January 1988

Partial Computation in Real-Time Database Systems

Susan B. Davidson
University of Pennsylvania, susan@cis.upenn.edu

Aaron Watters
University of Pennsylvania

Partial Computation in Real-Time Database Systems

Abstract
A critical component of real-time systems in the database, which is used to store external input such as environmental readings from sensors, as well as system information. Typically these databases are large, due to vast quantities of historical data, and are distributed, due to the distributed topology of the devices controlling the application. Hence, sophisticated database management systems are needed. However, most of the time database systems are hand-coded. Off-the-shelf database management systems are not used due in part to a lack of predictability of response [1, 2]. We motivate the use of partial computation of database queries as a method of improving the fault-tolerance and predictability of response in real-time database systems.

Partial Computation in Real-Time Database Systems

Susan B. Davidson and Aaron Watters
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

Abstract

A critical component of real-time systems is the database, which is used to store external input such as environmental readings from sensors, as well as system information. Typically, these databases are large, due to vast quantities of historical data, and are distributed, due to the distributed topology of the devices controlling the application. Hence, sophisticated database management systems are needed. However, most of the time the databases systems are hand-coded. Off-the-shelf database management systems are not used due in part to a lack of predictability of response [1, 2]. We motivate the use of partial computation of database queries as a method of improving the fault-tolerance and predictability of response in real-time database systems.

1 Introduction and Motivation

Real-time systems define correctness as providing the correct result in a timely manner. If the timing requirements for a computation cannot be met, the computation fails. To relax this definition of correctness and somehow *tolerate* the failure of meeting a deadline, one must either be willing to accept the results of computation late, or be willing to accept partial, poorer quality results in a timely manner. The first strategy interprets timing constraints as being "soft": the completion of a computation or set of computations has a value to the system which is expressed as a function of time. The system schedules these computations to maximize the total value to the system; however, it does not guarantee that all computations will be performed at their local maximum value [3]. The second strategy requires the computations to have an iterative or multi-phase nature. Furthermore, they should be *monotonic* in the sense that "goodness" of the answer is monotonically non-decreasing as the computation proceeds [4, 5]. An example of such a computation is the bisection method for finding a root of a function: The interval containing a root is initially very large, and keeps halving as the computation proceeds. At any point in the computation, the interval is valid; however, it is best defined when the endpoints of the interval converge to a single point, the root of the function.

While "soft" timing constraints have recently been proposed for transaction management in real-time database systems [6], little work has been done on generating *partial* or iterative answers to queries. However, we feel that the ability to give such answers is extremely important since the expected execution time of a query is very difficult to predict. In distributed databases (which is typically how real-time databases are structured), the unknown size of the relations to be transmitted makes it difficult to bound communication time; long-lived communication failures can also cause indeterminate delays. Indeterminate delays can also result from locking due to concurrency control in the local (centralized) databases. Unfortunately, in traditional databases systems, unless *all* structures necessary to answer the query are accessible, there is no notion of an answer. Our notion of partial computation for queries is therefore a "best-estimate" of the final answer, based on *the structures that are currently available.* Our interpretation of "monotonicity" is that any fact which is said to be true remains true as computation proceeds, and any fact which is said to be false remains false as computation proceeds.

As an example of how a partial answer to a database query could be useful, suppose that we have a distributed sytem of three blood bank databases. Each blood bank database maintains, among other information, a relation of how many pints of each blood type is currently on hand. Suppose that type O- is dangerously low at hospital X, and X is trying to find out if there is any available within the network of blood banks to meet a current crisis. This query could be expressed as: "Is there a blood bank that has blood of type O-?", Thus, the query "Do you have blood of type O- ?" would be broadcast to each of the three databases, and the final answer would be the logical "or" of the responses from each of the databases. The initial partial anwer to the query would be "I don't know yet." This partial answer can be changed to "Yes" immediately some blood bank responds with a "Yes", regardless of whether all blood banks have responded. The answer becomes "No" only when all of the blood banks have responded negatively. However, at any point in time, there is some answer to the query that is correct. If hospital X then wanted to know "How much blood of type O- is there in the system?", the initial partial answer would be "At least 0", and would be improved by adding the total amount from each database as the information became available. For instance, if the first blood bank responded with "10 pints", the answer would be improved to "At least 10 pints." If the second and third blood banks responded with 5 and 25 pints respectively, the answer would become "Exactly 40 pints."

This example illustrates several points:

- It is an example of a real-time process in which the response must be predictable: Hospital X cannot wait indefinitely for an answer from the blood banks since in the worst case that there is no O- blood it must start rounding up donors to cover any anticipated crisis. Note,

however, that the timing constraint is probably minutes (or hours, depending on how nervous hospital X is) rather than milliseconds.

- A "hand-coded" query system which anticipates this type of query probably would act as in the example, while a strict relational algebra system would not be optimized to give partial information.

- The "goodness" of the answer given to the user is monotonically non-decreasing with time. Given a partial answer "At least 10 pints." the user can infer that the total is *definitely not* less than 10 pints, and *possibly* any integer greater than or equal to 10 pints (11 or 1198, for example). Furthermore, the answer given at an earlier stage is never contradicted at a later stage.

2 Partial queries

The reason why conventional query languages (and the relational algebra in particular) do not seem to be amenable to an iterative method is that the relationship of individual relations (or whatever structure is used in the model) to the final result is not explicit. For example, in relational databases, a query $f(R_1, R_2, R_3, ..., R_n)$ can be thought of as some combination of the relations R_1,..., R_n using relational algebra operators. For simple expressions involving one binary operator, the relationship of relations R_1, R_2 to the final result is not difficult to reason about. If $f(R_1, R_2) = R_1 \cup R_2$, it is obvious that R_1 and R_2 each contain a part of the answer, although, in general, neither will contain the whole answer: R_1 and R_2 can be thought of as *consistent approximations* to the final result. If $f(R_1, R_2) = R_1 \bowtie R_2$, then every tuple in the join is contained in both R_1 and R_2 , but each relation may contain other tuples as well that do not participate in the join. Both can be thought of as *complete approximations* to the final result. Note in this case that a tuple of R_1 participating in the join with R_2 is a *partial description* of the tuple in the result since it may not contain all the fields in $R_1 \bowtie R_2$. However, for more complicated expressions like $f(R_1, R_2, R_3) = R_1 \bowtie (R_2 \cup R_3)$, it is difficult to reason about the information in R_2 and R_3 with respect to the final result.

Using the semantic notions of complete and consistent approximations, we have recently presented a method of iteratively combining structures as they become available [7, 8]. That is, the user first specifies the semantic relationship of the answer to the query to the individual structures in the database. The system then combines the approximations as structures become available in such a way that a partial answer is *always* available. The partial result is represented at any point in the computation by a *bounding pair* (A, B), where A is a complete approximation of the final result and B is a consistent approximation of the final result. From the set A, the user can infer

tuples that are definitely *not* in the answer to the query; from the set B the user can infer tuples that definitely *are* part of the answer. Given two bounding pairs for a query, (A_1, B_1), (A_2, B_2), we combine them into another bounding pair (A, B) where A is no "larger" a complete approximation than A_1 or A_2, and B is "at least as large" a consistent approximation as B_1 and B_2. That is, the new bounding pair is a better approximation of the final result since it squeezes the complete and consistent approximations closer together. This continues until there are no more bounding pairs to encorporate, or until A and B describe the same set of objects, *i.e.*, the answer is completely determined. Furthermore, the partial answer can be shown to improve monotonically.

Expanding on the blood bank example, suppose that hospital X maintained a relation $NEAR-BANKS(Code, Name, Phone)$ of blood banks in its area, and that a central authority on blood banks maintained a relation $SOURCES(Code, Address, Phone...)$ (to which hospital X had access). Furthermore, assume that the required computation from the each of local blood bank databases was represented by $AMOUNT_i(Code, Quantity)$, and that the query was "Give me the *Name*, *Phone* and *Address* of all local blood banks, as well as the *Quantity* of O- on hand." Then $NEAR-BANKS$ and $SOURCES$ are each complete approximations of the final result, and $AMOUNT_1$, $AMOUNT_2$ and $AMOUNT_3$ are each consistent approximations of the final result. The initial bounding pairs would be: $(A, \{\})$ for each complete approximation A, and $(\perp, B)$ for each consistent approximation B (where $\perp$ is a special set that underdescribes any set). If, for some reason, relations $AMOUNT_2$ and $AMOUNT_3$ could not be obtained by the deadline, a partial answer of $(SOURCES \bowtie NEAR-BANKS, AMOUNT_1)$ could be constructed. That is, the complete approximation of the final result would be a relation $TEMP(Code, Address, Phone...)$, where only the local blood banks would appear. If the first blood bank could supply enough O- blood, the answer would be sufficient; however, even if it couldn't, the complete approximation would at least give the phone numbers of the remaining two blood banks so hospital X could make a phone call to determine the amount on hand. That is, the partial answer may be useful.

The system has several advantages: (1) it is not tied in to any data model in particular (although the example given was relational in flavor); (2) it detects anomalies in the database, which can arise either due to incorrect semantic understanding of the structures in the database, or due to errors contained in the database; and (3) the deadline of a query can be met by providing a partial answer. The first advantage is especially important in real-time databases since many of the structures that need to be stored do not correspond to first-normal form relations, or accepted structures in other database models (*e.g.*, system information such as stacks and queueus, and historical data). The second advantage can be demonstrated through the blood bank example: If the telephone numbers recorded in $NEAR-BANKS$ and $SOURCES$ differed, the user may want to know that they

differed rather than having the system make an *ad hoc* decision about which was correct.

A disadvantage of this approach is that the complete approximation of the query may be a very large set, and could take too long to enumerate as a partial answer. We would therefore like to be able to use rules as a shortened, but accurate, description of this set (as proposed in [9]). For example, if a monitoring system in a hospital was asked to provide the results of a routine series of tests performed on a patient ("G-series"), all of which came back with normal results, the system should avoid listing each test individually but abbreviate with "G-series normal".

References

[1] H. Wedekind and G. Zoerntlein, "Prefetching in Realtime Database Applications," in *Proc. ACM-SIGMOD International Conference on Management of Data*, pp. 215–226, 1986.

[2] Presentations by TRW, IBM and Boeing. ONR Funding Workshop, San Jose, CA. December 1987.

[3] E. Jenson, C. Locke, and H. Tokuda, "A Time-Driven Scheduler for Real-Time Operating Systems," in *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 112–122, December 1986.

[4] K. Lin, S. Natarajan, and J. Liu, "Imprecise Results: Utilizing Partial Computations in Real-Time Systems," in *Proceedings of the Real-Time Systems Symposium*, pp. 210–217, December 1987.

[5] D. W. Leinbaugh and M. Yamini, "Guaranteed Response Times in a Hard-Real-Time Environment," *IEEE Transactions on Software Engineering*, vol. 6, pp. 1139–1144, December 1986.

[6] R. Abbott and H. Garcia-Molina, "What is a Real-Time Database System?," in *4'th Workshop on Real-Time Software and Operating Systems*, pp. 134–138, July 1987.

[7] P. Buneman, S. Davidson, and A. Watters, "Querying Independent Databases," *Information Sciences: An International Journal*, 1988. To appear.

[8] P. Buneman, S. Davidson, and A. Watters, "A Semantics for Complex Objects and Approximate Queries: Extended Abstract," in *Principles of Database Systems*, March 1988.

[9] T. Imielinski, "Intelligent Query Answering in Rule Based Systems," *The Journal of Logic Programming*, vol. 4, pp. 229–257, September 1987.