Parallelizing Extensible Query Optimizers

Florian M. Waas
Greenplum Inc.
flw@greenplum.com

Joseph M. Hellerstein
UC Berkeley
hellerstein@cs.berkeley.edu

ABSTRACT

Query optimization is the most computationally complex task in a database management systems. In many query optimizers, faster CPUs and increased RAM can translate directly to better query plans and thus better overall system performance. Although memory size continues to scale with Moore's Law, processor speeds are leveling off. Chip manufacturers are now focusing on multicore designs that integrate increasing numbers of cores in a single CPU. Query optimizers need to be parallelized in order to continue enjoying the growth trend of Moore's Law.

In this paper, we address this problem in the context of the extensible optimizer architectures found in many commercial database systems. We identify the key data dependencies inherent in the dynamic programming at the heart of these optimizers. We use this insight both to design a flexible parallel query optimization implementation, and to assess the opportunities for parallelism in this context.

The proposed solutions can serve as a blueprint for retrofitting existing industry-grade optimizers to leverage multicore architectures, without requiring significant rework of the underlying infrastructure.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems

General Terms

Algorithms, Performance

Keywords

Query optimization, parallel processing, depedencies, multicore, Erlang

1. INTRODUCTION

Moore's Law describes the exponentially increasing number of transistors that can be packed on an integrated circuit; it has been borne out by a doubling of transistors every 18-24 months over the last 50 years. Until recently, the most visible direct effect of Moore's Law was an exponential increase in processor speeds, roughly measured in instructions per second. In recent years, however, chip manufacturers have had difficulty continuing to scale processor performance while managing heat and energy effectively. Instead, they are using the increasing number of transistors to lay out multiple, cooler-running processor *cores* on a single chip. These *multicore* processors are becoming ubiquitous, and it is expected that over the coming years, Moore's Law will be reflected not by increasing processor speed, but by increasing numbers of cores per chip [10, 3]. This implies that if software systems want to continue enjoying growth in CPU performance, they need to be programmed for parallelism.

Database systems have long taken advantage of parallelism, in both shared-memory and cluster-based architectures. The focus of database parallelism has traditionally been on query execution, and implications on maintaining correctness during transaction processing [15]. Query optimization is one of the most computationally complex, CPU-intensive tasks in a database system, but until recently there was almost no work on parallel query optimizers. In a recent paper, Han, *et al.* [14] studied the problem of parallelizing the traditional optimization algorithm from System R [23] for Select-Project-Join blocks. They demonstrated that the algorithm can be effectively parallelized, with notable benefits in the sizes of queries that can be planned without resorting to unproven heuristics.

A query optimizer is a long-term investment, carefully evolved over a long period of time. Designed well, it provides extensibility in two dimensions. First, a good optimizer framework can accommodate new query optimizations and query language features as they are designed, helping keep pace with changes in the marketplace. This has been the subject of much research and engineering. Modern query optimization includes a broad class of techniques beyond the join ordering and access method selection considered in the classic System R work, including early aggregation and duplicate elimination [27], subquery decorrelation [8], materialized view matching [11], and many more. The Starburst project showed that a "bottom-up", System R-style optimizer can be made extensible [19, 22], and IBM DB2 has proven this concept in a commercial setting. A number of other commercial optimizers – including those of Microsoft SQL Server and HP Neoview – achieve extensibility via the ideas of Volcano [12] and Cascades [13], which modify the approach of System R to fill in a dynamic programming table in various orders other than strict bottom-up.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'09, June 29–July 2, 2009, Providence, Rhode Island, USA.
Copyright 2009 ACM 978-1-60558-551-2/09/06 ...$5.00.

The second dimension of extensibility involves exploiting innovations in computer hardware. This dimension has traditionally been implicit in optimizer design: faster CPUs and increased RAM support the application of more (and also more complex) optimizations *without* requiring software modifications. The shift to multicore requires this form of extensibility to be explicit in software: query optimizers need to exploit Moore's Law via parallelization. Given how heavily invested vendors are in their optimizer technology, they are looking for solutions to run in parallel on multicore architectures without having to give up hard-earned competitive advantages in their current optimizer architectures.

In this paper, we consider the challenge of parallelizing an extensible optimizer. Our approach is built on an explicit representation of the dependencies in dynamic programming, captured in a data structure we call the *search state dependency graph* (SSDG). The unfolding search of the dynamic programming algorithm is represented by a state-annotated frontier of nodes in an expanding SSDG, which exposes subtasks in a manner that enables a simple, flexible approach to parallel scheduling of optimizer subtasks.

Using this insight as a basic building block, our contributions in this paper are three-fold:

1. *Extensible, Parallel Optimization:* We consider extensible optimizers that can handle a variety of optimization rules, and even extensible search strategies [17, 6]. This is in contrast to prior work that is wedded to the structure of join enumeration [14].

2. *Flexibility:* Our approach allows for flexibility in the order of searching the optimizer state space. In addition to supporting multiple traditional search strategies, it enables dynamic scheduling of parallel search subtasks. This flexibility can ameliorate unpredictable task distributions arising from statistics-driven estimation and pruning of the search space.

3. *Analysis of Parallelizability:* By explicitly capturing the dependencies among subtasks, we are able to measure the available degree of parallelism during the course of optimization. We present empirical results from a prototype optimizer implementation in Erlang, showing extensive opportunities for exploiting a growing number of processor cores. This ability to dynamically monitor the degree of parallelism also points the way to future adaptive approaches to scheduling subtasks in parallel optimizers.

From a practitioner's point of view, these are important criteria for retrofitting existing optimizer technology with parallelism, and provide input into the effort and benefits involved in modifying a particular pre-existing optimizer.

2. QUERY OPTIMIZATION IN PRACTICE

Architecturally, a query optimizer is a software *framework* that has to accommodate an ever changing set of operations and optimization techniques, while maintaining a continuity of optimization performance over an extended period of time. Requirements for extension typically arise from additions to the query language used both to implement standards (e.g. OLAP extensions of SQL:2003) as well as proprietary language extensions (e.g. XQuery integration in the supported SQL dialect [4]). Over the last two decades, significant attention has been paid to designing flexible and extensible optimizer architectures, including a number of efforts documented in the research literature [7, 19, 22, 18, 20, 12, 17]. Frameworks based on enumeration via *Dynamic Programming (DP)* have emerged as the *de facto* standard in this domain.

CPU speeds have increased $1,000,000\times$ since Selinger's seminal paper was published [23], but the complexity of SQL queries has increased only modestly. This has rendered the use of optimization heuristics increasingly marginal over time—most queries can be optimized via enumerative approaches, which are guaranteed to find the global optimum of a well-defined search space according to a specific statistical model. As an example, consider query Q5 of the TPC-H benchmark suite, which contains a 5-way join. According to [26], a Cascades-based optimizer considers a total of over 230,000,000 alternative plans as a matter of a few 10s of seconds.

This number illustrates that enumeration-based optimization using DP is highly efficient. But more interestingly it illustrates that the growth of the search space at hand is not dominated by join order optimization: there are only a few dozen different join orders to consider. Instead, the plan space blows up due to a multitude of optimization techniques involving the treatment of aggregates. This latter point runs counter to textbook lessons about the complexity of query optimization. Although join ordering continues to be one of the significant components of query optimization complexity, it is no longer the sole component. And the growth of other sources of complexity drive the need to scale query optimization further.

It should be noted that there are design challenges in making DP scale. Many modern optimizers maintain an elaborate balance between resources used during optimization and the quality of the result of the optimization. In sophisticated optimizer implementations, the optimizer may make budgeting decisions during the course of optimization, considering how to trade off the quality of the best solution found so far against investing further resources for what might be only marginal improvements. In general, exhaustive enumeration needs to be guided carefully and certain simplifications may be applied to keep the search space to a manageable size. Queries with dozens of joins are rare, but a system has to be able to cope with them.

3. OPTIMIZER ARCHITECTURE

In this section we briefly survey the main components of a Cascades-style query optimizer. We assume the reader familiar with the basic principles of transformation-based query optimization and focus only on the elements that are relevant for our further exposition. For an in-depth discussion of the Cascades optimizer see e.g. [13].

3.1 Algebraic Optimization

Algebraic query optimizers like Cascades are distinguished by a number of key features, the most notable of which are listed below [9]:

- Using a search strategy the optimizer generates alternatives for the original algebra expression using transformations. A transformation applies to a (partial) input tree of operators and generates a one or more equivalent alternatives. Typically, transformations im-

plement small changes in an expression—e.g. commutative exchange of inputs to an operator—but can encompass arbitrarily sophisticated optimizations.

- Due to its organizing of optimizations as individual transformations that interoperate freely the optimizer is inherently extensible: any number of optimizations can be added without affecting previously existing ones in a negative way.

- Optimization is truly cost-based and does not rely on heuristics or engineering choices made by the implementers. Instead, all relevant alternatives are generated and costed and the plan that is optimal according to the cost model is chosen.

3.2 Representation of Alternatives

The input to the optimizer is an initial query tree. The representation can be generalized to DAGs instead of trees. However for the purpose of our research, considering trees only is fully sufficient. Each node in the tree corresponds to an operator of the extended relational algebra.

The core of the optimizer is a lookup table—sometimes referred to as *MEMO structure* or *blackboard*—that manages alternatives. Instead of maintaining alternative plans in their entirety, the optimizer uses an encoding that breaks plans down into *groups* of equivalent sub-expressions. The principle applies recursively in that groups are inputs to operators in other groups. This compact representation is similar to the one used for join-enumeration in [21].

To illustrate this principle, consider the input plan: when initializing the table each operator of the input plan is placed in a separate group (we ignore the case of common subexpression for the sake of clarity). The dependencies between operators in the initial plan correspond to references between groups. We call a group A *dependent* on a group B if any of the operators in A references group B. During optimization, alternative plans or plan fragments are generated by applying transformations. Each transformation may produce a set of alternatives that are also inserted into the table. As the optimization progresses, partial alternatives for which no equivalent group of expressions exist may be generated. In this case a new group is created and added to the lookup table.

The schedule according to which transformations are applied is determined by a *search strategy* that may prioritize certain optimizations based on their anticipated applicability, consider costs of the best partial solutions found so far, or take into account external factors such as overall memory consumption and elapsed time.

Once all alternatives have been explored—i.e. no more transformations are to be applied according to the search strategy—every group contains one or more alternatives. Following the *Principle of Optimality*, the best solution of each group is composed of optimal solutions to its dependent groups.

The representation is distinguished by being highly compact and very general as it is not restricted to certain classes of operators, e.g. join operators, but applies to any operator of the extended algebra. Transformations can implement arbitrarily complex optimizations ranging from simple join re-ordering to sophisticated matching strategies for materialized views [11, 9]. The modular architecture and strict separation of concerns result in enormous flexibility and extensibility.

3.3 Optimization Workflow

The actual workflow of optimizing a query comprises various conceptually different tasks. In [13], Graefe proposes a categorization of tasks that is based on a specific implementation. And although different implementations are likely to deploy a different structure of tasks using Graefe's taxonomy is very helpful to illustrate the concept of dependencies between tasks.

In our prototype, we implemented the following types of tasks:

- The execution of a transformation that applies to an individual operator of a group and generates one or more alternatives; this includes transformations that apply to partial plans that are rooted in these operators;

- The systematic exploration of a group by iterating over its operators and scheduling individual transformations of the previous type;

- The optimization of dependent groups using cost estimates for effective pruning of the search space; this type of task schedules the systematic exploration of a group as necessary;

- The extraction of the optimal solution of a group and assembly of the final plan;

As the above list emphasizes, tasks correspond to self-contained units of work and maintain a commonly shared state in the lookup table. Moreover, different tasks need to be executed in specific orders. For example, the exploration of a group must be complete before a cost bound can be established, which in turn is used to optimize other dependent groups. Furthermore, the structure of the overall query imposes dependencies between transformations, e.g. join ordering may get deferred until sub-query de-correlation is complete etc. We call a task that requires another task to be completed in order to proceed *parent task*, the other *dependent task*. Dependent tasks of the same parent are *siblings*. There is no requirement for siblings to be executed in any specific order.

We refer to the actual instance of a task as a *job*. The dependencies between tasks can be implemented by a general scheduling component. However, a more practical and robust approach is to have jobs create dependent jobs automatically. For example, the job of extracting the optimal solution in a given group may check if the group has been completely explored and create a job to do so if necessary. In general, there is no limitation on how many dependents a job may create.

3.4 Serial Scheduling

To illustrate job dependencies and their importance for parallel optimization we briefly discuss scheduling of optimization jobs in a serial (i.e. non-parallel) environment.

The scheduling of jobs—i.e. the assigning of jobs to processes or threads—is implemented by a scheduler component. The scheduler as proposed in [13] is strictly serial and uses a stack to manage dependencies between jobs. The set of pending jobs is maintained on the stack in reverse order

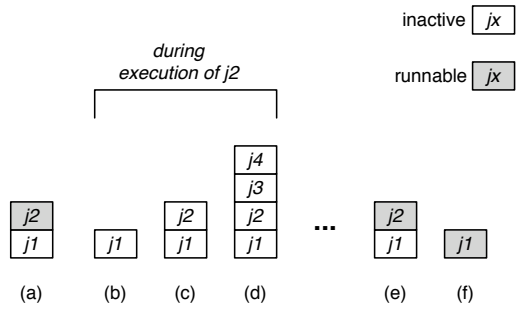

Figure 1: Stack of serial scheduler during course of execution

of their creation. The following simple invariant holds: between job executions, the top-most job on the stack is the next to be executed. However, *during* the execution of a job, the stack is typically being modified by the scheduler to accommodate new jobs, and the invariant is not valid.

Example. Consider 4 optimization jobs, $j_1, ..., j_4$. Assume j_1 is parent of j_2, and j_2 parent of both j_3 and j_4. After j_1 creates j_2 the stack is in the state as shown in Fig.1(a). j_1 is inactive and j_2, as the top-most job, *runnable*. The scheduler removes j_2 from the stack and assigns it to a thread for execution (b). Throughout steps (b)–(d) job j_2 is running. Since j_2 requires dependent jobs j_3 and j_4 to complete in order to proceed it reschedules itself (c) and adds jobs j_3 and j_4 (d). Once job j_2 returns control to the scheduler j_4 is started. After j_4 and j_3 are complete, j_2 is the top-most job again and can now proceed without spawning additional dependents (e). Once j_2 is complete control is returned to the scheduler who starts job j_1 (f). Once all jobs are executed and the stack is empty the optimization is complete. □

4. PARALLELIZING OPTIMIZATION

To exploit parallelism of the workflow and execute jobs that do not depend on each other we encode dependencies explicitly. We develop the notion of a *Search State Dependency Graph (SSDG)* in the following to facilitate detection of opportunities for parallel execution of jobs and their scheduling.

To motivate the need for a richer data structure, consider a simple attempt at using a stack-based scheduler to execute jobs in parallel. As we pointed out above, the stack is well-formed only between the execution of jobs. Simply scheduling the top-most job at any point in time while a job is running may break any of the implicitly encoded dependencies between tasks. In above example, starting the execution of j_1 while j_2 is running will lead to either the creating of redundant jobs (if j_1 concludes that it needs to create dependents) or to an incorrect order of execution (if it concludes that its dependents must have completed already). Also, a stack-based approach does not preserve any knowledge as to which jobs are dependents. In short, a stack-based scheduler cannot be used for scheduling more than one thread, because the stack reflects dependencies correctly only between the executions of jobs.

In order to exploit dependencies we define and record the states of jobs explicitly as follows:

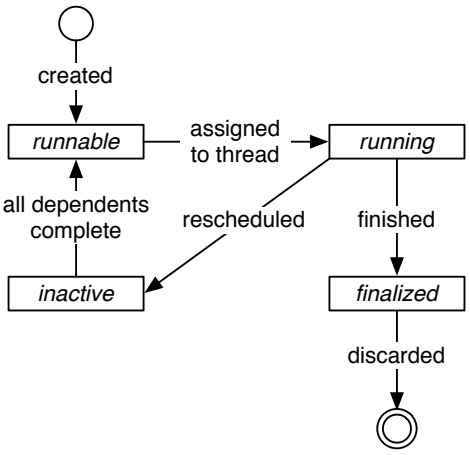

Figure 2: State transition diagram

- *runnable*: the job can be assigned to a thread
- *running*: the job is currently being worked on and cannot be assigned to another thread
- *inactive*: the job is waiting for dependent jobs to be completed
- *finalized*: the job is complete and can be discarded

The scheduling of a job can then be described as a state machine with transitions as depicted in Figure 2. By representing every job that is not yet finalized by a node and dependencies as directed arcs between them we encode all dependencies in a graph.

Example. Consider above example with jobs $j_1, ..j_4$. The scheduling steps are depicted in Figure 3. Leaf nodes, i.e. nodes with no outgoing edges, are runnable and can be assigned to a thread (a). The assignment is atomic. Unlike with a stack-based scheduler, running jobs are not removed from the data structure (b) but simply marked inaccessible to other tasks. Dependent jobs are added to their respective parent and are immediately runnable (c). All runnable jobs are executed in parallel pending availability of processing resources. Once the parent is rescheduled it becomes inactive (d) until all its dependents are complete (e). After the job completes it is removed from the graph (e, f). □

5. IMPLEMENTATION

In order to quantify the potential for parallelism as identified in the SSDG we implemented a complete prototype for an extensible optimizer based on the principles of Volcano and Cascades, replacing the conventional stack-based scheduler with an SSDG-based parallel approach.

5.1 Erlang

We wrote our prototype using the programming language *Erlang* [2]. The decision to use a language little known in the database community was driven by the need for a programming environment that (a) combines powerful abstractions of parallel primitives, (b) is highly efficient, and (c) offers enough control to study the aspects of multicore architectures in detail.

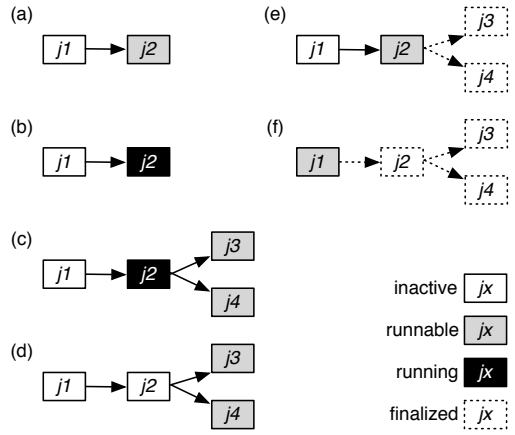

Figure 3: Search State Dependency Graph for Example; finalized jobs shown only for completeness

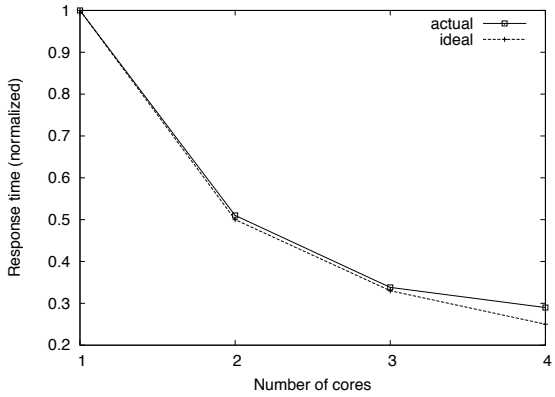

Figure 4: Actual scale-up of independent Erlang processes compared to ideal scale-up

Erlang is a parallel functional programming language with a concise and elegant process model. It has its roots in CSP [16] and Occam as well as in Prolog. Erlang was originally developed to meet requirements for high-availability and massive data throughput in switching gear for telecommunication systems but has recently attracted attention from various fields of computing due to its parallel nature. Erlang is *process-based*, i.e. its programming model is built on the notion of processes in constrast to, say, objects in object-oriented languages. All state is encapsulated as processes that communicate via a single, powerful, high-level communication primitive. As a result even programs that are not intended to be run on parallel hardware are always structured as a number of communicating processes. This programming model is highly conducive to software design that lends itself immediately to parallelization.

It is important to point out that the concept of a process in Erlang is very different from that in operating systems: Erlang processes are lightweight, have a very small resource footprint, and are interpreted by the Erlang Virtual Machine (EVM). The EVM has been designed to handle very large number of processes simultaneously—e.g. several tens of thousands [1]—and is highly optimized for communication.

Since there is no shared or global state that requires synchronization, Erlang programs are inherently easy to parallelize by simply instructing the EVM that interprets the program to leverage more processors or cores. The EVM does the mapping of Erlang processes to threads completely transparently.

In summary, we found Erlang to be an excellent choice for prototyping the SSDG and a general parallel optimizer framework: quick to write, simple to modify, easy to debug and verify.

5.2 Optimizer Framework

We implemented a basic yet complete Cascades-style optimizer framework along the lines of [12, 13]. In the following we briefly describe the most significant differences of our implementation with those used in other commercial products.

After the initial, canonical plan is copied into the MEMO, a number of optimization and exploration tasks are scheduled. Tasks perform various modifications of the plan fragements stored in the MEMO and schedule dependent tasks as necessary. The order of execution of the tasks is determined by a scheduler component that—in constrast to the original Volcano and Cascades scheduler—maintains the SSDG, identifies runnable tasks, and assigns them to processes. At any point in time, the next runnable task is chose randomly from any of the unassigned leaves of the SSDG.

Great care was taken to ensure a clean separation of the MEMO structure and search tasks from the actual transformations that manipulate plan fragements. The resulting system is easily extensible by adding additional transformations and operator encodings. In particular, the scheduler is oblivious of the type of optimizations implemented by individual tasks and, hence, is able to parallelize all optimizations purely based on their dependencies.

6. EXPERIMENTS

The SSDG is designed to allow implementers to retrofit existing optimizer technology with a parallel scheduler. But just how much parallelism is there? With the following experiments we try to find answers to these questions:

- What are the characteristics of the SSDG?
- How many runnable tasks are there at any given point in time, i.e. how many tasks could be worked on in parallel?
- Does our optimizer achieve sufficient speed-up despite being oblivious of the actual semantics of the optimizations it carries out?

Although our framework is general enough to handle various types of relational operators and associated optimizations, we chose to use only joins and join re-ordering optimizations to obtain results that make for easy comparison with existing literature such as [14]. Extending the optimizer to other operators and studying the effects on the opportunities for parallelization is on our agenda for future work.

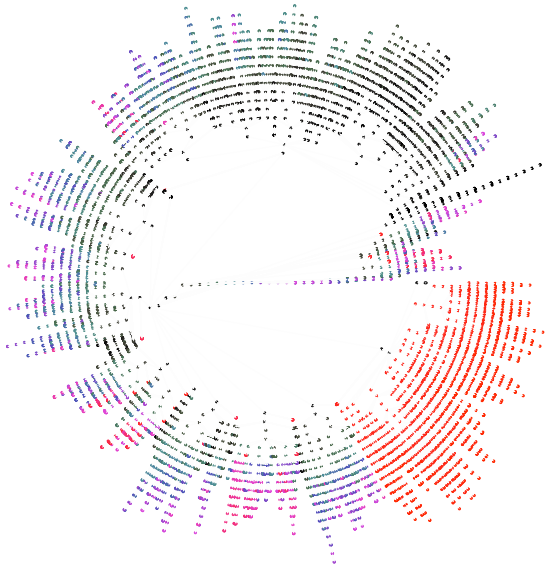

Figure 5: Radial plot of SSDG *(10-way join, star-shaped join graph)*

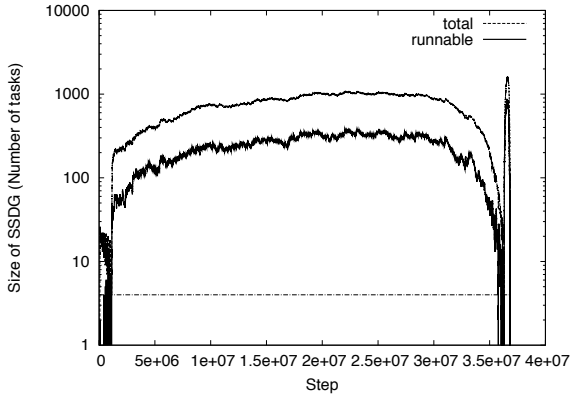

Figure 6: Number of tasks in SSDG as function of time *(10-way join, star-shaped join graph)*

6.1 Preliminaries

All experiments were carried out using a whitebox PC equipped with an Intel Core 2 Quad Core Q6600 processor running Erlang/OTP R12B, the public domain variant of the EVM.

In an initial experiment we demonstrate the scalability of the EVM itself. Given that all Erlang programs are interpreted and indirectly mapped to a thread model inside the EVM we found it useful to quantify the speed-up of a constant workload of several independent Erlang processes. In Figure 4 the run time for a program consisting of 4 processes that compute several thousand list operations each is shown as a function of the number of cores used by the EVM. The dotted line depicts the ideal speed-up. The graph shows that the scheduling overhead of the EVM is generally insignificant. The actual speed-up falls short of the ideal only when operating at maximum CPU capacity, i.e. on all four cores. This is to be expected: when the EVM uses all cores, it contends for at least one of them with background OS processes. This issue recurs in later experiments.

6.2 Characteristics of SSDG

As discussed in Section 4 the SSDG is a tree. Intuitively, a wide and shallow tree offers more parallelism to exploit than one that is a narrow and deep. Figure 5 shows the cumulative version of the SSDG—i.e. the graph including all terminated tasks at the end of an optimization—for a 10-way join defined by a star-shaped join graph. Given the enormous width and low depth of the graph, we chose a radial layout. Although not too many details can be discerned given the scale of the graph, it is apparent that almost all nodes are at a relatively narrow range of depth.

6.3 Opportunity for Parallelism

Next, we investigate the temporal aspect of the opportunity for parallelism. When do parallelizable tasks occur and how much parallelism is there?

To assess the potential more exactly, we compare total number of tasks vs. runnable tasks as a function of time for the same optimization problem, see Figure 6. The lower, bold line is the number of runnable tasks, that is the number of tasks that can be processed in parallel. The additional line (at $y = 4$) indicates the number of tasks needed to saturate all four cores of our experimentation platform.

Initially there is a phase that requires largely sequential processing, i.e. most processes are directly dependent on each other, before more and more runnable tasks become available. This behavior is to be expected as the DP component generates alternatives of increasingly larger semantic differences. During the middle period, there is ample opportunity for parallelism—anywhere between 10 and 50 runnable tasks in the SSDG at a time. The spike at the end of optimization is caused by the inherent sequentiality of finishing the optimization, via search tasks that find and combine optimal solutions for subproblems.

Figure 7 shows the same analysis for a linear join graph of the same size. Since the size of the search space and therefore the total number of tasks is significantly lower the previously discussed effects are magnified. Again, throughout most of the optimization a sufficiently large number of runnable tasks are available at any given point.

6.4 Speed-up of Optimization

In the final set of experiments we study the net effect of using the SSDG for query optimization. For both star-shaped and linear join graphs we optimize n-way joins where n is varied from 2 through 10 and 2 through 16 respectively. For each optimization problem, we took the average of 4 runs and varied the number of cores to be used by the EVM between 1 and 4.

All numbers are normalized to the execution time for 1 core. Figure 8 shows the results for star-shaped join graphs. For the trivially small problem of a single join, we see an initial improvement when going from 1 to 2 cores. It may seem surprising that there is any improvement at all given that there are only two alternatives to be considered. However, a few tasks can be executed in parallel contributing to a speed-up of almost 20%. Adding more cores results in

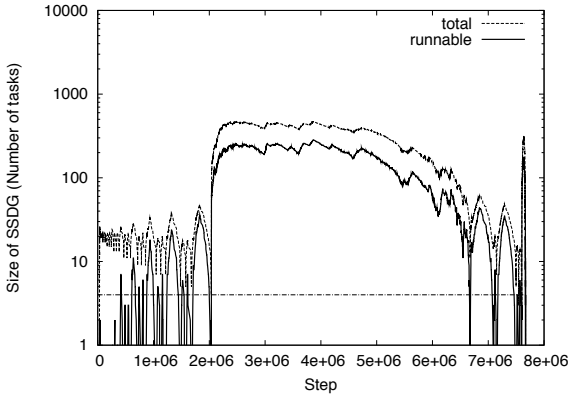

Figure 7: Number of tasks in SSDG as function of time *(10-way join, linear join graph)*

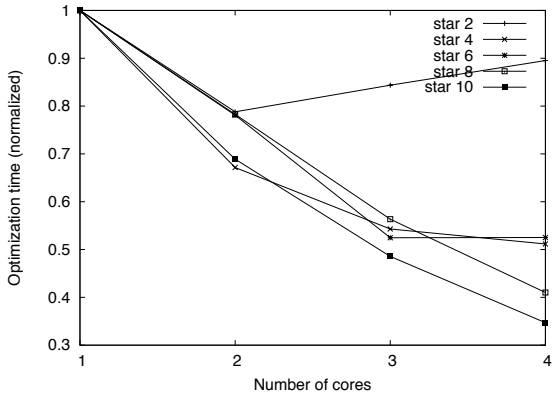

Figure 8: Optimization time as function of number of cores used *(n-way join, star-shaped join graph)*

a slow down for this small problem as the communication overhead of the different processes increases. For larger sizes of the problem we see a clear trend: the larger the problem the better the speed-up. Several factors contribute to this trend. First, the initial and terminal phases of the optimization that offer less parallelism dominate in small problems but diminish relative to the total number of steps for large problems. Second, the individual optimizations may cause contention when operating on closely related areas of the MEMO. With increasing size of the problem this effect becomes less pronounced. While improving with size, even at problem size 10 the optimization time is still slightly above the ideal speed-up with times at 60+% instead of 50% for 2 cores and between 40% and 60% instead of 33% for 3 cores. This is mainly due to (1) strong sequential dependencies at the beginning and the end of optimization and (2) communication and synchronization between tasks that interact with central processes such as the MEMO process.

The results for 4 cores are slightly weaker due to the effects discussed in Section 6.1, i.e. increasing interference of EVM and OS processes.

Figure 9 shows the analogous experiment for queries with linear join graphs. The graph shows a very similar trend. For the most trivial problem size, parallelization is of only limited effectiveness (linear 2). But even for problems of size 4 we see already substantial speed-up. For larger problem sizes the response times improves further. As with the star-shaped join graphs, the results for 4 cores are affected by increasing CPU contention between the EVM and the OS.

What is probably most remarkable about the results is the significance as well as the consistency of the speed-up across different sizes and, hence, widely varied number of tasks underlining the generality of our approach.

7. RELATED WORK

The potential benefits of parallelizing query optimization have long been recognized. In the past the possibility of parallelizing certain search algorithms has been hinted at repeatedly but until recently no concrete attempt has been made to leverage fine-grain parallelism. Especially earlier work on probabilistic and genetic optimization algorithms parallel processing cited parallelism as a way to mitigate the substantial CPU requirements of these algorithms, see e.g. [5].

The first practical approach to we are aware of is by Han *et al.* [14] focusing on SPJ queries. Based on the System R model of optimization authors identify opportunities for breaking down the join ordering problem into independent subproblems and study possible load-balancing strategies. Their approach is explicitly wedded to both (1) a particular type of query and (2) bottom-up enumeration of alternatives. In contrast to this work, our approach is not limited to a specific type of queries or search strategy.

In addition to parallelizing query optimization a number of attempts have been made to improve exhaustive enumeration see e.g. [25, 24].

8. CONCLUSION AND FUTURE WORK

In this paper, we assessed the parallel nature of query optimization. Specifically, we investigated possibilities to retrofit existing optimizer technology to leverage multicore architectures. We introduced the concept of a Search State Dependency Graph (SSDG) to capture dependencies between optimization tasks and identify independent tasks that can be worked on in parallel. Using a scheduler that maintains the SSDG we devised a variant of the widely used Volcano and Cascades optimizer frameworks that achieves substantial and consistent speed-up by utilizing commodity multicore architectures. The main advantages of our approach are its generality and transparency: we simply replaced the existing scheduler of the optimizer framework with a more sophisticated one but did have to come up with new optimization techniques that are parallelism-aware or specifically tailored to facilitate the load-balancing of optimization tasks.

Future Work. The prototype presented in this paper is based on the principles of Volcano as described in [12, 13]. However, the principles of identifying dependencies between optimization tasks appears to be a more general and may

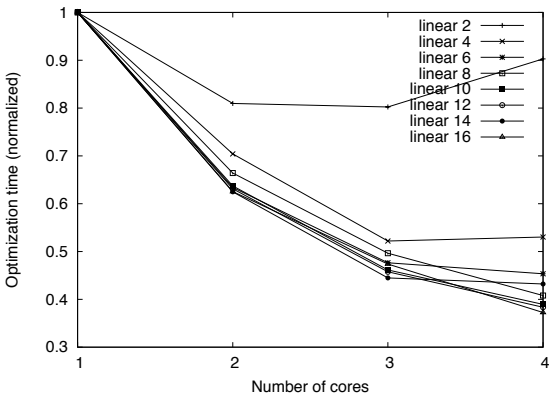

Figure 9: Optimization time as function of number of cores used *(n-way join, linear join graph)*

be applicable to other optimizer frameworks currently used in commercial database systems as well.

An implementation using C++ instead of Erlang is currently underway as part of the Greenplum Database development.

9. REFERENCES

[1] J. Armstrong. Apache vs. Yaws. Technical report, SICS, 2003.

[2] J. Armstrong, R. Virding, C. Wikstrom, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 2^{nd} edition, 1996.

[3] K. Asanovic, *et al.* The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report EECS Department, University of California, Berkeley, Dec 2006.

[4] A. Baras, C. A. Galindo-Legaria, T. Grabs, B. Krishnaswamy, and S. Pal. Optimizing Similar Scalar Subqueries for XML Processing in Microsoft SQL Server. In *Proc. ICDE*, pages 1164–1173, 2007.

[5] K. Bennet, M. Ferris, and Y. E. Ioannidis. A Genetic Algorithm for Database Query Optimization. In *Proc. Genetic Algorithms*, pages 400–407, 1991.

[6] T. Condie, D. Chu, J. M. Hellerstein, and P. Maniatis. Evita Raced: Meta-compilation for Declarative Networks. In *Proc. VLDB*, volume 1, pages 1153–1165, 2008.

[7] J.-C. Freytag. A Rule-Based View of Query Optimization. In *Proc. ACM-SIGMOD*, pages 173–180, 1987.

[8] C. A. Galindo-Legaria and M. Joshi. Orthogonal Optimization of Subqueries and Aggregation. In *Proc. ACM-SIGMOD*, pages 571–581, 2001.

[9] C. A. Galindo-Legaria, M. Joshi, F. Waas, and M.-C. Wu. Statistics on Views. In *Proc. VLDB*, pages 952–962, 2003.

[10] D. Geer. Chip Makers Turn to Multicore Processors. *IEEE Computer*, 28(5), May 2005.

[11] J. Goldstein and P.-A. Larson. Optimizing Queries Using Materialized Views: A practical, scalable solution. In *Proc. ACM-SIGMOD*, pages 331–342, 2001.

[12] G. Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[13] G. Graefe. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[14] W.-S. Han, W. Kwak, J. Lee, G. M. Lohman, and V. Markl. Parallelizing Query Optimization. In *Proc. VLDB*, 2008.

[15] J. M. Hellerstein, M. Stonebraker, and J. Hamilton. Architecture of a Database System. *Foundations and Trends in Databases*, 1(2), 2007.

[16] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[17] N. Kabra and D. J. Dewitt. OPT++: An Object-Oriented Implementation for Extensible Database Query Optimization. *VLDB Journal*, 8:55–78, 1999.

[18] A. Kemper, G. Moerkotte, and K. Peithner. A Blackboard Architecture for Query Optimization in Object Bases. In *Proc. VLDB*, pages 543–554, 1993.

[19] G. M. Lohman. Grammar-like Functional Rules for Representing Query Optimization Alternatives. In *Proc. ACM-SIGMOD*, 1988.

[20] G. Mitchell, U. Dayal, and S. B. Zdonik. Control of an Extensible Query Optimizer: A Planning-Based Approach. In *Proc. VLDB*, pages 517–528, 1993.

[21] A. Pellenkoft, C. A. Galindo-Legaria, and M. L. Kersten. The Complexity of Transformation-Based Join Enumeration. In *Proc. VLDB*, pages 306–315, 1997.

[22] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule-based Query Rewrite Optimization in Starburst. In *Proc. ACM-SIGMOD*, 1992.

[23] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. ACM-SIGMOD*, 1979.

[24] L. D. Shapiro, D. Maier, P. Benninghoff, K. Billings, Y. Fan, K. Hatwal, Q. Wang, Y. Zhang, H.-M. Wu, and B. Vance. Exploiting Upper and Lower Bounds In Top-Down Query Optimization. In *Proc. IDEAS*, pages 20–33, 2001.

[25] B. Vance and D. Maier. Rapid Bushy Join-order Optimization with Cartesian Products. In *Proc. ACM-SIGMOD*, pages 35–46, 1996.

[26] F. Waas and C. A. Galindo-Legaria. Counting, Enumerating, and Sampling of Execution Plans in a Cost-Based Query Optimizer. In *Proc. ACM-SIGMOD*, 2000.

[27] W. P. Yan and P.-A. Larson. Data Reduction Through Early Grouping. In *CASCON*, page 74, 1994.