Partial Join Order Optimization in the ParAccel Analytic Database

Yijou Chen, Richard L. Cole, William J. McKenna, Sergei Perfilov, Aman Sinha, Eugene Szedenits Jr.
ParAccel, Inc.
3 Results Way, Cupertino, CA 95014
{joe.chen, rick.cole, bill.mckenna, sergei.perfilov, aman.sinha, gene.szedenits}@paraccel.com

ABSTRACT
The ParAccel Analytic Database™ is a fast shared-nothing parallel relational database system with a columnar orientation, adaptive compression, memory-centric design, and an enhanced query optimizer. This modern object-oriented optimizer and its optimizer framework, known as Volt, provide efficient bulk and instance level query expression representation, multiple expression managers, and rule and cost-based expression transformation organized via multiple optimizer instances. Volt has been applied to the problem of ordering very large numbers of joins by partially ordering them for subsequent optimization using standard dynamic programming. Performance analyses show the framework's utility and the optimizer's effectiveness.

Categories and Subject Descriptors
H.2.4 [**Database Management**]: Systems – *parallel databases, query processing, relational databases*.

General Terms
Algorithms, Performance, Design, Experimentation.

Keywords
Query Optimization, Extensible Optimizer Framework.

1. INTRODUCTION
The ParAccel Analytic Database™ (PADB) is a new-generation relational database management system that combines leading-edge innovations with best practices to deliver a fast, simple, and cost-effective platform for analytic processing. PADB features address performance, high availability in case of disk or node failure, and solution simplicity via a true load-and-go design for normalized, de-normalized, and dimensional schemas. This design provides performance without auxiliary data structures such as indexes or materialized views. Performance and scalability are a function of columnar orientation, adaptive compression, shared-nothing massively parallel processing, a scalable communications fabric, and cost-based query optimization. Details of the features and capabilities of PADB may be found in the ParAccel Analytic Database technical whitepaper [8].

SIGMOD'09, June 29–July 2, 2009, Providence, Rhode Island, USA.

In this paper we focus on query optimization, including the design of a new PADB optimization capability addressing the large join order optimization problem via a new query optimization framework called Volt. The following sections describe the PADB query optimizer, how the PADB query optimizer and Volt interoperate, core Volt optimizer classes, the Volt Partial Join Order Optimizer, and three examples of query performance.

2. QUERY OPTIMIZATION IN PADB
Query optimization in PADB is based on an enhanced version of the Postgres [10] query optimizer. For example, the Postgres optimizer has been enhanced to address the query planning requirements of a shared-nothing parallel system. This enhanced Postgres optimizer is the baseline for cost-based query optimization in PADB.

A goal of the Volt query optimization framework is to enhance the Postgres/PADB query optimizer, while providing an entirely modern, object-oriented set of reusable software components. The framework is designed to simply and gradually support the introduction of new customized optimizer instances/rules into the context of the existing PADB optimizer. The first application of the Volt optimization framework is to address the problem of optimizing the order of joining very large numbers of tables, even thousands of tables.

2.1 Total and Partial Join Ordering
Choosing a good join order is crucial for processing queries that join a large number of tables, queries typically found in decision support and business intelligence applications. Although the Postgres optimizer uses dynamic programming to reduce the exhaustive search among all possible join orders, the optimizer is still highly resource intensive. In practice, its elapsed time is exponential as a function of the number of tables and it may run out of memory when ordering as few as 10 to 15 tables.

In order to avoid an exponential increase in resource consumption when there are a large number of tables, Postgres includes the Genetic Query Optimization (GEQO) module, which uses a genetic algorithm to converge on a join ordering solution. However, GEQO query plans may be of poor quality.

The Postgres optimizer considers all tables together when doing either exhaustive or heuristic (genetic) based optimization, a common algorithmic approach [3, 6, 11, 13]. Combinations of exhaustive and heuristic or randomized algorithms have also been developed [5, 12, 14].

The approach taken by the PADB/Volt optimizer has two phases: the first phase analyzes the set of tables being joined in a query and heuristically decomposes the entire set into smaller groups of tables,

i.e., a family of sets. In the second phase these groups are passed to the Postgres optimizer, which uses cost-based optimization to choose the best join order within each group *as well as between groups*. This phased approach also allows the Volt optimizer to guide the search of the Postgres optimizer with very little modification.

The Volt approach substantially reduces resource consumption in the Postgres optimizer, although it is possible that in some cases the quality of the plans may be affected. Testing on TPC-H, TPC-DS, and customer queries demonstrates that for large queries the PADB/Volt optimizer produces better plans in less time than the Postgres GEQO optimizer.

In order to leverage this new query optimizer capability, queries must be handed off to Volt and back to the core PADB optimizer, which is the topic of the following subsection.

2.2 Postgres to Volt

The Postgres-to-Volt layer mediates between the Postgres/PADB parser, planner and Volt optimizers. Its general task flow is: 1. Accept an input Postgres query (a tree-like structure). 2. Build the corresponding Volt expression. 3. Feed the expression to a suitable optimizer (Volt is comprised of multiple optimizers/rules), returning an optimized Volt expression. 4. Convert the optimized expression to a Postgres-digestible form to be returned to the caller.

Building the Volt expression occupies the greatest part of the code with the hub being a function to return a Volt expression for an input Postgres "node" structure. This function is mainly a large dispatcher that calls the function suitable for the input node type. If the particular node structure has fields that are also nodes (e.g., joins, predicates, and arithmetic expressions) then its particular function will call the dispatcher function for each such field and combine the returned expressions into a greater Volt expression. Otherwise the particular function for a simple node (e.g., a column or constant) simply builds the corresponding Volt expression. In this way the final Volt expression is constructed bottom-up by calling the dispatcher function for the topmost nodes of the query.

Volt expressions are not built directly but vended by various Volt manager classes and often require multiple steps to prepare inputs for a factory method [1]. The Postgres-to-Volt layer hides these details in an intermediate class that holds instances of the required Volt managers and provides a simplified interface for expression construction.

Volt Metadata Source Interfaces provide an extensible framework to manage schema definitions from different physical repositories. Separating logical data definition from physical storage will allow Volt to realize portable query optimization for different database engines. From a software engineering perspective, the isolation of Metadata definitions also facilitates rapid prototyping of new features and component-based test-driven development. For example, Volt Metadata Mock Interfaces and Query Repositories enable Volt to optimize queries without a SQL parser or physical database catalog interfaces.

2.3 The Volt Optimizer Framework

Based on the work described in [7], Volt is a collection of reusable software components (a software framework) that implement the abstractions central to query optimization [2, 4, 9, 15]. Volt's components are a relatively small number of C++ classes that can be composed to build query optimizers, which are also implemented as C++ classes. Volt is designed to be columnar-aware and MPP-aware and is being used to build a new optimizer for PADB. However, the Volt classes are stand-alone and can be used outside the scope of PADB. The benefits of this decoupling include optimizer platform independence and the capability to perform independent quality/performance testing.

The *Expression* class is central to Volt. This class and its subclasses are used to represent bulk-level logical and physical algebra expressions, as well as predicates, columns, tables, and scalar and numeric expressions. Expressions are allocated and stored in instances of the *ExpressionManager* class, which follows the Singleton design pattern [1] and thus guarantees that only a single instance of an Expression exists in a given ExpressionManager instance based on the specified sense of expression equality. For example, two instances of table expression T1 may be considered equal in the context of common subexpression detection across SQL blocks, but not equal if they are in the same FROM clause. In addition to implementing different notions of equality, Expressions also implement ordering functions, such as "less than." One result of this implementation is a guarantee that SQL generated (unparsed) from equivalent Volt expressions is lexicographically equal.

Expression instances maintain information about the other Expression instances that reference them, as well as the instances they reference. This property makes certain query rewrites that rely on column references simple to implement. For example, the elimination of joins from a fact to a dimension table depends on knowing that only the primary key column(s) of the dimension table is referenced and this reference information is always recorded with the single instance of the dimension table. Another rewrite that is easy to implement, via automatically maintained reference information, is the evaluation of expressions in the SELECT list of a non-composable view or derived table that can be eliminated. This reference information also makes garbage collection possible, giving Volt minimal memory requirements.

Collections of Expression instances can be stored as ordered/unordered sets or multisets, which are implemented using the *ExpressionList* class. One use of this class is to represent an Expression's inputs. For example, an *n*-ary OR predicate's inputs can be implemented as a set, which by definition eliminates duplicate disjuncts. Properties of operators (e.g., OR), such as whether duplicate inputs are allowed, are used by Volt to determine the appropriate type of ExpressionList in which to store an Expression's inputs.

Operator properties are used by Volt to perform rewrites at Expression *allocation time*. For example, knowing that operator ">=" is anti-symmetric allows an allocation request for "A>=B AND B>=A" to return "A=B". Similarly, knowledge that "<" is not reflexive allows a request to allocate "A<A" to return "FALSE" if A is not nullable or "UNKNOWN" if nullable.

An instance of Volt's *Optimizer* class accepts an Expression as input and returns an optimized Expression. This class has different subclasses, including the *RewriteOptimizer* and the *PartialJoinOrderOptimizer*. The former can perform rewrite tasks such as subquery decorrelation, join elimination, and column pruning. A PartialJoinOrderOptimizer instance analyzes an Expression's join graph and constructs table groupings using join selectivities and join types, e.g., FK-PK, Many-to-One, Many-to-Many, and One-to-One, which are used to build Expressions that are

inputs to the physical planning Optimizer. This type of Optimizer is described in more detail in the following subsection.

An Optimizer subclass is free to implement its optimizations in whatever manner the implementer decides is appropriate. Implementation of groups of heuristic (or cost-safe) query rewrites, e.g., view composition/folding, join elimination, outer-to-inner join conversion, may be done using rules and rule engines (drivers). To support these types of optimizations, Volt provides *Rule*, *RuleSet*, and *RuleDriver* classes. Rules transform an Expression from one consistent state to another consistent state. RuleSets implement sets of Rules, which are passed to a RuleDriver along with an Expression to be transformed. A RuleDriver understands Rule application order, how Rules interact with each other, and termination policies. Other types of optimizer strategies might be implemented without these rule-based components but instead be composed from other Volt components. An example of such an Optimizer is a traditional System-R [11] bottom-up dynamic programming join planner.

2.4 Partial Join Order Optimization in Volt

The Volt Partial Join Order (PJO) optimizer makes use of three graph structures: an FK-PK graph, a Join graph, and a Result graph. The algorithm further uses the Volt Multi-Join expression, a logical *n*-way join operator. The PJO rule, a Volt Rule as described above, is a member of a RuleSet comprising the Volt PartialJoinOrderOptimizer. The PJO rule is applied to a Volt expression by the optimizer's RuleDriver.

The PJO rule first performs an applicability test to determine if the partial join ordering can be done. For example, the rule checks the number of connected components. Once applicability succeeds, the PJO rule proceeds to the application phase.

For a given query, if foreign keys have been defined for the tables involved in a query, Volt builds an FK-PK graph for these tables and stores the graph as part of the query's context. If no foreign keys are defined for any of the tables, the FK-PK graph is empty. While the PJO rule may leverage FK-PK information when available, the rule is not limited to star schemas [13].

Volt next builds an undirected Join graph using all join predicates within a single query block. The Join graph represents joins between base tables or derived tables or any combination thereof. The Join graph edges are annotated with the join predicates such that an edge represents all joins between a pair of tables.

Volt then classifies the edges of the Join graph based on certain heuristics. The purpose of this classification is to identify whether a join is a pure FK-PK join, a One-to-One join, a Many-to-One join, or a Many-to-Many join. During this process, Volt uses the following information: 1. The FK-PK graph if it has been built for this query. 2. Uniqueness constraints defined in the schema. 3. Table and column statistics. 4. Estimated selectivity of local predicates. 5. Estimated selectivity of join predicates. Once this classification is complete, Volt produces a *directed* Result graph based on the undirected Join graph. The nodes and edges in the Result graph are then annotated with cumulative weights based on the selectivities of the local and join predicates.

Once all cumulative weights of nodes and edges have been computed, the next phase is to form groups of tables. Each group is represented by a Multi-Join expression and a single Multi-Join expression has a limit on the number of inputs that depends on a configurable *maximum group size* parameter. In order to populate the group, Volt performs a *weighted* depth-first traversal of the Result graph. When a node is visited, its corresponding table is added to the current group. New groups, i.e., Multi-Joins, are created as needed as the algorithm ensures that a table is added to a group only if it was joined to another table already in the group. An example grouping with a maximum group size of 3 might look like (((T1, T2, T3), T4, T5), T6) where each "("<table reference list>")" denotes a group. This grouping means the output of the joins of T1, T2, and T3 may be joined with T4 and T5, and then the output of the joins of all of the first 5 tables joined with T6. An alternative grouping might be ((T1, T2, T3), (T4, T5, T6)).

After a complete grouping has been formed, a second optimizer, such as the standard Postgres dynamic programming optimizer as used in the following performance examples, is free to optimize the various join order permutations within and between groups. Because the second phase of optimization is decoupled from the first, partial ordering phase, no particular type of second optimizer or approach to using the partially ordered groups is required.

2.5 Performance Examples

What follows are three short examples of performance results: for a single template query with increasing numbers of joins, performance data for many different queries having various numbers of joins, and one very large join query allocated and optimized within Volt space alone.

In the following experiments the Volt maximum group size was set to 10 and the Postgres GEQO optimizer was configured per the default Postgres parameters. Experiments were performed using a quad-core Intel® Xeon® CPU E5430 @ 2.66GHz on a system with 16 GB of memory running Red Hat Enterprise Linux ES release 4. Query optimization was single-threaded while query execution on PADB ran in parallel using four cores.

2.5.1 Example Join Query Analysis

The graph in Figure 1 shows the query elapsed time (the sum of query optimization time and execution time) in seconds with respect to the number of tables joined in a query. The query is TPC-DS query 78, modified by adding/removing tables and appropriate join predicates. Volt shows near linear scalability over a wide range of number of tables compared to GEQO, which degrades at 27 tables.

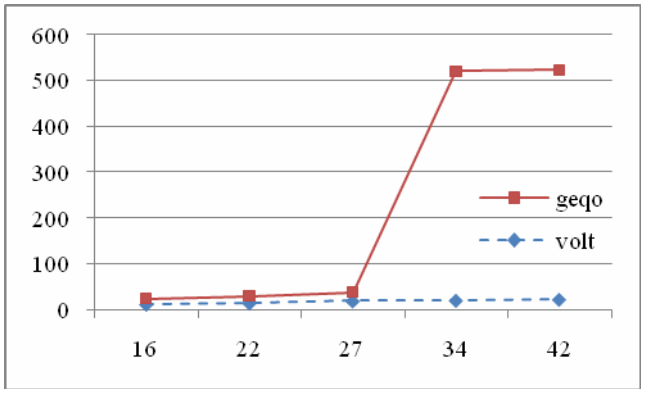

Figure 1. Elapsed time (seconds) versus number of tables.

2.5.2 Summary Data Analysis

The graph in Figure 2 is based on collating data obtained for join order optimization of 45 different queries derived from the TPC-H

and TPC-DS benchmarks, as well as representative customer queries. These queries joined anywhere from 2 to 42 tables. Ratios of total elapsed query time (sum of optimization and execution time) for each query are plotted versus the number of tables joined. Ratios of Volt Partial Join Order to GEQO are plotted as squares and ratios of Volt Partial Join Order optimization to the standard Postgres exhaustive dynamic programming optimizer are plotted as diamonds. Logarithmic solid and dashed trend lines depict the respective ratio trends.

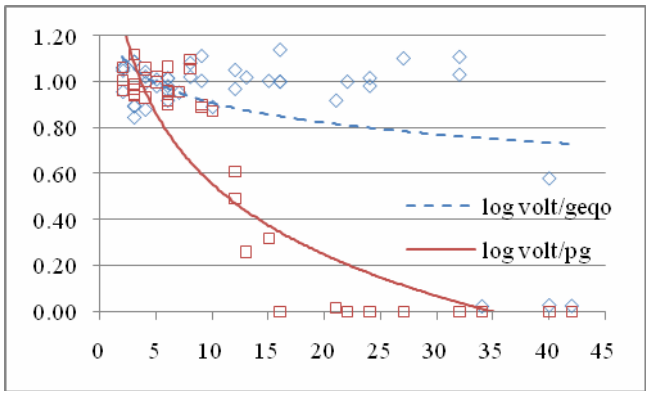

Figure 2. Ratio of elapsed times versus number of tables.

Volt Partial Join Ordering improves on standard dynamic programming techniques between 10 and 15 tables, where standard techniques fail to even finish query planning. Partial Join Ordering is comparable to the GEQO genetic algorithm until 20 to 30 tables and beyond, where GEQO plans are poor.

2.5.3 Very Large Queries

Elapsed time and peak memory usage data in Table 1 was obtained by optimizing a 1,000 table join expression constructed in Volt space alone, i.e., the expression was not parsed SQL input.

Volt was able to efficiently allocate and quickly optimize this very large join expression while maintaining a small memory footprint.

Table 1. Results for a 1,000 table join in Volt space.

Measurement Category	**Result**
Volt expression allocation time	3.89 seconds
Partial Join Order Optimizer run time	4.65 seconds
Total elapsed time	8.54 seconds
Peak memory usage	*46 MB*

3. CONCLUSION

The Volt optimizer framework extends the capabilities of the PADB query optimizer, facilitating a simple and gradual application to existing and new problem domains, such as total join ordering, physical query planning, and robust query execution. Volt Partial Join Order optimization is only the first example optimization that leverages this new framework.

4. REFERENCES

[1] Gamma, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995. ISBN 0-201-63361-2.

[2] Graefe, G. The Cascades Framework for Query Optimization. *Data Engineering Bulletin, Vol. 18, 1995*.

[3] Guttoski, P. B. et al. Kruskal's Algorithm for Query Tree Optimization. *IDEAS 2007*: 296-302.

[4] Kabra, N. and DeWitt, D. J. OPT++: an object-oriented implementation for extensible database query optimization. *The VLDB Journal, Vol. 8, No. 1 April 1999: 55-78.*

[5] Kossmann, D. and Stocker, K. Iterative Dynamic Programming: A New Class of Query Optimization Algorithms. *ACM TODS Vol. 25, No. 1 March 2000*:43-82.

[6] Krishnamurthy, R. et al. Optimization of Nonrecursive Queries. *VLDB 1986*: 128-137.

[7] McKenna, W. et al. EROC: A Toolkit for Building NEATO Query Optimizers. *VLDB 1996*: 111-121.

[8] ParAccel, Inc. *The ParAccel Analytic Database: A Technical Overview.* http://www.paraccel.com/. Whitepaper, 2009.

[9] Pires, C. G. and Machado, J. C. *DORS: Database Query Optimizer with Rule Based Search Engine*. SugarLoafPLoP 2002.

[10] The PostgreSQL Global Development Group. *PostgreSQL: Documentation.* http://www.postgresql.org/docs/. Online documentation, April 2009.

[11] Selinger, P. et al. Access Path Selection in a Relational Database Management System. *SIGMOD 1979*: 23-34.

[12] Swami, A. Optimization of large join queries: Combining heuristics and combinational techniques. *SIGMOD1989*: 367-376.

[13] Tao, Y. et al. Optimizing Large Star-Schema Queries with Snowflakes via Heuristic-Based Query Rewriting. *CASCON 2003*: 279-293.

[14] Vance, B. *Join-order optimization with Cartesian products.* Ph.D. Dissertation. OGI, Beaverton, OR. 1998.

[15] Xu, Y. *Efficiency in the Columbia Database Query Optimizer*. Master's Thesis. Portland State University, 1998.