1. **Topics**
   a. Inside the SQL Server 2005 Query Optimizer
   b. The Volcano Optimizer
   c. The Cascades Framework for Query Optimization
   d. Other resources

A. **Inside the SQL Server 2005 Query Optimizer**

The query optimizer is the component of a database management system that attempts to determine the most efficient way to execute a query. The optimizer considers the possible query plans for a given input query, and attempts to determine which of those plans will be the most efficient. In other words, an optimizer reorders operators and selects implementation algorithms. The general paradigm of database query optimization is to create alternative (equivalent) query evaluation plans and then to choose the best among the many possible plans. However, it is easy to waste a lot of computational effort in extensible query optimization. Real-world query optimizers are complex pieces of software, incorporating search heuristics with cost-evaluation models. In sum:

- **Given:** A complex query (i.e. joining n tables)
- **The "Plan Space":** We must search in a huge number of alternative, semantically equivalent plans.
- **The Perils of Error:** Running time of plans can vary by many orders of magnitude.
- **Ideal Goal:** Map a declarative query to the most efficient plan tree.
- **Conventional Wisdom:** You're OK if you avoid the rotten plans.

In what regards the plan space, we have that:

- **We must consider the available query processing algorithms:**
  - sequential & index (clustered/unclustered) scans
  - NL-join, (sort)-merge join, hash join
  - sorting & hash-based grouping
  - ...
- **Logical equivalences:**
  - joins are commutative and associative (reorderable) (modulo avoiding Cartesian products)
  - Selection and projection can be "pushed" below joins (modulo relevant info being preserved)
- **Note some heuristics can be folded:**
  - selections are always "pushed down"
  - projections are always "pushed down"
  - all this is only for single query blocks
- **Some other popular heuristics:**
  - only left-deep plan trees
  - avoid plans with cartesian products

Most query optimizers determine join order via a dynamic programming algorithm pioneered by IBM's System R database project. This algorithm generally works in three stages:

1. First, all ways to access each relation in the query are computed. Every relation in the query can be accessed via a sequential scan. If there is an index on a relation that can be used to answer a predicate in the query, an index scan can also be used. For each relation, the optimizer records the cheapest way to scan the relation, as well as the cheapest way to scan the relation that produces records in a particular sorted order.
2. The optimizer then considers combining each pair of relations for which a join condition exists. For each pair, the optimizer will consider the available join algorithms implemented by the DBMS. It will preserve the cheapest way to join each pair of relations, in addition to the cheapest way to join each pair of relations that produces its output according to a particular sort order.
3. Then all three-relation query plans are computed, by joining each two-relation plan produced by the previous phase with the remaining relations in the query.

Historically, System-R query optimizers would often only consider left-deep query plans, which first join two base tables together, then join the intermediate result with another base table, and so on. This heuristic reduces the number of plans that need to be considered (n! instead of 4^n) and is drawn from the observation that join algorithms such as nested loops only require a single tuple of the outer relation at a time. Therefore, a left-deep query plan means that fewer tuples need to be held in memory at any time: the outer relation's join plan need only be executed until a single tuple is produced, and then the inner base relation can be scanned (this technique is called "pipelining").

**The Volcano/Cascades Optimizer**

In the past, the Volcano project researched query processing techniques for database systems. Volcano proposed an optimizer generator (a tool that can generate query optimizers with basis on supplied parameters) that provides complete data model independence and natural extensibility. The data model, logical algebra, physical algebra, and optimization rules are translated by the optimizer generator into optimizer source code, which is then compiled and linked with the other DBMS components. Volcano provides effective support for non-trivial cost models and for physical properties such as sort order. At the same time, the generated optimizers are very efficient, as they combine dynamic programming with goal-directed search (i.e. top-down or backward-anchoring search, starting with the plan output and recursing on all possible subtrees) and branch-and-bound pruning. The search algorithm is driven by needs rather than possibilities.

In a follow-up project called Cascades, a cleaner and more object-oriented version of the Volcano engine was proposed. **MS SQL Server is currently based on a (simplified and streamlined) version of the Volcano/Cascades optimizer generator.**

The user queries to be optimized by an optimizer generated by Volcano are specified as a logical algebra expression (tree). The translation from a user interface into a logical algebra expression must be performed by a query parser. The output of the optimizer is a plan, which is an expression over a physical algebra. The set of algorithms, their capabilities and their costs represents the data formats and physical storage structures used by the database system.
The Volcano optimizer generator therefore uses two algebras, called the logical and the physical algebras, and generates optimizers that map an expression of the logical algebra (a query) into an optimal expression of the physical algebra (a query evaluation plan consisting of algorithms). To do so, it uses mappings within the logical algebra and cost-based mapping of logical operators to algorithms. The knowledge of algebraic laws required for equivalence transformations and mapping operators to algorithms is expressed using patterns and rules. The algebraic rules of expression equivalence, e.g., commutativity or associativity, are specified using transformation rules. The possible mappings of operators to algorithms are specified using implementation rules. Conditions may also be specified with both transformation and implementation rules, by attaching them to some condition code which will be invoked after a pattern match has succeeded. Both these rules are translated independently from one another and are combined by a search mechanism when optimizing a query, through a dynamic programming algorithm.

The results logical and physical algebra expressions are described using logical and physical properties, in turn derived using property functions (there must be one property function for each logical operator and for each algorithm). Logical properties are derived from the logical algebra expression and include the schema, expected size, etc. They are determined before any optimization is performed and are attached to equivalence classes, i.e. sets of equivalent logical expressions and plans. Physical properties depend on algorithms, e.g., sort order, partitioning, etc. They can only be determined after an execution plan has been chosen and are attached to specific plans and algorithm choices. The set of physical properties is summarized for each intermediate result in a physical property vector. Applicability functions determine the physical property vectors that the algorithm's inputs must satisfy. Cost functions estimate algorithm's cost, and the optimizer implementor can choose cost to be a number (e.g., estimated elapsed time), a record (e.g., estimated CPU time and I/O count), or any other type, providing that it defines appropriate comparison functions.

Each optimization goal (and subgoal) is a pair of logical expressions (i.e. logical and physical) and a physical property vector. In order to decide whether or not an algorithm can be used to execute a logical expression, the optimizer matches the implementation rule, executes the condition code associated with the rule, and then invokes an applicability function that determines whether or not the algorithm can deliver the logical expression with physical properties that satisfy the physical property vector. For example, when optimizing a join expression whose result should be sorted on the join attribute, hybrid hash join does not qualify while merge-join qualifies. After the optimizer decides to use an algorithm, it invokes the algorithm's cost function to estimate its cost.

The search algorithm employed by optimizers created with Volcano uses dynamic programming by storing all optimal subplans as well as optimization failures until a query is completely optimized. Since it is very goal-oriented through the use of physical properties and derives only those expressions and plans that truly participate in promising larger plans, the algorithm is more efficient than other approaches to using dynamic programming in database query optimization (e.g., IBM's System R). We'll now analyze it in more detail.

Volcano's directed dynamic programming approach derives equivalent expressions and plans only for those partial queries that are considered as parts of larger subqueries (and the entire query), not all equivalent expressions and plans that are feasible or seem interesting by their sort order. It uses backward chaining, exploring only those subqueries and plans that truly participate in a larger expression.

Algebraic transformation includes the possibility of deriving the same expression in several different ways. In order to prevent redundant optimization effort by detecting redundant derivations of the same logical expressions and plans during optimization, expression and plans are captured in a hash table of expressions and equivalence classes (hashed on physical/logical properties). An equivalence class represents two collections, one of equivalent logical and one of physical expressions. The logical algebra expressions are used for efficient and complete exploration of the search space, and plans are used for a fast choice of a suitable input plan that satisfies physical property requirements. For each combination of physical properties for which an equivalence class has already been optimized, the best plan found is kept.

Given these building blocks, Volcano works “top-down” as follows.

1. Consider the “top” of the query plan tree. We know the logical and physical properties from the query specification. Assume that a budget limit on the costs may be given to prune search. Use our rules and properties to recurse as follows:
   a. If the desired output is in the hashtable with cost below the limit, use it, else generate all subgoals based on:
      i. applicable transformations.
      ii. algorithms that give the required physical properties.
      iii. enforcers (i.e. specific algorithms such as sort), may be introduced, giving some required physical properties.
   b. Apply the "move", update the resulting physical and logical properties and costs. Add result to the hashtable. If cost is still within budget, recurse.

   As we can see, the cost limit is used to improve the search algorithm using branch-and-bound pruning. Once a complete plan is known for a logical expression (the user query or some part of it) and a physical property vector, no other plan or partial plan with higher cost can be part of the optimal query evaluation plan. Furthermore, cost limits are passed down in the optimization of sub-expressions, and tight upper bounds also speed their optimization.

   If a move to be pursued is the exploration of a normal query processing algorithm, its cost is calculated by the algorithm’s cost function. The algorithm's applicability function determines the physical properly vectors for the algorithms inputs, and their costs and optimal plans are found by finding the best plan for the inputs.

B. Other Resources