF. This somewhat reduces the optimization potential of the query, but at least allows it to execute.

opunion.c performs a variety of analyses to improve the efficiency of union queries.

4.8 opl

opl modules perform analytical tasks on outer joins.

oplinit.c allocates and formats the structure that anchors information about outer joins in a subquery.

oplojoin.c performs a wide variety of functions pertaining to the compilation of outer joins. It contains the code that searches for outer joins that can be transformed to inner joins (see section 6.4, below). It also coordinates outer join specific cost analysis and maintains numerous structures used in the compilation of outer joins.

4.9 opn

opn modules perform the enumeration phase of OPF. They contain the code that builds the join trees to be subjected to cost analysis and applies many heuristics to reduce the size of the enumeration search space. They also manage the memory containing the cached plan fragments during cost analysis.

opnarl.c assigns the leaf nodes of a join tree. Successive calls result in successive assignments of the leaf nodes. It also contains a function that incorporates numerous heuristics for trimming the search space. For example, the TID join used to access a base table from a secondary index requires that the index access be located below and to the left of the base table. So any leaf node assignment in which a secondary index is above or to the right of the corresponding base table will be rejected.

opnbfhkget.c

opncalcost.c contain much of the cost analysis code. Processes a join, computing the cost of performing the join, the need for sorts on the inputs of the join and so forth. If the result is a complete query plan that is cheaper than the previous best query plan, it replaces the old one with the new one.

opnceval.c is the entry point to the cost analysis of a join tree. It also contains the code that determines if the compilation has timed out (the time estimate for executing the current best query plan is less than the time spent in compilation).

opncksrt.c determines the need for a sort at the top of a query plan and creates it if *necessary.
opncmemory.c, opncoins.c, opndcmem.c, opndeltrl.c, opndemem.c, opndrmem.c, opndsmem.c, opnememory.c various functions involving memory allocation and deallocation and enumeration cache management.

opncost.c transforms disk and cpu time estimates into a single unified cost value for comparisons with other costs.

opndiff.c computes difference measures between pairs of values. This is used in histogram processing for doing interpolation and join estimation.

opnum.c contains entry code to the enumeration phase. It does final preparation of the subquery for enumeration, determines which enumeration technique to use (OPF classic or greedy enumeration) and finishes the enumeration phase by assuring the best query plan has been copied to permanent memory and won’t be lost in the enumeration cache.

opneprime.c executes a statistical heuristic called “balls and cells” that is used in several disparate places in OPF. Among other things, it makes a statistical estimate of the number of disk reads required to randomly read m rows from a table occupying n pages.

opneqtamp.c returns a histogram for a given equivalence class.

opnextit.c cleans up prior to exit from enumeration processing.

opnegnperm.c generates successive permutations of a set of tables/indexes, applying some search space trimming heuristics on the way.

opnmerge.c search for potentially useful orderings in a plan subtree (columns that are referenced inside and outside the subtree and might make a useful ordering later in the plan).

opnjmaps.c sets bits in a variety of maps to define characteristics of joins in a join tree.

opncommon.c common logic to both joins and base table access executed during the cost analysis of a join tree.

opncost.c performs recursive analysis on a join tree, costing each table and join and attaching costs to the overall tree.

opnewenum.c performs the new greedy enumeration on a subquery. It contains all the logic to build the 3-table plan fragments, consolidate them into the list of tables and iterate the process until a plan is found.

opnpart.c is service routine that drives the generation of permutations and combinations in original enumeration algorithm.

opnprocost.c computes costs and result set size of projection/restriction operations applied to a plan node.

opnprleaf.c computes costs of performing projection/restriction on leaf node of a plan.

opnprocess.c takes a binary tree shape of degree n and drives the adornment and cost analysis of all valid leaf node assignments to the tree.

opnrefmt.c computes the cost of reformatting (usually sorting) the result of some portion of a query plan.

opnsavest.c, opnsrchst.c save and locate plan fragments in the enumeration cache.
opnsm1.c sets various bit maps and flags in plan node defining applicable Boolean factors and other node characteristics.

opnsm2.c sets more bits for join nodes in query plan.

opntpbk.c is utility function for computing the number of rows that will fit in a page of given size.

opntree.c allocates join tree node structures and drives the enumeration process by successively generating the different tree shapes of degree n. Tree shape generation is the outermost of the enumeration loops. It also implements a heuristic that guesses a reasonable set of secondary indexes to accompany the base tables in the first set of enumerated plans. The intent for complex queries is to start with a viable set of indexes likely to generate an efficient plan earlier in the enumeration process, giving the timeout mechanism a better opportunity to produce a decent plan quickly.

opnukey.c determines if keyed access can be performed to satisfy a given join.

4.11 opo

opo modules are concerned with ordering of data and are part of the enumeration phase.

opocombine.c combines the orderings of 2 join inputs into a single ordering that best describes the join output.

opocopyco.c copies a CO tree from enumeration memory to permanent (within the OPF session) memory.

opocordr.c contains various functions for creating orderings in a variety of situations (from a base table, from a list of equivalence classes, et. c).

opofordr.c locates multi-attribute orderings or makes one, if necessary.

opominc.c finds minimum cost CO in a list.

4.12 opq

opq contains the modules that make up the optimizedb utility that constructs histograms. The modules in this directory make up a standalone utility that can be executed from command line and are not part of OPF in the Ingres server. They are part of the OPF codeline as a historical artifact.

opqdata.sc contains global constant allocations and initializations.

opqoptdb.sc contains the majority of the histogram construction code. Included are functions that layout the data areas needed for histogram construction, locate appropriate catalog information, compose the query text used to retrieve column values for histogram construction, assemble the data into histograms and split it up and store it in the appropriate istatistics and ihistogram catalog rows.

opqscanf.c performs lexical scanning functions while processing an input file of histogram information to be stored in the catalogs.

opqstatd.sc contains the majority of the statdump utility code that reads existing histograms and displays them on the standard output file.

opqutils.sc contains utility functions shared between the optimizedb and statdump utilities. This
includes command line option interpretation and some catalog operations.

4.13 ops

1. 5. Major Structures and Concepts

This section discusses some of the major data structures and architectural concepts of OPF.

5.1 Subqueries

As described earlier, subqueries are the unit of optimization used by OPF. The rewrite phase of OPF (largely in the opa subdirectory) breaks a query into the pieces for which optimized query plans will be generated. By way of example, a query that joins a few tables with a view whose definition only consists of a simple select (no unions, no aggregates), the view definition can be merged into the referencing query and the entire combined query can be optimized as a single unit. However, if the view contains a union or aggregate functions, Ingres has no optimization strategies for combining the view into the referencing query. The view would be spun off as a separate subquery and an optimal plan would be constructed to compute it. The referencing query would then generate another subquery in which the result of the view would be represented as a “virtual” table whose results might be accessed from an internal temporary table or, with some types of subquery, the results of one subquery may be streamed into another.

Subqueries are categorized to define whether they represent unions, simple aggregates (no group by), function aggregates (aggregate function plus group by) and several more esoteric types. They anchor all lists and structures required to generate a query plan to execute the subquery. For example, they anchor a list of tables and indexes in the query that are local to the subquery. Many other entities specific to the subquery are also anchored in the subquery structure – Boolean factors, equivalence classes, columns and expressions, equijoin maps, outer joins and the lowest cost plan tree compiled to date for the subquery.

The optimization and code generation phases of OPF operate on subqueries as their root structures. Enumeration loops over the subqueries producing a plan for each. Code generation does likewise, generating a single query plan that ties the results of each subquery’s plan into the final plan for execution.

5.2 Equivalence Classes

Many OPF concepts date back to the Ph.D. dissertation of the late Bob Kooi who worked on Ingres in the mid 1980’s. Equivalence classes are a concept introduced to OPF during the first restructuring of commercial Ingres and came straight from Kooi’s thesis.

An equivalence class is a set of columns and/or expressions that are known to contain the same value at various points of query processing. Equivalence classes are determined by conjunctive factors in the where clause that are “=” predicates. The comparands of such “=” predicates are, by definition, in the same equivalence class. For example, the predicate “a.x = b.y” places a.x and b.y in the same equivalence class. A subsequent predicate “a.x = c.z” places all 3 of a.x, b.y and c.z in the same equivalence class.

Equivalence classes are a very useful abstraction from columns and expressions that are heavily used in performing join analysis during the enumeration phase of OPF. They are also used to determine index
structures that will support keyed access. For example, if a constant value is in the same equivalence class as an indexed column, that index can be probed with the constant value.

Equivalence classes are also used during code generation to determine the most easily accessible column to produce a value used in the query plan. Using the example of the equivalence class containing a.x, b.y and c.z, if a select list contains a.x, the value can be supplied from any of tables a, b or c, whichever is most convenient for the query plan. This allows greater flexibility in materializing values from rows and increases the optimization potential of a query.

5.3 Secondary Indexes

Ingres uses a novel technique to incorporate secondary indexes into a query plan. Secondary indexes are structurally identical to base tables in an Ingres database. They include the key columns of the index, optional non-key columns and the TID of the corresponding base table row. The index structure is built on the key columns and the leaf nodes are rows consisting of the columns described above.

Rather than include an execution primitive that uses the secondary index as an access path into the base table, Ingres reads the secondary index as it would any base table. If the key and non-key columns of the index alone cover the base table columns requested in the query, the base table will simply be dropped from the query in favour of the secondary index. If not, a special join will be performed later in the query plan to retrieve the corresponding base table rows (those with the same TID value as the TID column in the secondary index). By treating their secondary index access this way, OPF can perform operations such as index combination (effectively ANDing or ORing the TID lists returned from different indexes on the same base table) and deferred base table access to reduce the number of rows that need to be processed.

Secondary indexes are introduced to the query by a function that examines each base table in the query for potentially useful secondary indexes. If there are restriction predicates that map onto the key columns of an index, or if the columns of an index cover all referenced base table columns in the query, it will be added to the list of “tables” to be compiled into the query plan. The enumeration phase will then build plans from all combinations of 0 or more potentially useful secondary indexes, along with the base tables. This technique allows it to build join trees with all permutations of the various combinations of the indexes and base tables, allowing plans incorporating the sort of index combination techniques as outlined above.

5.4 Histograms

Ingres was the first commercial dbms to use histograms as a representation of column value distribution. Histograms are an integral component of the enumeration phase of OPF. They are used to estimate the number of rows qualified by restriction predicates in a query and also the number of rows in the join of 2 tables. OPF will also determine when histograms will be useful after the join of 2 tables (for example, when a 3rd table will be joined using some other join column) and will dynamically construct a histogram to estimate the value distribution of columns in the join output.

Ingres histograms are classed as equi-depth. They are constructed so that each cell represents roughly the same number of rows. They are built by the optimizedb utility which is an embedded SQL program.
that simply reads the values of a column using a variety of user-selected options including random sampling, then constructs the histogram data structures and stores them in 2 system catalogs – iistatistics and iihistogram. In addition to histograms on individual columns, Ingres also supports “composite” histograms on the concatenation of the key columns of a secondary index or base table index. This allows more accurate estimation of the number of rows qualified by restrictions on different columns in the same table.

5.5 Memory Streams

Several facilities of the Ingres server (PSF, OPF and QEF, being the most obvious) use a memory stream concept to make allocation and freeing of memory an easier task. A stream is a set of memory allocations for a related purpose. Each such allocation is chained to the others and to a stream anchor. When memory is released, a single request will release the whole stream, rather than require individual freeing of each allocated piece.

OPF uses several distinct streams. The use of different streams allows some classes of memory to be allocated and freed more locally than others – conserving memory for longer term use. The first is OPF’s overall utility stream from which the majority of its memory requirements are satisfied (subquery structures, equivalence classes, local table descriptors, parse tree nodes allocated during rewrite, etc.). The enumeration phase uses 3 distinct memory streams. One is used to contain histogram memory allocated during the enumeration process. It will be freed at the end of enumeration. The second is used to hold the join tree structures that describe the tree shape and leaf node assignments of a potential query plan. It is also freed at the end of enumeration.

The 3rd enumeration memory stream is the most interesting. It contains the “cost ordering” (CO) trees that are the actual proposed query plans. It is this information that is cached to reduce the number of intermediate plan computations that must be made. But large queries may require huge amounts of CO memory to hold all the intermediate results and OPF is designed to purge the cache when memory is exhausted during enumeration processing. If this happens, the current best plan is saved into the main OPF memory stream and the enumeration memory stream is closed and re-opened. This destroys all the intermediate cost structures and requires more work to continue the cost analysis. However, it insulates OPF from memory failure during the lengthy enumeration process.

5.6 Partitioned Tables

OPF does 2 types of special processing when a query contains references to partitioned tables. It searches the query for restriction predicates that match the partitioning scheme of a partitioned table. This will allow it to build information into the query plan that allows the execution facility to only open partitions known to contain qualified rows.

It also looks for joins between partitioned tables in which the join columns of the tables match their respective partitioning schemes. In such circumstances, OPF builds query plans that allow individual partitions of the tables to be joined pairwise, rather than the whole tables being joined in a single operation. Such a “divide and conquer” approach enables join algorithms such as hash and sort-merge to be processed much more efficiently.
5.7 Parallel Query Processing

Ingres supports parallel processing of long running, complex queries. Parallel processing strategies include pipelined parallelism in which query nodes on the same path in a query plan are executed in parallel, effectively piping rows from one node to the next. Bushy parallelism is also supported in which different portions of a query plan are executed in parallel – for example, different selects of a union query. Finally it also supports partitioned parallelism in which different partitions of a single partitioned table may be read by several processes simultaneously. For partition compatible joins (see the discussion of partitioned tables above) a whole join may be executed on several processes simultaneously.

OPF still generates serial query plans. After the best serial plan is produced, it is subject to further analysis to determine the potential for parallel execution. An estimate is made of the potential savings in execution elapsed time for each node, were it to be executed in parallel. A data structure is built containing all such estimates and is sorted on the estimated reduction. The nodes with greatest savings potential are then modified to execute in parallel. This is done by inserting “exchange” nodes to identify points of parallelism in the plan. (Exchange nodes define points in the query plan where intermediate result rows are exchanged between processes responsible for executing different sections of the query plan. They were proposed by Goetz Graefe in the design papers for the Volcano experimental dbms.)

5.8 Exception Handling

6 Example Optimizations

1. 6.1 Subselect Flattening

OPF uses fairly standard techniques for flattening subselects, with some notable exceptions. It is quite evident that optimizing a join query is easier than optimizing a query with an unflattened subselect. The semantics of a correlated, unflattened subselect effectively demand nested evaluation algorithms that can be quite inefficient. Techniques for transforming different classes of subselect queries into join queries were outlined in a paper by Won Kim (Sept. 1982 TODS) and are used by most commercial RDBMS’ (including Ingres).

Some subselects are more problematic than others. Specifically, subselects that test non-inclusion (NOT EXISTS, NOT IN, <> ANY) require more sophisticated transformations to process. Ingres effectively builds a temporary copy of the subselect result set with an added column indicating whether the row matches or doesn’t match a corresponding row of the containing query. This temporary table is then joined with the containing query, selecting only those rows which don’t match.

A more ingenious transformation has since been implemented to process non-inclusion subselects. OPF transforms such queries into outer joins between the containing query and the subselect, joined on the correlation columns. The result set is then restricted to the main query rows that don’t join to a subselect row. This allows the query to be optimized as a simple join, rather than the more arcane transformation inspired by Kim’s paper.

6.2 Pushing Predicates to Base Tables
A relatively recent enhancement to OPF attempts to place restriction predicates as close to base table access as possible. One of the axioms of relational query optimization says that restrictions should be applied as soon as possible to limit the amount of data processed in a query plan. Complex queries involving aggregate and union views make this more difficult than might otherwise be apparent. In particular, the OPF technique of splitting a query into independently optimized subqueries required moving predicates from one parse tree fragment (in one subquery) to another.

This is now achieved by a final pass over the subquery structures at the end of the rewrite phase. Compatible subqueries are identified and examined for the presence of restrictions in one that may apply to the other. The restrictions are then replicated in the other subquery to assure they are applied as soon in query execution as possible.

6.3 Union Elimination

OPF makes a pass over the subqueries that originate from a union query (or a query containing a union view) checking for subqueries that can’t possibly return any rows. These may arise from badly coded where clauses, for example something like “a > 95 and a < 50”. OPF will simply drop such subqueries from the plan, effectively eliminating them from further processing. A subsequent minor optimization flags any query that can’t return any rows so that no attempt is made to access the database before returning 0 rows.

6.4 Outer Join Processing

Ingres offered one of the first commercial implementations of SQL 1992 outer join syntax. The Ingres execution engine supports numerous outer join algorithms giving OPF the opportunity to compile efficient query plans even in the presence of outer join syntax. OPF even detects outer joins that, by virtue of restriction predicates, can be treated as inner joins with even more optimization potential. For example, in the query “select * from a left outer join b on a.x = b.y where b.z > 25”, the “b.z > 25” where clause restriction implies that result rows may only contain non-null b.z column values. Therefore the join can be compiled as an inner join.

6.5 Greedy Enumeration

The exhaustive search technique used by OPF during enumeration causes problems as the query gets larger. OPF only passes complete join trees to the cost analysis part of enumeration. The search space of all possible join trees contains all permutations of leaf node assignments of all combinations of 0 or more potentially useful secondary indexes with the original base tables to all possible structurally unique binary tree shapes of degree n (n being the number of secondary indexes and base tables in the current combination). This quickly becomes a very large number. In an 8 table query with 4 potentially useful secondary indexes, there are 2.5 X 10^{11} different join trees to pass through the cost analysis. This underscores the importance of the various heuristics used to trim the search space, as well as the timeout heuristic that allows OPF to terminate enumeration before all possible plans have been examined. However, there will always be queries that take an unacceptably long time to produce a decent query plan.

To address this problem, OPF has been enhanced with a new greedy enumeration technique that greatly
reduces the search space for large queries, yet still produces reasonably efficient query plans. Whereas the original enumeration algorithm constructs entire join trees for the query before performing cost analysis, the new heuristic does localized optimization by building all plans involving 3 tables/indexes at a time. It chooses the lowest cost 3 table plan fragment, then replaces the 2 tables at the bottom of the plan fragment with their join in the array of tables to be joined in the query. The size of the array is reduced by one and the process is repeated (with the join of the 2 eliminated tables being treated as a “composite” table in the next round of cost analyses). Eventually, a plan is built containing all the tables from the query, but in far less time (the 8 table, 4 index example quoted before would require cost analysis of just over 4000 join trees using the new heuristic).

7 Code Architecture

1. 7.1 ops_sequencer (entry processing)
2. 7.2 opa_aggregate (query rewrite)
3. 7.3 opj_joinop (pre-, post-enumeration processing)
4. 7.4 opn_enum (enumeration)
5. 7.5 opc_querycomp (code genera

For more information, please visit www.ingres.com