A QUERY-DRIVEN APPROACH TO SIMULATIONS

Alexander Tuzhilin

P. Balasubramanian

Information Systems Department
Stern School of Business
New York University

Working Paper Series
STERN IS-93-44

A Query-Driven Approach to Simulations

Alexander Tuzhilin [*] P. Balasubramanian

Information Systems Department
Leonard N. Stern School of Business
New York University

Abstract

This paper describes a Query-Driven Simulation (QDS) approach to asking questions about outcomes of business processes. In this approach a user issues a query about outcomes of simulation runs and, based on the query asked, appropriate simulations are launched and the answer to the query is determined from the outcomes of these simulations. It is argued that Query-Driven Simulations provide a more declarative, flexible, and interactive approach to asking questions about simulation outcomes than the traditional approaches of letting the end-users run simulations and gather statistics about simulation outcomes. The paper also presents a new simulation system development lifecycle based on the QDS approach.

KEY WORDS: Query-Driven Simulations, Discrete-Event Simulations, Temporal Databases, Query Languages.

1 Introduction

Discrete-event simulations have been extensively used for analyzing performance of various complex industrial systems in situations when it is difficult or impossible to obtain explicit solutions for the analytical models of these systems. For example, in manufacturing organizations decision makers might be interested in the utilization ratios of the machines in their plants over a period of time, the average waiting time for jobs in queues, the scrap rates for their plants, detection of bottlenecks, and so on. To answer these types of questions about future outcomes of processes in a manufacturing system, a simulation model of the system is built, and simulations of this model are run several times. Based on these runs, statistics related to the questions of interest to decision makers (e.g. the utilization ratios, throughput, waiting times, etc.) are collected and presented to the user. We will call this traditional approach *simulate-and-gather-statistics (SAGS)* approach.

[*]Address: 44 West 4th Street, Room 9-78, New York, NY 10012, e-mail: atuzhili@rnd.stern.nyu.edu, pbalasub@rnd.stern.nyu.edu

Traditionally, simulations collect summary statistics in one of the following two ways. In the first approach, summary statistics are computed inside the simulation program, and the program presents these statistics to the user. The main problem with this approach is that the end-user has to modify the program if he or she wants to ask a question about simulation outcomes that goes beyond the set of statistics generated by the program. For example a simulation program may output the total time spent in the system (makespan) for a job but the user may want a breakup of the time spent in the queue, in transit and in process.

In the second approach, various simulated events are recorded in the *trace files*, and then statistics are collected from these trace files by either writing programs in one of the programming languages, such as Fortran or C, or by using one of the statistical packages, such as SAS [SAS89]. The main problem with this approach is that the end-user has to know either a programming language or a statistical package to be able to collect statistics or ask any other questions about the trace files. Otherwise, he or she has to rely on the IS department which provides the end-users with a set of "canned" questions.

Since most of the people who ask questions about future outcomes of business processes in their organizations, such as a foreman, a salesman, or personnel manager, do not know much about simulations, programming languages, or statistical packages, they cannot ask *ad-hoc* questions about future outcomes of their business processes as the questions arise *"on-the-fly"*. Clearly, this situation is unsatisfactory in many organizations, such as manufacturing, transportation, or in the military, where various users want to ask many different questions about simulation outcomes of various models [BT93b].

In this paper, we describe the *Query Driven Simulations (QDS)* approach, that addresses this problem. QDS is an approach to simulations in which the user first asks queries about outcomes of simulations expressed in a *declarative* query language, such as SQL [Dat89], and then appropriate simulations are launched depending on the query, and events necessary to answer the query are recorded in the trace file(s). After the simulation runs are completed, the query is evaluated on the trace files(s) of events recorded by the simulation program. We also present a specific QDS system *Cassandra*$^+$ that implements the QDS approach just described. We also present its query language about simulation outcomes, called *SimQL*, and describe how SimQL queries are processed by the Cassandra$^+$ system. In addition, we describe issues related to performance of the QDS systems, and present the *Query-Driven Modeling Lifecycle* of the model development process.

In the next section, we present the SimQL language using a series of examples. A formal description of the language can be found in [BT93a]. In Section 3 we describe how SimQL queries are processed by the Cassandra$^+$ system. In Section 4, we compare the QDS and SAGS approach. In Section 5, we discuss some optimization issues. In Section 6, we explain what tasks have to be performed to use a QDS system. We describe the Query-Driven modeling lifecycle in Section 7. Finally, in Section 8 we describe some related work.

2 Description of SimQL Language

The SimQL language consists of two subcomponents: the *core* query language subcomponent and the shell into which the core query language is embedded. The core query language subcomponent is the "heart" of SimQL and is used to ask temporal queries[1] about simulation traces. In this paper, we use SQL [Dat89] with timestamps to express core queries, but, as will be explained below, we could have used *any* temporal query language as long as it supports the same data model as the simulation component.

The second subcomponent of SimQL is the *shell* into which the temporal query language is embedded. This shell provides an *interface* between the querying and simulation parts of Cassandra$^+$ that integrates the two components into one system. For example, we specify in the shell such information as the simulation model against which the query is asked, the parameters for that model, for how long simulations should be run, what answer we expect back, i.e. a full relation or just a number, and various additional information that the simulation component of Cassandra$^+$ needs in order to provide the answer to a query.

Example 1 Consider the following query:

How many parts can be finished in the next 10 hours?

It can be expressed in SimQL as:

Initialization:	Real-time
Type:	Event-based
Answer-Semantics:	Numeric
Core-query:	
SELECT	COUNT(Part#)
FROM	FINISHED
Model-Name:	Mfc-Model-4
Confidence-coefficient:	90
Error-of-estimation:	20

[1]Simulation methods deal with process evolving in time and hence we need a temporal query language to ask questions about these processes.

The core query in this example is expressed in SQL as

```
SELECT   COUNT(Part#)
FROM     FINISHED
```

Note that this core-query is embedded in the SimQL shell that provides additional information about the meaning of the query. For example, the parameter **Model-Name** in the shell specifies the name of the simulation model. It tells Cassandra$^+$ that the query is asked against the model Mfc-Model-4.

The parameter **Initialization** = Real-time, specifies that simulations should be done in "real-time," i.e. they should start from the initial state of the system in the model Mfc-Model-4 that represents the current state of the physical system. Alternatively, they could be done "off-line," meaning that the initial state of the system is not specified, and simulations should be run for some time until, e.g., the steady state is reached, and only then the query should be evaluated.

The second parameter in the query, **Type** = Event-based, specifies that the trace file of the simulation model Mfc-Model-4 must be stored as historical event relations [Sno87]. In this case, the simulation trace file(s) are copied into the temporal database without any conversion. Alternatively, the **Type** parameter can be "predicate-based," and this requires conversion from the event-based to the predicate-based representation as will be described in Section 3.2.

The value of the **Answer-Semantics** parameter in Example 1 is *numeric*. It specifies that the query returns back a single number (the number of finished parts in our case). Alternatively, the answer-semantics can be *non-numeric* if the query returns back a relation. We will discuss this semantics in Example 2. We have to distinguish between numeric and non-numeric semantics because the types of answers are different in these two cases as Example 2 will show.

Finally, the parameters **Error-of-estimation** and **Confidence-coefficient** specify what the estimation error of the answer can be and with what confidence we can provide the answer [MWS90]. In our example, the user wants the estimation error to be within 20% of the mean and the confidence-coefficient of the answer to be 90%.

A possible answer to this query can be

> The average number of parts produced within the next 10 hours is 32 ± 3, and we can make this statement with confidence 90%.

In other words, the probability that the answer to the query falls between 29 and 35 parts is 90%.

□

In the first example of a SimQL query, we have described some of the shell parameters. It turns out that there are other parameters in the query which are taken as *default* parameters. For example, **TIME** is one such parameter. If not specified, it is "extracted" from the query (10 hours in our case). If it is present then we assume that the simulations are run only for that time and that the time domain is restricted by this parameter, as the next example shows.

Example 2 Consider the following question that a foreman in a manufacturing plant may want to ask:

What are the parts that will always stay in Cell-1 for the next 5 hours.

This query can be expressed in SimQL as

Initialization:	Real-time
Type:	Predicate-based
Time:	5 hours
Answer-Semantics:	Relational
Core-query:	
SELECT	Part#
FROM	VISITS
WHERE	Cell# = Cell-1
	AND Begin_Time $\leq$ \$NOW
	AND End_Time $\geq$ \$NOW + 5 hours;
Model-Name:	Mfc-Model-2
Parameters:	number_of_cells = 5, job_arrival_rate = 10
Confidence-coefficient:	95
Error-of-estimation:	20
Number-of-answers:	2

The core query in this example is

```
SELECT  Part#
FROM    VISITS
WHERE   Cell# = Cell-1
        AND Begin_Time ≤ $NOW
        AND End_Time ≥ $NOW + 5 hours;
```

Note that the core query is unbounded in the sense that we need to know the values of the VISITS predicate at all the (arbitrarily removed) points in the future to evaluate its value at present [2]. To solve this problem, we specify the **Time** parameter in the shell. The **Time** parameter restricts

[2]Unless we provide some intelligent query processing strategies that can recognize that only the instances of VISITS within the next 5 hours are needed to answer the query.

the temporal domain to the bounded set of times (up to 5 hours from now), and the core query is evaluated on *that* domain.

This query has additional parameters that did not appear in the previous example since default values were assumed for them in Example 1. One of these parameters is **Parameters** that specifies the parameters passed to the simulation model specified in the query. For example, the parameters `number_of_cells = 5` and `job_arrival_rate = 10` in the query are passed directly to the `Mfc-Model-2` model.

The **Answer-semantics** parameter in the query in this example has `relational` as its value. This means that the query returns relations (tables) as its answer. Also, the **Type** parameter has the value `predicate-based`. This means that the relations in the core-query are predicates with two timestamp attributes, specifying the times when a tuple was added to and removed from a relation (unlike events that have only one timestamp attribute). For example, predicate `VISITS(Part#,Cell#,Begin_time,End_time)` has two times associated with it: when a part begins (Begin_time) and ends (End_time) its visit to a cell. Finally, the parameter **Number-of-answers** specifies the number of the most likely answers the user wants specified in the order of decreasing probabilities of these answers. This parameter can appear only in the SimQL queries that have non-numeric values in the **Answer-semantics** parameter.

A possible answer to the query from this example can be

> Most likely, parts PY346, PY378, and PZ216 will always be in Cell-1 within the next five hours; the probability of this is 24% ± 2%, and we make this statement with confidence 95%. The second most likely answer is that parts PZ289 and PY378 will always be in Cell-1 within the next five hours; the probability of this is 21% ± 2%, and we make this statement with confidence 95%.

The query returns two most likely answers because the parameter **Number-of-answers** is 2 in this case. Furthermore, the answers are returned in the decreasing order of their average probability estimates.

Note that the answer to this query is different from the answer to the query in Example 1. This query returns the relation that is the most likely answer to the query and an estimate of the probability of that answer. In contrast to this, the answer to the query with the numeric value of the **Answer-semantics** parameter returns the average estimate of the value of the numeric parameter and the estimated error for this value (32 ± 3 in Example 1).

□

In Example 2, we considered the relational value of the **Answer-semantics** parameter. This value directs Cassandra$^+$ to return the most likely answer(s) to the query. However, the user may sometimes want a different kind of the answer, as the following example shows.

Example 3

Consider the query

> How many days would it take to complete order number JC-243 by each of the three manufacturing plants (PL-1, PL-2, PL3)?

The relational semantics would return a certain answer, e.g. { `(PL-1, 10days)`, `(PL-2, 14days)`, `(PL-3, 12days)` }, and would assign a probability estimate for this answer, e.g. probability 26% $\pm$ 2%. However, we may need a different answer. We may want to know probability estimates for each plant *separately*, e.g., { `(PL-1, 10days)` with probability 23 $\pm$ 2%, `(PL-2, 14days)` with probability 34 $\pm$ 3%, `(PL-3, 12days)` with probability 21 $\pm$ 2% }.

To accommodate for this type of answer, we provide the *tuple* value for the **Answer-semantics** parameter, as the following SimQL query shows

Type:	Predicate-based
Time:	30 days
Answer-Semantics:	Tuple
Core-query:	
SELECT	Plant#, (End_time - Begin_time)
FROM	PROCESS
WHERE	Order# = JC-243;
	AND Plant# IN (PL-1, PL-2, PL-3);
Model-Name:	Mfc-Model-1

where PROCESS is a relation with schema PROCESS(Order#, Plant#, Begin_time, End_time).

□

When the user issues a SimQL query, Cassandra$^+$ determines the simulation model to which the query refers to, determines how many simulation runs N are needed to obtain the answer within the estimates specified by the user, runs this simulation model for N simulation runs, storing simulation traces in trace files, converts the resulting simulation trace files into the temporal database format based on the **Type** parameter, issues the temporal query against each simulation trace, and statistically analyses the answers to these queries.

There are two additional issues related to the process of interaction between SimQL queries and simulations. First, there is the *model management* issue [Bla92]. Cassandra$^+$ must store a set of simulation models against which the user can ask queries. For instance, in Example 2, the query was issued against the manufacturing model Mfc-Model-2 and in Example 1 against model Mfc-Model-4. Therefore, Cassandra$^+$ must provide the capability to store, query and update various models. We will discuss this issue further in Section 3.1.

Second, different models in the modelbase can be written in different simulation languages. For example, Mfc-Model-2 can be written in MODSIM [BDMR90], Mfc-Model-4 in Simscript [Con87], and Bank-Model-12 in Simkit [Int85]. As was stated already in the introduction, one of the important advantages of Cassandra$^+$ is that it can support *any* temporal query language and *any* simulation language as long as the two agree on the data model (so that temporal queries can be asked against the corresponding traces)[3].

The next example shows that SimQL queries can be asked not only about the future but also about the past *and* the future.

Example 4 How many parts will be produced by the end of September, assuming that it is now September 15.

Initialization:	Real-time
Type:	Event-based
Answer-Semantics:	Numeric
Time:	Combined(Past(15), Future(15))
Core-query:	
SELECT	COUNT(PART)
FROM	FINISHED
WHERE	$NOW − 15 < Time
	AND Time < $NOW + 15 days;
Model-Name:	Mfc-Model-1

Note that the **Time** parameter in the query indicates that the simulation model Mfc-Model-1 should be run for 15 days and then the simulation results should be *combined* with the history of relation `FINISHED` over the past 15 days[4]. Finally, the core query is evaluated on the combined relation that has the lifespan of 30 days.

[3]The only convention is that the trace files generated by programs written in different simulation languages must have a certain format. The structure of this format will be discussed in Section 3.2.

[4]The historical data can be obtained by gathering the transactional real-time data about all the events and activities happening on the manufacturing floor [FDJG$^+$92] and then processing this data and storing it in the historical relational database format.

□

The next example shows how SimQL queries can be used for experimental design.

Example 5 How many customers will be serviced in the bank per day if we vary the number of tellers between 5 and 8?

Initialization:	Off-line(Steady-state)
Type:	Event-based
Answer-Semantics:	Numeric
Time:	1 day
Core-query:	
SELECT	COUNT(Customers#)
FROM	SERVICED
Model-Name:	Bank-Model-7
Model-parameter:	number_of_tellers = 5..8
Confidence-coefficient:	95
Error-of-estimation:	10

This query is called a *range* query because it gives rise to *four* individual queries, one query per *each* number of tellers (5, 6, 7 and 8) specified in the **Model-parameter** parameter. As a result of this, SimQL returns to the user four different answers, one answer for each value of the parameter.

Also note that this is an off-line query. This means that simulations are run initially until Bank-Model-7 reaches a steady state. Only after that, simulations will be run for one day of simulated time and traces will be generated starting from that time.

□

In this section we described SimQL language using a series of examples. In the next section we describe how SimQL queries are executed.

3 Execution of SimQL Queries

The algorithm that evaluates a SimQL query is presented in Fig. 1. Initially, a SimQL query (Q) is parsed and the model against which the query is to be evaluated is determined together with the parameters necessary to evaluate the query. After that, Cassandra$^+$ determines if the query is a *range* query, and evaluates each instance of the range query on the given simulation model in a loop (the outermost FOREACH loop). The key part in this process is the evaluation of the query Q on a single simulation run (procedure `compute_answer_for_single_run`). It starts with the *initialization*

```
evaluate_query(Q)

determine the model $M$ against which query $Q$ is evaluated;

determine parameters of query $Q$:
                $T$ - simulation time
                $R$ - range of model instances in $Q$
                $c$ - error of estimation
                $\alpha$ - confidence coefficient

FOREACH model instance $r$ in $R$ DO

      determine the number of simulation runs $N_r$ needed to evaluate
      $r$ within estimation bounds $c$ and $\alpha$;

      FOR $i = 1$ TO $N_r$ DO
        $ANSW_{ri}$ = compute_answer_for_single_run(Q,r,T,i)
      END_FOR

      compute the answer $STAT\_ANSW_r$ to $Q$ for model instance $r$ based on $\{ANSW_{ri}\}_{i=1,N_r}$

      return the answer $STAT\_ANSW_r$ to the user

END_FOREACH

END

compute_answer_for_single_run(Q,r,T,i)

      initialize $i$-th run of model instance $r$

      run simulation of run $i$ for time $T$ and store the results in trace file $TR_{ri}$

      convert the trace file $TR_{ri}$ into the database format $DB_{ri}$

      evaluate the core query core(Q) on $DB_{ri}$ and store the results in $ANSW_{ri}$

      RETURN($ANSW_{ri}$)

END
```

Figure 1: Evaluation of SimQL queries in Cassandra$^+$.

of the simulation model to be executed by retrieving the *default* parameters for this model from the modelbase and overriding them with the simulation parameters specified in the query. After that, Cassandra$^+$ launches the initialized simulation model for time T. As a result of this simulation, the model generates trace files. We assume that these trace files are regular ASCII or EBCIDIC files generated by the WRITE statements of the simulation program and that they are stored in a certain format that will be described in detail in Section 3.2. After that, Cassandra$^+$ converts the trace files into the database tables having the format of the DBMS being used to express core queries. For example, if we use Oracle [Ora87] as a DBMS and the trace file is an ASCII file, then the ASCII trace file is converted into Oracle's relational table. Finally, the core query is evaluated against the trace files converted into the database format. After all N_r simulation runs are executed for the model instance r and all the answers $ANSW_{ri}$ are determined, the final statistical answer $STAT\_ANSW_{ri}$ is computed based on the **Answer-Semantics** parameter of the query.

So far we presented a short overview of the query processing strategy used in Cassandra$^+$. In the rest of this section, we describe parts of this strategy in detail. We start in Section 3.1 with the description of the modelbase and the information it stores to aid in the evaluation of SimQL queries. Then we describe in Section 3.2 the format of the temporal database files and how core queries are evaluated on these files. After that, we describe how a SimQL query is evaluated on a single simulation run in Section 3.3. In Section 3.4 we describe how Cassandra$^+$ evaluates a SimQL query based on the results of multiple runs. Finally, we describe the implementation of the Cassandra$^+$ system in Section 3.5.

3.1 A Modelbase

The modelbase contains information about the simulation models that the user can query. It is a central repository of all the information about all the models used in an organization. The modelbase is needed in order to instantiate and run simulation models and to convert their outputs into historical relational database formats. In Cassandra$^+$, we store the modelbase in a relational database as a *set* of tables since the information we need in the modelbase is not normalized [Ull88], and thus it is better not to place all the information pertaining to the models in a single table[5].

The modelbase contains the main table with one record per one model, and other tables that "link" additional information about the model (such as the description of the events for the model) to the main table. The main table in the modelbase contains the following fields:

[5]For example, we store the descriptions of all the events that a model traces, and it is better to keep this information in a separate table.

1. *Model name*, that serves as a key. For instance, Mfc-Model-4, Banking-Model-6 are examples of model names.

2. *Target simulation language*: the language in which the simulation model is written, e.g. Modsim, Simscript, etc.

3. *Default simulation parameters*: parameters that are used in the model. These parameters are taken as defaults. They can be over-written by the parameters that the user specified in the query (such as *number_of_cells=5* and *job_arrival_rate=10* in Example 2).

4. *Name of the simulation program*: this field contains the *name* of the object module for the simulation model, as stored in the secondary storage. When the actual simulation is ready to be run, this object module is dynamically linked to Cassandra$^+$ module using the name of the module stored in this field.

5. *Past information*: this field provides the name of the relation that contains the names of the relations that stores past information about events and predicates that concerns the simulation model.

6. *Events traced by the model*: this field contains the name of the relation that specified the names of the events that are traced by the simulation model.

7. *Event-to-predicate conversion programs*: this field contains the name of the relation that contains the names of the programs that builds the predicates from the events. For instance, in Example 2, predicate *VISITS* can be computed from two event predicates *ARRIVES* and *DEPARTS*.

8. *Online-vs-offline flag*: the flag specifying if real-time queries can be asked against this model; if the value of the flag is "online" then the modelbase must contain initialization programs described in Item 9.

9. *Initialization programs for real-time queries*: if the flag in Item 8 is "online" then real-time queries can be asked on this model; in this case, this field provides the *names* of initialization programs that compute the initial state of the system from which simulations start.

10. *Optimization flag*: this is a boolean field specifying if queries on the simulation model can be optimized. In order for a query to be optimized on a simulation model, the model should have its PRINT statements written according to a certain convention so that the query could pass the optimization information to the simulation model.

Event Parameter(s)

Time Stamp

Figure 2: Event File Format

11. *Working model_base*: this field contains the name of the relation that contains the particulars of all the model instances that have to be executed based on the current query.

Typically, information is retrieved from a modelbase using some query language [Len93, MW89, Geo87]. For example, [Geo87] describes a querying language for the structured modeling approach that allows the end-user to retrieve, and update information about the models stored in the modelbase. However, we want to point out the difference between this type of a query language and SimQL. SimQL is used to query outcomes of simulations: to answer a SimQL query, we have to retrieve the model from the modelbase and *run* it. In contrast to this, the query language on the modelbase is used to retrieve information about the models stored in the modelbase. Unlike SimQL, it does not require execution of these models.

However, we provide only very limited model retrieval and maintenance capabilities in the current version of Cassandra$^+$ since our major goal was to develop a working prototype of a QDS system and since model management capabilities are not directly related to this goal.

3.2 Temporal Databases

SimQL queries ask questions about outcomes of simulations, and these outcomes are written into trace files in the form of events. Therefore, it is natural to use a *temporal* query language [TCG$^+$93] as a core query language in SimQL since events occur in time and since user questions are temporal in their nature. Since a query language comes as part of a database management system, it is also natural to use a *temporal* database [TCG$^+$93] to store query traces.

However temporal databases are not commercially available at present. Therefore, we selected SQL with timestamps as the core query language in the paper. Another important reason for this choice is that SQL is a very popular query language and is used as a standard in relational databases. Furthermore, many temporal queries can be expressed in SQL with timestamps, and therefore we do not constrain ourselves by this choice. An obvious disadvantage of using SQL with timestamps is that some queries will look quite "ugly" when expressed in it.

Part#	Cell#	Time
P3	Cell-1	10:24
P4	Cell-3	10:45
P5	Cell-5	10:58
P6	Cell-1	11:05
P10	Cell-4	12:01
P22	Cell-6	12:55

Figure 3: Trace File for Event ARRIVAL

Predicate Parameter(s)
Begin-Time
End-Time

Figure 4: Predicate File Format

As we saw in Section 2, some core SimQL queries are asked against events and others against predicates. To handle this distinction, we consider two types of SQL relations in this paper. One type has a single time column (see Figure 2) and corresponds to a *historical event relation* of TQuel [Sno87]. The time column specifies the time at which the rest of the tuple belongs to the relation. For example, the relation ARRIVAL(Part#,Cell#,Time) (Figure 3) specifies the time at which a part arrives at a cell.

The other type of relation has two time columns (see Figure 4) and corresponds to a *historical interval relation* of TQuel [Sno87]. The two time columns indicate the beginning and the end of the time when the tuple is true. For example, the relation LOCATED(Part#,Cell#,Begin_time,End_time) (Figure 5) specifies the beginning and the end of the time period when a part was located in a cell.

In the rest of the paper, we will follow TQuel's terminology and call the first type of historical relation *event* relation. The second type of historical relation will be called either *predicate-based* or *interval-based.*

We have described two important "building blocks" of a QDS system: the modelbase and the temporal database. We are ready to describe the details of the algorithm presented in Figure 1, i.e. explain how a query is actually evaluated on a simulation model. We begin with the description of this process for a single simulation run.

Part#	Cell#	Begin-time	End-time
P3	Cell-1	10:24	10:30
P4	Cell-3	10:45	10:55
P5	Cell-5	10:58	11:10
P30	Cell-1	10:45	NOW
P10	Cell-4	12:01	12:31
P22	Cell-6	12:55	13:15

Figure 5: Predicate Located

3.3 Evaluation of the Core Query on a Single Simulation Run

In this section, we describe the details of the procedure `compute_answer_for_single_run` from Figure 1.

When the simulation program is executed, it writes various events that the query asked it to trace into the *trace* files, one file per event. For example various occurrences of event FINISHED for the query from Example 1 are recorded into the trace file FINISHED that may have the events { `Finished(P3, 10:23)`, `Finished(P6, 10:47)`, `Finished(P8, 11:13)` } recorded in it. These trace files are stored as ASCII (or EBCIDIC) files.

After that, the trace files containing events are converted into historical database relations. If **Type** parameter in the query is "event-based" then the conversion process is simple and is done on a record-by-record basis: one event in the trace file generates the corresponding record in the historical relation. If **Type** parameter in the query is "predicate-based" then we have to convert events into predicates. To do this, Cassandra$^+$ accesses the conversion routines for the simulation model that are stored in the modelbase. For each predicate-based relation in the query, it checks whether the appropriate conversion routine exists in the modelbase. If all of the necessary routines exist, Cassandra$^+$ invokes them and does the conversion. After the conversion is finished, the resulting trace files are stored as interval-based relations. For example, the events ARRIVAL(Part#,Cell#,Time) and DEPARTURE(Part#,Cell#,Time) can be converted into the interval-based relation LOCATED(Part#,Cell#,Begin_time,End_time) by setting its attribute Begin_time to the time when Part# arrives at Cell#, and setting End_time to the time when Part# departs from Cell#.

After the trace files are converted into historical database relations, the core query is evaluated on the temporal database according to the semantics of the query language in which the core query is expressed. This completes the description of how a SimQL query is evaluated on a single simulation run.

3.4 Statistical Evaluation of a SimQL Query

Once we know the answer to a SimQL query for a single simulation run, we can explain how it can be evaluated on multiple simulation runs. However, as we mentioned before, we have to distinguish between the two cases when the core query returns a *numeric* and when it returns a *non-numeric* answer because the answers for multiple runs are quite different in these two cases.

3.4.1 Numeric Answers

In this case, a query returns a number as an answer for a single simulation run. Therefore, multiple simulation runs generate a set of numbers as answers, one number per run. Also, some of these numbers may be repeated in the set. For instance, assume we do five simulation runs for the query from Example 1 and assume we get the answers { 18, 20, 19, 18, 19 } for these runs. To determine the answer to a "numeric" query, we assume that the average of this answer is normally distributed. Then we estimate the mean and variance of this normal distribution from the sample of answers for individual runs and determine the confidence interval for the average answer based on the bounds specified in the query. If this confidence interval is within these bounds (e.g. 20% of the mean), we stop the simulations. If not, we increase the number of simulations to be run, as we will describe in Section 3.4.3, in order to get the confidence interval within the limits and run that many simulations again.

Therefore, the semantics of a SimQL query for the *numeric* type of an answer is defined by the confidence interval for that value that lies within the estimation error specified in the query. For example, if the mean value of the number of parts that will be finished within the next 10 hours is 20, the estimation error is 30%, and the probability that this number is between 17 and 23 is 95% then the answer that Cassandra$^+$ returns to the user who asked this query is

> The average number of parts produced within the next 10 hours is 20 ± 3, and we can make this statement with confidence 95%.

3.4.2 Non-Numeric Answers

In this case, the query returns a relation, and not a single number[6]. Since we consider relations instead of numbers, we cannot make statements about averages for these relations. Instead, we

[6] Of course, the relation can also consist of a single number in the degenerate case. For example, we could specify **Answer-Semantics** = `Relational` for the query in Example 1. However, if we did so, we could not talk about an *average* number of parts produced in 10 hours.

ANSWER	*FREQUENCY*
{PY346, PY378, PZ216}	0.4
{PY346, PZ216}	0.2
{PU629, PY378, PZ216}	0.4

Figure 6: Frequencies of Different Answers for the Query From Example 2.

determine the *most likely* answers in one of the following two ways. The first way is to determine which answer, as a *relation*, is the most likely one. This alternative can be selected by specifying **Answer-semantics** = `relational` in the query. The other choice is to determine which tuples in the answer are the most likely ones. This alternative can be selected by specifying **Answer-semantics** = `tuple`. We describe relational and tuple semantics of answers now, assuming that the parameter **Number-of-answers** in the query is equal to N.

If *relational semantics* is selected in the query then we compute frequencies for each *relation* returned as an answer in at least one simulation run. This type of semantics is needed when the user treats query answers as outcomes of scenarios and wants to know what is the chance of each outcome. For instance, assume that we made five simulation runs for the query from Example 2 "what are the parts that will always stay in Cell-1 for the next 5 hours," and assume we get the frequencies for each of the resulting answers as shown in Figure 6.

As we increase the number of simulation runs, the distribution of the estimate of the frequencies of each simulation outcome converges to a normal distribution[MWS90]. Then our goal is to estimate N largest frequencies based on the estimation parameters *error-of-estimation* and *confidence coefficient* specified in the query.

The semantics of a SimQL query for the *relational* type of an answer is defined by N relations having N *largest* frequencies in this distribution (based on the estimation of the means of these frequencies), i.e. the query returns the first N *most likely* answers. If $N = 1$ then the query returns a most likely answer. If there is more than one most likely answer then either all of them can be returned, or one of them selected at random. In our example, either both most likely answers, { PY346, PY378, PZ216 } and { PU629, PY378, PZ216 }, having frequency 0.4, or one of them chosen at random is returned if relational semantics is selected. We assume that $N = 1$ as the *default* value for the relational semantics of answers.

If *tuple semantics* is selected in the query, then we compute frequency of occurrence of each tuple in the set of answers. In other words, we want to know the chance of each tuple belonging

TUPLE	FREQUENCY
PZ216	100%
PY378	80%
PY346	60%
PU629	40%

Figure 7: A Sample Tuple Semantics Answer

to the answer. For example, if we made K simulation runs, and the tuple $(a_1, \ldots, a_n)$ occurred in answers for k runs, then the frequency of $(a_1, \ldots, a_n)$ is $\frac{k}{K}$. Then the semantics of a SimQL query for the *tuple* type of the **Answer-semantics** parameter is defined by the first N tuples having the highest frequencies. Furthermore the default value for the parameter **Number-of-answers** is *All*, i.e. the user wants to know frequencies of occurrence of all of the tuples in the answers. For example, if five simulation runs produce answers as presented in Figure 6, then the answer to the query from Example 2 based on the tuple semantics is shown in Figure 7 for **Number-of-answers** = *All*. In other words, the tuple semantics for this query specifies the chance various parts will always stay in Cell-1 for the next 5 hours.

In summary, we showed how the same non-numeric query can have two different answers depending on whether the semantics is relational or tuple-based. The two examples presented above show that the two semantics are complimentary to each other, that both of them are needed in practice, and that it is up to the user to select the semantics he or she wants.

Once we know the semantics of answers for SimQL queries, our next task is to determine the number of simulation runs necessary to answer a non-numeric query with the estimation error and the confidence interval specified in the query.

3.4.3 Determination of the Number of Simulation Runs Needed to Answer the Query

An important task for Cassandra$^+$ is to determine the number of simulations to be run so that the answer returned by the query has the error of estimation as specified by the user and this answer can be stated with the confidence specified by the user. To do this, we have to distinguish between *numeric* and *non-numeric* answer-semantics. We first describe the *numeric* case.

To determine the number of simulation runs for the numeric case, we proceed as follows. We start with an initial number of runs, and run simulations for that many runs. Then we compute the mean and variance for these runs and test if the result satisfies the constraints as specified in the query. If it does, we stop; otherwise, we recompute the number of simulation runs and repeat the process again until we either satisfy the query constraints or run out of time. Formally, we proceed as follows.

Let N_k be the number of simulation runs selected at the k-th trial. Let $\hat{y_k}$ and $\sigma_{\hat{y_k}}$ be estimations of mean and variance of the answer to the *numeric* query for the k-th trial. Also, let p be the error of estimation measured as a percentage of the real (absolute) error to the mean, and let α be the confidence coefficient.

We describe the process of selection of the number of simulation runs inductively. We start the process with some set of initial runs, i.e. select N_0 to be some initial number, e.g. $N_0 = 50$. Assume that we have done k trials already and that we selected N_k. We do N_k simulation runs and compute the estimations of mean and variance for these runs, i.e. $\hat{y_k}$ and $\sigma_{\hat{y_k}}$. Then we check if the answer for N_k simulation runs satisfies the query constraints. The formulae for the lower and upper confidence limits for $\hat{y}$ are [MWS90]:

$$\text{lower confidence limit (LCL)} = \hat{y} - Z_{\frac{\alpha}{2}}\ \sigma_{\hat{y}}$$

$$\text{upper confidence limit (UCL)} = \hat{y} + Z_{\frac{\alpha}{2}}\ \sigma_{\hat{y}}$$

Therefore, the confidence limit itself is $2Z_{\frac{\alpha}{2}}\ \sigma_{\hat{y}}$, and it should be less than or equal to $p\hat{y}$ (where p is the error-of-estimation specified by the user). Combining these observations, we obtain the test specifying if the answer for N_k simulation runs is within the query constraints:

$$\frac{2Z_{\frac{\alpha}{2}}\ \sigma_{\hat{y_k}}}{\hat{y_k}} \leq p \tag{1}$$

If this condition holds, then we select N_k as the number of simulation runs. If this condition does not hold, then we compute N_{k+1} using [MWS90]:

$$N_{k+1} = (\frac{2\ \sigma_{\hat{y_k}}}{p\hat{y_k}})^2 \tag{2}$$

If it turns out that N_{k+1} is too large, then we keep running simulations until we reach the time limit specified as a default parameter by the user in the modelbase. In this case, we examine the number of completed runs N'. If $N' > N_k$ then Cassandra$^+$ evaluates the query based on N' simulation runs by estimating the mean and variance of the sample of N' runs; otherwise, it returns the answer based on the N_k runs. In either case, the system returns the answer of the form $\hat{y} \pm d$ with confidence α, where d is the confidence interval for the normal distribution $N(\hat{y}, \sigma_{\hat{y}})$ and confidence coefficient α, and $\hat{y}$ and $\sigma^2_{\hat{y}}$ are the mean and variance of the sample.

As an example of this process, assume that we do five simulation runs, $N_0 = 5$, for the query from Example 1 and we get the following answers {18, 22, 19, 25, 17 } for these runs. The mean value of this sample, $\hat{y_0}$, is 20.2 and the variance, $\sigma^2_{\hat{y_0}}$, is 10.7. We then calculate the 95% confidence interval for the parameter which in this case is 20.2 ± 5.38. If the user wants the estimation error of the answer to be within 20% of the mean, then we have not satisfied that constraint and we have to increase the number of simulation runs N_1 and continue the simulations. Substituting the numbers into equation (2), we obtain $N_1 = (\frac{2\ \sigma_{\hat{y_0}}}{p\hat{y_0}})^2 \ = \ 10.5 \ \approx \ 11$

Assume that we make 11 simulation runs and get the following answers {18, 19, 18, 20, 19, 18, 19, 17, 19, 20, 19 }. We again compute the confidence interval which in this case turns out to be 18.72 ± 1.48. This value is within the bounds that the user specified (20%). Therefore, we terminate the simulation runs. This completes our description for the *numeric* case of answer semantics.

If the answer-semantics parameter is non-numeric, i.e. either *relational* or *tuple*, then we proceed as follows. We first estimate *probabilities* for each possible answer. To do this, we proceed as follows. We divide simulation runs in N batches comprised on n runs each. For each batch, we run n simulations and compute the proportions of each distinct outcome. Then we determine the total set of unique outcomes $R_1, \ldots, R_m$ as a union of distinct outcomes for all N individual batches and determine the proportion p_{ij} of outcome j in batch i, for $i = 1, \ldots, N$ and $j = 1, \ldots, m$. After that, we compute the mean μ_j and variance σ_j^2 of the sample $p_{1j}, \ldots, p_{Nj}$ $(j = 1, \ldots, m)$ and select the first q largest means μ_j, where q is the **Number-of-answers** parameter specified in the SimQL query (see Example 2). For each of the selected outcomes, we test if it satisfies the constraints specified in the query by using equation (1), in which we substitute the mean μ_j and the variance σ_j^2. If *all* of them satisfy this constraint, we terminate the process of query evaluation and return the list of q largest μ_j's to the user along with the confidence intervals. If even one of the constraints is not satisfied, we recompute the batch size N and repeat this process all over again. The new batch size N' is determined as follows. For each of the largest selected means μ_j, $j = 1, \ldots, q$, we compute the size of the next simulation batch N_j using equation (2) in which we substitute the mean μ_j and the variance σ_j^2. We set the size of the next batch N' to be equal to:

$$N' \ = \ \max_{1 \leq j \leq q}\{ \quad N_j\}$$

This completes the description of the procedure of how we compute the batch size for the *non-numeric* type of **Answer-semantics** parameter.

3.5 Implementation of Cassandra$^+$

The Cassandra$^+$ system described in this paper was implemented in C under UNIX. We selected Ingres [Ing89, Sto86] as the database that stores simulation traces and SQL with timestamps as the core query language for SimQL.

We store the modelbase as an Ingres database because the modelbase can be quite large and because we want to store the models in a central location and want to provide uniform access to it. The modelbase is stored in several tables (3 in the current implementation) because, as we pointed out in Section 3.1, the data stored in it is unnormalized. The Cassandra$^+$ system interacts with the modelbase by using *dynamic SQL* [Ing89, EN90] since it is necessary to formulate SQL queries against the modelbase from within Cassandra$^+$ dynamically "on-the-fly."

The modelbase can store simulation models written in *any* simulation language, such as Modsim [BDMR90], Simscript [Con87], Siman [PSS90], as long as their WRITE statements satisfy the conventions described in Section 3.2. However, it is more difficult to switch from one core query language to another (and hence from one DBMS to another) than to switch between two simulation languages. For example, it is not possible to switch from Ingres to Oracle in the current implementation of Cassandra$^+$ without rewriting (small) portions of its code for the following two reasons. First of all, we have to use dynamic SQL in order to access the modelbase, and the implementation of dynamic SQL is system dependent. Also, the conversion routines from ASCII trace files to the database format are also system dependent. For these reasons, we have to provide some modifications to the Cassandra$^+$'s code when we move from one DBMS to another in the current version of Cassandra$^+$. However, these modifications are relatively small and are quite "local" to the code. Therefore, we believe that they can be easily automated in the future.

4 Comparison of QDS and SAGS Approaches

In Sections 2 and 3, we described a Query-Driven Simulation (QDS) System Cassandra$^+$ and presented its query language SimQL. It follows from this description that Query-Driven Simulations, and Cassandra$^+$ in particular, have the following advantages over traditional simulate-and-gather-statistics (SAGS) approaches.

First, QDS approach provides a more *declarative* way of asking questions about outcomes of simulations than the SAGS approach. The user formulates questions in a declarative general-purpose query language that tells the QDS system *what* the user wants to know. If the user uses this language he/she does not have to specify *how* the system has to obtain the answer. In particular, the user does not have to know any simulation and statistical packages, or write any programs.

Second, QDS approach gives the user extra *flexibility*. The user can ask *any* query expressible in the query language of the QDS system (e.g. SimQL for Cassandra$^+$). This flexibility makes end-users less dependent of the MIS department since they do not have to rely on the simulation specialists when they want to ask an additional question not supported by the available information system.

Third, QDS approach is more *interactive* than the traditional SAGS approach. The user of the QDS system can ask queries "on-the-fly" as they arise without any help from the simulation specialist.

Finally, QDS approach *automatically* provides statistical answers to the questions asked by the user without any extra work on his/her part. Unlike the SAGS approach, in which the user has to determine how many simulation runs are needed in order to obtain the answer within the user-specified constraints, the QDS approach does all this work for the user.

However, the QDS approach has certain limitations in comparison to the traditional SAGS approach. First of all, some of the questions expressible in the SAGS approach cannot be expressed as QDS queries. To explain why this is the case, consider Cassandra$^+$, SimQL as its query language, and SQL as the core language for SimQL. It is well-known that SQL is not Turing-complete, i.e. SQL queries cannot compute arbitrary recursive functions. For example, SQL queries cannot compute transitive closure [AU79]. In contrast to this, the SAGS approach can compute an arbitrary recursive function (including the transitive closure) since most of the simulation languages, such as MODSIM [BDMR90], SIMSCRIPT [Con87], GPSS [IBM70], are Turing-complete. Therefore, the QDS approach has less *expressive power* than the SAGS approach in general.

However, this limitation of SQL and other relational query languages is well-known. It is solved in practice by *embedding* SQL queries in programming languages, such as C, COBOL, and ASSEMBLER. Therefore, we expect that the limited expressive power of the QDS approach will be solved in the same way as it is solved in practice for SQL.

Another limitation of the QDS approach is that it requires the generation of large trace files. In the worst case, the QDS system must trace *every* event occurring in the simulation model. In contrast to this, the SAGS approach can simply compute summary statistics inside the simulation model and thus generate no trace files *at all.* Furthermore, if the trace files are generated by the SAGS approach, the model developer has a control over the events he or she wants to trace. This means that only a small fraction of all the events may end up being recorded in the trace files in the SAGS approach. Since recording events in the trace files can slow performance of QDS systems in comparison to SAGS systems, it is important to develop query optimization techniques for the QDS approach. These techniques would allow recording only those events that are necessary for answering the query. We will discuss these query optimization techniques in the next section.

5 Improving Performance of a QDS System

If a simulation model does not know what types of events the query requires to trace, then it must record *all* the events occurring during the simulation process into the trace file(s); otherwise, the QDS system may not have enough information in order to answer some of the queries. This approach can involve a large overhead since recording all the events into the trace file(s) is quite a time consuming operation. For example, assume that a simulation model contains 100 different types of events, and the query refers only to 3 events. This means that an unoptimized query processing strategy has to trace all 100 events instead of 3.

An alternative approach would be to determine for the QDS system what events the simulation model should trace and pass this information to the simulation model. In the context of the previous example, this would mean that the simulation model would trace only 3 events relevant to the query. However, this information passing can be done only if the simulation model is "ready" to accept this information, i.e. if the model is written according to the following conventions. Each event in the optimized simulation model has an *event flag* associated with it. Furthermore, each "WRITE event" statement of the unoptimized program must be replaced with the conditional write statement "*IF* event-flag is on *THEN* WRITE event" in the optimized program. In addition to this convention, an optimized simulation program must accept a list of events that the query wants it to trace as one of its arguments. Finally, each of the events passed to the simulation program sets its corresponding event-flag "on" at the beginning of the program. If a simulation model follows these conventions then it is "ready" for optimization.

To distinguish models that are "ready" for the optimization from the models that are not "ready," the modelbase has an *optimization flag* as one of the fields in the record describing the model. If the flag is "on," then the model is ready for the optimization; otherwise, it is not.

The query processing strategy for the optimized simulation model (i.e. with the optimization flag "on") works as follows. At first, the query determines which events the simulation model should trace. This depends on whether the query is event- or predicate-based. If the query is event-based, the QDS system parses the core query and determines all the events referenced in it. If the query is predicate-based, the QDS system parses the query and determines all the predicates referenced in it. Then it accesses the modelbase and, for each predicate, determines the events that are used to compute the predicate. The union of all the events taken over all the predicates determines the set of events that the simulation model should trace. After the QDS system determines the set of events needed by the query, this information is passed to the simulation model as one of the parameters. After that, the optimized simulation model will trace only the events that were passed to it by the query.

The optimization scheme described in this section imposes minimal constraints on the model developer. The model developer has to include events passed by the query as one of the arguments to the program, add a piece of code that sets the event flags "on" for the events passed to the program, and add conditional WRITE statements to trace the events. These changes that an optimized simulation model requires are minimal and can either be checked by some model validation tool or automatically inserted into unoptimized model by a pre-processing software.

The scheme just described helps the QDS system trace only the events referenced in the query. However, we can reduce the number of events traced by the system even further if we trace only the *instances* of events necessary to answer the query. For example, consider the query "in which machine part PJ-374 will be 5 hours from now?" In order to answer this query, we have just to run simulations for 5 hours without tracing any events *at all* and at the end see in which machine part PJ-374 is located at that time.

To solve this kind of optimization problem we have to do three things:

1. Parse the query and determine under what conditions each event pertinent to the query must be recorded in the trace file. These conditions should be written as programs in the simulation language of the model, and we will call them *optimization routines.*

2. The event flags in the previous optimization scheme must be replaced with calls to the corresponding optimization routines described in Part 1.

3. The optimization routines generated by the query optimization module must be compiled and dynamically linked to the simulation program while the query is being processed by the QDS system.

As we can see from this description, this optimization problem is a difficult one. It requires parsing the query and determination of preconditions for the events the query needs, which is a difficult problem in itself. Furthermore, it requires the implementation of the complex dynamic linking scheme described above. Therefore, we do not support this type of optimization in the current version of Cassandra$^+$ system, leaving it as a topic of future research.

Note that the query optimization techniques described in this section improve the performance of the system but also have certain costs associated with them. First, the simulation models must comply with the standards described in this section that model developers must follow to make the model suitable for optimization. Second, it becomes more difficult to switch from one core query language to another because queries have to be parsed now, and each language needs its own parser.

6 Running a QDS System

In this section we address the question of what it takes for a model developer to build a new simulation model and for the model administrator to install it into the modelbase. In particular, we describe what extra steps are needed over and above the traditional SAGS approach to develop a model that can be used in a QDS system.

When a model developer builds a simulation model for a QDS system he or she should follow the following guidelines. First of all, an unoptimized simulation model must trace *all* the events in that model. Furthermore, events must have a format described in Section 3.2. In addition, if the user intends to optimize queries on that model then the model must follow the rules described in Section 5.

Secondly, the simulation program must have a set of arguments that are used by the QDS system to pass simulation parameters from a query to the simulation program. These arguments must include (i) the number of simulation runs that the program should run, (ii) for how long to run them, (iii) the arguments of the model itself (e.g. the number of tellers, average service times, etc. in a banking model), (iv) random seed, (v) the list of events to be traced by the optimized model. We need to pass the random seed to the simulation model as an external parameter so that we would not get the same results each time we run the model. The random seed is generated at random by the initialization module of the QDS system that prepares the simulation model for the execution.

Thirdly, if the model is used to do real-time simulations, then the model developer must supply the initialization routines. These routines set the initial state of the simulation model to the current state of the system that is obtained by accessing the historic data about the system stored in the modelbase and retrieving the data that is currently valid, i.e. that has the timestamp *now*.

Once a simulation model is developed, it is installed into a modelbase by the model administrator. As part of this installation procedure, the model administrator must supply all the necessary

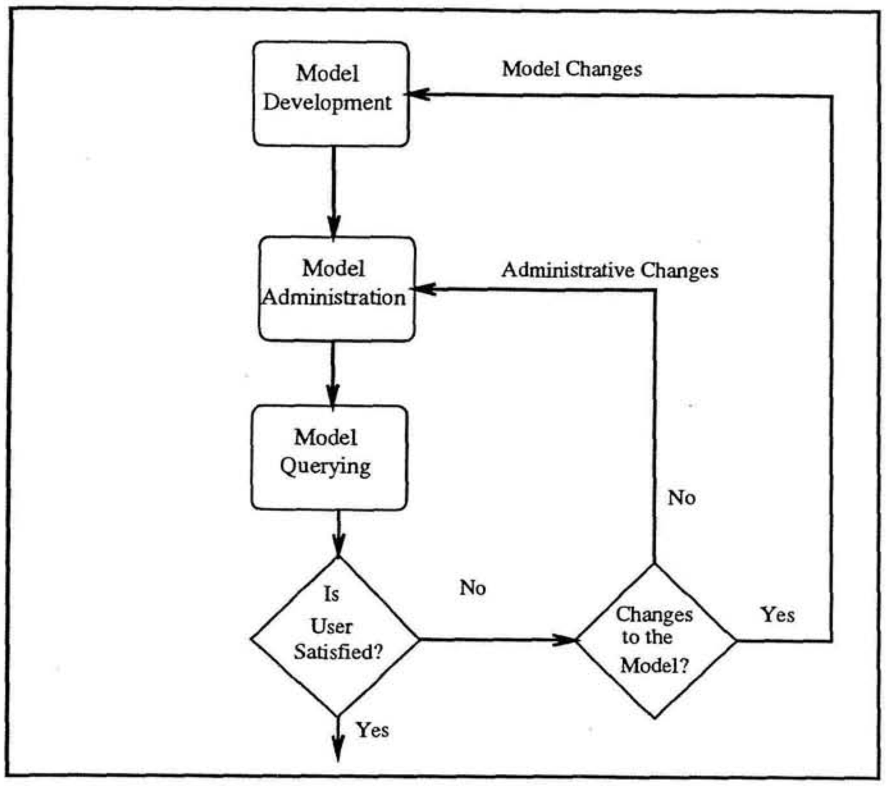

Figure 8: Query-Driven Modeling Lifecycle

information about the model needed by the modelbase, such as the name of the model, the language in which it is written, the list of events used in the model, the list of predicates that the user can query (see Section 3.1 for the detailed description of these parameters). As part of this process, the model administrator must provide a set of routines converting events into predicates, as described in Section 3.1.

7 Query-Driven Modeling Lifecycle

In the paper, we described several activities required to build a working QDS system, such as development of simulation models by the model developer, installation of these models into the modelbase, and querying simulation outcomes of the models in the modelbase by the end-user. It turns out that these activities are not independent of each other but are related in the way shown in Figure 8.

As Figure 8 shows, the model developer initially designs a simulation model and delivers it to the model administrator who installs the model into the modelbase. Part of this installation procedure is the specification of all the information that the modelbase requires to know about the newly installed model, such as the list of events used in the model, the name of the object module containing the model, and the list of the default parameters. After the model administrator installs the model in the modelbase, end-users can start using it by issuing queries about simulation outcomes expressed in SimQL.

At this point, either the user is satisfied with the model and keeps using it, or he/she might experience some problems. There are two types of problems the user can run into. First, the information, as specified in the modelbase, does not satisfy user's needs. For example, it may turn out that the user wants to ask a query about a predicate that does not appear in the modelbase but can be easily computed from the events that the simulation model traces. As another example, the user may want to run a real-time query on a model, but discovers that the QDS system would not let him/her do this because the online-vs-offline flag for the model is set off in the modelbase. These two problems can be solved by the *model administrator* who makes appropriate changes to the *modelbase*. In particular, he or she writes a conversion routine for the new predicate and installs this predicate and the routine into the modelbase. To solve the problem in the second example, the administrator supplies the initialization routines for the simulation model and sets the online-vs-offline flag in the simulation model "on". Note that the simulation model itself is not changed for this type of a problem.

The second problem that the user can encounter arises when he or she asks a query that the *simulation model* cannot handle because it does not have enough information to answer the query. For example, the user may want to know how many parts will be painted in red color within the next 10 hours, and the simulation model does not keep track of the colors of different parts and the painting information. In this case, the *model developer* must adjust the *simulation model* accordingly.

In both cases, Query-Driven Simulations provide a *feedback loop* in the process of model development: the models are modified based on the feedback coming from the user after the user asks various questions about these models. We call the process of development, installation, usage, and model adjustment based on the users' feedback a *Query-Driven Modeling Lifecycle*. The model development process can go through several iterations before it converges to a stable simulation model satisfying end-user's needs.

8 Related Work

Query languages in the context of simulations were studied before. In [Len93], a database of simulation models, called a modelbase, was constructed based on the structured modeling approach [Geo87]. As part of the structured modeling approach, [Len93] uses the query language defined for this approach by Geoffrion[Geo87]. Although the system supports queries, these queries are used in a totally different context than SimQL queries: they are used for asking questions about the models *themselves* (e.g. which models stored in the modelbase are manufacturing models), not about simulation traces produced by running the models, as is the case with SimQL.

In [MW89], Miller and Weyrich developed the SIMODULA system that has its own SQL-like query language for asking questions about simulations (with object-oriented features added to it). Each model has a relation of input parameters and outcomes of previously executed simulations associated with that model. For example, a banking model may have a BankScenario relation associated with it that has input parameters, such as number of tellers, mean interarrival rate, mean service time, and the output parameters, such as throughput and the service time, as its attributes. If the user wants to ask a question about throughput and average waiting time for the banking model with input parameters mean interarrival rate being 4.0, mean service time being 6.0 and the number of tellers equal to 2, then SIMODULA checks in the BankScenario relation if this model has been run before. If it was, it retrieves the answer from relation BankScenario (values of attributes Throughput and AverageWaitingTime). Otherwise, SIMODULA launches the simulation with the input parameters retrieved from the query and the rest of them set to defaults.

In this paper we present a more extensive approach to Query-Driven Simulations by allowing SimQL queries to *drive* simulations and not just *launch* them as is done in [MW89]. We also allow the user to query simulation traces in an ad-hoc manner instead of letting him/her ask a fixed set of questions on summary statistics about a single simulation run. Furthermore, we express answers in statistical terms that require more than a single simulation run to obtain the answer. Finally, we allow a loose coupling between *any* database query language and *any* simulation language as long as trace files generated by simulation programs conform to the standard described in Section 3.2.

In [Tuz92, Tuz93], the idea of asking queries on simulation traces was proposed, and a SimTL language was presented. SimTL consists of a simulation and a querying components. The simulation component is based on a temporal logic programming language [AM89], and the querying component is based on temporal logic [MP92]. Thus queries about simulation outcomes expressed in temporal logic are asked about simulations generated by temporal logic programs. This means that SimTL is a tightly coupled simulation and querying system, in which both components depend on the formalism of temporal logic.

In this paper, we extend the work of [Tuz92, Tuz93] by integrating an *arbitrary* temporal query language with an *arbitrary* simulation language. Therefore, unlike SimTL, where the interface between querying and simulating components is well-understood and is based on temporal logic, we have to develop a proper interface between these components in order to achieve independence between the query and simulation languages. Also, unlike the work in [Tuz92, Tuz93], we provide answers to SimQL queries in statistical terms.

9 Conclusions

In this paper, we described a Query-Driven Simulation system Cassandra$^+$ that allows end-users to ask various questions about outcomes of simulations. We described the query language SimQL, explained how SimQL queries are executed, and presented some query optimization strategies. We also described how model development, installation, usage, and model adjustment are integrated into a Query-Driven Modeling Lifecycle.

One of the important features of Cassandra$^+$ is that it can support *any* temporal relational query language asked about simulation models written in *any* simulation language as long as trace files generated by these models conform to a certain standard.

Query-driven simulations provide more declarative, flexible, and interactive ways of asking questions about simulation outcomes than the traditional simulate-and-gather-statistics approaches. They allow end-users to ask various questions in a declarative query language in an ad-hoc manner "on the fly," just as relational query languages allow the users to ask questions about the data stored in databases.

As a future work, we want to add *Query-Driven Animation* capabilities to Cassandra$^+$. Traditionally, animations are done for the entire system being simulated. In contrast to this, we plan to do animations only of the parts of the system pertaining to the query being asked. For example, if the user wants to know how many parts will go through the manufacturing cell C3 within the next shift, it may be sufficient for him or her just to see only the parts arriving at and departing from cell C3. The challenging question here is to determine for a given query what parts of the system should be animated.

References

[AM89] M. Abadi and Z. Manna. Temporal Logic Programming. *Symbolic Computation*, 8:277–295, 1989.

[AU79] A. Aho and J. Ullman. Optimal partial match retrieval when fields are independently specified. *ACM Transactions On Database Systems*, 4(2):168–179, 1979.

[BDMR90] R. Belanger, B. Donovan, K. Morse, and D. Rockower. *MODSIM II Reference Manual.* CACI, 1990.

[Bla92] Blanning, R. and Whinston, A. and Ai-Chang, M. and Dhar, V. and Holsapple, C. and Jarke, M. and Kimbrough, S. and Lerch, J. and Prietula, M. Model management systems. In Edward A. Stohr and Benn R. Konsynski, editors, *Information Systems and Decision Processes.* IEEE Computer Society Press, 1992.

[BT93a] P. Balasubramanian and A. Tuzhilin. Cassandra$^+$: A System for doing Query Driven Simulation. Working Paper IS-93-40, Stern School of Business, NYU, 1993.

[BT93b] P. Balasubramanian and A. Tuzhilin. Using Query-Driven Simulations for Querying Outcomes of Business Processes. Working Paper IS-93-38, Stern School of Business, NYU, 1993.

[Con87] Consolidated Analysis Centers, Inc. *UNIX SIMSCRIPT II.5 User's Manual*, 1987.

[Dat89] C.J. Date. *A Guide to the SQL Standard.* Addison-Wesley, 1st edition, 1989.

[EN90] R. Elmasri and S. Navate. *Fundamental of Database Systems.* The Benjamin/Cummings Publishing Company, 2nd edition, 1990.

[FDJG$^+$92] K. Fordyce, R. Dunki-Jacobs, B. Gerard, R. Sell, and G. Sullivan. Logistics Management System (LMS): An Advanced Decision Support System for Dispatch or Short Interval Scheduling. *Production and Operations Management*, 1(1):70–86, Winter 1992.

[Geo87] A.M. Geoffrion. An Introduction to Structured Modeling. *Management Science*, 33(5):547–588, May 1987.

[IBM70] IBM. *General Purpose Simulation System/360 User's Manual*, 1970.

[Ing89] Ingres. *INGRES/OpenSQL Reference Manual for the UNIX and VMS Operating System.* Relational Technology Inc., 1989.

[Int85] IntelliCorp, Mountain View, Calif. *The SIMKIT System: Knowledge-Based Simulation Tools in KEE*, 1985.

[Len93] M.L. Lenard. A Prototype Implementation of a Model Management System for Discrete-Event Simulation Models. In *Proceedings of the 1993 Winter Simulation Conference*, pages 33–39, 1993.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1992.

[MW89] J.A. Miller and O.R. Weyrich. Query Driven Simulation Using SIMODULA. In *Proceedings of the* 22^{nd} *Annual Simulation Symposium*, 1989.

[MWS90] W. Mendenhall, D.D. Wackerly, and R.L. Scheaffer. *Mathematical Statistics with Applications.* PWS-KENT, 4th edition, 1990.

[Ora87] Oracle Corporation. *ORACLE Overview and Introduction to SQL ORACLE Part No. 3801 User's Manual*, 1987.

[PSS90] C.D. Pegden, R.E. Shannon, and P.P. Sadowski. *Introduction to simulation using SIMAN.* McGraw-Hill, New York, 1990.

[SAS89] SAS Institute, Raleigh, NC. *SAS User's Guide*, 1989.

[Sno87] R. Snodgrass. The temporal query language TQuel. *ACM Transactions On Database Systems*, 12(2):247–298, 1987.

[Sto86] M. Stonebarker. *The INGRES Papers: Anatomy of a Relational Database System.* Addison Wesley Publishing Company, Inc., 1986.

[TCG+93] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass. *Temporal Databases.* Benjamin/Cummings, 1993.

[Tuz92] A. Tuzhilin. SimTL: A Simulation Language Based on Temporal Logic. *TRANSACTIONs of The Society for Computer Simulation*, 9(2):086–099, 1992.

[Tuz93] A. Tuzhilin. Applications of temporal databases to knowledge-based simulations. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal Databases.* Benjamin Cummings, 1993.

[Ull88] J Ullman. *Principles of Database and Knowledge-Base Systems (Vol. I).* Computer Science Press, 1st edition, 1988.