Optimizing Performance-Polymorphic Declarative Database Queries

Thomas Padron-McCarthy, Tore Risch
Department of Computer and Information Science
Linköping University
S-581 83 Linköping, Sweden
E-mail: {tompa,torri}@ida.liu.se

November 22, 2004

Performance polymorphism, where a system can select between several given implementations of the same conceptual operation, has been used in real-time programming languages, such as Flex. A related but more limited mechanism has been used in active database systems, such as HiPAC. We have introduced performance polymorphism into a database query language. We show how performance-polymorphic queries are specified and optimized in our system. We have shown the feasibility of the concept by implementing a general, performance-polymorphic query optimizer.

1 Introduction

1.1 Real-time databases

A *real-time system* is a system where the operations not only have correctness requirements, but also requirements on timeliness. For example, a task needs to be completed before a given deadline. Real-time systems are not restricted to applications where the time spans in question are short (fractions of a second), although many real-time applications fall in this domain, but also include applications with a longer time range (several seconds or much longer) [27].

A *real-time database management system* (RTDBMS) [21] is a DBMS that meets timeliness requirements, in addition to the traditional DBMS functionality. Alternatively, it can be defined as a real-time system that includes database capabilities, such as transaction management, index structures, and query capabilities. Despite the apparent similarity, in practice these two definitions are not co-extential. The approach of starting with the DBMS concept, and adding real-time functionality, tends to put emphasis on database facilities, such as transaction processing and defining a declarative query language as interface to

the database, while the other approach favors a programming-language interface, typically from C++.

1.2 Performance polymorphism

In time-critical applications it is sometimes possible to find a simplified algorithm that can be used when there isn't enough time to run the normal algorithm. This simplified algorithm performs the same conceptual operation as the normal one, but in shorter time. The trade-off is that the result may be of a lower quality with respect to precision, completeness, or data consistency.

For example, such different algorithms can be used in a control application that reads input from a slow physical sensor. If the control application doesn't always have time to wait for the next sensor reading, it could instead use an extrapolation of previous values. This will produce a value within the allowed time-frame, but the value may deviate from the actual physical value.

Another example is a numerical computation that may be set to produce results with different precisions depending on the execution time spent on the computation. Thus we have a trade-off between time and precision, and this trade-off can be used in a time-critical application to find an acceptable result within the allowed time-span, instead of a more exact result that arrives too late.

The different implementations can be defined by the programmer, and under some circumstances it is possible for the system to find them automatically [5] [18] [17].

The term *performance polymorphism* refers to "a scheme were all the versions of a particular computation are identified as candidates for binding to a generic name. We call this technique *performance polymorphism* by analogy to the conventional polymorphism where different functions of the same name operate on different data types" [8]. A similar definition is given in [29].

The essence of the definition is that the system must be able to automatically select among several different given implementations, and that it should find the best, or at least an acceptable, trade-off between the different performance measures. This can be done either at run-time, or at compile-time. A mixture is also possible, where the selections that can be made early are made early, and the rest is deferred to run-time [8].

While these definitions seem to be best suited for use in programming languages or in object-oriented databases, with named functions or methods, the concept could be extended to cover e. g. contingency plans in the ECA-rules of an active database system [2], and the query partitioning in CASE-DB [17]. However, in order to naturally capture this and other forms of polymorphism, an object-oriented data model is advantageous. Several real-time object-oriented data models have been defined, e. g. RTSORAC [20]. Some real-time systems combine performance polymorphism and object orientation, such as the Flex programming language [10] and the ROMPP data model [29].

1.3 Performance-polymorphic queries

By a *query* we mean an operation against a database that is formulated in a declarative query language. This declarative query cannot be executed directly, so a *query optimizer* is required to translate the query into an executable, procedural program, the *execution plan*.

Since there are usually many possible execution plans for a given declarative query, and since these plans can have widely varying performance, it is important that the query optimizer finds a good (i. e. cheap) plan, ideally the best. A traditional database query optimizer works with a single performance measure, the "cost", which usually reflects the expected execution time [23], dominated by the number of disk accesses. The optimizer uses a *cost model* to estimate the cost of the execution plans.

By a *performance-polymorphic query* we mean a query, that is formulated in a declarative query language, and that involves operations that may exist in several implementations with different performance.

A *performance-polymorphic query optimizer* is, in addition to the functionality of a traditional query optimizer, required to choose between the different implementations of each performance-polymorphic operation.

To the best of our knowledge, all previous work concerning object-oriented performance polymorphism where it is possible for the user to define multiple versions of a function or method with different performance, has concentrated on providing a programming-language interface. No declarative query language, and therefore no query optimization, has been provided.

While a programming-language interface may be sufficient for many applications, there are important advantages with a declarative query language, such as a simpler interface, increased data independence, and the possibility for better optimization than for hand-coded procedural programs, especially for large amounts of data and non-trivial schemas. Since the system includes a cost model for the optimizer, it can use this to automatically estimate the execution time.

Since query optimization is a potentially time-consuming task, it is usually important for a real-time database with a declarative query language to do the optimization at an early time, so the optimization time does not have to be included in the time constraints at execution time. Even if there may exist applications where this is tolerable, for example when the ranges of the time in the time constraints are very large (minutes), optimization should be done early, if possible.

2 Related Work

An early example of a mechanism where different implementations of an operation can be defined by the user, and then automatically chosen by the system, is the contingency plans for alternative executions of reactive rules in the active DBMS **HiPAC** [2], also discussed in some later publications, e. g. [1]. In the

ECA rules described in the HiPAC project, more than one version of the action part could be specified. When the time constraints could not be met by the normal action of a rule, the system could instead chose an alternative, the contingency plan. While HiPAC provided a declarative query language, the use of different-performance implementations was limited to the ECA rules, and could not be used in other parts of the system. Contingency plans do not fall under the definition of performance polymorphism, but it is a related concept.

CASE-DB [5] [18] [17] is a real-time relational prototype DBMS that permits the specification of time constraints for queries expressed in relational algebra. Given a deadline, the system can automatically partition a query, and then does iterative improvement using these partitions, while handling the risk of over-spending its time budget. For non-aggregate queries this requires a previous, user-defined partitioning of the data. CASE-DB has no user-declared performance polymorphism, i. e. it is not possible to define multiple implementations of operations. The quality of the answer, and trade-offs between time and quality, is not discussed.

As mentioned above, an object-oriented data model is advantageous for expressing performance polymorphism. Several real-time object-oriented data models have been defined, e. g. **RTSORAC** [20].

Other real-time systems combine object orientation with a more explicit performance polymorphism, such as the Flex programming language [11] [8] [10] [15] [7] [9] in the Concord project [12] and the ROMPP data model [29] in the MDARTS project [16].

Flex is an experimental programming language based on C++, which has been extended with real-time functionality and with performance polymorphism. In Flex it is possible to define several implementations of the same function, with different timing measures, and with different figures of merit. The system will, at run-time, chose an implementation that fits within the given time constraint. If several implementations are feasible, the one with the highest figure of merit is chosen. By performing normal compiler optimizations on the code for choice of implementation, some of the binding decisions can be done at compile-time.

[8] introduces the term performance polymorphism, and contains a discussion of the concept and how to implement it.

Since Flex isn't a declarative query language, it does not do any query optimization. However, the "allocation problem" is discussed: when more than one performance-polymorphic functions is to be executed, more than one choice has to be made, and the trade-off between them considered.

ROMPP uses a similar solution, but from a database approach with a schema describing the data. ROMPP has a mechanism of envelope and letter classes to handle performance polymorphism in several specialization dimensions, not just (or even necessarily) time. [29] mentions "value propagation" of the specialization dimensions, i. e. the combination of performance characteristics, as an "open question to be answered". ROMPP does not provide a declarative query language, and thus no optimization.

[29] also contains an overview of previous work on performance polymor-

phism.

CHAOS [22] is a system for developing and executing real-time applications. CHAOS has support for different implementations of an operation, and for configuring and re-configuring an application by replacing these implementations. While the re-configuration can be done "dynamically" at run time, the system cannot do this automatically.

3 Our implementation

We have implemented a performance-polymorphic query optimizer within our research platform AMOS [14] [25] [26] [19] which is a main-memory object-oriented active DBMS, with a relationally complete, object-oriented query language. The optimizer uses dynamic programming [23], which has been modified to handle operations that are polymorphic in any number of user-defined performance dimensions, e. g. time, precision, quality. The performance dimensions can have both numeric and symbolic values. Constraints can be stated on all performance dimensions, and any one of these can be used as the optimization objective.

For each performance dimension, the user specifies

- its name,
- its starting value, which is used as the value of this performance dimension for an empty execution plan, i. e. a partial plan to which no operations have been added yet,
- a default value, which is used when the value of this performance dimension for a certain operation is not specified,
- a combination function that combines the performance values of two operations, to be used when more steps are added to a partial execution plan,
- a comparison function that determines which of two values of this performance dimension is better,
- a switch indicating if the value of this performance dimension is monotonically worsening as the incomplete plan grows by adding operations to it, if it is improving, or if its monotonicity is unknown.

As an example, the performance measure **TIME** will typically have the starting value 0, no default value, a combination function that numerically adds two values, and will use the function **LESS-THAN** as a comparison-function. Since an execution plan can never be made faster by adding operations, the value is monotonically worsening.

A typical use, in the context of a real-time system, is to define one performance measure called **TIME**, which expresses either the expected or the worst-case actual execution time, and another measure called **QUALITY**, which

expresses the quality or "goodness" of the result. The optimizer is then typically required to either chose the execution plan that gives a result with the highest possible quality within some given time limit, or to chose the execution plan with the fastest execution time, given some minimum quality.

Since any number of performance measures can be defined, nothing prevents us from defining several different time measures, for example average time (which can be minimized by the optimizer) and worst-case time (which we might put constraints on in a real-time system).

During optimization, the space of possible execution plans is investigated by building a search tree using best-first search with respect to the performance measure that is used as objective. Each node in this search tree represents an execution plan for the declarative query that is being optimized. The leaf nodes represent complete, executable plans, while the internal nodes of the tree represent partial plans.

During the search, each plan is examined with respect to the stated constraints on all performance measures. If a constraint is overrun, the sub-tree rooted in that plan can be pruned when either (1) the plan is a complete plan, or (2) the plan is a partial plan and the constraint in question is monotonically worsening. In those cases we know that the constraint is irrevocably broken.

4 An example

In this example we will show how the AMOS system translates a declarative query into an unoptimized execution plan, how the possible execution plans can be represented as a search tree, and how the query optimizer finds an optimal execution plan by partially constructing, and traversing, this tree.

A mobile telephone network consists of a number of base stations, each covering an certain area, and a number of mobile telephones. At all times, each base station needs to know which of the mobile telephones are present in the area it covers.

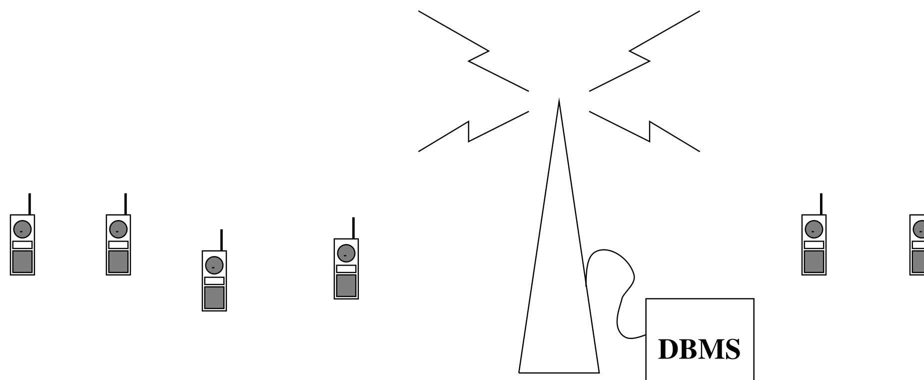

Figure 1: A cell in a mobile telephone network

We assume that the base station has the ability to find a certain mobile telephone by sending out a radio message that the telephone responds to, if that telephone is present in the area. We call this operation `present`.

We also assume that the base station can use a different operation, `signal_strength`, to determine the strength of the signal received from the telephone.

In this scenario, it can be useful to have multiple implementations of both these operations. For example, it will often be enough to know that a telephone was present in the area some time ago, and thus a previously stored value can be used, but at other times it will be necessary to be more certain, which requires actually sending a radio message to the mobile telephone to receive a reply. This is expensive from a battery consumption and frequency utilization point of view, in addition to the time required.

Therefore we assume that the conceptual operation `present` has been implemented in three different ways, each with a different performance measure for time ($\mathbf{t}$) and quality ($\mathbf{q}$):

- `p1`: the procedure `was_present` gets the previously stored value ($\mathbf{t} = 0$, $\mathbf{q} = 0.2$)
- `p2`: `search_once` sends one radio message ($\mathbf{t} = 0.2$, $\mathbf{q} = 0.6$)
- `p3`: `search_many` sends several radio messages ($\mathbf{t} = 3.0$, $\mathbf{q} = 0.99$)

We also assume that the conceptual operation `signal_strength` has been implemented in two different ways:

- `s1`: `old_signal_strength` gets the previously stored value ($\mathbf{t} = 0$, $\mathbf{q} = 0.2$)
- `s2`: `measure_signal_strength` measures the actual signal strength by sending a radio message and measuring the reply ($\mathbf{t} = 0.3$, $\mathbf{q} = 0.9$)

In this example, radio communication is very slow in comparison with internal data lookup and calculations. We can therefore assume that internal operations take time 0.

The quality measure $\mathbf{q}$ that is used here varies between 0 (lowest quality) and 1 (highest quality), and is combined using the function **MIN**. The starting value and the default value are both 1.

We also assume that the number of telephones in the database is 100, and that previous values of `present` and `signal_strength` have been stored for 10 of these (for use by `p1` and `s1`).

As an example query, we need to find which of the mobile telephones that are present in the area but have a signal strength less then 25. Assuming that the data type `telephone` and the performance-polymorphic functions `present` and `signal_strength` have been defined, this query can be formulated using AMOS' query language, AMOSQL:

```
select p
for each telephone p
where present(p)
and signal_strength(p) < 25
with t better than 2.0
optimize q;
```

We want the query to be executed in at most 2 seconds, with the best possible quality within that time limit. We have therefore defined a constraint `t <= 2.0`. We have also declared the quality **q** as the optimization objective. This means that the optimizer will attempt to find the execution plan with the best quality that has an estimated execution time of less than 2 seconds.

The AMOSQL query is compiled into a domain calculus language called ObjectLog [14], which is a variant of Datalog where facts and Horn Clauses are augmented with type signatures:

$$
\begin{array}{l}
\texttt{answer}_{telephone}\texttt{(P) :-} \\
\quad \texttt{signal_strength}_{telephone,integer}\texttt{(P, _G1) \&} \\
\quad \texttt{present}_{telephone}\texttt{(P) \&} \\
\quad \texttt{<}_{object,object}\texttt{(_G1, 25).}
\end{array}
$$

The functions in ObjectLog are not only performance-polymorphic, but also both type-polymorphic in the regular way (notice the subscripts) and *binding-polymorphic* (with different implementations depending on whether the arguments are bound or free).

This representation of the query is still declarative and not directly executable, since the order among the predicates and the bindings are not determined. The predicates will be re-ordered by the optimizer, and the polymorphic functions will be resolved. The result, the execution plan, will be run as a nested-loop join.

The query optimizer starts by creating an empty execution plan, which is used as root of the search tree (figure 2). The space of possible execution plans will then be traversed as the tree is constructed. In the figure, **plan** denotes the list of functions in the partial plan, **t** denotes the estimated execution time for the partial plan, and **q** denotes its quality. **Fanout** is the expected number of objects in the result of an operation when the plan is executed. The fanout is operator-dependent, and fanout is actually treated as yet another performance dimension in our system.

plan = ( ), t = 0, q = 1, fanout = 1

Figure 2: The root of the search tree

The optimizer has six choices as the first step in the execution plan: the operation `<` (less-than), the three implementations of `present`, and the two implementations of `signal_strength`. So far, no parameters are bound, and

the operation `<` cannot be executed with two unbound parameters. All the implementations of `present` and `signal_strength` can be executed.

For each of the five partial plans that result from adding one of these operations to the empty plan, **t**, **q** and **fanout** are calculated by looking up the specified values of these performance dimensions for the operation, and calling the combination function for that performance dimension. Of the five plans, all but two will break the time constraint `t <= 2.0`. Those that don't are added to the search tree, which then consists of two partial plans (figure 3).

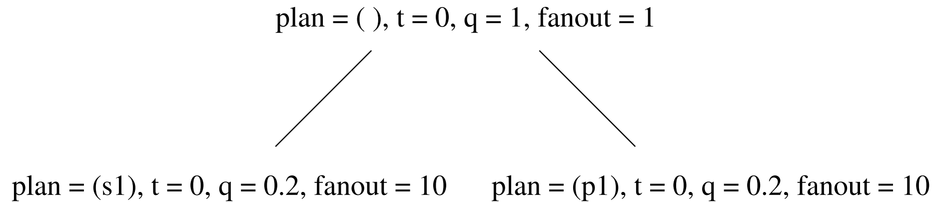

Figure 3: The search tree after the first iteration

In the next iteration we continue building on the partial plan with the best quality **q**, since this is the optimization objective. In this case both plans have the same quality, so it doesn't matter which one is chosen.

We take the plan that consists of the operation `s1`, and expand it. It already contains an implementation of `signal_strength`, so we can expand it with either the operation `<` (less-than) or one of the three implementations of `present`. `p3` would break the time constraint `t < 2`, so it is not used. The three new partial plans are added to the search tree (figure 4).

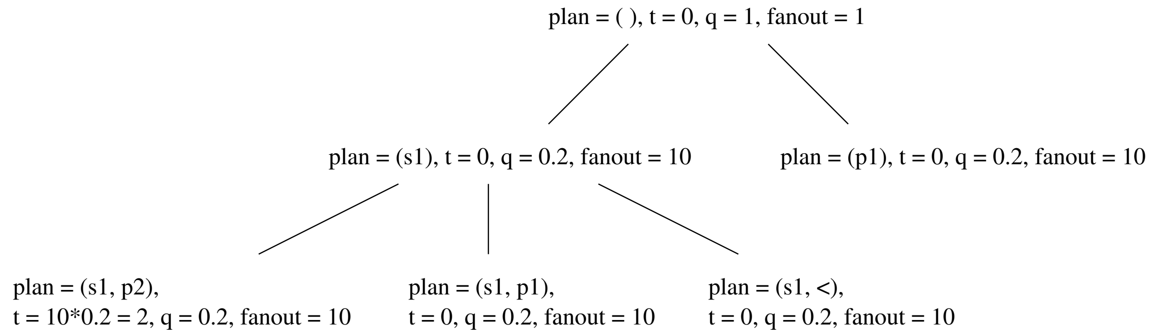

Figure 4: The search tree after the second iteration

In the next iteration, the partial plan `(p1, s1)` is added (figure 5).

In the iteration after that, the partial plan `(s1, p2)` is expanded to `(s1, p2, <)`, which is a complete execution plan (figure 6).

Since we have done a best-first search with respect to the quality measure **q**, and we know that **q** is monotonically worsening as the plan grows, we can

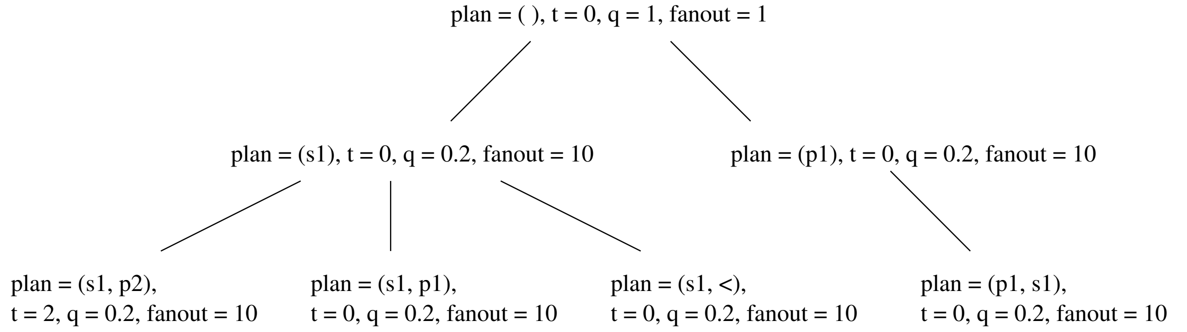

Figure 5: The search tree after the third iteration

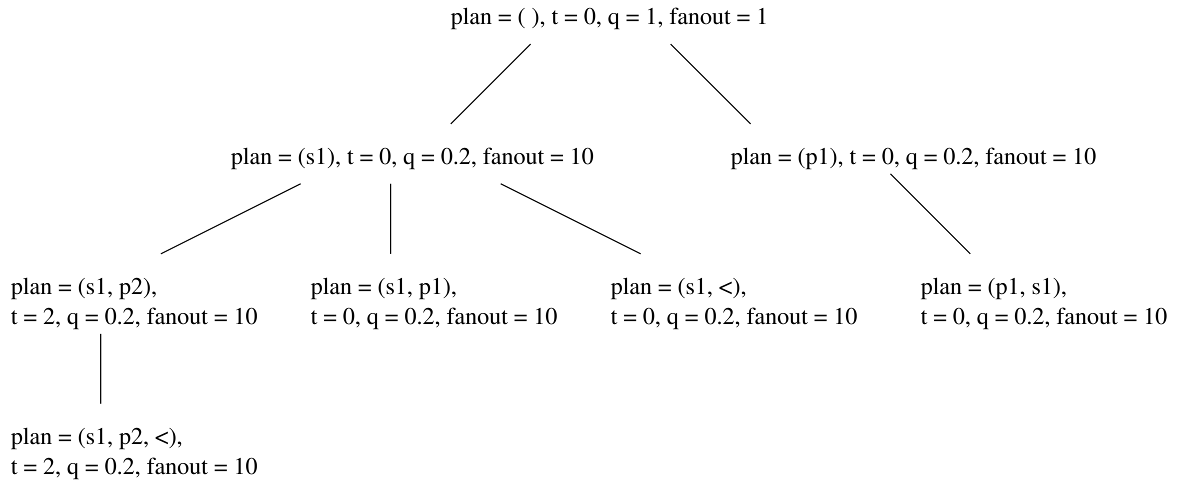

Figure 6: The search tree after the fourth iteration

return this result. None of the partial plans can be extended to a complete plan with better quality than this plan.

If we instead use a quality measure with a value that can improve as the plan grows, the algorithm will continue expanding the partial plans in the search tree even after finding this plan. This is a difference from the standard dynamic programming algorithm [23].

A possible simplification to our algorithm is to return the first found complete plan no matter what the monotonicity of the optimization objective. This plan is within the time limit, and with some probability it does also have a good quality compared to the other plans (because of the best-first search).

5 Further work

The optimization of a declarative query is a combinatorical problem, and performance polymorphism adds additional complexity. n predicates can be ordered in $n!$ different orders, and for each of these predicates that exists in several performance-polymorphic versions, the expression $n!$ has to be multiplied by the number of different versions of that predicate.

Since optimization is a hard problem in this sense, and the optimizer itself in its present implementation is not time-constrained, query optimization and the resolving of performance-polymorphic predicate implementations is expected to be done early, at query compile time. In some cases, however, late binding is advantageous, [4] and then strategies are needed to estimate the performance of late bound performance-polymorphic function calls. The query optimizer should automatically choose early binding when possible. When late binding cannot be avoided the system can optimize each resolvent separately and then estimate the time to execute the performance-polymorphic call in terms of the actual time to execute its resolvents. Such partial evaluation of the estimates of cost and quality would minimize the amount of work that has to be done at run time.

For complex queries, the exhaustive search done by dynamic programming will not be feasible. Therefore, alternative optimization algorithms should be investigated, such as randomized and heuristic algorithms. Among the candidates are hill-climbing with multiple random starting points and simulated annealing [6].

The present work leaves the choice of a quality measure, or several quality measures, to the application implementer. However, different quality measures can be studied. Among these are the rules of fuzzy logic [28] [3], or some ad-hoc measure, like the certainty factors of EMYCIN [24].

Timing estimates should be developed for the internal operations on data, including operations, such as lookup and insertion, on the DBMS's internal data structures. These data structures should be modified for an improved and well-analyzed worst-case behavior.

The cost model that is used to find timing estimates should be verified against actual, measured execution times [10] [13].

Even if we in this work concentrate on the real-time aspect of performance polymorphism, where a trade-off in quality is made in order to get the operation done within a time limit, performance polymorphism is not limited to real-time applications. If we e. g. are searching for information from sources on the Internet, with different monetary costs and data quality, this can be modeled using performance polymorphism, and a performance-polymorphic query optimizer can be used to find an acceptable trade-off between monetary cost and data quality. In the general case, the implementations of the performance-polymorphic operations are specified along any number of performance dimensions.

6 Conclusions

We have defined the concept of performance-polymorphic queries, and compared it to other similar approaches.

We have extended a query language with performance-polymorphic queries.

We have developed a performance-polymorphic query optimizer based on extensions to an object-oriented query optimizer.

We have showed the steps of the performance-polymorphic query optimizer through a telecom example.

Performance-polymorphic declarative queries can be optimized and executed by a real-time DBMS, as demonstrated by our optimizer. The technique is general and can also be used outside the real-time domain, since any number of performance measures can be defined and handled by the optimizer.

References

[1] H. Branding and A. Buchmann. On providing soft and hard real-time capabilities in an active DBMS. In *International Workshop on Active and Real-Time Database Systems*, Sweden, June 1995. University of Skovde, Sweden.

[2] U. Dayal et al. The HiPAC project: Combining active databases and timing constraints. *SIGMOD Record*, 17(1):51–70, March 1988.

[3] R. Fagin. Combining Fuzzy Information from Multiple Systems. In *Proceedings of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS 1996*, pages 216–226, Montréal, Canada, June 1996. ACM Press.

[4] S. Flodin and T. Risch. Processing Object-Oriented Queries with Invertible Late Bound Functions. In *Proceedings of VLDB-95*, 1995.

[5] W.-C. Hou, G. Özsoyoğlu, and B. K. Taneja. Processing aggregate relational queries with hard time constraints. In *Proc. of ACM SIGMOD Conf. 1989*, pages 68–77, Portland, Oregon, May 1989.

[6] Y. E. Ioannidis and Y. C. Kang. Randomized algorithms for optimizing large join queries. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 312–321, Atlantic City, NJ, May 1990.

[7] K. Kenny and K.-J-Lin. Measuring and Analyzing Real-Time Performance. *IEEE Software*, 8(5):41–49, September 1991.

[8] K. Kenny and K. J. Lin. Structuring large real-time systems with performance polymorphism. In *Proc. 11th IEEE Real-Time Systems Symp.*, pages 238–246, Orlando, FL, December 1990.

[9] K. Kenny and K.-J. Lin. Implementing and Checking Timing Constraints in Real-Time Programs. *Microprocessing and Microprogramming*, (38):477–484, 1993.

[10] K. B. Kenny and K.-J. Lin. Building Flexible Real-Time Systems Using the Flex Language. *Computer*, 24(5):70–78, May 1991.

[11] K.-J. Lin and S. Natarajan. Expressing and Maintaining Timing Constraints in FLEX. In *Proceedings of the 9th IEEE Real-time Systems Symposium*, pages 96–105, Los Alamitos, CA, July 1988. IEEE Computer Society Press.

[12] K. J. Lin, S. Natarajan, J. W. S. Liu, and T. Krauskopf. Concord: A system of imprecise computations. In *Proc. 1987 IEEE Compsac*, October 1987. Japan.

[13] S. Listgarten and M.-A. Neimat. Modelling Costs for a MM-DBMS. In *Proceedings of the First International Workshop on Real-Time Databases*, pages 77–83, Newport Beach, CA, USA, March 1996.

[14] W. Litwin and T. Risch. Main memory oriented optimization of OO queries using typed datalog with foreign predicates. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):517–528, December 1992.

[15] J. W. S. Liu, K.-J. Lin, W.-K. Shih, A. Chuang shi Yu, J.-Y. Chung, and W. Zhao. Algorithms for Scheduling Imprecise Computations. *Computer*, 24(5):58–68, May 1991.

[16] V. Lortz. An object-oriented real-time database system for multiprocessors. Technical Report CSE-TR-210-94, University of Michigan, April 1994.

[17] G. Özsoyoğlu, S. Guruswamy, Kaizheng Du, and Wen-Chi Hou. Time-constrained query processing in CASE-DB. *IEEE Transactions on Knowledge and Data Engineering*, 7(6):865–884, December 1995.

[18] G. Özsoyoğlu, K. Du, S. Guru swamy, and W.-C. Hou. Processing real-time, non-aggregate queries with time-constraints in CASE-DB. In F. Golshani, editor, *Proceedings of the International Conference on Data Engineering*, volume 8, pages 410–417, Los Alamitos, CA, February 1992. IEEE Computer Society Press.

[19] T. Padron-McCarthy and T. Risch. Performance-Polymorphic Execution of Real-Time Queries. In *Proceedings of the First International Workshop on Real-Time Databases*, pages 50–53, Newport Beach, CA, USA, March 1996.

[20] J. J. Prichard, L. C. DiPippo, J. Peckham, and V. F. Wolfe. RTSORAC: A real-time object-oriented database model. *Lecture Notes in Computer Science*, 856:601–610, 1994.

[21] K. Ramamritham. Real-time databases. *Distributed and Parallel Databases*, 1(2):199–226, April 1993.

[22] K. Schwan, P. Gopinath, and W. Bo. CHAOS – Kernel support for objects in the real-time domain. *IEEE Transactions on Computers*, 36(8):904–916, August 1987.

[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. of SIGMOD Conf. 1979*, pages 23–34, Boston, MA, 1979.

[24] E. H. Shortliffe. *Computer-Based Medical Consultations: MYCIN*. Elsevier/North-Holland, Amsterdam, London, New York, 1976.

[25] M. Sköld, E. Falkenroth, and T. Risch. Rule Contexts in Active Databases — A Mechanism for Dynamic Rule Grouping. In *RIDS'95 (Rules in Database Systems)*, Athens, Greece, September 1995.

[26] M. Sköld and T. Risch. Using partial differencing for efficient monitoring of deferred complex rule conditions. In *Proceedings of the 12th International Conference on Data Engineering*, pages 392–401, Washington - Brussels - Tokyo, February 1996. IEEE Computer Society.

[27] J. A. Stankovic. Misconceptions About Real-Time Computing. A Seriuos Problem for Next-Generation Systems. *Computer*, 21(10):10–19, October 1988.

[28] L. A. Zadeh. Fuzzy Sets. *Information and Control*, 8(3):338–353, June 1965.

[29] L. Zhou, E. A. Rundensteiner, and K. G. Shin. Schema evolution for real-time object-oriented databases. Technical Report CSE-TR-199-94, University of Michigan, March 1994.